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. Dipl.-Ing. Alexander Hartl

Vienna, 29th May, 2022

Erklärung zur Verfassung der Arbeit

Jonas Ferdigg, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Mai 2022

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich auf dem langen Weg zu diesem Abschluss begleitet haben.

Besonders möchte ich mich bei Frau Prof. Tanja Zseby und Herrn Dipl.-Ing. Alexander Hartl bedanken, die mich in den letzten Monaten vor dem Abschluss dieser Thesis mit sehr viel konstruktivem Feedback und Anregungen unterstützt haben und ohne die diese Arbeit zu diesem Zeitpunkt wohl noch längst nicht abgeschlossen wäre.

Als nächstes möchte ich mich sehr herzlich bei meinen Eltern bedanken, die mir den nötigen finanziellen und mentalen Rückhalt gegeben haben und mich zu einem Menschen erzogen haben, der es fertig bringt, einen Master an der TU Wien abzuschließen.

Bedanken möchte ich mich auch bei meiner Freundin, ohne die diese Arbeit wohl weit mehr Rechtschreibfehler enthalten würde und die es mir in turbulenten Zeiten ermöglicht hat, mich trotzdem auf den Abschluss dieser Arbeit zu konzentrieren.

Ein dankendes Wort möchte ich auch an meine Studienkollegen richten dafür, dass sie die Freuden und manches Leiden des Studiums mit mir geteilt und mich stets motiviert haben, ein besserer Ingenieur zu werden.

Besonders bedanken möchte ich mich auch bei all meinen Professorinnen und Professoren sowie Tutorinnen und Tutoren der TU Wien dafür, dass sie ihr Wissen mit mir geteilt haben und den doch oft sehr trockenen Stoff mit viel Humor etwas verdaulicher gemacht haben.

Ein besonderer Dank gilt auch meiner WG aus der Schweglerstraße und all meinen Freunden, die dafür gesorgt haben, dass ich mich wärend des Studiums nicht nur fachlich, sondern auch menschlich weiterentwickelt habe und ohne die diese Arbeit schon vor Jahren fertig geworden wäre.

Kurzfassung

Techniken des Machine Learnings (ML) und Deep Neural Networks (DNNs) haben in verschiedene Disziplinen Einzug gehalten und ihre möglichen Vorteile werden für eine Vielzahl von Anwendungen untersucht. Die Fähigkeiten moderner ML Modelle in der Mustererkennung haben Expertensysteme oder sogar Menschen in speziellen Anwendungsbereichen längst überholt. Ihre Fähigkeit, scheinbar komplexe Daten genau zu klassifizieren, eignet sich auch für den Einsatz im Zusammenhang mit Network Intrusion Detection (NID). Ein häufig verwendetes Muster beim Trainieren von Neuronalen Netzen im Bereich des Natural Language Processing (NLP) ist, ein Modell mit vielen nicht-gelabelten Daten selbstüberwacht vorzutrainieren (to *pre-train*) und dann nur noch mit einem kleinen markierten Datensatz zu verfeindern (*fine-tuning*). Im Zuge des Vortrainierens kann ein Modell beauftragt werden, verhüllte Informationsteile aus den Eingabedaten zu rekonstruieren, künftige Eingaben vorherzusagen, oder andere Fragen zu den Eingabedaten zu stellen, deren Antwort aus den unmarkierten Daten ableitbar ist. Im nächsten Schritt wird eine kleine Menge von gekennzeichneten Daten zur Feinabstimmung des Modells verwendet, um die angestrebte eigentliche Aufgabe zu erfüllen. Inspiriert von den Erfolgen von Modellen wie BERT und seinen Nachfolgern haben wir die gleichen Methoden verwendet, um die Klassifizierungsgenauigkeit für Deep-Learning basierte Network Intrusion Detection Systeme (NIDS) zu erhöhen. In unserer Forschung beantworten wir die Frage, ob Pre-Training Paradigmen, die im NLP verwendet werden, auch im Kontext von Deep-Learning basierten NIDS anwendbar sind. Wir haben ein Vortraining für *Long Short-Term Memory (LSTM)* und *transformer encoder* Modellen mit einer Reihe von selbstüberwachten Trainingsmethoden auf Basis von Autocodierung und Autoregression durchgeführt, um die binäre Klassifizierung von Netzwerkverkehrsaufzeichnungen zu verbessern. Nach dem Vortraining verwenden wir eine überwachte Feinabstimmung mit einer kleinen Menge markierter Daten, um dem Modell beizubringen, wie es die Daten in angreifende und gutartige Datenströme einteilen kann. Als Trainingsdaten haben wir eine *flow*-Darstellung der NID-Datensätze CIC-IDS2017 und UNSW-NB15 mit dem *flow*-Schlüssel *<dstIp, srcIp, dstPort, srcPort, protocolId>* verwendet. Unsere flows bestehen aus einer Folge von Tensoren, die paket- und flussspezifische Merkmale enthalten. Unsere Ergebnisse zeigen, dass die Klassifizierungsgenauigkeit durch Vortraining verbessert werden kann, aber nur in bestimmten Instanzen. Weitere Untersuchungen sind erforderlich, um zu sehen, ob unsere Ergebnisse verallgemeinert werden können.

Abstract

Machine learning techniques and Deep Neural Networks (DNNs) have found their way into various disciplines. Their possible benefits are explored for a diverse range of applications. The pattern matching capabilities of modern day machine learning models have long surpassed expert systems or even humans in narrow applications. Their ability to accurately classify seemingly complex data makes them well suited to also be used in the context of Network Intrusion Detection (NID). While supervised learning is still most effective when training machine learning models, its feasibility is often stifled by a lack of expensive labeled data. For this, among other reasons, researchers at the forefront of machine learning development, especially in the field of Natural Language Processing (NLP), have began to pre-train their models on large amounts of unlabeled data to overcome the scarcity of labeled data. A commonly used pattern e.g. used to train Google's Bidirectional Encoder Representations from Transformers (BERT) model, is to pre-train large scale machine learning models in a self-supervised manner. This is done by tasking the model to either reconstruct omitted parts of information from the input data, predicting future input or asking other questions about the input data to which the answer is derivable from the unlabeled data. Only a small amount of labeled data is then used to fine-tune the model to perform the target downstream task. Inspired by the achievements of models like BERT and its successors, we used the same methods to increase classification accuracy for deep learning based Network Intrusion Detection System (NIDS). In our research we try to answer the question whether pre-training paradigms used in NLP can improve classification accuracy for deep learning based NIDS. We performed pre-training on Long Short-Term Memory (LSTM) and transformer encoder models with a set of devised auto encoding and auto regression based self-supervised training methods to improve binary classification of network traffic records. After pre-training we use supervised fine-tuning with a small amount of labeled data to teach the model how to classify the data into attack and benign flows. As training data we used flow representations of the CIC-IDS2017 and UNSW-NB15 NID datasets with the flow key <dstIp, srcIp, dstPort, srcPort, protocolId>. Our flows consist of a sequence of tensors containing packet and flow specific features. Our results show that classification accuracy can be improved through pre-training, but only in specific instances. Further inquiry is needed to see if our results can be generalized.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Questions 2

1.3 Approach 3

1.4 Contribution 3

1.5 Structure 3

2 Background **5**

2.1 Notation 5

2.2 Machine Learning 5

2.3 Artificial Neural Networks 6

2.4 Stochastic Gradient Descent 7

2.5 Backpropagation 7

2.6 Recurrent Neural Networks 8

2.7 Long Short-Term Memory 9

2.8 Attention and Transformers 10

2.9 Self-supervised Learning 12

2.10 Auto Encoder 13

2.11 Pre-Training and Fine-Tuning 13

2.12 Performance Metrics 14

3 State of the art **17**

3.1 Machine Learning for Network Intrusion Detection 17

3.2 Self-supervised Pre-training for LSTMs and Transformer Networks 21

4 Methodology **29**

4.1 Datasets 29

4.2 Data Representation 33

4.3 Machine Learning Models 33
4.4 Framework and Training 36
4.5 Metrics and Validation 38

5 Experiments 39
5.1 Self-supervised Pre-training for Long Short-Term Memory Networks . 40
5.2 Self-supervised Pre-training for Transformer Encoder Networks 49

6 Results 55
6.1 Long Short-Term Memory Model 56
6.2 Transformer Model 69
6.3 Explainability 72

7 Discussion 85

8 Conclusion 93

A Appendix 96
A.1 Transformer per Category Results 96

List of Figures 103

List of Tables 105

Acronyms 109

Bibliography 113

CHAPTER 1

Introduction

1.1 Motivation

With the progressing digitalization of more and more aspects of society, cyber security continues to 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 like it is the case with e.g. DoS and brute force attacks. To stop such attacks it is necessary to identify them within the vast flow of benign 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. A machine learning based approach has the potential to combine both and overcome the downsides each one has individually. As Machine Learning (ML) is a rapidly developing field, its steady improvement fueled the advance of Neural Network (NN) based IDSs, which start to show promising results [SYZ20], [MDES18], [PKBB19]. 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 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/pre-training is seeing increased use in the realm of ML as unlabeled data can be used to boost the performance without need 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 NLP is 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, i.e. without using labeled data. The pre-trained network can then be fine-tuned for more specific tasks like question answering or text classification. Analogously, 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 (GPUs) 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 LSTMs and transformer encoder networks can be improved with self-supervised pre-training in a scenario where only little labeled and a lot more unlabeled data is available. In our experiments we are looking at scenarios where the ratio of labeled to unlabeled data is 1:8, 1:80 and an extreme case where the ratio is over 1:100, depending on the dataset. For performance we are mainly looking at the accuracy of classification, but we are also keeping track of other commonly used evaluation metrics like the F1 score, recall, precision, *etc.* The problem to solve is a binary classification problem for which the model is to group flows into *attack* and *benign* classes.

- 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 models 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 pre-training, we fine-tune the network with very little labeled training data to teach it how to classify the flows into benign and attack categories. 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 without pre-training. They also show which pre-training methods are more and which are less beneficial for classification accuracy. While we are only training the models for binary classification, we are also tracking the specific accuracy for all of the different types of attacks occurring in the datasets. We then take a closer look at how and why classification behavior differs for the pre-trained models and what the reasons might be for cases where pre-training did not improve performance.

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 NIDSs.
- Development of new pre-training methods for LSTMs and transformer encoder models in the context of NIDSs.
- Close inspection of the used datasets.

1.5 Structure

After this introduction section, we provide some background information and define terminology, acronyms and mathematical notation used throughout the thesis. Subsequently we provide an overview of the current state-of-the-art of NNs research for sequence-to-sequence modeling, pre-training for such models and ML supported NIDSs in general in the *State of the Art* section 3. Reasoning behind our methodology, data representation and other decisions made can be found in the dedicated *Methodology* section 4. A detailed description of the conducted experiments and the parameters used 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 considerations for future research.

CHAPTER 2

Background

Artificial Neural Networks (ANNs) have shown great improvements over the last years due to increasing computational power, more sophisticated models and smarter training algorithms [KSCP21], [ASR21]. 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 [JEP$^+$21], [SGSG19], [SYS$^+$20]. 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 [MDES18], [PKBB19]. ANNs are especially suited for IDSs due to their capability to classify data with high accuracy based on seemingly complex patterns. 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.

2.1 Notation

Throughout the thesis, sequences are denoted $x^{(n)}$ while an element in a vector or matrix would be denoted in subscript e.g. x_i or $W_{i,j}$ respectively, with matrices always written in capital letters. Superscript letters (except for T, which is the *transpose* operator and will never be used as label) without parenthesis are part of the variable name, e.g. W^Q is the *query* matrix in the attention function. Deviations from this notation are stated explicitly.

2.2 Machine Learning

Machine learning describes the study of computer algorithms which are *trained* or fitted to optimize a given criterion without the need to specifically program them. The algorithm constructs a model based on input-output pairs which describe how the model should behave. To ensure that the algorithm is learning patterns and structures of problems 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 produces a best guess for the output. In the validation phase, the network processes unseen data, i.e. data not used during training, to ensure that the model did not just learn the training data by heart. Popular traditional machine learning algorithms and models used for classification are k-Nearest Neighbors (KNN), Support Vector Machine (SVM), Naive Bayes or Decision Tree Classifier (DTC)

2.3 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*.

During training, a metric of difference, often called *loss*, is then calculated between the output of the model and the ground truth, i.e. the expected output. The function used to calculate the loss is called a *loss function* or *cost function* and must be differentiable. The network is trained to perform a desired transformation by adjusting its weights/parameters through virtue of *backpropagation* and Stochastic Gradient Descent (SGD). The network produces output $\hat{y}$ at the last layer after processing input x. A scalar cost/loss value is calculated by the 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 (CEL), and for regression Squared Error Loss (SEL) or L1 loss is typically used. Backpropagation 2.5 computes the gradient of the loss function, which is then used by a gradient method like SGD to iteratively update all weights in order to minimize (or maximize) $C(\hat{y}, y)$. As we only aim to distinguish between attack and no-attack flows, and therefore have only two classes, we are using binary CEL with *mean* reduction which is defined as:

$$C(\hat{y}, y) = -\frac{1}{N}\sum_{n}^{N}[y_n \cdot log(\hat{y}_n) + (1 - y_n) \cdot log(1 - \hat{y}_n)] \tag{2.1}$$

With $\hat{y}$ and y being predicted, and target data and N the batch size.

Other methods to train an ANN are e.g. the Conjugate gradient method or the Levenberg-Marquardt algorithm, but SGD is by far the most popular.

2.4 Stochastic Gradient Descent

Stoachstic Gradient Descent is by far the most popular algorithm used for training modern neural networks. It is an iterative first-order optimization algorithm aimed to iteratively improve an objective function by updating its parameters towards its lowest point, i.e. the local maximum (or minimum). For non-stochastic gradient descent it must be possible to calculate an exact gradient. In the case of machine learning, the function to minimize is the sum (or mean) of the loss of all records in the dataset. The function, which is to be optimized, would then be $Q(\theta) = \sum_i^N C(g(x_i), y_i, \theta)$, where N is the number of records in the dataset. Input x_i and output data y_i are seen as constants and θ represents all weights in the model. The length of the resulting gradient vector $\nabla_\theta Q(\theta)$ would therefore be the cardinality of theta $|\theta|$ which is the number of parameters used in the model. Function $f(\theta)$ is then a function that accumulates the loss for every record in the dataset. To put this into perspective: the datasets we use contain around 2 million records and the LSTM model contains 5 million weights. One can see that it is not possible for computers at this date to calculate an exact gradient. Therefore, a stochastic approach is used to estimate a gradient $\nabla_\theta \hat{Q}(\theta)$. The parameters of the objective function are then iteratively updated by a small amount towards the steepest slope, i.e. the gradient:

$$\theta = \theta - \eta \nabla_\theta Q(\theta) \tag{2.2}$$

η is called the *learning rate* and is an important hyper parameter in machine learning, which must be tuned to the model and data at hand.

2.5 Backpropagation

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 is processed and information is flowing *forward* through the network, producing output $\hat{y} = g(x)$ 2.3, hence this is called a *forward-pass* or *forward-propagation*. The model output culminates into a single scalar cost after applying a loss function $C(\hat{y}, y)$, which can be interpreted as a measure of distance between the model output $\hat{y}$ and the target output y. For ML the backpropagation 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 the record that was processed. For this purpose, the weights w are deemed parameters of the forward-propagation and inputs 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.

2.6 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* or *hidden state*. 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. In the case of the Elman network, two parameter matrices are involved in the calculation of the output:

$$h^{(t)} = \sigma^h(W^h x^{(t)} + U^h h^{(t-1)} + b^h) \tag{2.3}$$

$$\hat{y}^{(t)} = \sigma^{\hat{y}}(W^{\hat{y}} h^{(t)} + b^{\hat{y}}) \tag{2.4}$$

With W^h, $W^{\hat{y}}$ and U^h being the parameter matrices, and b^h and $b^{\hat{y}}$ being a parameter vectors. σ^h and $\sigma^{\hat{y}}$ constitute (potentially different) activation functions. 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 general2.3, a differentiable loss function $C(\hat{y}^{(t)}, y^{(t)})$ can be calculated which can again be minimized by applying backpropagation 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.

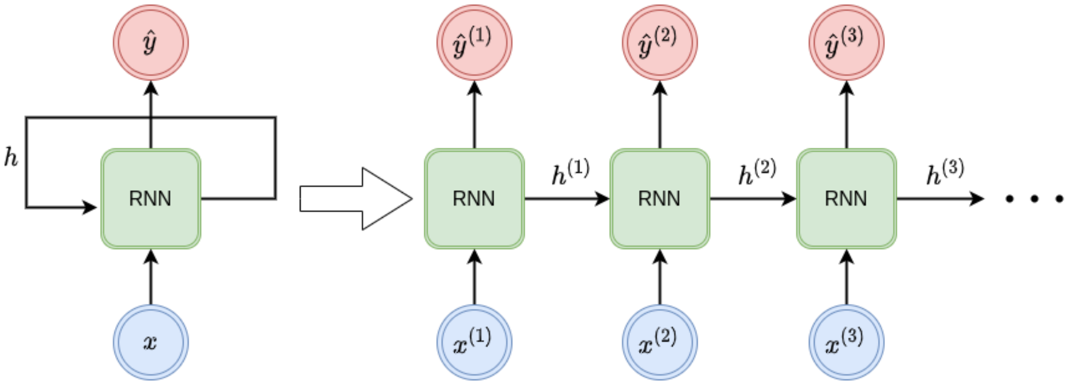

Figure 2.1: Depiction of an unrolled RNN with $x^{(t)}$ being the input sequence, $\hat{y}^{(t)}$ the output sequence, and $h^{(t)}$ the internal state of the RNN after each processing stage.

2.7 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 2.2. 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.

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 pointwise multiplication with a vector that holds values between 0 and 1. The three sigma activations seen in 2.2 produce the gate vectors. The input gate $i^{(t)}$ controls whether the memory cell is updated. The forget gate $f^{(t)}$ controls how much of the old state is to be forgotten. The output gate $o^{(t)}$ 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 $\bar{C}$ after multiplication with the input gate vector $i^{(t)}$. The repeated addition of a *tanh* activation distributes gradients and vanishing/exploding gradients are mitigated.

$$i^{(t)} = \sigma(W^i[h^{(t-1)}, x^{(t)}] + b^i) \tag{2.5}$$

$$f^{(t)} = \sigma(W^f[h^{(t-1)}, x^{(t)}] + b^f) \tag{2.6}$$

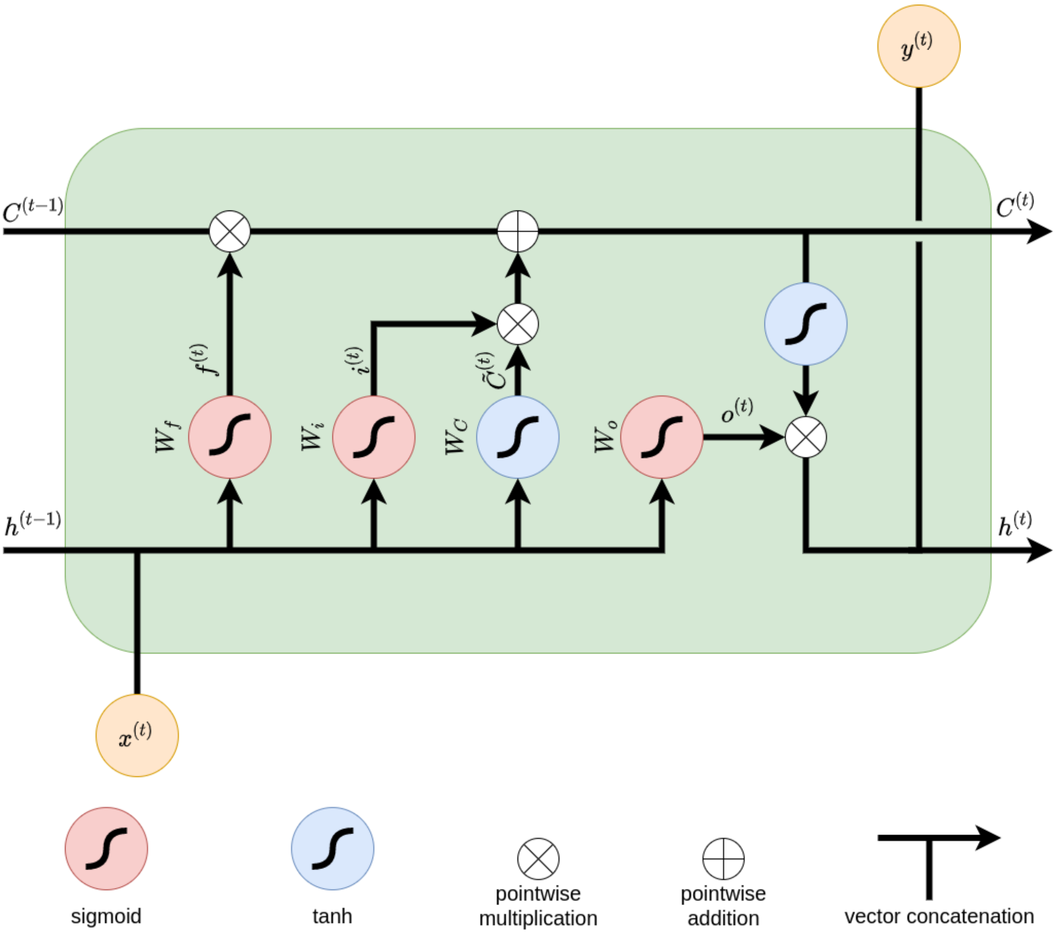

Figure 2.2: One LSTM memory cell [Lip15]

$$o^{(t)} = \sigma(W^o[h^{(t-1)}, x^{(t)}] + b^o) \tag{2.7}$$

$$\bar{C} = \tanh(W^C[h^{(t-1)}, x^t] + b^C) \tag{2.8}$$

2.8 Attention and Transformers

In 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 or packets in a flow. 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

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.9}$$

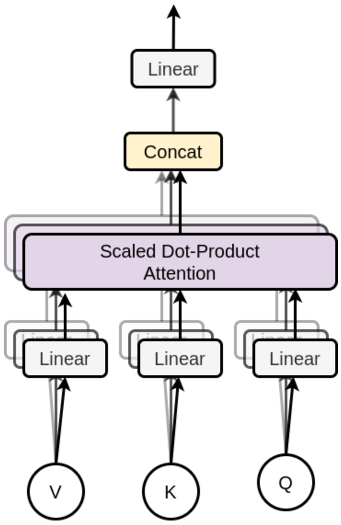

Figure 2.3: Self attention layer of Transformer by [VSP+17b]

"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

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V), i = 1, ..., h \tag{2.10}$$

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{2.11}$$

depicted in figure 2.3. The Multi-Head Attention block from 2.3 is used in the transformer encoder block 2.4 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 2.4. 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.

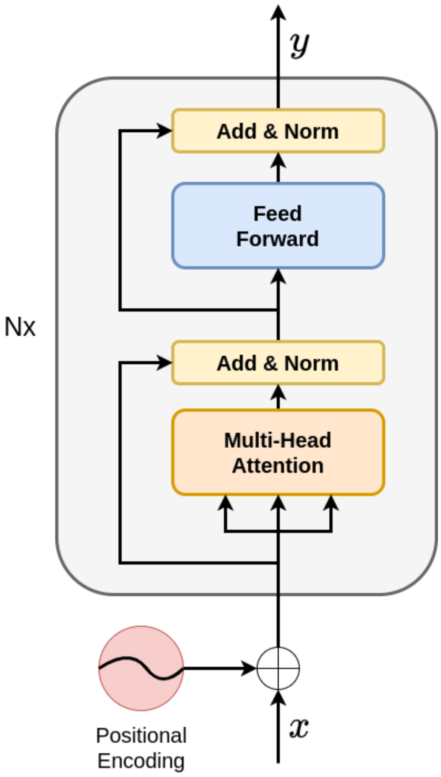

Figure 2.4: Transformer encoder model as proposed by [VSP+17b]

2.9 Self-supervised Learning

Supervised learning is most effective when teaching a NN the desired projection, 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 or alternative. 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 calculated 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, i.e. abstract features of the data which will help solve the actual task.

2.10 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 2.5. 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 or context of the data.

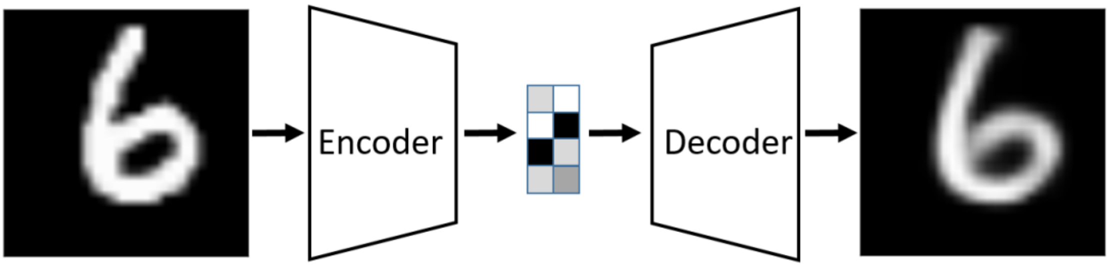

Figure 2.5: Visualization of an auto encoder. The input is encoded and subsequently decoded yielding and approximate reconstruction of the image [BKG20]

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 or the next layer of auto encoder.

2.11 Pre-Training and Fine-Tuning

Pre-training with subsequent *fine-tuning* describes a methodology of training a NN in two separate phases. E.g. Google's BERT for NLP is pre-trained in a self-supervised fashion with vast amounts of text (3.3 billion words) [DCLT18]. *Self-supervised* in this context means that the input, or parts of it, are also used as the training target. Depending on the task of the model, i.e. translation, question answering, text generation, the model's parameters are then fine-tuned with labeled data to fit the given task. *Fine-tuning* then involves updating the weights of a pre-trained model by training it on a task specific labeled dataset, which is usually much smaller than the dataset used for pre-training [BMR+20]. Up to the release date of this paper, the pre-training - fine-tuning approach is still among the most effective approaches available when it comes to training large scale (>1 billion parameters) NLP models, but researchers have since aimed to decrease the need for labeled data even further by only presenting the model with very few, or even just one, example of a correctly executed downstream task [BMR+20] instead of fine-tuning on labeled data.

2.12 Performance Metrics

To measure the effectiveness of different IDSs and machine learning models in general, 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 [Pow08]. Commonly used metrics are

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

  $$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.12}$$

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

  $$Precision = \frac{TP}{TP + FP} \tag{2.13}$$

- **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.

  $$Recall = DR = \frac{TP}{TP + FN} \tag{2.14}$$

- **Specificity** is defined as the ration of true negative samples to total negative samples. The metric describes the probability that a benign flow will be categorized as such by the IDS.

  $$Specificity = \frac{TN}{TN + FP} \tag{2.15}$$

- **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.

$$FNR = MAR = \frac{FN}{TP + FN} \tag{2.16}$$

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

$$FPR = FAR = \frac{FP}{TP + FP} \tag{2.17}$$

- **F1 Measure or F Score** is calculated from the precision and recall of the test and is an alternative description of the accuracy of a statistical analysis.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.18}$$

CHAPTER 3

State of the art

As the topic of this thesis is rather specific and novel in the context if NID, comparable research is hard to find. Overall, the thesis works on the two subjects of self-supervised pre-training for NNs and for Deep Learning (DL) supported NIDS. Here we are looking at state-of-the-art research of both aspects individually. Special focus lays on achievements in machine learning based NLP due to the similar structure in input data and semantic.

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 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 follows: the data source is a network based IDS with flow based detection using deep learning methods as depicted in figure 3.1. As can be see in figure 3.2, the detection method is machine learning based anomaly detection and the machine learning model is a deep learning model based on LSTM and transformer networks, 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..

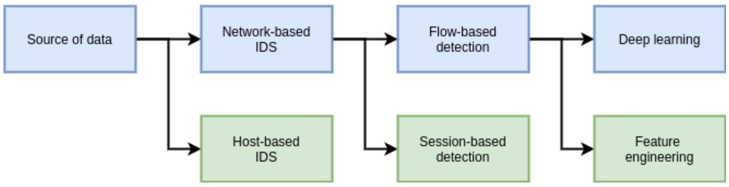

Figure 3.1: Our design decisions (blue) and alternatives (green) based on the data source taxonomy proposed by Hongyu Liu et al. [LL19].

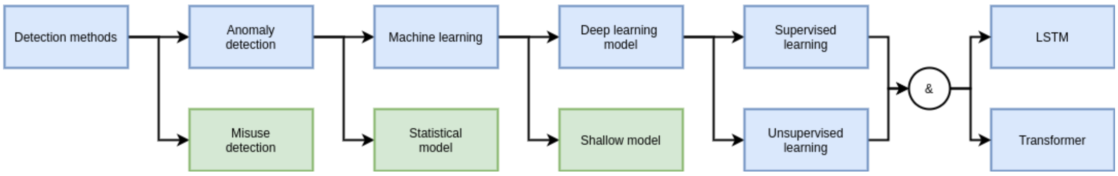

Figure 3.2: Our design decisions (blue) and alternatives (green) based on the detection method taxonomy proposed by Hongyu Liu et al. [LL19].

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 approaches is much greater, we are focusing on 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 the paper focuses 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 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 were 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 2.12 to assess the performance of the different models. This research was especially relevant for u, 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.

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

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%	1.2%	98.97%
CIC-IDS2017 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%

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

	DNN			LSTM			Bi-LSTM		
Attacks	**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**		
Attacks	**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%	85.42%	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%

Table 3.2: Evaluation metrics of DL models with the five data sets in binary classification. P denotes precision, R recall and F1-M the F1 measure. [SYZ20]

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. Additionally, we use a transformer encoder model for classification which was not included in the comparison done by Ahmed Samy et al. Furthermore, the models in their paper were trained exclusively in a supervised manner with at least 70% of the data available in each dataset which differs from our approach of using very little labeled data and relying on self-supervised pre-training.

Compared to other machine learning models there is much less research published on

Model	Precision (%)	Recall (%)	F-Score (%)	FPR (%)
Bi-LSTM	88.86	91.85	90.33	0.15
CRF	48.09	15.41	23.34	0.22
ANID	97.29	94.40	95.28	0.03

Table 3.3: Model performance comparison for CIC-IDS2017 dataset [TICE19]

transformer or attention based networks for IDS. In one of the few papers published on the topic by Mangxuan Tan et al., the authors constructed their own attention based model, similar to the classic transformer model [TICE19], which they coined "Attention for Network Intrusion Detection (ANID)". The model includes a Feed Forward (FF), a Scaled Dot-Product Attention (SDA) and a Scaled Dot-Product Self-Attention (SDSA) component which are the same as in the traditional transformer model by Vaswani et al. but combined differently. Like us, the authors used the CIC-IDS2017 dataset for their experiments but use a fundamentally different approach to data pre-processing and feature selection. For pre-processing, the authors constructed a sequenced of feature vectors by dividing the raw pcap traces into time slots of 0.5 seconds and labeling them as 1 (attack class) if an attack occurred in that time frame and 0 (benign class) otherwise. In the resulting pre-processed dataset, each record contains a sequence of 10 sequential time slots. As feature representation, they use a selection of statistical features aggregated over each time slot resulting in 19 features [TICE19]. Furthermore, they removed 95% of attack slots of each attack type to simulate a real world scenario where attacks are much rarer than they are in the dataset. They compared the performance of their model to a conventional Bi-LSTM model and a Conditional Random Fields (CLF). Their results can be seen in table 3.3.

Even though the authors used the same dataset as we did for our research, comparability with our models is limited due to the different approach to dataset pre-processing and feature selection. The paper still shows that attention based models can achieve a performance improvement over LSTM models in certain cases which gives us reason to look further into them.

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 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]. Today, the limiting factor is often the amount of labeled data available to train a network. Hence, the topic of unsupervised or self-supervised training is explored by researchers of different fields in hope of enhancing their models without the need for more labeled data. Special attention was given to papers in the NLP sector as the methods we use are successfully deployed there.

In the following section we are looking at some of the related papers on the topic of self-supervised or unsupervised training.

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 words. 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 do not 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.

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 sequence-to-sequence 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 propose 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 fine-tuned for their classification task on the mentioned training datasets. With the pre-trained and fine-tuned 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 3.4.

For our thesis, we used the same auto-encoder and composite model for pre-training as explained in sections 5.1.5 and 5.1.6.

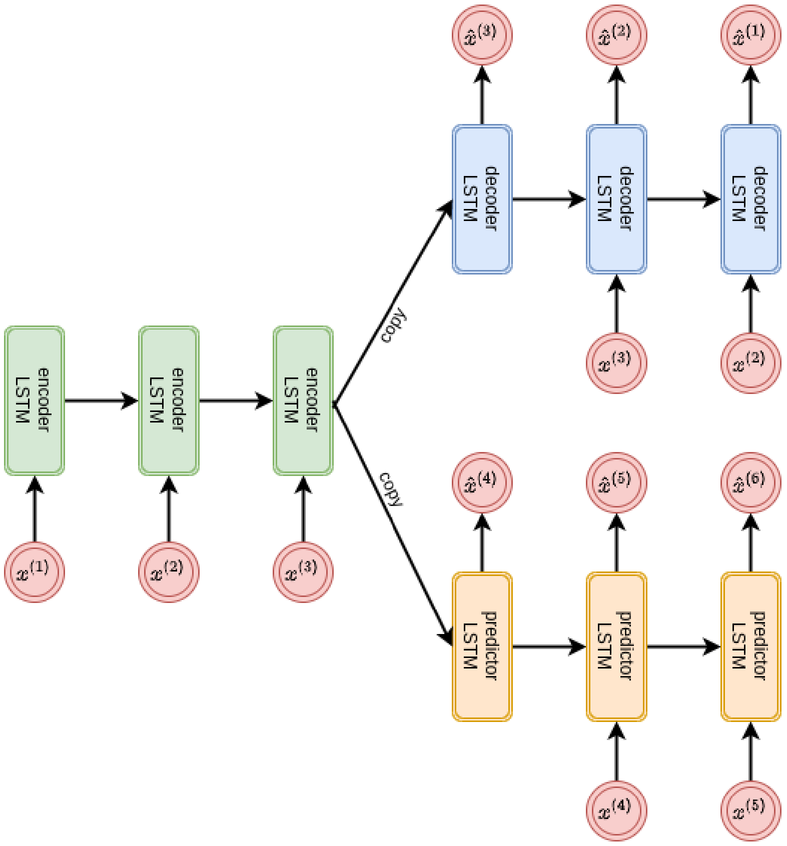

Figure 3.3: Composite model for input reconstruction and future prediction [SMS15]

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 + fine-tuning	75.8	74.9	44.1

Table 3.4: Summary of results on Action Recognition [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 fine-tuning. 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

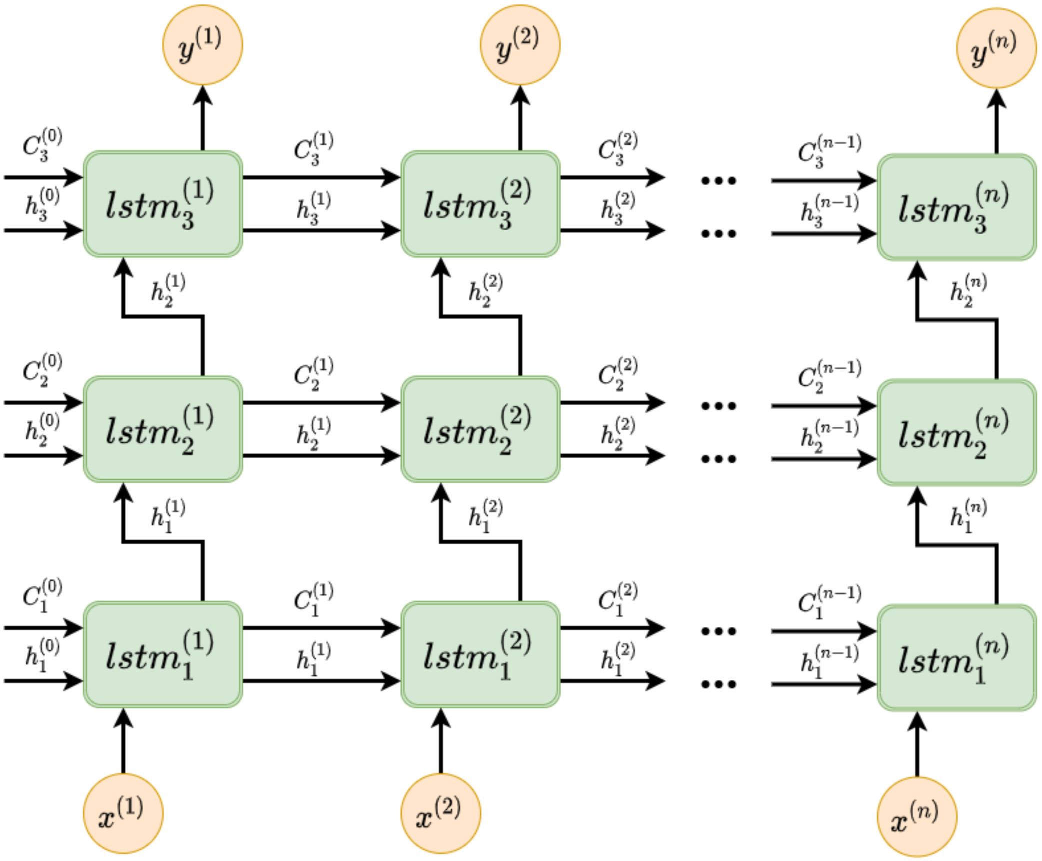

Figure 3.4: Data flow in a three layered LSTM network.

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 2.10, a stacked auto encoder model uses multiple encoder and decoder layers as can be seen 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

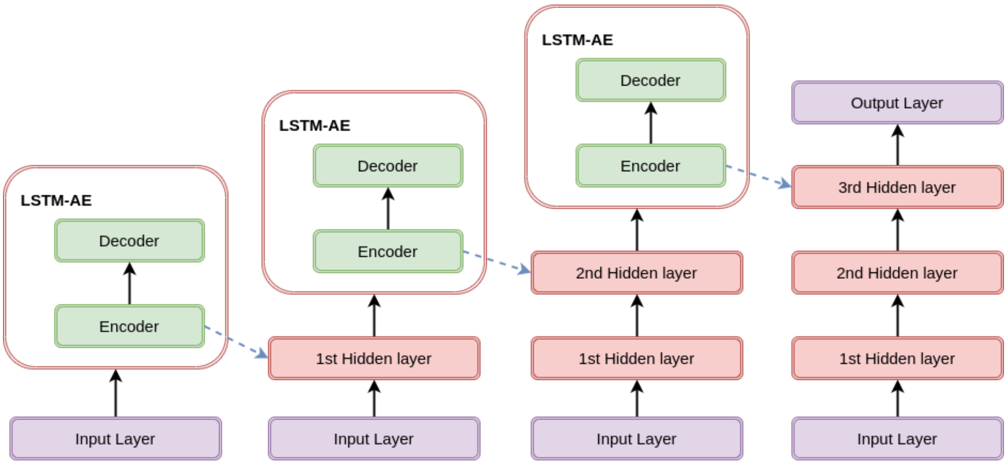

Figure 3.5: Layer-wise pre-training of LSTM-SAE model. [SK19b]

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 fine-tuned 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 fine-tuning and validation, they used two datasets:

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 bikes 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 3.5 and 3.6.

The results in tables 3.5 and 3.6 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	0.4	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 3.5: The results of DLSTM and LSTM-SAE using data set 1 [SK19b]

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

CHAPTER 4

Methodology

As summarized by the survey paper [LL19] of Hongyu Liu and Bo Lang in section 3.1, 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 this thesis is 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. Derived from state-of-the-art research, this seems to be a uni-directional multi-layer DLSTM in the context of NID. Most of the state-of-the-art research on DL, however, and especially on self-supervised/unsupervised and transfer learning, is done in the context of NLP [DCLT18], [PNI$^+$18], [VSP$^+$17a], [BMR$^+$20]. 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 i.e a word or packet conveys some semantic meaning when in the context of a sentence or flow. Other researchers have made similar observations [RATS18]. Based on the results achieved in NLP with attention based models, we deemed the transformer encoder to be a potentially powerful new tool for network traffic classification.

4.1 Datasets

We used the two NIDS datasets *UNSW-NB15* [MS15] and *CIC-IDS2017* [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 footprint attacks at time of publication. The records are bidrectional 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.* After preprocessing, where some incomplete records are removed, the dataset contains a total of 2.06 million records, of which 1.99 million are normal transactions labeled as benign and 0.072 million attack records meaning 96.64% of data is classified as benign and 3.36% as attack. Attack records can be further divided into nine attack categories as listed in table 4.1.

#	**Class**	**No. Records**	**% w.r.t. benign class**	%	**% w.r.t. majority atk. class**	**% w.r.t. total instances**
1	Analysis	534	0.03%		1.82%	0.03%
2	Backdoors	397	0.02%		1.36%	0.02%
3	DoS	3972	0.20%		13.57%	0.19%
4	Exploits	29281	1.47%		100.00%	1.42%
5	Fuzzers	20848	1.05%		71.20%	1.01%
6	Generic	4301	0.22%		14.69%	0.21%
7	Benign	1991349	100.00%		6800.82%	96.46%
8	Reconnaissance	11971	0.60%		40.88%	0.58%
9	Shellcode	1546	0.08%		5.28%	0.07%
10	Worms	185	0.01%		0.63%	0.01%

Table 4.1: UNSW-NB15 dataset record distribution by attack category [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-IDS2017** 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 worm attacks among others. It contains a total of 2.31 million records of which 1.73 million are labeled as benign and 0.58 million are attack records. In other words, 74.72% of the dataset are benign records and 25.28% attack records. Attack records are further classified as shown in table 4.2.

#	Class	No. Records	% w.r.t. benign class^S	% w.r.t. majority atk. class	% w.r.t. total instances
1	Botnet ARES	755	0.04%	0.32%	0.03%
2	FTP-Patator	254	0.01%	0.11%	0.01%
3	SSH-Patator	2,556	0.15%	1.09%	0.11%
4	DDoS LOIT	94,327	5.46%	40.39%	4.08%
5	DoS GoldenEye	7,451	0.43%	3.19%	0.32%
6	DoS Hulk	233,521	13.53%	100.00%	10.11%
7	DoS Slowhttptest	4,209	0.24%	1.80%	0.18%
8	DoS slowloris	3,886	0.23%	1.66%	0.17%
9	Heartbleed	2	0.00%	0.00%	0.00%
10	Infiltration	76,237	4.42%	32.65%	3.30%
11	Benign	1,726,226	100.00%	739.22%	74.72%
12	PortScan - Firewall off	159,648	9.25%	68.37%	6.91%
13	PortScan - Firewall on	381	0.02%	0.16%	0.02%
14	SQL Injection	13	0.00%	0.01%	0.00%
15	XSS WebAttack	678	0.04%	0.29%	0.03%

Table 4.2: CIC-IDS2017 dataset record distribution by attack category. [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, and a separated attack network containing a set of PCs with public IPs and running Windows 10 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 pre-training, 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 fine-tuning as described in section 5.

4.1.1 Subsets

Pre-training and training in our experiments is conducted on subsets of the original dataset, i.e. 80% for pre-training, 10%, 1% and even smaller splits for fine-tuning. The split of the datasets is performed by stratified sampling so the ratio of each attack class stays roughly the same in the different splits as in the original dataset.

In hope of making any possible positive results more salient, we devised an extreme scenario where we limit the labeled data available for fine-tuning to a bare minimum of

10 records at most (for some categories, the dataset contains less than 10 samples) per attack category present in the original dataset. We then include an amount of benign records to keep the ratio of attack/benign flows of the original dataset. This results in the two subsets elaborated in tables 4.3 and 4.4 labeled **CIC17_10** and **UNSW15_10** for the corresponding datasets.

#	**Class**	**No. Records**
1	Botnet ARES	10
2	FTP-Patator	10
3	SSH-Patator	10
4	DDoS LOIT	10
5	DoS GoldenEye	10
6	DoS Hulk	10
7	DoS Slowhttptest	10
8	DoS slowloris	10
9	Heartbleed	1
10	Infiltration	10
11	Benign	391
12	PortScan - Firewall off	10
13	PortScan - Firewall on	10
14	SQL Injection	1
15	XSS WebAttack	10

Table 4.3: Subset **CIC17_10** devised for CIC-IDS2017 to include a minimal amount of records amounting to approximately 0.023% of the total dataset.

#	**Class**	**No. Records**
1	Analysis	10
2	Backdoors	10
3	DoS	10
4	Exploits	10
5	Fuzzers	10
6	Generic	10
7	Benign	2543
8	Reconnaissance	10
9	Shellcode	10
10	Worms	10

Table 4.4: Subset **UNSW15_10** devised for UNSW-NB15 to include a minimal amount of records amounting to approximately 0.11% of the total dataset.

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]. We decided not to include any packet payload features except the packet length in bytes, as in a real world scenario most of the traffic would be encrypted anyway. This leads to poor classification performance for U2R and R2L [TBLG09] like SQL injection, XSS and other payload based attacks which 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. However, we use a packet sequence representation instead of an aggregate flow tensor due to its similarity with natural language as mentioned before, which enables us to apply state-of-the-art methods and sequence2sequence models from DL based NLP.

The sending direction *pktDirection* of the packet is a binary feature where 0 is the direction of the first packet in the flow and 1 is the complementary direction.

We used the data pre-processing from the paper [HBFZ19] by Tanja Zseby et al. as it fit the requirements for our experiments and was easily modifiable. Starting from captured network traffic in the form of PCAP files, we used the tool go-flows [oT19] to extract the specified features and used the same python script as was used for the experiments in the paper [HBFZ19] to assign labels to the extracted flows.

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 min-max 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 pre-training for ML-based intrusion detection we chose to take a look at LSTM and 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 [DCLT18], [TICE19]. Both types of networks are generally susceptible to improvements through self-supervised pre-training as prior research has shown [DCLT18]

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

Table 4.5: Packet features [PB18].

[SMS15] [SK19b]. Whether pre-training 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]. A depiction of the data flow and layers of a three layered LSTM can be seen in figure 3.4.

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

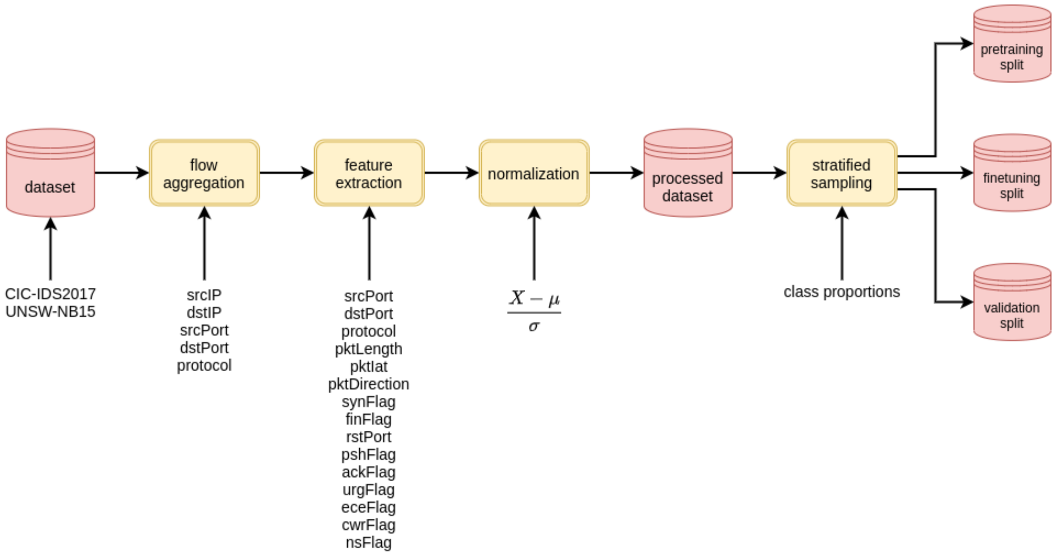

Figure 4.1: All steps performed in dataset preprocessing to yield pre-training, training and validation splits.

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 weights of an LSTM network, the most common ones are random initialization, orthogonal initialization or some form of transfer learning which in our case is self-supervised pre-training.

For pre-training, the output layer is a linear layer with 15 nodes, equal to the number of features, and no activation function. This is necessary as for pre-training the output of the model is a sequence of feature vectors representing network packets e.g. when calculating the identity function the model is tasked with producing the input packet sequence, or parts of it, as the model output. For binary classification the output layer is replaced with a linear layer with a single node because only one probability value is needed to distinguish between *attack* and *no-attack*. The node has no activation function on its own because we use the PyTorch implementation *torch.nn.BCEWithLogitsLoss* of Binary Cross Entropy (BCE) which applies a sigmoid activation function as first step of the calculation. This results in a sequence2sequence model which generates output sequences equal to the length of the input sequence. For supervised fine-tuning the target sequence is one of length n equal to the input sequence length where every element is the target label of the binary classification e.g. if the sequence is classified as an attack the target label would be 1 and the target sequence for supervised training would be $y^{(t)} = (1, ..., 1)$ of length n. For validation however it would only require a sequence2scalar model so 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.

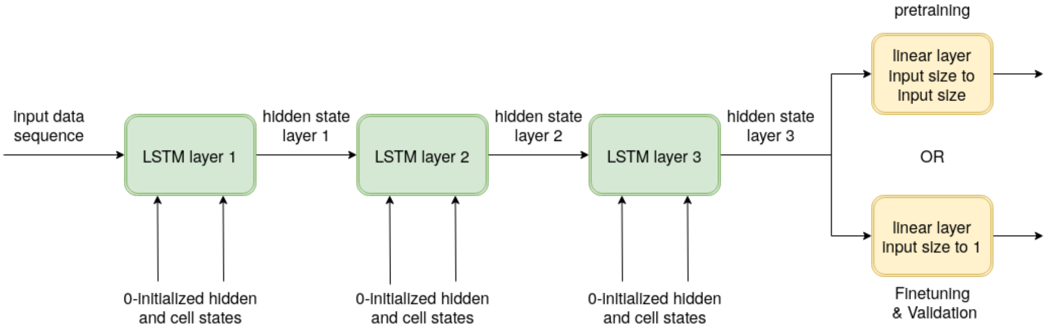
pretraining
linear layer input size to input size
Input data sequence
LSTM layer 1
hidden state layer 1
LSTM layer 2
hidden state layer 2
LSTM layer 3
hidden state layer 3
OR
linear layer input size to 1
0-initialized hidden and cell states
0-initialized hidden and cell states
0-initialized hidden and cell states
Finetuning & Validation

Figure 4.2: Depiction of the LSTM model.

The model as proposed by Vaswani et. al [VSP+17a] and its derivative score high on the leader boards of NLP benchmarks like GLUE [WSM+18] or SQuAD [RJL18] and are still considered state-of-the-art in many regards. Following the example of BERT we only used the encoder part of the *network* since the decoder does not provide any benefits for classification problems. We tuned the model parameters to be 10 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. A depiction of the model can seen in figure 4.3, where N denotes the number of layers. For pre-training, the output layer is a linear layer with 15 nodes, equal to the number of features, and no activation function is used. 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 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 [CLS] 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 unweighted 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.

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.Encoder*. 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

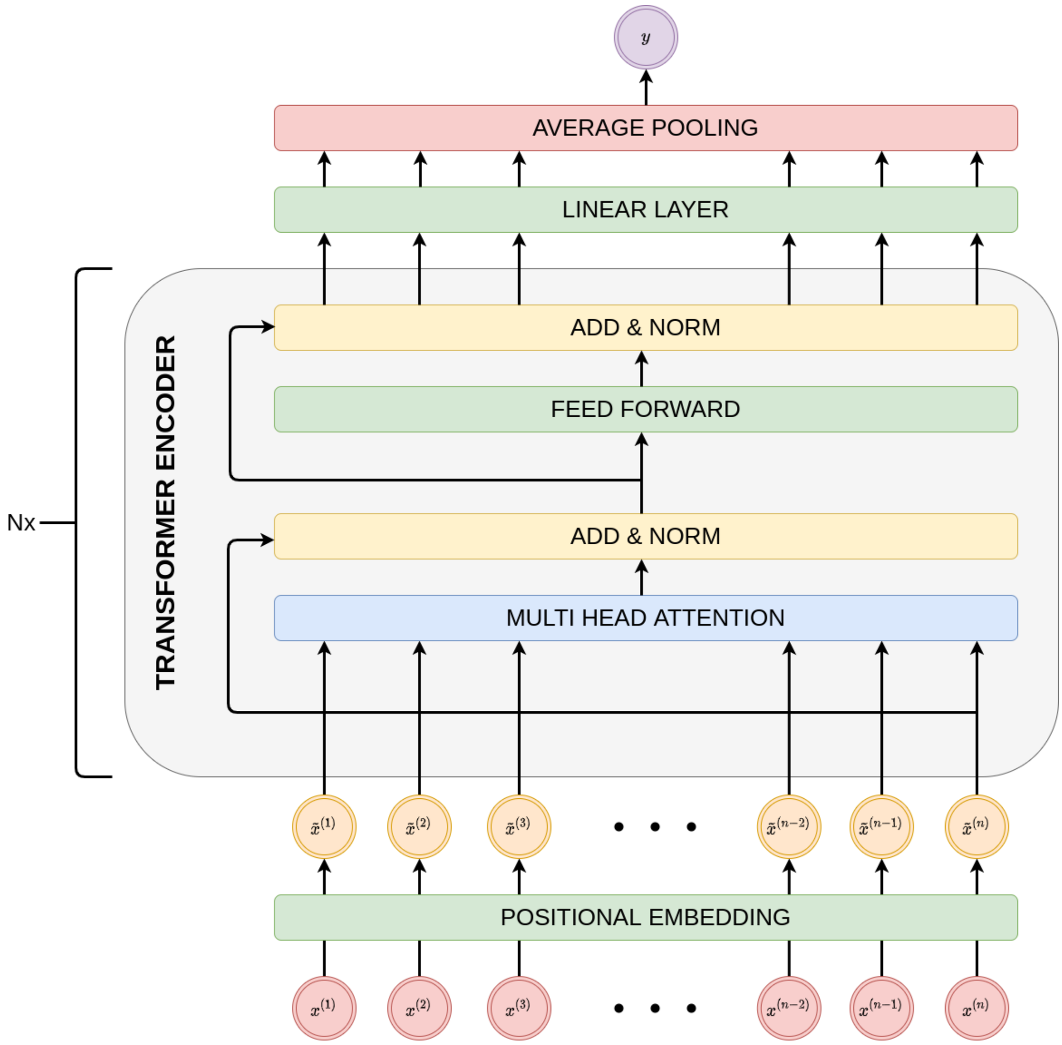
y
AVERAGE POOLING
LINEAR LAYER
TRANSFORMER ENCODER
Nx
ADD & NORM
FEED FORWARD
ADD & NORM
MULTI HEAD ATTENTION
$\tilde{x}^{(1)}$ $\tilde{x}^{(2)}$ $\tilde{x}^{(3)}$ $\tilde{x}^{(n-2)}$ $\tilde{x}^{(n-1)}$ $\tilde{x}^{(n)}$
POSITIONAL EMBEDDING
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(n-2)}$ $x^{(n-1)}$ $x^{(n)}$

Figure 4.3: Depiction of the transformer encoder model used for classification.

use of a learning rate scheduler which decreases the learning rate by a factor of 0.1 if mean batch loss is plateauing during training. As optimization criterion for pre-training 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 also tried to apply [**?**] which did not seem to have an effect on learning. 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. For some instances, i.e. for pre-training with the proxy tasks MASK on the transformer model and for pre-training with the proxy task IDENTITY and PREDICT we use 16bit float (FP16) values during training together with the *torch.cuda.amp.GradScaler*. This significantly decreases training time and GPU load but introduced numerical instability in some cases and for most cases 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 transformer encoder model. Further increasing batch size did not improve final accuracy nor did it decrease the time in which training converges.

4.5 Metrics and Validation

All tables, graphs and plots are automatically generated by a results script which trains the models with parameters set in a configuration file and generates the necessary data for all subsequent analysis methods i.e. performance metrics, per attack class statistics, Partial Dependence Plot (PDP), neuron activation plots and DTC. It subsequently generates latex tables and graphical elements which are included in this thesis. During supervised training the model is validated every 6, 2 and 1 epoch(s) for trainings with a total of 600, 200 and 100 training epochs respectively. The metrics described in section 2.12 are generated and stored for every validated epoch. The results in the following sections and all plots are generated with the model from the epoch with the highest recorded accuracy score. Hence, the tables in the results chapter 6 also include the number of the training epoch after which the numbers were generated and the training time needed on the setup described in the previous section 4.4 to reach that epoch. All results were generated and validated with the same random seed for the Python, PyTorch and Numpy Random Number Generator (RNG). Due to this, pre-training and validation splits contain exactly the same records for all experiments which maximizes comparability.

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 50, 50 and 200 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. Training with the specialized subsets CIC17_10 and UNSW15_10 4.1.1 was performed for 600 epochs. A full overview of all experiments to establish a comparison baseline can be seen in table 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 reconstruct 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 for 10 epochs in all instances. Supervised fine-tuning is performed with 10%, 1% and with even smaller subsets as described in the previous section 4.1.1.

The assumption is: The less the model can rely on information acquired through supervised training the more it has to rely on information acquired during self-supervised pre-training. Overviews of all conducted experiments can be found in tables 5.1, 5.3 and 5.4. Experiments are numbered so they can be referenced in later sections. Experiments in table 5.1 where the model is not pre-trained are referenced again in table 5.3.

As pre-training 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-IDS2017	128	-	90	50
2.1.1	LSTM	CIC-IDS2017	128	-	10	50
2.2.1	LSTM	CIC-IDS2017	128	-	1	200
2.3.1	LSTM	CIC-IDS2017	128	CIC17_10	-	600
1.2.1	LSTM	UNSW-NB15	128	-	90	50
2.4.1	LSTM	UNSW-NB15	128	-	10	50
2.5.1	LSTM	UNSW-NB15	128	-	1	200
2.6.1	LSTM	UNSW-NB15	128	CIC17_10	-	600
1.3.1	Transformer	CIC-IDS2017	1024	-	90	50
3.1.1	Transformer	CIC-IDS2017	1024	-	10	50
3.2.1	Transformer	CIC-IDS2017	1024	-	1	200
3.3.1	Transformer	CIC-IDS2017	1024	UNSW15_10	-	600
1.4.1	Transformer	UNSW-NB15	1024	-	90	50
3.4.1	Transformer	UNSW-NB15	1024	-	10	50
3.5.1	Transformer	UNSW-NB15	1024	-	1	200
1.6.1	Transformer	UNSW-NB15	1024	UNSW15_10	-	600

Table 5.1: List of baseline training runs used for comparison with results from pre-trained models.

which features are more important than others and correct weights accordingly. A list of all proxy tasks can be seen in table 5.2. Each of them will be explained in detail in the sections below.

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 weights of each LSTM layer are initialized with a uniform random distribution $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$. Because we assume that in some cases, i.e. for some sequences, no reasonable prediction can be made we don't want the model to react too strongly to those outliers. Hence, the MAE is used to determine the divergence between prediction and target data instead of Mean Squared Error (MSE). Translating to PyTorch this means we used *L1Loss* with *mean* reduction as the loss function for pre-training. After some initial trials we set the training

Section(s)	Label	Name	Description
5.1.1	IDENTITY	Identity Function	Reconstruct exact input feature vector at each stage
5.1.2	PREDICT	Predict Packet	Predict the next packet at each stage of the LSTM
5.1.3, 5.2.3	MASK	Mask Packets	Reconstruct masked packets in the sequence
5.1.4, 5.2.1	OBSCURE	Obscure Features	Reconstruct obscured 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 pre-training of DL models.

hyper-parameters 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.

5.1.1 Identity Function (IDENTITY)

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 i.e. $x^{(t)} = y^{(t)}$ with the same 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 in some instances when compared to the case where the model was not pre-trained, i.e. where weights are initialized randomly with a uniform distribution.

5.1.2 Predict Packet (PREDICT)

For this proxy task, the model has to predict the next packet in the flow. This is also called *autogregression* and it assumes *autocorrelation* between the current state of the model, i.e. the encoded input sequence up to this point, and future inputs. We can see that the *synFlag*, *ackFlag* the *pktDirection* and also to some extend *pktIat* and *pktLength* should be predictable in many instances since they either follow a certain

#	Dataset	Subset	Training %	Training Eps.	Proxy Task	Pretr. %	Pretr. Eps.
2.1.1	CIC-IDS2017	-	10	50	NONE	0	0
2.1.2	CIC-IDS2017	-	10	50	PREDICT	80	10
2.1.3	CIC-IDS2017	-	10	50	OBSCURE	80	10
2.1.4	CIC-IDS2017	-	10	50	AUTO	80	10
2.1.5	CIC-IDS2017	-	10	50	IDENTITY	80	10
2.1.6	CIC-IDS2017	-	10	50	COMPOSITE	80	10
2.2.1	CIC-IDS2017	-	1	200	NONE	0	0
2.2.2	CIC-IDS2017	-	1	200	PREDICT	80	10
2.2.3	CIC-IDS2017	-	1	200	OBSCURE	80	10
2.2.4	CIC-IDS2017	-	1	200	AUTO	80	10
2.2.5	CIC-IDS2017	-	1	200	IDENTITY	80	10
2.2.6	CIC-IDS2017	-	1	200	COMPOSITE	80	10
2.3.1	CIC-IDS2017	CIC17_10	-	600	NONE	0	0
2.3.2	CIC-IDS2017	CIC17_10	-	600	PREDICT	80	10
2.3.3	CIC-IDS2017	CIC17_10	-	600	OBSCURE	80	10
2.3.4	CIC-IDS2017	CIC17_10	-	600	AUTO	80	10
2.3.5	CIC-IDS2017	CIC17_10	-	600	IDENTITY	80	10
2.3.6	CIC-IDS2017	CIC17_10	-	600	COMPOSITE	80	10
2.4.1	UNSW-NB15	-	10	50	NONE	0	0
2.4.2	UNSW-NB15	-	10	50	PREDICT	80	10
2.4.3	UNSW-NB15	-	10	50	OBSCURE	80	10
2.4.5	UNSW-NB15	-	10	50	AUTO	80	10
2.4.6	UNSW-NB15	-	10	50	IDENTITY	80	10
2.4.7	UNSW-NB15	-	10	50	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	OBSCURE	80	10
2.5.4	UNSW-NB15	-	1	200	AUTO	80	10
2.5.5	UNSW-NB15	-	1	200	IDENTITY	80	10
2.5.6	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	OBSCURE	80	10
2.6.4	UNSW-NB15	UNSW15_10	-	600	AUTO	80	10
2.6.5	UNSW-NB15	UNSW15_10	-	600	IDENTITY	80	10
2.6.6	UNSW-NB15	UNSW15_10	-	600	COMPOSITE	80	10

Table 5.3: Training and pre-training configurations for LSTM model with different proxy tasks.

pattern over the course of the flow or are expected to remain within a small deviation range. 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: $y^{(t)} = x^{(t+1)} : t = 1, .., n-1$. 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 $\hat{x}^{(t)} = \hat{x}^{(2)}, \hat{x}^{(3)}, ..., \hat{x}^{(n)}$ of equal length $n-1$ between which a differentiable loss can be calculated. This way, a lot of information is conveyed to the network when compared to only predicting the

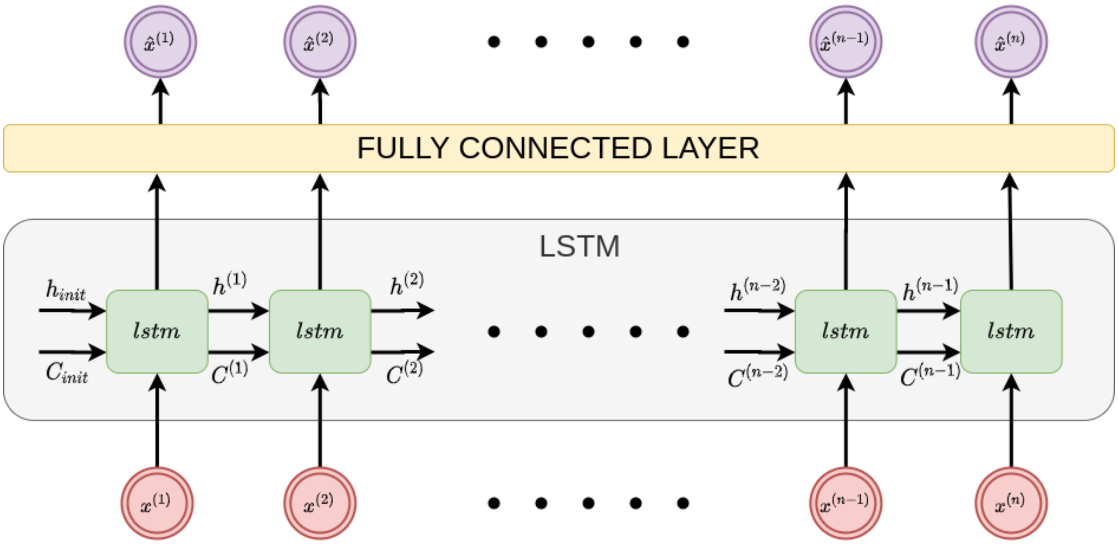

Figure 5.1: Depiction of data flow, input and output of the model during pre-training on the identity function proxy task (ID).

last packet in a flow. At first glance, this looks similar to the identity function above. 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. A depiction of the proxy task can be seen in figure 5.2.

5.1.3 Mask Packets (MASK)

Similar to the pre-training in BERT, a random token in the input sequence is masked with a value of -1, i.e. the packet token is replaced with a feature vector $x_{mask} = [-1, ..., -1]^T$ containing -1 for each element and the model is to reconstruct the masked packet. 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. This is possibly due to the model having to retain information to solve two different tasks, an approach also used to pre-train Google's BERT, which prevents the model from attuning itself to one specific task in a form of over-fitting. A depiction of the proxy task can be seen in figure 5.3.

5.1.4 Obscure Features (OBSCURE)

For this pre-training task, the model is to predict masked features of some packets in the sequence as depicted in figure 5.4. We have tried multiple masking values but -1

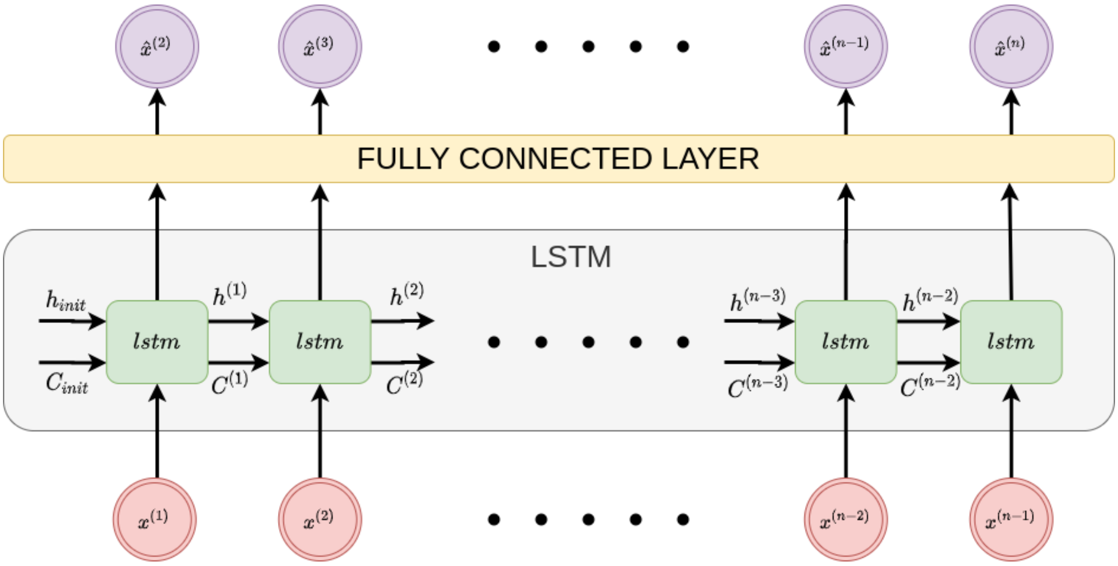

Figure 5.2: Depiction of data flow, input and output of the model during pre-training on the prediction proxy task (PREDICT).

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, but we tried some of the most intuitive approaches. The task proved to be quite difficult for both models, so we decided to predict only the TCP SYN flag. 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 obscured features of predicted *logits* and the target data sequences. A depiction of the proxy task can be seen in figure 5.4.

5.1.5 Auto Encoder (AUTO)

As explained in section 2.10, 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 depicted in figure 5.5. 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

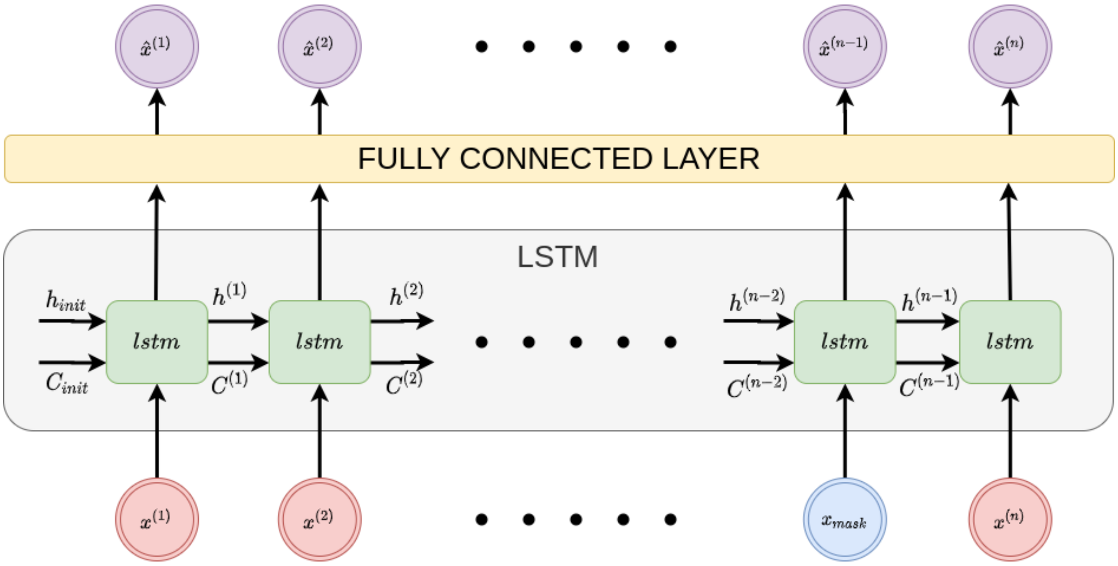

Figure 5.3: Depiction of data flow, input and output of the LSTM model during pre-training on the *Mask Packet* proxy task (MASK).

the hidden state $h_e^{(n)}$ and cell state $C_e^{(n)}$ of the last stage 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 $h_d^{(1)} = h_e^{(n)}, C_d^{(1)} = C_e^{(n)}$ and is tasked with reconstructing the input sequence in reverse order. The reverse order target sequence is used to shorten the paths of the input tokens to their respecting target tokens. After every stage of the decoder, either the output $\hat{x}^{(t)}$ or the target token of the current stage $x^{(t)}$, the ground truth, is then fed into the model as input token $x_d^{(t)} = \hat{x}^{(t-1)}$ for the next stage. Providing the ground truth again at the decode stage is also called *teacher forcing*. Note that for the decoder indices are processed in reverse order. The first input token for the decoder is a zero vector which functions as a start-of-sequence token $x_{zero} = [0, ..., 0]^T$. 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. After trying both approaches, we decided to use teacher forcing as it produced slightly better results.

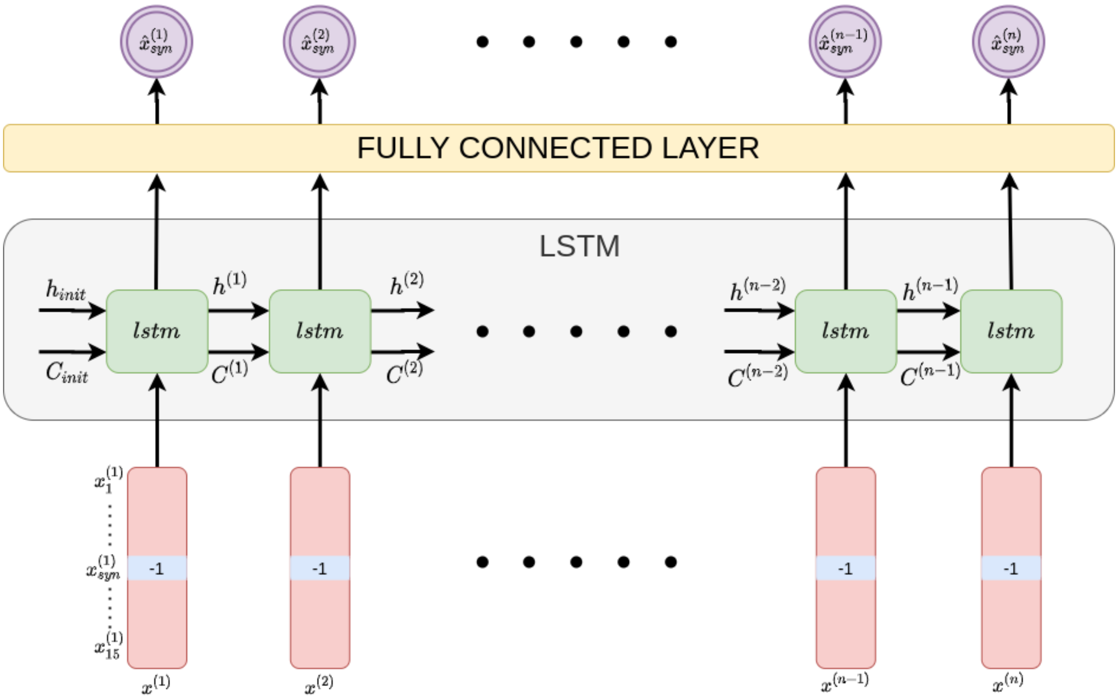

Figure 5.4: Depiction of data flow, input and output of the LSTM model during pre-training on the *Obscure Feature* proxy task (OBSCURE).

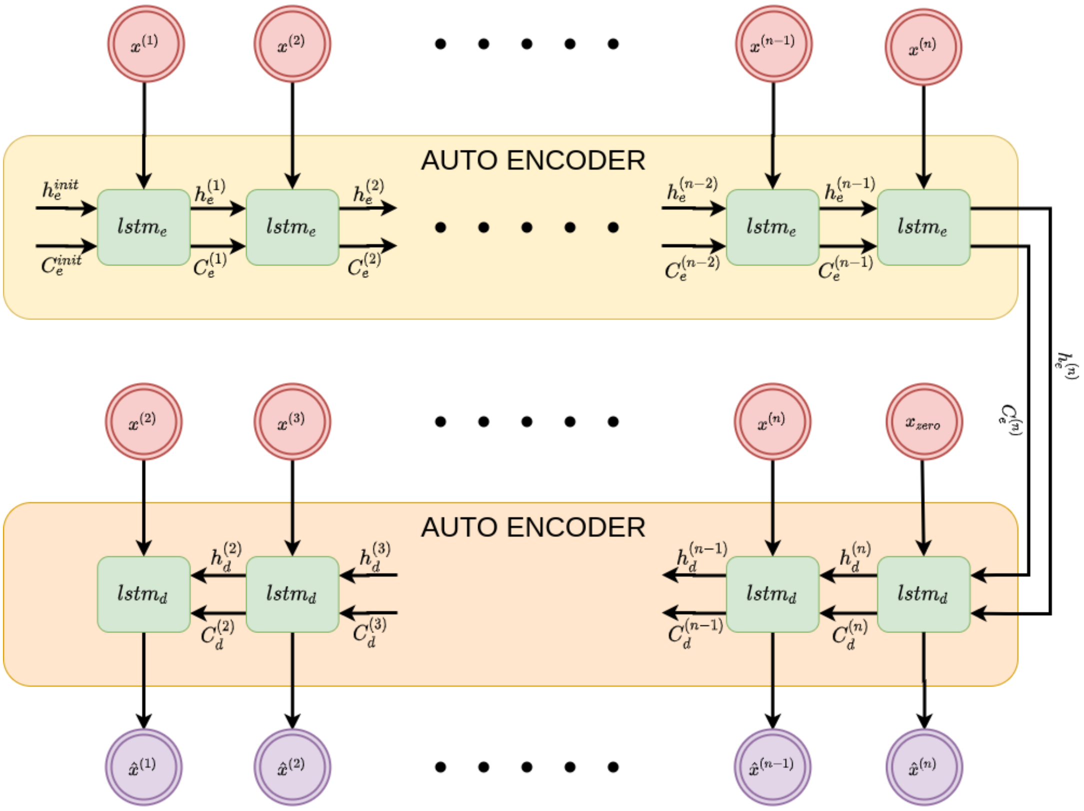

Figure 5.5: Depiction of data flow, input and output of the LSTM model during pre-training on *Auto Encoder* proxy task (AUTO). The fully connected layer is omitted from the graphic.

5.1.6 Composite Model (COMPOSITE)

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 the first part of 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 $\hat{x}^{(t)}$ 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 5.6. For an input sequence of length n the encoder processes the first m tokens of the sequence, constructing an abstract representation in its hidden state. The hidden state of the last stage of the encoder LSTM is then copied to both the decoder and predictor LSTMs as initial hidden state. Like the auto encoder from the previous section 5.1.5 the decoder LSTM tries to recreate the partial input sequence $x^{(t)} : t = 1, ..., m$. Initialized with the final hidden state of the encoder, the predictor LSTM tries to foretell future packets of the flow, i.e. target sequence $x^{(t)} : t = m, ..., n$. At every stage of the predictor (or decoder) LSTM (except the first), either the output $\hat{x_p}^{(t-1)}$ (or $\hat{x_d}^{(t+1)}$) of the previous stage or the target token $x^{(t-1)}$ (or $x^{(t+1)}$) 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. We tried both approaches and found that the model performs better with conditioning so for our experiments both the predictor and decoder model uses the output of its previous stage as input for the next. In our case, m was chosen to be half the sequence length rounded up so $m = \lceil \frac{n}{2} \rceil$. The data flow in the model is depicted in figure 5.6.

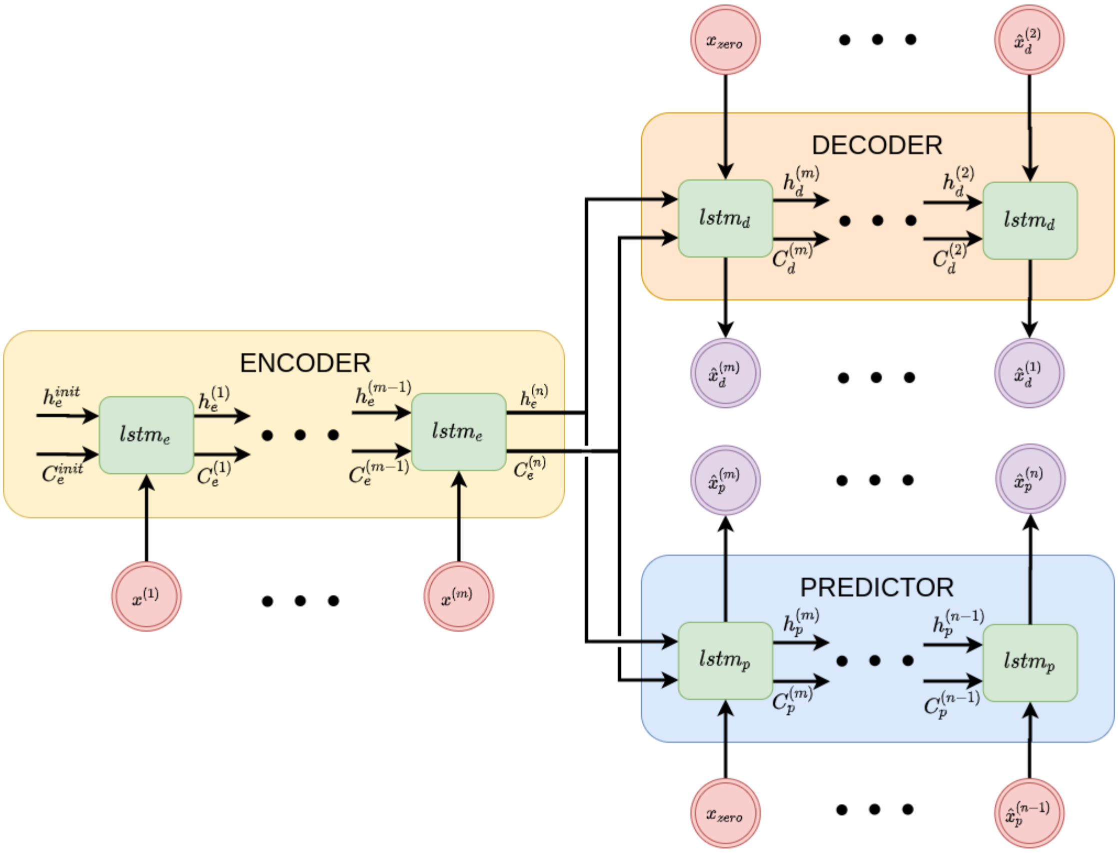

Figure 5.6: Depiction of data flow, input and output of the LSTM model during pre-training with the *Composite* proxy task (COMPOSITE). The fully connected layer is omitted from the graphic.

5.2 Self-supervised Pre-training for Transformer Encoder 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 transformer models than it is in LSTMs, the pre-training task *Predict Packet* 5.1.2 we used for the LSTM is no longer feasible. While the LSTM at each stage only has 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% of the dataset on the transformer takes longer to converge than on the LSTM in terms of real time. 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-IDS2017	-	10	50	NONE	0	0
3.1.2	CIC-IDS2017	-	10	50	AUTO	80	10
3.1.3	CIC-IDS2017	-	10	50	OBSCURE	80	10
3.1.4	CIC-IDS2017	-	10	50	MASK	80	10
3.2.1	CIC-IDS2017	-	1	200	NONE	0	0
3.2.2	CIC-IDS2017	-	1	200	AUTO	80	10
3.2.3	CIC-IDS2017	-	1	200	OBSCURE	80	10
3.2.4	CIC-IDS2017	-	1	200	MASK	80	10
3.3.1	CIC-IDS2017	CIC17_10	-	600	NONE	0	0
3.3.2	CIC-IDS2017	CIC17_10	-	600	AUTO	80	10
3.3.3	CIC-IDS2017	CIC17_10	-	600	OBSCURE	80	10
3.3.4	CIC-IDS2017	CIC17_10	-	600	MASK	80	10
3.4.1	UNSW-NB15	-	10	50	NONE	0	0
3.4.2	UNSW-NB15	-	10	50	AUTO	80	10
3.4.3	UNSW-NB15	-	10	50	OBSCURE	80	10
3.4.4	UNSW-NB15	-	10	50	MASK	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	OBSCURE	80	10
3.5.4	UNSW-NB15	-	1	200	MASK	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	OBSCURE	80	10
3.6.4	UNSW-NB15	UNSW15_10	-	600	MASK	80	10

Table 5.4: Training and pre-training configurations for transformer model with different proxy tasks.

To make the effect of pre-training on the transformer more salient we tried different approaches like using two consecutive transformer encoder models in serial configuration. Only the first transformer model is pre-trained and for supervised fine-tuning it feeds its output, which hopefully constitutes an abstraction of the data, into the second model which is then trained with labeled data. Another approach was to reset the deeper layers of the transformer model, as an abstract data representation might have formed in the earlier layers. The later layers might have been used to solve the proxy task based on this representation which are not needed and even disadvantageous for classification. We also tried tweaking the dropout rate, the number of attentions heads and the number of layers to improve performance. None of these methods achieved consistent improvements in our case.

5.2.1 Obscure Features (OBSCURE)

Analogous to the *Mask Features* proxy task for the LSTM, we used the same method for pre-training the transformer. A depiction for the data flow during pre-training can be seen in figure 5.7.

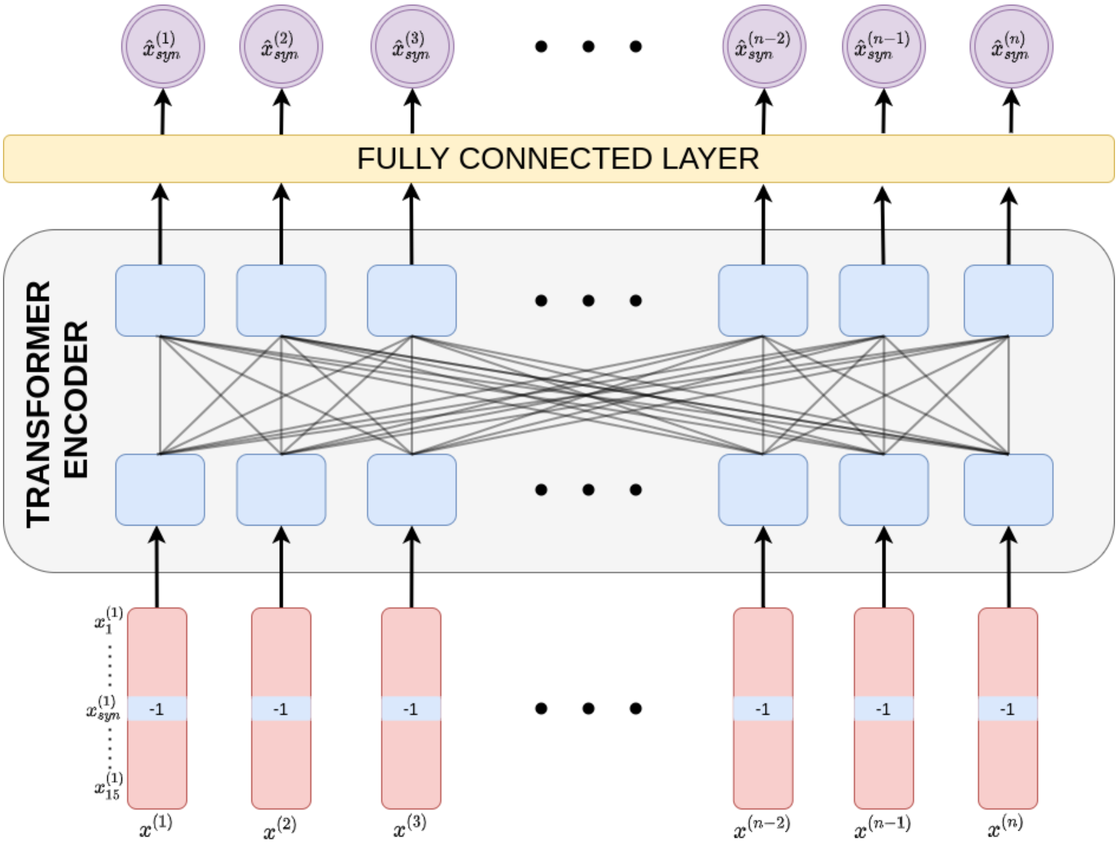

Figure 5.7: Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Obscure Feature* proxy task (OBSCURE).

5.2.2 Auto Encoder (AUTO)

Auto encoders are an established concept when it comes to self-supervised learning 2.10. 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 a dropout rate of 20% to introduce artificial noise into the input data.

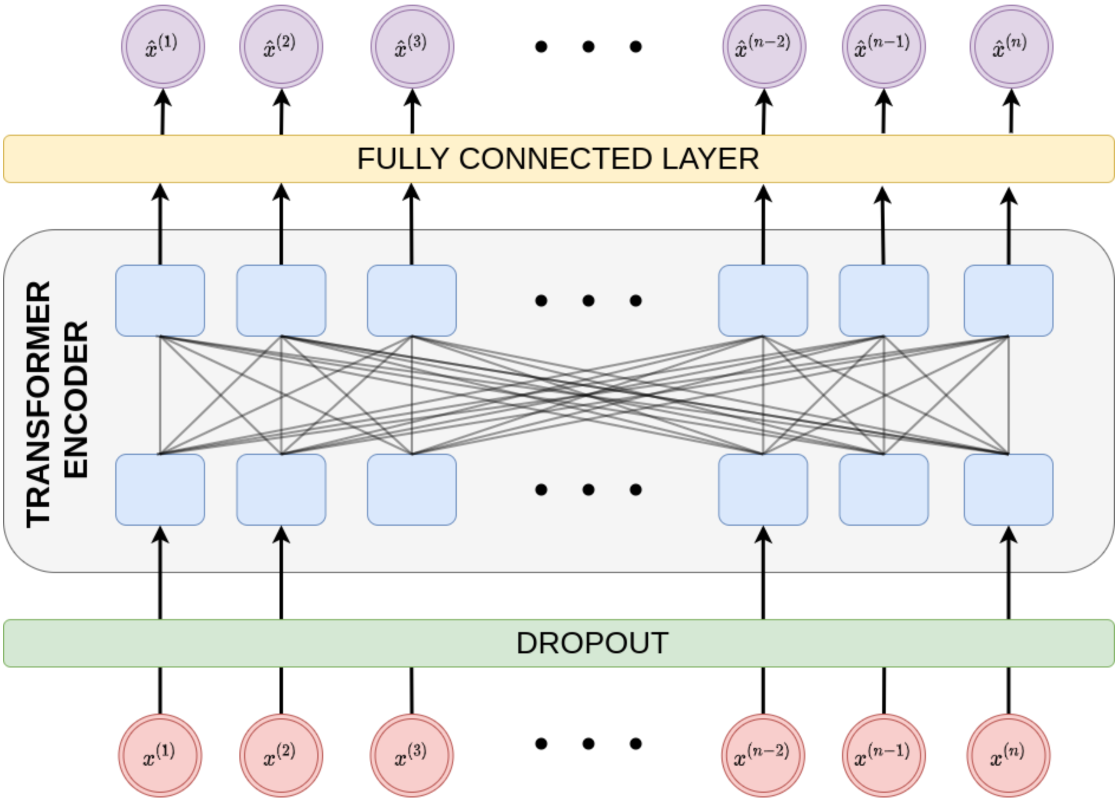

Figure 5.8: Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Auto Encoder* proxy task (AUTO).

5.2.3 Mask Packet (MASK)

Same as with for the LSTM, one packet token is replaced with a mask vector $x_{mask} = [-1, ..., -1]^T$ containing -1 for each feature. The model is then tasked with reconstructing the omitted input. While for our LSTM model we found that this proxy task performs better if we also include the non-masked tokens into the loss calculation, here we calculate the loss only based on the difference between the masked token and the predicted token. For the LSTM, this task is very similar to the PREDICT task in section 5.1.2 with the

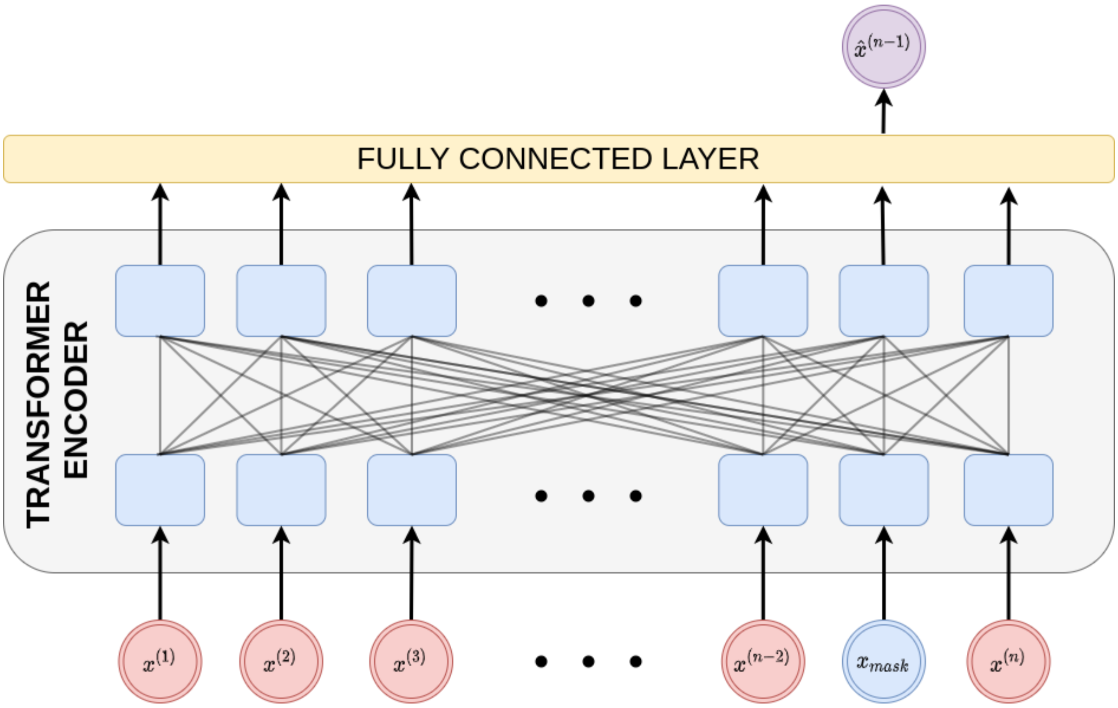

Figure 5.9: Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Mask Packet* proxy task (MASK).

difference that it is to predict only one token instead of every subsequent token. The transformer however is able to also take "future" packets, i.e. later packet tokens in the input sequence into account. Since a packet in a flow can be seen as a word in a sentence, and the feature representation of a packet is similar to an embedded word vector, this pre-training task is analogous to the method used in BERT [DCLT18].

CHAPTER 6

Results

In this chapter, we present the results from experiments proposed in the previous chapter. Like with the experiments section, we will be looking at the results from LSTM and the transformer model independently. An in-depth look into the datasets and the workings of the differently trained models based on Partial Dependence (PD) plots, neuron data plots and the outputs of a fitted DTC follow in the section 6.3. For a complete list of parameters used during training we refer to tables 5.1, 5.1, 5.3 and 5.4 and the experiments section in general. To analyze the results, we devised two types of tables: In one we are analyzing all result metrics which are listed in section 2.12 together with some training metrics like the time it took the model to produce the best sampled results and in the other we are looking at attack category specific accuracy results. *Epochs Supervised* states how many epochs the model was trained during fine-tuning i.e. on the amount of labeled data it was provided, denoted by *Training percentage*. The label *Best epoch* constitutes the training epoch in which the highest accuracy score was achieved by the model on the validation subset and *Time to best epoch* constitutes the time it took during fine-tuning to achieve this result. To track in which epoch the model performed best, validation accuracy was tested every 6 epochs for trainings with 600 epochs, every 2 epochs for trainings with 200 epochs and every epoch for trainings with <100 epochs. This was done to keep training times reasonable, as, especially in the cases where only the specialized subsets were used for fine-tuning, validation takes longer than an epoch of training. Higher accuracy scores might have occurred during epochs in which validation accuracy was not tested. In the tables we are looking for cases where pre-training affected the classification behavior of the model i.e. where scores differ from the NONE baseline column.

6.1 Long Short-Term Memory Model

As a baseline, we look at results where the model has been trained in a purely supervised fashion with different amounts of data of the two datasets. The results are comparable to previous experiments with deep neural networks on these datasets [SYZ20] and even slightly better in some instances. Looking at tables 6.1, 6.3 we can already see that very little supervised data is needed to achieve fairly high accuracy. For the LSTM model, going from 90% of training data (exp. 1.1.1 and 1.2.1) to 10% (exp. 2.1.1 and 2.4.1) only amounts to an absolute drop of 0.164% and 0.276% accuracy for CIC-IDS2017 and UNSW-NB15 datasets respectively. Most astounding are also the results when dropping from millions of records when training with 90% of the datasets to just the specialized subsets containing a couple of hundred entries in total and only 10 records of each attack class. Withholding most of labeled data in the datasets, this constraint only amounts to an absolute accuracy decrease of 4.114% and 1.176% for datasets CIC-IDS2017 and UNSW-NB15 respectively. While this is fairly pleasant in general, it means that results will be harder to improve as any benefit pre-training provides might be overshadowed by the effectiveness of supervised training, even with very little data.

	1.1.1	2.1.1	2.2.1	2.3.1
Proxy task	NONE	NONE	NONE	NONE
Epochs Supervised	50	50	200	600
Training percentage	90.00 %	10.00 %	1.00 %	
Specialized subset				CIC17_10
Training metrics				
Best epoch	45	49	191	593
Time to best epoch	10h 8m	1h 34m	1h 14m	0h 43m
Performance metrics				
Accuracy	99.796 %	99.632 %	99.385 %	95.682 %
Detection rate	99.306 %	98.828 %	98.408 %	90.088 %
Precision	99.885 %	99.711 %	99.153 %	92.624 %
Specificity	99.961 %	99.903 %	99.716 %	97.574 %
F1-Measure	99.595 %	99.268 %	98.779 %	91.338 %
False alarm rate	0.115 %	0.289 %	0.847 %	7.376 %
Missed alarm rate	0.694 %	1.172 %	1.592 %	9.912 %

Table 6.1: Experiments 1.1.1, 2.1.1, 2.2.1 and 2.3.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset CIC-IDS2017.

		1.1.1	2.1.1	2.2.1	2.3.1
Class	#	NONE	NONE	NONE	NONE
Botnet ARES	0	94.667%	94.667%	98.667%	94.737%
FTP-Patator	1	92.000%	72.000%	30.769%	96.000%
SSH-Patator	2	100.000%	98.024%	93.254%	100.000%
DDoS LOIT	3	100.000%	99.979%	99.979%	99.530%
DoS GoldenEye	4	100.000%	100.000%	99.460%	97.981%
DoS Hulk	5	100.000%	99.996%	100.000%	98.740%
DoS Slowhttptest	6	100.000%	99.763%	99.284%	99.286%
DoS slowloris	7	100.000%	100.000%	96.875%	92.727%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%
Infiltration	9	94.906%	92.913%	91.557%	63.608%
Benign	10	99.961%	99.903%	99.716%	97.574%
PortScan - Firewall off	11	99.981%	99.299%	98.839%	83.707%
PortScan - Firewall on	12	100.000%	83.784%	78.947%	81.579%
SQL Injection	13	100.000%	100.000%	0.000%	100.000%
XSS	14	88.060%	91.045%	50.000%	80.303%
Benign	10	99.961%	99.903%	99.716%	97.574%
Attack	!10	99.306%	98.828%	98.408%	90.088%
Overall	ALL	99.796%	99.632%	99.385%	95.682%

Table 6.2: Per category accuracy analysis of experiments 1.1.1, 2.1.1, 2.2.1 and 2.3.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset CIC-IDS2017.

	1.2.1	2.4.1	2.5.1	2.6.1
Proxy task	NONE	NONE	NONE	NONE
Epochs Supervised	50	50	200	600
Training percentage	90.00 %	10.00 %	1.00 %	
Specialized subset				UNSW15_10
Training metrics				
Best epoch	41	25	159	77
Time to best epoch	8h 38m	0h 44m	0h 52m	0h 6m
Performance metrics				
Accuracy	98.930 %	98.654 %	98.305 %	97.754 %
Detection rate	82.936 %	80.684 %	78.846 %	65.595 %
Precision	86.315 %	81.166 %	74.673 %	69.315 %
Specificity	99.517 %	99.313 %	99.019 %	98.934 %
F1-Measure	84.592 %	80.924 %	76.703 %	67.404 %
False alarm rate	13.685 %	18.834 %	25.327 %	30.685 %
Missed alarm rate	17.064 %	19.316 %	21.154 %	34.405 %

Table 6.3: Experiments 1.2.1, 2.4.1, 2.5.1 and 2.6.1 with LSTM model trained in a purely supervised fashion on different percentages of data for UNSW-NB15.

		1.2.1	2.4.1	2.5.1	2.6.1
Class	#	NONE	NONE	NONE	NONE
Analysis	0	26.415%	28.846%	43.396%	80.392%
Backdoors	1	97.436%	90.000%	80.000%	85.000%
DoS	2	93.862%	84.733%	78.261%	81.679%
Exploits	3	95.878%	93.625%	87.431%	85.498%
Fuzzers	4	50.459%	51.254%	59.632%	47.315%
Generic	5	98.131%	93.897%	80.189%	69.484%
Normal	6	99.517%	99.313%	99.019%	98.934%
Reconnaissance	7	98.825%	96.888%	92.017%	44.155%
Shellcode	8	96.104%	74.510%	82.237%	39.610%
Worms	9	94.737%	94.737%	78.947%	57.895%
Benign	6	99.517%	99.313%	99.019%	98.934%
Attack	!6	82.936%	80.684%	78.846%	65.595%
Overall	ALL	98.930%	98.654%	98.305%	97.754%

Table 6.4: Per category accuracy analysis of experiments 1.2.1, 2.4.1, 2.5.1 and 2.6.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset UNSW-NB15.

6.1.1 Pre-training of the LSTM model with CIC-IDS2017 Dataset

Next, we will be looking at results for pre-training with the different proxy tasks and different amounts of data used for supervised fine-tuning. Tables 6.5, 6.7 and 6.9 show results for experiments 2.1.1 - 2.1.6, 2.2.1 - 2.2.6 and 2.3.1 - 2.3.6 on dataset CIC-IDS2017 conducted with 10%, 1% and only a very small fraction of data as defined in subset CIC17_10. Looking at the performance metrics, we can see that there is some variance in the resulting data. The NumPy and PyTorch random seeds are the same for all experiments which means that pre-training, supervised fine-tuning and validation have been conducted with the exact same subsets of the original dataset which means that differences in results can only come from pre-training with different proxy tasks. This establishes the fact that pre-training in general, and also different methods of pre-training have an effect on final performance. Starting with table 6.5, we can see that pre-training with some proxy tasks improves performance while others have almost no effect or even a negative effect.

For accuracy, the highest positive delta 0.101% in experiments 2.1.1-6 in table 6.5 can be observed for pre-training with the COMPOSITE proxy task 5.1.6 closely followed by pre-training with the ID proxy task 5.1.1 with a delta of 0.095%. The highest negative delta in accuracy, -0.010%, can be observed for the OBSCURE feature proxy task 5.1.4. It should be noted, that for detection rate, the highest delta is 0.343% also occurring after COMPOSITE pre-training. This shows that the improvement in accuracy stems from improved attack detection capability achieved through pre-training.

	2.1.1	2.1.2	2.1.3	2.1.4	2.1.5	2.1.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	50	50	50	50	50	50
Training percentage	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %
Specialized subset						
Training metrics						
Best epoch	49	47	49	46	48	47
Time to best epoch	1h 34m	1h 35m	1h 40m	1h 59m	1h 43m	3h 13m
Performance metrics						
Accuracy	99.632 %	99.705 %	99.620 %	99.630 %	99.727 %	99.733 %
Detection rate	98.828 %	99.108 %	98.877 %	98.871 %	99.146 %	99.171 %
Precision	99.711 %	99.722 %	99.616 %	99.663 %	99.771 %	99.771 %
Specificity	99.903 %	99.907 %	99.871 %	99.887 %	99.923 %	99.923 %
F1-Measure	99.268 %	99.414 %	99.245 %	99.265 %	99.457 %	99.470 %
False alarm rate	0.289 %	0.278 %	0.384 %	0.337 %	0.229 %	0.229 %
Missed alarm rate	1.172 %	0.892 %	1.123 %	1.129 %	0.854 %	0.829 %

Table 6.5: Experiments 2.1.1-6 with LSTM model finetuned with 10% of dataset CIC-IDS2017.

		2.1.1	2.1.2	2.1.3	2.1.4	2.1.5	2.1.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Botnet ARES	0	94.667%	93.333%	94.737%	93.243%	94.737%	93.333%
FTP-Patator	1	72.000%	96.154%	53.846%	3.846%	88.462%	92.308%
SSH-Patator	2	98.024%	99.206%	99.206%	99.209%	99.608%	99.209%
DDoS LOIT	3	99.979%	99.989%	100.000%	100.000%	99.989%	99.989%
DoS GoldenEye	4	100.000%	100.000%	100.000%	100.000%	99.865%	100.000%
DoS Hulk	5	99.996%	99.996%	99.996%	99.996%	99.996%	99.996%
DoS Slowhttptest	6	99.763%	99.761%	99.282%	99.284%	99.282%	99.284%
DoS slowloris	7	100.000%	99.740%	99.740%	99.741%	99.740%	99.742%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
Infiltration	9	92.913%	93.865%	93.549%	93.209%	93.924%	94.181%
Benign	10	99.903%	99.907%	99.871%	99.887%	99.923%	99.923%
PortScan - Firewall off	11	99.299%	99.874%	99.179%	99.578%	99.943%	99.905%
PortScan - Firewall on	12	83.784%	70.270%	86.486%	73.684%	86.486%	86.486%
SQL Injection	13	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
XSS	14	91.045%	86.765%	94.118%	59.701%	91.045%	91.176%
Benign	10	99.903%	99.907%	99.871%	99.887%	99.923%	99.923%
Attack	!10	98.828%	99.108%	98.877%	98.871%	99.146%	99.171%
Overall	ALL	99.632%	99.705%	99.620%	99.630%	99.727%	99.733%

Table 6.6: Per category accuracy analysis of experiments 2.1.1-6 with LSTM model finetuned with 10% of dataset CIC-IDS2017.

Looking at table 6.7 we can inspect for which attack categories improvement is most salient. When comparing training with supervised methods only in experiment 2.1.1 with the on the COMPOSITE proxy task pre-trained model in experiment 2.1.6 we can see major improvements for detection of FTP-Patator brute force attacks. Accuracy jumped from 72% for supervised only training to 92.308% for the COMPOSITE trained model and even 96.154% for the PREDICT proxy task, constituting a positive delta of 24.154%. For experiment 2.1.4, pre-training with the auto encoder task, the accuracy for detection of the FTP-Patator attack category dropped to 3.846%. Such high variance in results shows again that the LSTM model is susceptible to different pre-training strategies. Other attack classes which have seen improvement in detection accuracy are port scans with firewall on and off (#11-12) with positive deltas of 2.702% and 0.606% respectively and infiltration (#9) and SSH-Patator (#2) with deltas of 1.269% and 1.185%.

Looking at results for experiments 2.2.1-6 (table 6.7) fine-tuned with 1% of the CICIDS-2017 dataset - *ceteris paribus* - the maximum delta in accuracy increased to 0.178%, which in this iteration is observed after pre-training with the PREDICT proxy task 5.1.2. The positive delta for the COMPOSITE proxy tasks increased to 0.136% and the delta for the ID proxy task remained almost the same at 0.094%. PREDICT, ID and COMPOSITE proxy tasks have shown improvements in accuracy and performance overall for experiments 2.1.1-6 and 2.2.1-6, but the pattern breaks down when looking at experiments 2.3.1-6 with fine-tuning performed with subset CIC17_10 where improvement is now only present for pre-training with the ID proxy task where the positive deltas in

	2.2.1	2.2.2	2.2.3	2.2.4	2.2.5	2.2.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	200	200	200	200	200	200
Training percentage	1.00 %	1.00 %	1.00 %	1.00 %	1.00 %	1.00 %
Specialized subset						
Training metrics						
Best epoch	191	185	199	173	177	171
Time to best epoch	1h 14m	1h 19m	1h 30m	1h 14m	1h 15m	2h 25m
Performance metrics						
Accuracy	99.385 %	99.563 %	99.310 %	99.378 %	99.479 %	99.521 %
Detection rate	98.408 %	98.781 %	98.386 %	98.192 %	98.617 %	98.686 %
Precision	99.153 %	99.486 %	98.876 %	99.339 %	99.319 %	99.416 %
Specificity	99.716 %	99.827 %	99.622 %	99.779 %	99.771 %	99.804 %
F1-Measure	98.779 %	99.132 %	98.631 %	98.762 %	98.967 %	99.050 %
False alarm rate	0.847 %	0.514 %	1.124 %	0.661 %	0.681 %	0.584 %
Missed alarm rate	1.592 %	1.219 %	1.614 %	1.808 %	1.383 %	1.314 %

Table 6.7: Experiments 2.2.1-6 with LSTM model finetuned with 1% of dataset CIC-IDS2017.

		2.2.1	2.2.2	2.2.3	2.2.4	2.2.5	2.2.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Botnet ARES	0	98.667%	97.333%	98.649%	98.684%	96.053%	94.737%
FTP-Patator	1	30.769%	44.000%	0.000%	45.833%	40.000%	44.000%
SSH-Patator	2	93.254%	98.431%	99.605%	91.732%	100.000%	98.425%
DDoS LOIT	3	99.979%	99.979%	99.989%	100.000%	100.000%	100.000%
DoS GoldenEye	4	99.460%	99.865%	96.486%	99.322%	98.787%	99.866%
DoS Hulk	5	100.000%	99.892%	99.991%	99.948%	99.935%	99.966%
DoS Slowhttptest	6	99.284%	99.760%	99.761%	99.758%	99.762%	99.762%
DoS slowloris	7	96.875%	100.000%	97.416%	97.172%	97.674%	99.742%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
Infiltration	9	91.557%	92.904%	90.980%	91.028%	91.943%	92.051%
Benign	10	99.716%	99.827%	99.622%	99.779%	99.771%	99.804%
PortScan - Firewall off	11	98.839%	99.489%	99.357%	98.594%	99.294%	99.413%
PortScan - Firewall on	12	78.947%	65.789%	57.895%	62.162%	75.676%	76.316%
SQL Injection	13	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%
XSS	14	50.000%	64.706%	1.493%	1.471%	75.000%	65.672%
Benign	10	99.716%	99.827%	99.622%	99.779%	99.771%	99.804%
Attack	!10	98.408%	98.781%	98.386%	98.192%	98.617%	98.686%
Overall	ALL	99.385%	99.563%	99.310%	99.378%	99.479%	99.521%

Table 6.8: Per category accuracy analysis of experiments 2.2.1-6 with LSTM model finetuned with 1% of dataset CIC-IDS2017.

	2.3.1	2.3.2	2.3.3	2.3.4	2.3.5	2.3.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	600	600	600	600	600	600
Training percentage						
Specialized subset	CIC17_10	CIC17_10	CIC17_10	CIC17_10	CIC17_10	CIC17_10
Training metrics						
Best epoch	593	407	407	35	257	41
Time to best epoch	0h 43m	0h 30m	0h 30m	0h 2m	0h 19m	0h 5m
Performance metrics						
Accuracy	95.682 %	94.073 %	93.355 %	94.938 %	96.276 %	94.229 %
Detection rate	90.088 %	82.494 %	79.470 %	84.706 %	93.690 %	82.552 %
Precision	92.624 %	93.274 %	93.236 %	94.708 %	91.750 %	93.874 %
Specificity	97.574 %	97.988 %	98.050 %	98.399 %	97.151 %	98.178 %
F1-Measure	91.338 %	87.553 %	85.804 %	89.428 %	92.710 %	87.850 %
False alarm rate	7.376 %	6.726 %	6.764 %	5.292 %	8.250 %	6.126 %
Missed alarm rate	9.912 %	17.506 %	20.530 %	15.294 %	6.310 %	17.448 %

Table 6.9: Experiments 2.3.1-6 with LSTM model finetuned with subset CIC17_10 of dataset CIC-IDS2017.

accuracy and detection rate have increased to 0.594% and 3.602% respectively. All other pre-training resulted in strongly reduced performance most salient with the OBSCURE proxy task with a negative delta in accuracy of -2.327%.

It shall be noted that training with this little data entails a high variability in validation accuracy and loss over the course of training as the model might start to over-fit very early. From figures 6.1 and 6.2 showing the mean training and validation loss per epoch over the duration of fine-tuning, we can derive that this is indeed the case. While training loss diminishes for all models over almost the whole fine-tuning period, the validation loss progresses very differently for the different models, even though they are fine-tuned with the exact same subset. While models pre-trained with the AUTO and COMPOSITE proxy task converge quite early after around 40 epochs, deteriorating monotonically afterwards, the mean loss of other models fluctuates heavily. For the model which was not pre-trained and the model pre-trained with the OBSCURE proxy task, the fluctuation even seems to have a downward trend which, even though least validation loss does not necessarily mean highest validation accuracy, is also congruent with the results from table 6.9 where the best epoch for NONE and OBSCURE where 593 and 407. Plots depicting training and validation loss progression can be found in the appendix for every experiment.

It also shall be noted that for training with the CIC17_10 and UNSW15_10 the ratio of samples between categories has changed when compared to the original dataset or the stratified sampled 10% and 1% subsets. For CIC17_10 and UNSW15_10, the same amount of samples per category are included. This does not impact the comparison

		2.3.1	2.3.2	2.3.3	2.3.4	2.3.5	2.3.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Botnet ARES	0	94.737%	100.000%	100.000%	93.421%	95.946%	98.667%
FTP-Patator	1	96.000%	100.000%	65.385%	65.385%	100.000%	100.000%
SSH-Patator	2	100.000%	99.219%	99.219%	98.800%	100.000%	98.425%
DDoS LOIT	3	99.530%	97.181%	97.811%	99.915%	99.989%	99.861%
DoS GoldenEye	4	97.981%	99.194%	96.774%	97.294%	96.900%	98.113%
DoS Hulk	5	98.740%	74.322%	65.846%	66.460%	98.434%	70.648%
DoS Slowhttptest	6	99.286%	100.000%	98.333%	98.565%	100.000%	98.568%
DoS slowloris	7	92.727%	100.000%	93.782%	96.073%	97.135%	99.741%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
Infiltration	9	63.608%	77.447%	69.639%	89.872%	80.716%	77.644%
Benign	10	97.574%	97.988%	98.050%	98.399%	97.151%	98.178%
PortScan - Firewall off	11	83.707%	86.145%	91.537%	98.643%	88.805%	90.155%
PortScan - Firewall on	12	81.579%	71.053%	68.421%	75.676%	71.053%	70.270%
SQL Injection	13	100.000%	100.000%	100.000%	100.000%	100.000%	0.000%
XSS	14	80.303%	80.597%	0.000%	60.294%	82.353%	89.706%
Benign	10	97.574%	97.988%	98.050%	98.399%	97.151%	98.178%
Attack	!10	90.088%	82.494%	79.470%	84.706%	93.690%	82.552%
Overall	ALL	95.682%	94.073%	93.355%	94.938%	96.276%	94.229%

Table 6.10: Per category accuracy analysis of experiments 2.3.1-6 with LSTM model finetuned with subset CIC17_10 of dataset CIC-IDS2017.

between pre-trained and non-pre-trained models but takes from the comparability between results of experiments 2.3.1-6 in table 6.9 and the results of experiments 2.1.1-6 in table 6.5 and 2.2.1-6 in table 6.7.

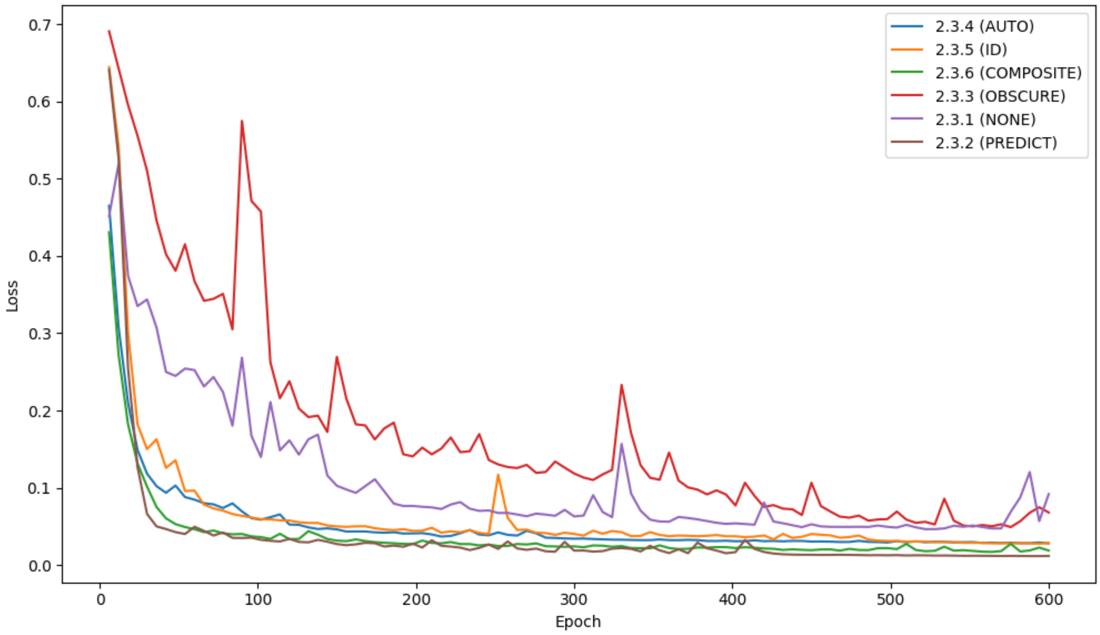

Figure 6.1: Plot of mean training loss per epoch during fine-tuning on the LSTM model with specialized subset CIC17_10.

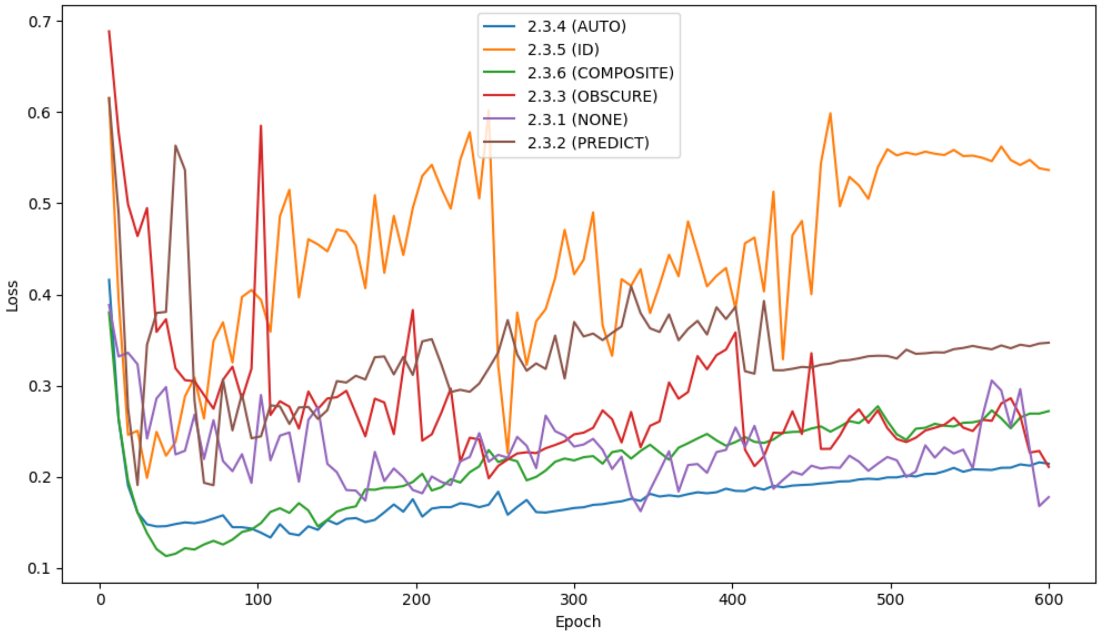

Figure 6.2: Plot of mean validation loss per epoch during fine-tuning on the LSTM model with specialized subset CIC17_10.

6.1.2 Pre-training of the LSTM model with UNSW-NB15 Dataset

A similar pattern emerges from the results of tests on the UNSW-NB15 dataset. For experiments 2.4.1-6 in table 6.11 fine-tuned with 10% of the dataset minor improvement can again be observed for the pre-trained models. The highest positive delta of 0.086% occurred after pre-training with de identity function proxy task in experiment 2.4.5 when comparing to the purely supervised training in experiment 2.4.1. Training with other proxy tasks shows comparably minor improvements to accuracy.

Looking at class specific accuracy in table 6.12 we can see that contrary to the results from experiments 2.1.1-6 on dataset CIC-IDS2017, here the increase in overall accuracy stems from minor improvements on benign flow classification i.e. an increase in specificity rather than an increase in detection rate. In fact, detection rate drops for all pre-trained models in experiments 2.4.1-6 most salient in results from experiment 2.4.6 trained on the COMPOSITE proxy-task where detection rate drops by 1.273% but specificity increased by 0.084% when compared to supervised only training in experiment 2.4.1 resulting in an improvement in accuracy of 0.037% due to 96.64% of the UNSW-NB15 dataset being benign flows.

The maximum delta between supervised only trained and pre-trained models increases to 0.137% in experiments 2.5.1-6 in table 6.13, this time occurring for proxy task AUTO with all other proxy tasks also showing minor improvements in accuracy. Just looking at experiment 2.5.4, contrary to experiments 2.4.1-6 in table 6.11, specificity and detection rate increased slightly when compared to baseline experiment 2.5.1. While in

	2.4.1	2.4.2	2.4.3	2.4.4	2.4.5	2.4.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	50	50	50	50	50	50
Training percentage	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %	10.00 %
Specialized subset						
Training metrics						
Best epoch	25	23	25	13	25	15
Time to best epoch	0h 44m	0h 40m	0h 44m	0h 29m	0h 44m	0h 56m
Performance metrics						
Accuracy	98.654 %	98.737 %	98.721 %	98.699 %	98.740 %	98.691 %
Detection rate	80.684 %	80.227 %	80.397 %	80.542 %	80.342 %	79.411 %
Precision	81.166 %	83.374 %	82.971 %	82.258 %	83.460 %	82.802 %
Specificity	99.313 %	99.414 %	99.394 %	99.364 %	99.415 %	99.397 %
F1-Measure	80.924 %	81.770 %	81.664 %	81.391 %	81.871 %	81.071 %
False alarm rate	18.834 %	16.626 %	17.029 %	17.742 %	16.540 %	17.198 %
Missed alarm rate	19.316 %	19.773 %	19.603 %	19.458 %	19.658 %	20.589 %

Table 6.11: Experiments 2.4.1-6 with LSTM model finetuned with 10% of dataset UNSW-NB15.

		2.4.1	2.4.2	2.4.3	2.4.4	2.4.5	2.4.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Analysis	0	28.846%	38.462%	28.302%	30.189%	26.415%	21.154%
Backdoors	1	90.000%	87.500%	77.500%	73.684%	80.000%	75.000%
DoS	2	84.733%	82.952%	81.888%	85.751%	80.612%	81.980%
Exploits	3	93.625%	94.055%	93.134%	93.061%	93.272%	92.610%
Fuzzers	4	51.254%	50.820%	52.972%	53.256%	51.812%	51.092%
Generic	5	93.897%	89.647%	91.589%	91.589%	91.355%	91.274%
Normal	6	99.313%	99.414%	99.394%	99.364%	99.415%	99.397%
Reconnaissance	7	96.888%	95.101%	95.784%	95.186%	97.386%	95.956%
Shellcode	8	74.510%	78.710%	71.429%	69.079%	73.377%	62.338%
Worms	9	94.737%	100.000%	94.737%	94.737%	94.737%	94.737%
Benign	6	99.313%	99.414%	99.394%	99.364%	99.415%	99.397%
Attack	!6	80.684%	80.227%	80.397%	80.542%	80.342%	79.411%
Overall	ALL	98.654%	98.737%	98.721%	98.699%	98.740%	98.691%

Table 6.12: Per category accuracy analysis of experiments 2.4.1-6 with LSTM model finetuned with 10% of dataset UNSW-NB15.

	2.5.1	2.5.2	2.5.3	2.5.4	2.5.5	2.5.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	200	200	200	200	200	200
Training percentage	1.00 %	1.00 %	1.00 %	1.00 %	1.00 %	1.00 %
Specialized subset						
Training metrics						
Best epoch	159	121	191	89	183	109
Time to best epoch	0h 52m	0h 40m	1h 3m	0h 33m	1h 1m	1h 15m
Performance metrics						
Accuracy	98.305 %	98.413 %	98.348 %	98.442 %	98.404 %	98.329 %
Detection rate	78.846 %	81.994 %	81.274 %	79.185 %	81.114 %	78.066 %
Precision	74.673 %	75.308 %	74.419 %	77.306 %	75.531 %	75.491 %
Specificity	99.019 %	99.014 %	98.974 %	99.148 %	99.037 %	99.071 %
F1-Measure	76.703 %	78.509 %	77.696 %	78.234 %	78.223 %	76.757 %
False alarm rate	25.327 %	24.692 %	25.581 %	22.694 %	24.469 %	24.509 %
Missed alarm rate	21.154 %	18.006 %	18.726 %	20.815 %	18.886 %	21.934 %

Table 6.13: Experiments 2.5.1-6 with LSTM model finetuned with 1% of dataset UNSW-NB15.

		2.5.1	2.5.2	2.5.3	2.5.4	2.5.5	2.5.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Analysis	0	43.396%	72.222%	64.151%	51.852%	65.385%	55.556%
Backdoors	1	80.000%	76.923%	72.500%	72.500%	75.000%	65.000%
DoS	2	78.261%	82.952%	78.426%	79.442%	82.995%	75.448%
Exploits	3	87.431%	91.129%	89.687%	89.184%	90.358%	86.885%
Fuzzers	4	59.632%	63.702%	65.394%	57.358%	60.941%	59.478%
Generic	5	80.189%	90.888%	84.813%	88.345%	86.150%	86.150%
Normal	6	99.019%	99.014%	98.974%	99.148%	99.037%	99.071%
Reconnaissance	7	92.017%	90.833%	90.244%	91.253%	92.172%	91.906%
Shellcode	8	82.237%	64.516%	71.711%	74.026%	78.710%	48.684%
Worms	9	78.947%	88.889%	83.333%	83.333%	88.889%	88.889%
Benign	6	99.019%	99.014%	98.974%	99.148%	99.037%	99.071%
Attack	!6	78.846%	81.994%	81.274%	79.185%	81.114%	78.066%
Overall	ALL	98.305%	98.413%	98.348%	98.442%	98.404%	98.329%

Table 6.14: Per category accuracy analysis of experiments 2.5.1-6 with LSTM model finetuned with 1% of dataset UNSW-NB15.

	2.6.1	2.6.2	2.6.3	2.6.4	2.6.5	2.6.6
Proxy task	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Epochs Supervised	600	600	600	600	600	600
Training percentage						
Specialized subset	UNSW15_10	UNSW15_10	UNSW15_10	UNSW15_10	UNSW15_10	UNSW15_10
Training metrics						
Best epoch	77	203	449	29	161	47
Time to best epoch	0h 6m	0h 18m	0h 40m	0h 2m	0h 14m	0h 8m
Performance metrics						
Accuracy	97.754 %	97.759 %	97.639 %	97.764 %	97.665 %	97.586 %
Detection rate	65.595 %	62.626 %	64.033 %	63.501 %	68.435 %	60.904 %
Precision	69.315 %	70.788 %	67.511 %	70.390 %	66.479 %	67.698 %
Specificity	98.934 %	99.050 %	98.871 %	99.021 %	98.736 %	98.933 %
F1-Measure	67.404 %	66.457 %	65.726 %	66.768 %	67.443 %	64.122 %
False alarm rate	30.685 %	29.212 %	32.489 %	29.610 %	33.521 %	32.302 %
Missed alarm rate	34.405 %	37.374 %	35.967 %	36.499 %	31.565 %	39.096 %

Table 6.15: Experiments 2.6.1-6 with LSTM model finetuned with subset UNSW15_10 of dataset UNSW-NB15.

experiments 2.4.1-6 in table 6.11, all pre-trained models show a decrease in detection rate, in experiments 2.5.1-6 in table 6.13 all pre-training methods except the COMPOSITE proxy-task result in an improved detection rate.

The pattern of improved accuracy breaks again when looking at fine-tuning with the specialized subset UNSW15_10 in table 6.15 for experiments 2.6.1-6. Here, almost no improvement is measurable and for the COMPOSITE proxy task accuracy even dropped

		2.6.1	2.6.2	2.6.3	2.6.4	2.6.5	2.6.6
Class	#	NONE	PREDICT	OBSCURE	AUTO	ID	COMPOSITE
Analysis	0	80.392%	81.132%	83.019%	78.846%	82.353%	74.510%
Backdoors	1	85.000%	80.000%	65.000%	72.500%	65.000%	71.795%
DoS	2	81.679%	76.081%	72.843%	70.229%	74.169%	69.289%
Exploits	3	85.498%	84.933%	82.993%	86.811%	84.419%	77.747%
Fuzzers	4	47.315%	37.735%	45.516%	33.799%	54.493%	38.443%
Generic	5	69.484%	81.352%	72.131%	78.960%	75.177%	70.960%
Normal	6	98.934%	99.050%	98.871%	99.021%	98.736%	98.933%
Reconnaissance	7	44.155%	39.193%	44.229%	51.222%	51.345%	53.070%
Shellcode	8	39.610%	56.494%	56.863%	48.684%	46.753%	45.161%
Worms	9	57.895%	94.737%	73.684%	84.211%	94.737%	84.211%
Benign	6	98.934%	99.050%	98.871%	99.021%	98.736%	98.933%
Attack	!6	65.595%	62.626%	64.033%	63.501%	68.435%	60.904%
Overall	ALL	97.754%	97.759%	97.639%	97.764%	97.665%	97.586%

Table 6.16: Per category accuracy analysis of experiments 2.6.1-6 with LSTM model finetuned with subset CIC17_10 of dataset UNSW-NB15.

by 0.168% when for fine-tuning with 1% and 10% it showed slight improvement. The only increases in accuracy are measurable for the PREDICT and AUTO proxy tasks with very minor accuracy increases of 0.005% and 0.010% with all other pre-training methods resulting in lower accuracy scores best represented by experiment 2.6.6 with the COMPOSITE proxy-task where accuracy dropped by 0.168%.

6.2 Transformer Model

The transformer model without pre-training produces state of the art results, similar to experiments with the LSTM model but performs slightly worse: For 90% of data for supervised training only 0.138% and 0.187% absolute difference for CIC-IDS2017 and UNSW-NB15 datasets respectively 6.17. For UNSW-NB15 for 10% and 1% fine-tuning no improvements were achieved, except for a 0.03% plus in accuracy for 3.5.4 (table 6.23) which is most likely just variance.

	1.3.1	3.1.1	3.2.1	3.3.1
Proxy task	NONE	NONE	NONE	NONE
Epochs Supervised	50	50	200	600
Training percentage	90.00 %	10.00 %	1.00 %	
Specialized subset				CIC17_10
Training metrics				
Best epoch	45	47	83	125
Time to best epoch	7h 35m	1h 16m	0h 28m	0h 11m
Performance metrics				
Accuracy	99.658 %	99.448 %	99.189 %	93.154 %
Detection rate	98.884 %	98.576 %	98.640 %	82.865 %
Precision	99.762 %	99.236 %	98.161 %	89.271 %
Specificity	99.920 %	99.743 %	99.374 %	96.633 %
F1-Measure	99.321 %	98.905 %	98.400 %	85.949 %
False alarm rate	0.238 %	0.764 %	1.839 %	10.729 %
Missed alarm rate	1.116 %	1.424 %	1.360 %	17.135 %

Table 6.17: Experiments 1.3.1, 3.1.1, 3.2.1 and 3.3.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset CIC-IDS2017.

6.2.1 Pre-training of the transformer model with CIC-IDS2017 Dataset

In experiments with the CIC-IDS2017 dataset and increase in classification accuracy was only achieved in experiment 3.3.3 (table 6.21) for pre-training with the OBSCURE proxy task and fine-tuning with the specialized subset CIC17_10. Results for pre-training with 10 an 1 percent of the dataset can be seen in tables 6.19 and 6.20 respectively. Results for training with specialized subset CIC17_10 can be found in table 6.21.

	1.4.1	3.4.1	3.5.1	3.6.1
Proxy task	NONE	NONE	NONE	NONE
Epochs Supervised	50	50	200	600
Training percentage	90.00 %	10.00 %	1.00 %	
Specialized subset				UNSW15_10
Training metrics				
Best epoch	47	47	191	563
Time to best epoch	7h 0m	1h 5m	1h 0m	0h 52m
Performance metrics				
Accuracy	98.743 %	98.545 %	98.328 %	97.684 %
Detection rate	79.300 %	88.111 %	84.212 %	58.910 %
Precision	84.283 %	75.079 %	72.790 %	70.721 %
Specificity	99.457 %	98.928 %	98.845 %	99.106 %
F1-Measure	81.716 %	81.075 %	78.086 %	64.277 %
False alarm rate	15.717 %	24.921 %	27.210 %	29.279 %
Missed alarm rate	20.700 %	11.889 %	15.788 %	41.090 %

Table 6.18: Experiments 1.4.1, 3.4.1, 3.5.1 and 3.6.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset UNSW-NB15.

	3.1.1	3.1.2	3.1.3	3.1.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	50	50	50	50
Training percentage	10.00 %	10.00 %	10.00 %	10.00 %
Specialized subset				
Training metrics				
Best epoch	47	49	49	48
Time to best epoch	1h 16m	1h 17m	1h 21m	1h 13m
Performance metrics				
Accuracy	99.448 %	99.411 %	98.724 %	99.313 %
Detection rate	98.576 %	98.584 %	97.927 %	98.427 %
Precision	99.236 %	99.079 %	97.052 %	98.851 %
Specificity	99.743 %	99.690 %	98.994 %	99.613 %
F1-Measure	98.905 %	98.831 %	97.488 %	98.639 %
False alarm rate	0.764 %	0.921 %	2.948 %	1.149 %
Missed alarm rate	1.424 %	1.416 %	2.073 %	1.573 %

Table 6.19: Experiments 3.1.1-6 with transformer encoder model finetuned with 10% of dataset CIC-IDS2017.

	3.2.1	3.2.2	3.2.3	3.2.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	200	200	200	200
Training percentage	1.00 %	1.00 %	1.00 %	1.00 %
Specialized subset				
Training metrics				
Best epoch	83	109	123	83
Time to best epoch	0h 28m	0h 38m	0h 43m	0h 29m
Performance metrics				
Accuracy	99.189 %	99.105 %	99.168 %	99.091 %
Detection rate	98.640 %	98.746 %	98.718 %	98.679 %
Precision	98.161 %	97.736 %	98.004 %	97.748 %
Specificity	99.374 %	99.226 %	99.320 %	99.231 %
F1-Measure	98.400 %	98.239 %	98.360 %	98.211 %
False alarm rate	1.839 %	2.264 %	1.996 %	2.252 %
Missed alarm rate	1.360 %	1.254 %	1.282 %	1.321 %

Table 6.20: Experiments 3.2.1-6 with transformer encoder model finetuned with 1% of dataset CIC-IDS2017.

	3.3.1	3.3.2	3.3.3	3.3.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	600	600	600	600
Training percentage				
Specialized subset	CIC17_10	CIC17_10	CIC17_10	CIC17_10
Training metrics				
Best epoch	125	191	419	281
Time to best epoch	0h 11m	0h 18m	0h 39m	0h 26m
Performance metrics				
Accuracy	93.154 %	92.099 %	93.832 %	92.398 %
Detection rate	82.865 %	84.338 %	84.675 %	82.987 %
Precision	89.271 %	84.401 %	90.310 %	86.396 %
Specificity	96.633 %	94.725 %	96.928 %	95.581 %
F1-Measure	85.949 %	84.370 %	87.402 %	84.657 %
False alarm rate	10.729 %	15.599 %	9.690 %	13.604 %
Missed alarm rate	17.135 %	15.662 %	15.325 %	17.013 %

Table 6.21: Experiments 3.3.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset CIC-IDS2017.

	3.4.1	3.4.2	3.4.3	3.4.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	50	50	50	50
Training percentage	10.00 %	10.00 %	10.00 %	10.00 %
Specialized subset				
Training metrics				
Best epoch	47	1	48	49
Time to best epoch	1h 5m	0h 2m	1h 5m	1h 7m
Performance metrics				
Accuracy	98.545 %	96.728 %	98.184 %	98.527 %
Detection rate	88.111 %	25.951 %	98.897 %	81.500 %
Precision	75.079 %	58.556 %	66.337 %	77.864 %
Specificity	98.928 %	99.326 %	98.157 %	99.151 %
F1-Measure	81.075 %	35.964 %	79.409 %	79.641 %
False alarm rate	24.921 %	41.444 %	33.663 %	22.136 %
Missed alarm rate	11.889 %	74.049 %	1.103 %	18.500 %

Table 6.22: Experiments 3.4.1-6 with transformer encoder model finetuned with 10% of dataset UNSW-NB15.

6.2.2 Pre-training of the transformer model with UNSW-NB15 Dataset

For experiments 3.6.1-4 6.24 on dataset UNSW-NB15 with subset UNSW15_10 both the AUTO proxy task yielded minor improvements of 0.295% accuracy but with the MASK proxy task the model accuracy fell by 1.213% as the model defaulted to guessing negative i.e. detection rate: 0%. The accuracy improvements in experiment 3.6.4 stems from a major increase in detection rate of an absolute 29.424% when compared to experiment 3.6.1 i.e. performance of the model without pre-training.

Surprisingly, the MASK proxy task yielded the worst results overall when looking at accuracy as it dropped for 4/6 experiment series even though it is the most similar proxy task used compared to the one used in Google BERT. To avoid cluttering this section with tables we moved the per category results for the transformer model to the appendix.

6.3 Explainability

In this section we are trying to analyze the mechanisms involved in generating the results in the previous section and analyze the datasets used. As with machine learning in general, complete explainability is hard to achieve, hence the following attempts are our best efforts to make sense of the results. We try to answer the questions

	3.5.1	3.5.2	3.5.3	3.5.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	200	200	200	200
Training percentage	1.00 %	1.00 %	1.00 %	1.00 %
Specialized subset				
Training metrics				
Best epoch	191	129	197	197
Time to best epoch	1h 0m	0h 41m	1h 2m	1h 3m
Performance metrics				
Accuracy	98.328 %	98.165 %	98.256 %	98.331 %
Detection rate	84.212 %	98.260 %	85.553 %	82.106 %
Precision	72.790 %	66.208 %	71.088 %	73.749 %
Specificity	98.845 %	98.162 %	98.722 %	98.927 %
F1-Measure	78.086 %	79.111 %	77.653 %	77.704 %
False alarm rate	27.210 %	33.792 %	28.912 %	26.251 %
Missed alarm rate	15.788 %	1.740 %	14.447 %	17.894 %

Table 6.23: Experiments 3.5.1-6 with transformer encoder model finetuned with 1% of dataset UNSW-NB15.

- Did the pre-training have any effect on the model behavior and if yes: how did predictions change?
- Which features were relevant for classification?
- Why did the model perform so well, even with very little training data?
- Were the datasets suited to be used for the conducted experiments?

To answer these questions, we conducted a series of tests including plotting neuron activation data, a PD analysis, and fitting a DTC to discern the value thresholds of features which lead to correct classifications.

6.3.1 Neuron Activation Plots

To answer the question whether pre-training had any effect at all on the behavior of the LSTM model, we looked at neuron activations of the last stage of the LSTM, i.e. the hidden state of the last stage of the last layer of the model after having processed the whole data sequence, after pre-training and after fine-tuning. The plots are category specific and where generated by processing all records of the stated attack category in the respective dataset. The plot is then the average of all obtained values and describes the mean neuron activation response of the model to this category.

	3.6.1	3.6.2	3.6.3	3.6.4
Proxy task	NONE	MASK	OBSCURE	AUTO
Epochs Supervised	600	600	600	600
Training percentage				
Specialized subset	UNSW15_10	UNSW15_10	UNSW15_10	UNSW15_10
Training metrics				
Best epoch	563	155	161	137
Time to best epoch	0h 52m	0h 13m	0h 15m	0h 12m
Performance metrics				
Accuracy	97.684 %	96.471 %	97.456 %	97.979 %
Detection rate	58.910 %	0.000 %	69.498 %	88.334 %
Precision	70.721 %	100.000 %	62.702 %	65.989 %
Specificity	99.106 %	100.000 %	98.482 %	98.333 %
F1-Measure	64.277 %	0.000 %	65.925 %	75.544 %
False alarm rate	29.279 %	0.000 %	37.298 %	34.011 %
Missed alarm rate	41.090 %	100.000 %	30.502 %	11.666 %

Table 6.24: Experiments 3.6.1-6 with transformer encoder model finetuned with subset UNSW15_10 of dataset UNSW-NB15.

The assumption was, that if the information the model learned during pre-training proved useful for classification the neurons which were activated after pre-training would still be activated after fine-tuning. The same neuron activations not being present after fine-tuning does however not mean, that pre-training had no positive effect on classification. As we are only looking at the last stage of the last layer of the LSTM, the learned information might have been used in a previous layer of the LSTM to influence later layers towards a more accurate classification which would not show in these results. To have some form of quantitative metric, we calculated the $L1$ loss, as a measurement of difference between the neurons activation after pre-training and the same neurons after fine-tuning. Instances where the neurons were not activated in either tensor, i.e. activation level < 0.1, are omitted from the plots (but not the L1 loss calculation) to increase readability. $L1$ is defined as $L1 = mean(|h_{pretraining} - h_{supervised}|)$.

As we can see in figures 6.3 and 6.4, neurons that are activated after pre-training tend to remain activated after fine-tuning which was the case in most instances. This gives us slightly more confidence that the model deems the information learned during pre-training usable and does not tend to overlearn it during fine-tuning. For the reasons stated above, no clear conclusions can be derived from these plots and metrics about the effectiveness of pre-training. We only wanted to ensure that the models did not completely *unlearn* all the information gathered during pre-training in the fine-tuning stage. This does not appear to be the case but can also not be ruled out completely as the neuron activations might

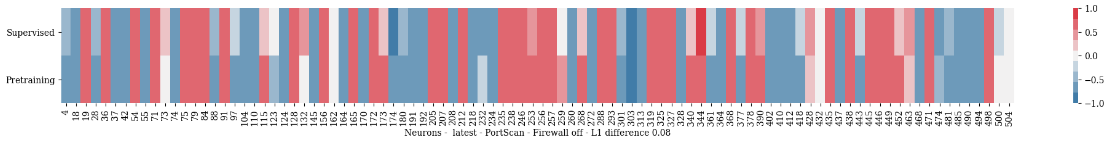

Figure 6.3: Neuron comparison plot of the average neuron activation level of the latest stage of the LSTM model after processing all flows of category *PortScan - Firewall off* in dataset CIC-IDS2017. The model was pre-trained with the ID proxy task and afterwards fine-tuning with specialized subset CIC17_10.

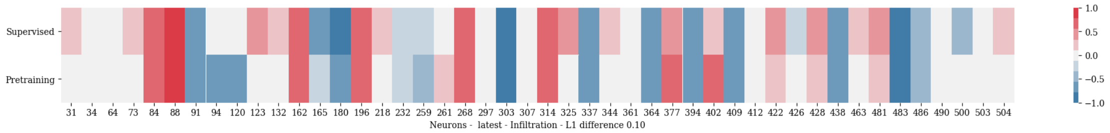

Figure 6.4: Neuron comparison plot of the average neuron activation level of the latest stage of the LSTM model after processing all flows of category *Infiltration* in dataset CIC-IDS2017. The model was pre-trained with the ID proxy task and afterwards fine-tuning with specialized subset CIC17_10.

coincidentally align after pre-training and fine-tuning even though they don't correlate. We can't rule this out completely as we don't know the expected value distributions of the activated neurons.

6.3.2 Partial Dependency Plots

We started with the **PD** analysis on some of the features to discern their effect on the classification of each attack type in the dataset. The resulting PDPs were generated with the models from experiments 6.9, 6.15, 6.21, 6.24 because we assumed that pre-training has the most impact on models trained on very little labeled training data. The features we looked at were *source port number*, *destination port number*, *protocol identifier* and *packet length*.

For both datasets, we generated plots to inspect how the predictions of the model change after altering the value of the feature - *ceteris paribus*. The value range of each feature was within the highest and lowest occurring value of said feature in the dataset. The colored graph lines represent the predictions, i.e. the activation level of the output neuron of the model trained with the respective proxy task denoted in the graph legend in the top left corner. The histogram depicted in gray together with the right-hand side vertical axis describes the number of flows in the used testset with this feature value. The testset used to generate each plot contained all records of the specified category contained in the dataset. Examples of these plots can be seen in figures 6.5 and 6.6. After we generated the data for each tuple of ($dataset$ x $model$ x $feature$ x $attacktype$) we generated plots to compare the results of the different models yielding a total of 160 plots. Most of them are of little interest as often the inspected feature had no or very little impact on the classification of the attack type. A full list of all generated PD plots can be found in the appendix.

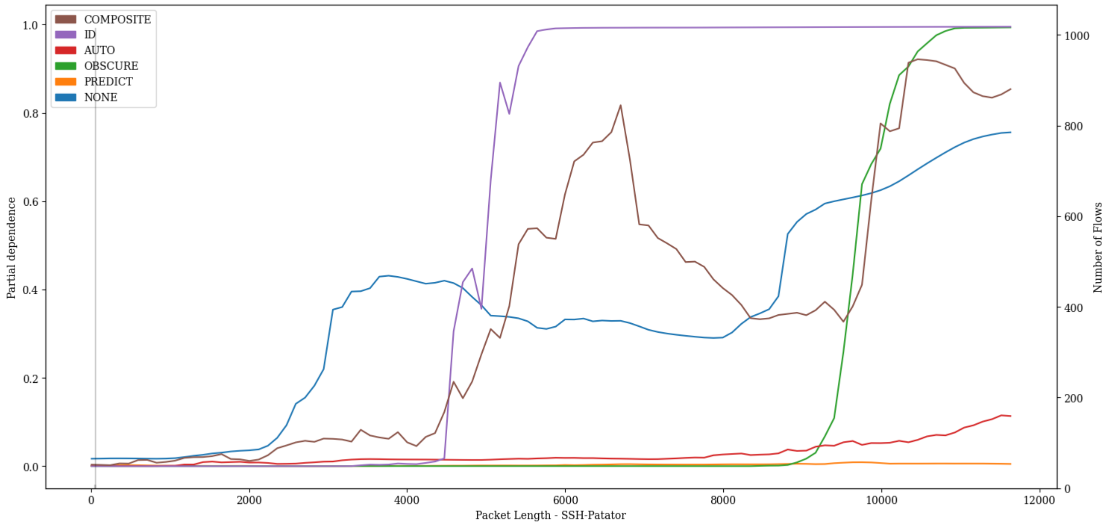

Figure 6.5: Partial dependency plot between feature *packet length* and classification of *SSH Patator* attacks in the CIC-IDS2017 dataset. The histogram describes the distribution of occurring values for feature *packet length* in flows of type *SSH Patator*.

What is immediately apparent from the the plots is that even though the final metrics of the experiments with different proxy tasks were similar, the general behavior of the models trained with different proxy tasks is affected by pre-training. In the range where the actual feature values relevant for classification of specific attack categories lay, the models behave very similar which leads to correct classification. An example of this can be seen in figure 6.6 where, deducing from the plots and from a later inspection of the decision tree thresholds of the fitted DTC, the source port is an important feature when it comes to classifying *SSH Patator* attacks. Varying this feature leads to different behavior of the differently trained models, but in the value range the feature actually takes on in the dataset, i.e. the relevant section, the models behave very similar. We will take a deeper look into this issue later in the section when we look at the structure of the decision tree used to classify each attack category. If no such clear pattern in the plot emerges like e.g. in plot 6.5, it means that there the value of the feature has little impact on the classification of the attack type which is also congruent with the results generated by the DTC which can be found later in this chapter.

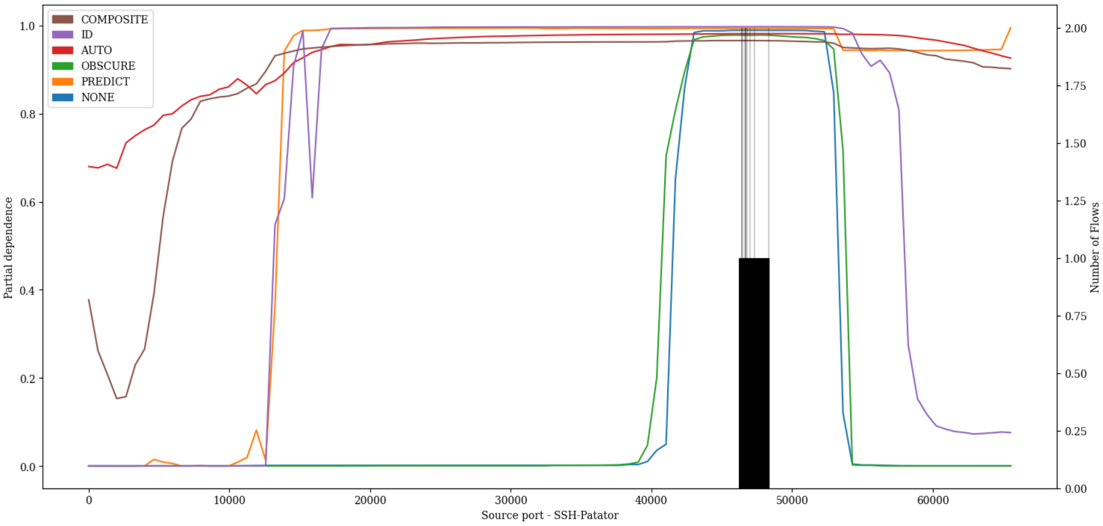

Figure 6.6: Partial dependency plot between feature *source port* and classification of *SSH Patator* attacks in the CIC-IDS2017 dataset. The histogram describes the distribution of occurring values for feature *source port* in flows of type *SSH Patator*.

6.3.3 Decision Tree Classifier

Due to the high accuracy rates the models achieved with very little training data i.e. the specialized subsets as specified in section 4.1.1, we suspected that the datasets might simply be easy to classify. To get additional information on how the models might classify the data and discern benign from attack traffic, we fitted a DTC on both used datasets and plotted the resulting decision trees. In table 6.27 we evaluated the accuracy of the decision tree classifier with different maximum depths when tasked with binary classification of the datasets.

The accuracy scores listed in tables 6.25 and 6.26 are based on per packet classification so they are not completely comparable to the results from the DNN models as their validation metrics are based on the classification of the extracted flows and not packets. The results were generated with the exact same stratified training and validation splits as were used for training the DL models to keep some comparability. The results were generated with a maximum depth of 20 and 16 for datasets CIC-IDS2017 and UNSW-NB15 respectively. With these parameters the DTC achieves peak performance before it starts to overfit as depicted in table 6.28. This indicates that records of CIC-IDS2017 dataset might be slightly harder to classify. The high accuracy achieved by the DTC on a packet basis without having any concept of flows or sequences might indicate that the datasets indeed might be simply very easy to classify.

A more detailed comparison between results yielded from the DTC and the DL models without pre-training can be found in table 6.28, again with the caveat that the DTC is evaluated on packets and not flows, hence shifting the ratio of attack to benign records

	90.00 %	10.00 %	1.00 %
Actual depth	20	20	20
Fittings time [s]	288.68	29.28	2.38
Accuracy	98.193 %	97.849 %	96.874 %
Detection rate	95.101 %	94.507 %	92.718 %
Precision	94.399 %	93.140 %	89.513 %
Specificity	98.832 %	98.547 %	97.739 %
F1-Measure	94.749 %	93.819 %	91.087 %
False alarm rate	5.601 %	6.860 %	10.487 %
Missed alarm rate	4.899 %	5.493 %	7.282 %

Table 6.25: Results of a DTCs with max. depth 20 fitted to different amounts of data of the CIC-IDS-2017 dataset.

	90.00 %	10.00 %	1.00 %
Actual depth	16	16	16
Fittings time [s]	453.49	47.94	3.58
Accuracy	99.472 %	99.373 %	99.186 %
Detection rate	82.131 %	78.172 %	72.275 %
Precision	85.627 %	84.000 %	77.801 %
Specificity	99.766 %	99.741 %	99.647 %
F1-Measure	83.842 %	80.981 %	74.936 %
False alarm rate	14.373 %	16.000 %	22.199 %
Missed alarm rate	17.869 %	21.828 %	27.725 %

Table 6.26: Results of a DTCs with max. depth 16 fitted to different amounts of data of the UNSW-NB15 dataset.

slightly. The validation subset for CIC-IDS2017 consisting of 2,493,032 packets and is composed of 82.86% benign packets and 17.14% attack packets. In flow representation, the dataset consists of 74.72% benign and 25.28% attack records. The validation subset for UNSW-NB15 consisting of 6,228,573 packets and is composed of 98.33% benign packets and 1.67% attack packets. In flow representation, the dataset consists of 96.64% benign and 3.36% attack records. Only looking at accuracy, the DTC performs similarly to the DL models, but other metrics e.g. FAR and MAR, are considerably worse especially when comparing results for the cases where the models had access to 90% of the datasets.

Next we tried to fit a DTC on subsets of the datasets containing only benign flows and flows of one attack category to obtain thresholds for the feature values which lead to the correct classification of the different attack types. We used a maximum depth of 5 as a

	CIC-IDS2017		**UNSW-NB15**	
max. depth	**accuracy**	**fitting time**	**accuracy**	**fitting time**
1	82.8558%	202.97s	98.7241%	212.02s
2	82.8558%	216.38s	98.7695%	227.32s
3	89.3871%	216.94s	98.8434%	249.41s
4	90.9993%	226.19s	98.8562%	268.7s
5	91.6236%	242.34s	98.9273%	285.73s
6	93.3365%	241.75s	99.0839%	312.4s
7	93.6416%	252.43s	99.1811%	330.9s
8	96.4964%	259.8s	99.2847%	347.64s
9	96.9901%	271.28s	99.3517%	363.32s
10	97.3088%	266.96s	99.3894%	381.01s
11	97.6354%	265.23s	99.4337%	399.33s
12	97.7696%	269.61s	99.4546%	415.14s
13	97.9199%	273.56s	99.4596%	429.8s
14	98.0323%	277.4s	99.4671%	454.4s
15	98.0646%	274.03s	99.4715%	455.26s
16	98.0982%	281.64s	99.472%	453.49s
17	98.1472%	274.37s	99.472%	457.97s
18	98.1662%	275.86s	99.468%	461.35s
19	98.1792%	276.26s	99.4609%	457.48s
20	98.1928%	288.68s	99.4531%	457.15s
21	98.1906%	297.94s	99.4447%	475.83s
22	98.167%	309.79s	99.4244%	502.92s
23	98.1783%	310.07s	99.4131%	498.34s
24	98.1524%	313.42s	99.4028%	506.54s
25	98.154%	312.18s	99.4058%	486.02s

Table 6.27: Performance of DTC for binary classification fitted on 90% of data from the respective dataset with different maximum depth values. Accuracy was calculated on the remaining 10% of data not used for fitting.

trade-off between accuracy and readability of the resulting decision trees. An overview of all the results can be seen in tables 6.29 and 6.30.

The resulting trees can be inspected to discern which features are important for classifying the different attack types. E.g. in figure 6.7 the decision tree for the classification of *SSH-Patator* flows in the CIC-IDS2017 dataset is depicted. The tree is to be interpreted as following: Each node constitutes a binary split in the data. The first line in each node contains a threshold of a feature value e.g. *Destination port <= 22.5* in the root node. All samples in the branch to the left fulfill the criterion, all samples to the right don't. The label *gini* indicates the quality of the split measured with the *Gini impurity*. The

Trained with	**CIC-IDS2017**			**UNSW-NB15**		
90.00 %	**LSTM**	**Transformer**	**DTC***	**LSTM**	**Transformer**	**DTC***
Accuracy	99.796 %	99.658 %	98.193 %	98.930 %	98.743 %	99.453 %
DR	99.306 %	98.884 %	95.101 %	82.936 %	79.300 %	82.125 %
Precision	99.885 %	99.762 %	94.399 %	86.315 %	84.283 %	84.628 %
Specificity	99.961 %	99.920 %	98.832 %	99.517 %	99.457 %	99.747 %
F1-Measure	99.595 %	99.321 %	94.749 %	84.592 %	81.716 %	83.358 %
FAR	0.115 %	0.238 %	5.601 %	13.685 %	15.717 %	15.372 %
MAR	0.694 %	1.116 %	4.899 %	17.064 %	20.700 %	17.875 %
10.00%	**LSTM**	**Transformer**	**DTC***	**LSTM**	**Transformer**	**DTC***
Accuracy	99.632 %	99.448 %	97.849 %	98.654 %	98.545 %	99.322 %
DR	98.828 %	98.576 %	94.507 %	80.684 %	88.111 %	76.731 %
Precision	99.711 %	99.236 %	93.140 %	81.166 %	75.079 %	82.350 %
Specificity	99.903 %	99.743 %	98.547 %	99.313 %	98.928 %	99.714 %
F1-Measure	99.268 %	98.905 %	93.819 %	80.924 %	81.075 %	79.441 %
FAR	0.289 %	0.764 %	6.860 %	18.834 %	24.921 %	17.650 %
MAR	1.172 %	1.424 %	5.493 %	19.316 %	11.889 %	23.269 %
1.00%	**LSTM**	**Transformer**	**DTC***	**LSTM**	**Transformer**	**DTC***
Accuracy	99.385 %	99.189 %	96.874 %	98.305 %	98.328 %	99.124 %
DR	98.408 %	98.640 %	92.718 %	78.846 %	84.212 %	69.783 %
Precision	99.153 %	98.161 %	89.513 %	74.673 %	72.790 %	76.189 %
Specificity	99.716 %	99.374 %	97.739 %	99.019 %	98.845 %	99.626 %
F1-Measure	98.779 %	98.400 %	91.087 %	76.703 %	78.086 %	72.846 %
FAR	0.847 %	1.839 %	10.487 %	25.327 %	27.210 %	23.811 %
MAR	1.592 %	1.360 %	7.282 %	21.154 %	15.788 %	30.217 %

Table 6.28: Comparison between model performances without pre-training for 90%, 10% and 1% of training data with random seed 500 and stratified sampling. DTC performance is only partly comparable as it operates on packets and not flows.

label *samples* states how many samples have been passed on to this node. The label *value* the current guess of the classifier for how the current set is comprised. The first value constitutes a guess of the DTC on how many *benign* samples are still in the set and second value of how many *attack* samples remain. The proportion of remaining *benign* and *attack* values is also depicted in the color of the node. The label *class* indicates the best guess of the DTC at this point in the tree. As the graphical representation is too small to be printed if all leafs of the tree are present, we present only the textual tree representation as generated by the *scikit-learn* ML python library [BLB$^+$13] for those instances. A list of all generated trees can be found in the appendix.

Category	#	dth	fit.t.[s]	val.acc.	tr.acc.	benign[%]	attack[%]	> guess
Botnet ARES	0	5	154.09	99.9865	99.9865	99.9455	0.0545	Yes
FTP-Patator	1	5	152.0	99.99	99.9901	99.9762	0.0238	Yes
SSH-Patator	2	5	160.35	99.7815	99.7939	99.3285	0.6715	Yes
DDoS LOIT	3	5	180.66	96.5979	96.6016	94.2669	5.7331	Yes
DoS GoldenEye	4	5	169.65	99.5204	99.5242	99.4889	0.5111	Yes
DoS Hulk	5	5	181.88	95.1116	95.1345	90.6977	9.3023	Yes
DoS Slowhttptest	6	5	185.27	99.9043	99.9052	99.8125	0.1875	Yes
DoS slowloris	7	5	176.51	99.8859	99.878	99.7809	0.2191	Yes
Heartbleed	8	4	153.24	100.0	100.0	100.0	0.0	No
Infiltration	9	5	188.35	99.5406	99.5552	98.8194	1.1806	Yes
PortScan - Firewall off	11	5	201.47	99.8179	99.8208	98.477	1.523	Yes
PortScan - Firewall on	12	5	181.81	99.9951	99.9942	99.9934	0.0066	Yes
SQL Injection	13	5	155.59	99.9995	99.9994	99.9995	0.0005	No
XSS	14	5	166.3	99.9713	99.9717	99.9713	0.0287	No

Table 6.29: Results of the DTC discerning between benign packets and packets of a certain attack type of dataset CIC-IDS-2017

Category	#	dth	fit.t.[s]	val.acc.	tr.acc.	benign[%]	attack[%]	> guess
Analysis	0	5	301.7	99.9933	99.9932	99.9933	0.0067	No
Backdoors	1	5	299.67	99.9942	99.9949	99.994	0.006	Yes
DoS	2	5	296.34	99.9013	99.8971	99.8954	0.1046	Yes
Exploits	3	5	305.68	99.3061	99.2947	99.0393	0.9607	Yes
Fuzzers	4	5	295.38	99.7286	99.7259	99.679	0.321	Yes
Generic	5	5	287.15	99.8845	99.8843	99.8711	0.1289	Yes
Reconnaissance	7	5	300.48	99.952	99.9531	99.8951	0.1049	Yes
Shellcode	8	5	289.53	99.9941	99.9938	99.9895	0.0105	Yes
Worms	9	5	290.39	99.9961	99.9936	99.9961	0.0039	No

Table 6.30: Results of the DTC discerning between benign packets and packets of a certain attack type of dataset UNSW-NB15

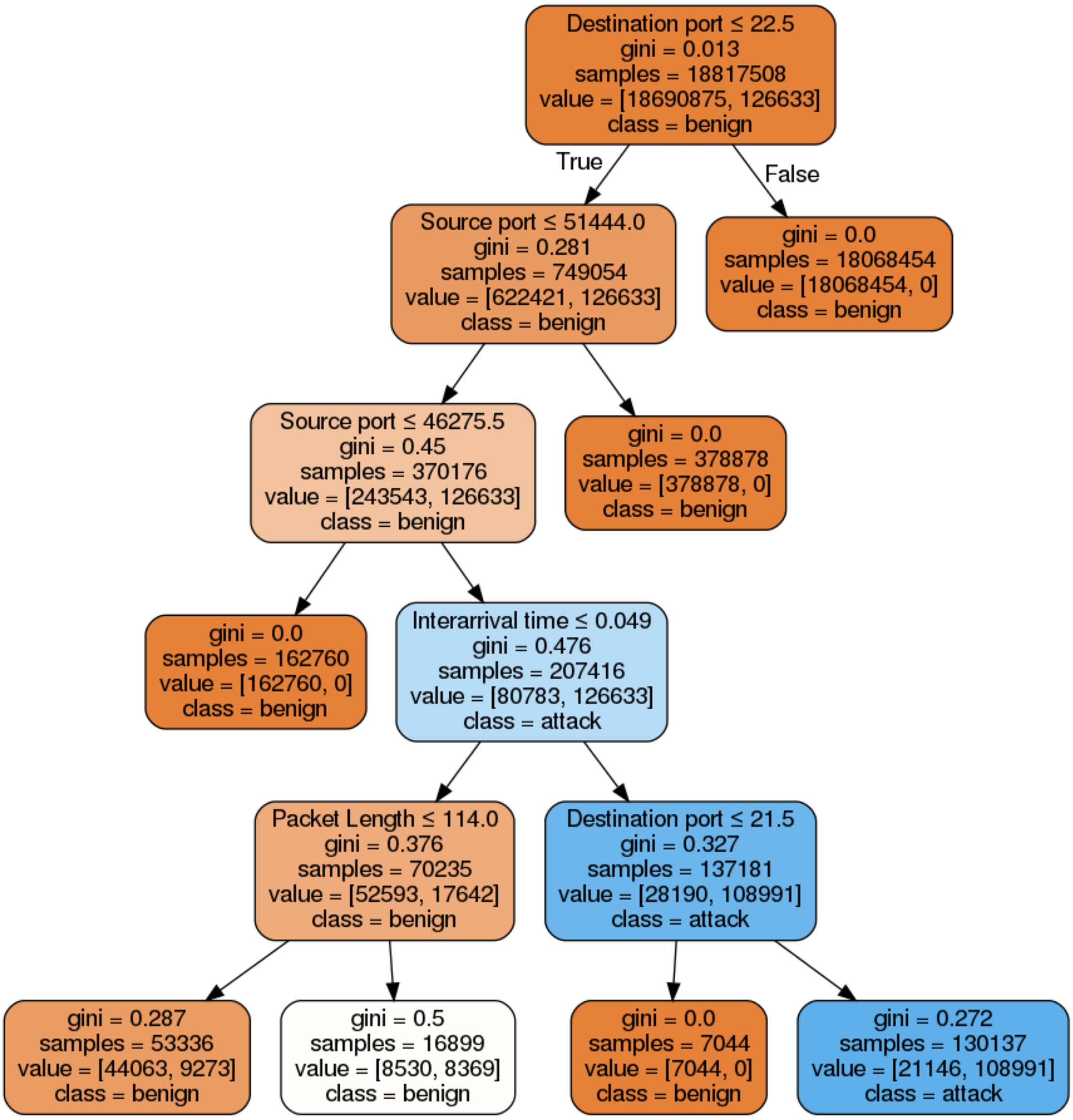

Figure 6.7: Decision tree yielded from a DTC fitted to the a subset of the CIC-IDS2017 dataset (99.329% benign samples, 0.671% attack samples) containing only benign flows and attack flows of category *SSH-Patator*. The DTC achieved an accuracy of 99.781%.

Tables 6.31, 6.32 and 6.33 constitute a closer look at the importance of different features for binary classification. The values in the tables constitute the *Gini importance* or *mean decrease impurity* which are defined as the total decrease in node impurity i.e. the Gini impurity in our case [BLB+13].

In table 6.31 the importances of the different features from both datasets are compared. The destination port and the packet length were the most valuable features for binary classification for the CIC-IDS2017 and UNSW-NB15 datasets respectively. The URG, ECE, CWR and NS flags seem to be of no importance to classification, but that is probably due to the fact that they are rarely used. Noteworthy is that for the two datasets different features are of high importance which suggests the conclusion that they are easy to classify due to different reasons.

	Gini importance	
	CIC-IDS2017 (max depth = 20)	UNSW-NB15 (max depth = 16)
Source port	0.1007	0.0270
Destination port	**0.3987**	0.1916
Protocol	0.0081	0.0043
Packet Length	0.1279	**0.5227**
Interarrival time	0.2409	0.0848
Direction	0.0373	0.0083
SYN Flag	0.0084	0.0057
FIN Flag	0.0174	0.0005
RST Flag	0.0261	0.0004
PSH Flag	0.0234	0.1318
ACK Flag	0.0110	0.0230
URG Flag	0.0000	0.0000
ECE Flag	0.0000	0.0000
CWR Flag	0.0000	0.0000
NS Flag	0.0000	0.0000

Table 6.31: Normalized Gini importances of features resulting from a DTC fitted on 90% of data from the respective dataset. Highest values are marked bold.

In tables 6.32 and 6.33 the normalized feature importance was calculate for trees fitted on subsets containing only benign flows and one selected attack category, processed by a DTC or depth 5. The most important feature for classifying the respective attack category is again marked bold. These results are mostly consistent with our results from the PD plots, even though in that case evaluation was done on flows and not packets. E.g. the high importance of *Source Port* feature when classifying the *SSH-Patator* attack category is reflected in the PD plot 6.6 above. Features might change in importance when looked at in a flow context instead of only looking at single packets.

	Botnet ARES	FTP-Patator	SSH-Patator	DDoS LOIT	DoS GoldenEye	DoS Hulk	DoS Slowhttptest	DoS slowloris	Heartbleed	Infiltration	PortScan - Firewall off	PortScan - Firewall on	SQL Injection	XSS
Source port	0.14	**0.82**	**0.58**	0.04	0.35	0.0	0.05	0.05	**0.95**	0.06	0.31	0.0	**0.58**	0.0
Destination port	**0.58**	0.03	0.26	0.16	**0.45**	**0.5**	0.27	0.02	0.05	0.29	**0.34**	0.0	0.22	0.23
Protocol	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.13	0.0	0.0
Packet Length	0.0	0.02	0.01	**0.45**	0.15	0.17	0.02	**0.75**	0.0	0.13	0.15	**0.42**	0.0	0.14
Interarrival time	0.0	0.12	0.14	0.18	0.01	0.22	**0.53**	0.02	0.0	0.0	0.16	0.07	0.2	**0.36**
Direction	0.0	0.0	0.0	0.0	0.04	0.0	0.05	0.0	0.0	0.04	0.0	0.35	0.0	0.0
SYN Flag	0.0	0.0	0.0	0.07	0.0	0.0	0.02	0.04	0.0	**0.4**	0.0	0.01	0.0	0.0
FIN Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.27
RST Flag	0.13	0.02	0.0	0.09	0.0	0.0	0.01	0.02	0.0	0.09	0.04	0.01	0.0	0.0
PSH Flag	0.0	0.0	0.0	0.0	0.0	0.01	0.04	0.1	0.0	0.0	0.0	0.0	0.0	0.0
ACK Flag	0.14	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
URG Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ECE Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CWR Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NS Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 6.32: Normalized Gini importances of features for classification of attack categories as decerned by a DTC of max. depth 5 fitted on 90% of dataset CIC-IDS-2017. The highest value is marked bold.

	Analysis	Backdoors	DoS	Exploits	Fuzzers	Generic	Reconnaissance	Shellcode	Worms
Source port	0.23	0.1	0.02	0.0	0.06	0.0	0.01	0.03	0.1
Destination port	**0.61**	0.3	**0.55**	0.21	0.3	**0.39**	**0.73**	**0.46**	0.2
Protocol	0.0	0.0	0.0	0.0	0.0	0.06	0.0	0.0	0.01
Packet Length	0.16	**0.47**	0.34	**0.55**	**0.6**	0.3	0.26	0.23	0.05
Interarrival time	0.0	0.06	0.02	0.06	0.01	0.08	0.0	0.28	0.11
Direction	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0	0.15
SYN Flag	0.0	0.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0
FIN Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RST Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
PSH Flag	0.0	0.0	0.07	0.16	0.02	0.16	0.0	0.0	**0.38**
ACK Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
URG Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ECE Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CWR Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NS Flag	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 6.33: Normalized Gini importances of features for classification of attack categories as decerned by a DTC of max. depth 5 fitted on 90% of dataset UNSW-NB15. The highest value is marked bold.

CHAPTER 7

Discussion

In this thesis we set out to inspect the possible benefits of pre-training on LSTM and transformer encoder models to answer the questions:

- 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?

The results of our experiments appear to be inconclusive. Some experiments have shown minor improvements, but later experiments with the same proxy task and less data have shown either no improvement or worse performance. As can be seen in tables 7.1 and 7.2, the LSTM model seems to be more receptive for pre-training but no clear pattern emerges which could point at a proxy-task that generally improves accuracy when used in pre-training. The first and second research questions, R1 and R2, must therefore be answered with "it depends":

- A1: Yes, pre-training can improve flow classification accuracy of LSTM models in some instances, but it can not be concluded that this is generally so.

Referring to section 7.2 it can be seen that pre-training seems to have even less positive impact on the transformer model compared to the LSTM model in general, but still yielded some positive results for experiments with very little supervised data i.e. the dedicated subsets used for experiments 3.3.3 and 3.6.4 as can be seen in tables 6.21 and 6.24.

- A2: Yes, pre-training can improve flow classification accuracy of transformer encoder models in very specific instances, but in most cases, it does not.

We could name the predict packet (PREDICT) and surprisingly the identity function (ID) proxy task as most likely to improve final accuracy of the LSTM model. In table 7.1 we can see that the highest overall absolute increase in accuracy of 0.594% was also achieved with pre-training with the ID proxy task in experiment 2.3.5 6.9.

- A3: The *predict packet (PREDICT)* and *identity function (ID)* proxy tasks where most effective in increasing flow classification accuracy for LSTM models in our experiments. For transformer encoder models it was the *auto encoder (AUTO)* proxy task with 20% dropout rate.

As is often the case, complete explainability for NNs is hard to achieve. From the PD plots we can deduce that pre-training had a significant impact on the internal behavior of the models. The differences in neuron activations after pre-training with different proxy tasks suggests the same thing. We can therefore conclude that the internal representation of data is influenced in a positive way in the instances where pre-training improved classification accuracy when compared to a model with randomly initialized weights.

- A4: Pre-training can be seen as a non-random weight initialization before conducting supervised training which leads to an alternate internal data representation in the hidden state and ultimately to an increase in flow classification accuracy.

The fact that any improvement was observable at all is already promising, when considering that the baseline results of exclusively supervised trained models are already very high when compared to other state-of-the-art results for the used datasets in contemporary research. Increasing accuracy ever so slightly without the need for more labeled data might make an ML based IDS feasible in a real world scenario when before it was not. There could be several reasons why no conclusive pattern can be found in the results:

The achieved accuracy scores by the models when fine-tuned with 90% of data were in the high 99.x percent 6.28 and even with only one percent of data used for training, accuracy was still over 99%. The flow representation used omits the packet payload, so payload based attacks like *SQL Injection* or *XSS* attacks are very hard, if not impossible, to detect. There might simply be very little room for improvement. So even if pre-training were effective, it would not show up in our results. One approach to mitigate this effect would be to switch to multinomial classification of the exact attack type instead of binary classification. This approach increases fine-tuning training time, but since unsupervised pre-training takes disproportionately longer than fine-tuning at the moment, overall training time would not increase significantly.

Experiments (#)	PREDICT(2)	OBSCURE(3)	AUTO(4)	ID(5)	COMPOSITE(6)
		CIC-IDS2017			
10% (2.1.x)	0.073%	-0.012%	-0.002%	0.095%	**0.101%**
1% (2.2.x)	**0.178%**	-0.075%	-0.007%	0.094%	0.136%
CIC2017_10 (2.3.x)	-1.609%	-2.327%	-0.744%	**0.594%**	-1.453%
		UNSW-NB15			
10% (2.4.x)	0.083%	0.067%	0.045%	**0.086%**	0.037%
1% (2.5.x)	0.108%	0.043%	**0.137%**	0.099%	0.024%
UNSW15_10 (2.6.x)	0.005%	-0.115%	**0.010%**	-0.089%	-0.168%
Max. absolute accuracy increase	0.178%	0.067%	0.137%	**0.594%**	0.136%

Table 7.1: Absolute differences of validation accuracy between differently pre-trained LSTM model and the same model without pre-training. The highest value of each row is marked bold.

Experiments (#)	MASK(2)	OBSCURE(3)	AUTO(4)
	CIC-IDS2017		
10% (3.1.x)	**-0.037%**	-0.724%	-0.135%
1% (3.2.x)	-0.084%	**-0.021%**	-0.098%
CIC2017_10 (3.3.x)	-1.055%	**0.678%**	-0.756%
	UNSW-NB15		
10% (3.4.x)	-1.817%	-0.361%	**-0.018%**
1% (3.5.x)	-0.163%	-0.072%	**0.003%**
UNSW15_10 (3.6.x)	-1.213%	-0.228%	**0.295%**
Max. absolute accuracy increase	0.037%	**0.678%**	0.295%

Table 7.2: Absolute differences of validation accuracy between differently pre-trained transformer model and the same model without pre-training. The highest value of each row is marked bold.

In accordance with the last point, the datasets we used seem to be easy to classify. Even though not completely comparable, the results from the DTC alone show that records in both datasets can be classified with high accuracy, i.e. about 97% or more 6.28 even without access to flow information. The lack of complexity of the data might be another reason why pre-training in this case showed no improvement as supervised training was always sufficient to extract all relevant information from the input data. To inspect this conjecture, a more challenging dataset should be used or a combination of different datasets. Ideal would be actual captured traffic of a mid size company during a penetration test.

Word embeddings are an integral part of modern NLP systems. This aspect is not included in our process since our feature vectors were already given. As our feature representation contains some qualitative features which are mapped to a mostly arbitrary number e.g. the IP protocol identifier and port numbers, creating embeddings for qualitative features and tuning them to the task at hand might lead to improvement in overall accuracy, independent of whether pre-training is used or not.

Insufficient data or model complexity might be another reason why pre-training has produced no consistent improvement. Google BERT [DCLT18] uses a model with 110 million parameters in its *BASE* version and 340 million in for its *LARGE* version. In comparison, our LSTM model consists of 5.3 million parameters and the transformer encoder model only contains 74 thousand parameters. In the original paper about BERT, google claims to have pre-trained their model on a corpus of about 3.3 billion words. During our work on this thesis many new NLP models like Googles T5 [RSR+19], Nvidias Megatron [SPP+19] were presented with the largest one being OpenAIs GPT-3 with 175 billion parameters, pre-trained on 410 billion tokens [BMR+20]. The datasets we are using contain around 25 million packets each which is two magnitudes smaller than the corpus Googles BERT was trained on. Larger models or significantly larger datasets would not have been feasible in our case due to lack of computational resources. One of these two reasons, i.e. either lack of model complexity or pre-training data, might be the cause of our inconclusive results. This hypotheses is of course not easily checked without the necessary resources. As already stated above, the dataset used for unsupervised pre-training could be magnified by merging multiple datasets together or even using available unlabeled network traffic data for pre-training.

Unsuited proxy tasks could also be a reason why pre-training showed little effect. The used proxy tasks appeared to us as most intuitive but other choices might have been more effective. There is also the possibility of pre-training using labeled data where e.g. a custom dataset is constructed where flows are labeled with the application layer protocol the flow is part of. Other unsupervised training methods might also be feasible e.g. energy-based unsupervised learning [RBCL07]. The possible approaches here are many and we have by no means explored all our options.

Looking at the validation and training loss progression during supervised fine-tuning, it shows that especially in experiments where models are trained on only 1% of the dataset or even less, i.e. with the specialized subsets, the models start to overfit heavily. It might be that the effects of over-fitting in these scenarios mitigates any positive effect pre-training might have had. During fine-tuning the model learns two things: Patterns in the data based on the label, but also how to classify data at all. At the very least, the fully connected layer must be trained for classification in the fine-tuning phase. During pre-training the model might have constructed a perfectly useful latent feature space, but it has not yet learned how to do actual classification based on it. This poses the difficulty of teaching the model how to classify, but not overfit it on the little data it has, overwriting any possible gains from pre-training. Deducted from this hypothesis, it should be a requirement that the time the model takes to converge should be greater than the time the model learns to classify at all. This problem is implicitly mitigated by increasing the size of the dataset.

Furthermore, pre-training was performed with parts of the same dataset which was also used for supervised fine-tuning and for validation. Even though the labels where ignored, the data used for pre-training contained attack flows. If anything, this is most likely beneficial for possible positive effects on the final accuracy and metrics and has to be considered when trying to reproduce the results. This effect could be mitigated by e.g. using one dataset for pre-training and a different dataset for fine-tuning. In a real world setting however our scenario is more likely to apply as for pre-training a NIDS it would make most sense to train the model with network traffic captured within the network the system is going to protect. This would also mean that it has to be assumed that the data also contains some attack flows as there is no way of asserting otherwise. For a real world application this approach also assures that the models learn common patterns of the specific networks they are going to be used in. It might even make sense to re-train or at least update the model periodically with recent data. Unfortunately network protocols are not as universal as natural language which makes transfer learning more difficult .i.e. it would be hard to teach the model universal patterns of network traffic. An ad-hoc approach tailored to the traffic of a single specific network is more likely to yield usable results in the near future.

An unsuited data representation might stifle the effectiveness of the used models. Even after deciding to use a flows representation, there is a wide selection of feature spaces [MZIV18]. Our analysis of the datasets in section 6.3.3, in particular table 6.31, shows that a lot of importance is accumulated in a few features while others are mostly irrelevant which shows that the selection of the correct feature and data representation has a significant effect on the performance of the models. Omitting unused or irrelevant features from the data representation would drastically decrease training time, but it is of course impossible to tell *a priori* if a feature is going to be relevant for classification of the data at hand. There are however feature selection and feature reduction methods that can and should be applied.

We used a flow representation of per-packet feature vectors instead of the often used approach of aggregating all packets of a flow into a single feature vector containing statistical data. This was, as already explained in earlier sections, to enable the use of machine learning techniques used in NLP which almost exclusively expect a sequences of

Experiment #	PREDICT(2)	OBSCURE(3)	AUTO(4)	ID(5)	COMPOSITE(6)
CIC-IDS2017					
10% (2.1.x)	Yes	No	No	Yes	Yes
1% (2.2.x)	Yes	No	No	Yes	Yes
CIC2017_10 (2.3.x)	No	No	No	Yes	No
UNSW-NB15					
10% (2.4.x)	Yes	Yes	Yes	Yes	Yes
1% (2.5.x)	Yes	Yes	Yes	Yes	Yes
UNSW15_10 (2.6.x)	Yes	No	Yes	No	No
# Cases in which pre-training improved accuracy	5/6	2/6	3/6	5/6	4/6

Table 7.3: Table of comparisons whether accuracy improved for pre-trained LSTM models when compared to supervised only trained baseline experiments.

Experiments (#)	MASK(2)	OBSCURE(3)	AUTO(4)
CIC-IDS2017			
10% (3.1.x)	No	No	No
1% (3.2.x)	No	No	No
CIC2017_10 (3.3.x)	No	Yes	No
UNSW-NB15			
10% (3.4.x)	No	No	No
1% (3.5.x)	No	No	Yes
UNSW15_10 (3.6.x)	No	No	Yes
# Cases in which pre-training improved accuracy	0/6	1/6	2/6

Table 7.4: Table of comparisons whether accuracy improved for pre-trained transformer models when compared to supervised only trained baseline experiments.

tokens as input. When considering the immense overhead of using sequences instead of a single vector and looking at the results of the DTC which has no concept of flows but still performs reasonably well, there might be better ways to build meaningful sequences. A possible approach is to use a sequence constituting of aggregated data over specified consecutive time frames comes to mind, which has already been used in other papers like [TICE19], [MDES18].

While machine learning researchers in the realm of NLP are already trying to move past the pre-training / fine-tuning paradigm [BMR$^+$20], it is yet to be explored in the context of NID. We contributed by applying these established methods in an attempt to harness them for the very needed improvement of machine learning based NID. Although by no means exhaustive, we managed to achieve a useful primer by using both state-of-the-art models and techniques combined with careful inspection of the obtained results and the used data. The tables 7.3 and 7.4 serve as a final overview on the instances in which pre-training actually improved classification accuracy. Considering the difficulties and shortcomings stated above, the fact that it was possible to improve classification accuracy in some instances remains an encouraging pointer towards the feasibility of the approach overall. Taking into account that the datasets used seem to be a mismatch for our experiments and the fact that results from experiments with the specialized subsets should be ignored due to the negative effects of over-fitting, the results for the LSTM model become a lot more promising. The facts that improvements where only minor and that our experiments where quite narrow in the sense that we only performed them on two datasets and with the same random seed, remain. For these reasons further inquiry is needed, applying the lessons we learned during our research, to confirm that the results are generalizable for sequence2sequence models in the context of deep learning based NID.

CHAPTER 8

Conclusion

The goal of our research was to explore possible applications of advancements made in other domains like NLP or visual computing to the field of NIDS. In particular we looked at pre-training with proxy tasks that has been successfully applied to solve NLP problems. Possible combinations of different data representations, models, parameterization and techniques span a vast design space which we carefully navigated to cover as much ground as was possible with the resources, computational and temporal, at hand. For this we selected two promising models for sequence processing: A RNN with LSTM cells and the attention based transformer model to perform binary classification on the NIDS datasets CIC-IDS2017 and UNSW-NB15. For pre-training, we devised different tasks for the models which would force them to find patterns and structure in the data and which could be evaluated without the need for human made labels. With the powerful PyTorch suite at its core, we developed a framework in about 5000 lines of Python code to automate training and results generation to make them as reproducible as possible. With an array of 66 experiments we tried to unearth any potential improvements pre-training might yield. As even this was not enough to give us a definitive direction to pursue, we dug deeper into the inner workings of the models and the structure of the data to maybe shed light on what worked and what did not and why. The result of our endeavor is a broad overview of possible approaches to self-supervised training on state-of-the-art machine learning models and an in-depth look into the patterns and structures in the data which allow the models to learn and improve during training. Although results where mostly inconclusive or insufficient for generalization, it is to consider that our experiments were far from exhaustive. As we discussed in previous sections, the datasets might simply be too easy to classify and there might have been little room for results to be improved by pre-training when compared to purely supervised training. One might try pre-training with more data, or with a different dataset than is used for supervised training. Different, cleverer proxy tasks might be a way to make pre-training effective. Just trying out different kinds of auto encoders, of which there are many by now, might yield interesting results. We contributed to the topic at hand by delivering a promising primer and ideas which might act as a venture point for further inquiries. The code we produced and the lessons we learned during the way are documented and will hopefully guide future approaches of self-supervised machine learning based NID.

APPENDIX A

Appendix

A.1 Transformer per Category Results

		1.3.1	3.1.1	3.2.1	3.3.1
Class	#	NONE	NONE	NONE	NONE
Botnet ARES	0	93.243%	93.333%	96.053%	97.333%
FTP-Patator	1	64.000%	44.000%	3.846%	73.077%
SSH-Patator	2	99.203%	96.850%	92.126%	98.431%
DDoS LOIT	3	99.989%	99.915%	99.936%	99.679%
DoS GoldenEye	4	100.000%	99.325%	96.486%	90.566%
DoS Hulk	5	100.000%	99.996%	99.970%	66.554%
DoS Slowhttptest	6	99.760%	98.804%	98.329%	92.271%
DoS slowloris	7	100.000%	99.742%	94.241%	90.181%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%
Infiltration	9	94.491%	93.066%	92.637%	90.164%
Benign	10	99.920%	99.743%	99.374%	96.633%
PortScan - Firewall off	11	99.142%	98.889%	99.736%	92.416%
PortScan - Firewall on	12	73.684%	73.684%	86.486%	94.737%
SQL Injection	13	0.000%	0.000%	0.000%	100.000%
XSS	14	4.412%	0.000%	0.000%	29.412%
Benign	10	99.920%	99.743%	99.374%	96.633%
Attack	!10	98.884%	98.576%	98.640%	82.865%
Overall	ALL	99.658%	99.448%	99.189%	93.154%

Table A.1: Per category accuracy analysis of experiments 1.3.1, 3.1.1, 3.2.1 and 3.3.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset CIC-IDS2017.

		3.2.1	3.2.2	3.2.3	3.2.4
Class	#	NONE	MASK	OBSCURE	AUTO
Botnet ARES	0	96.053%	92.000%	88.158%	93.333%
FTP-Patator	1	3.846%	0.000%	44.000%	8.000%
SSH-Patator	2	92.126%	94.882%	94.444%	98.425%
DDoS LOIT	3	99.936%	99.989%	99.925%	99.968%
DoS GoldenEye	4	96.486%	98.649%	97.820%	97.564%
DoS Hulk	5	99.970%	99.927%	99.957%	99.948%
DoS Slowhttptest	6	98.329%	99.282%	97.108%	94.724%
DoS slowloris	7	94.241%	96.891%	95.822%	94.026%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%
Infiltration	9	92.637%	93.096%	93.001%	92.772%
Benign	10	99.374%	99.226%	99.320%	99.231%
PortScan - Firewall off	11	99.736%	99.748%	99.773%	99.817%
PortScan - Firewall on	12	86.486%	76.316%	72.973%	75.000%
SQL Injection	13	0.000%	0.000%	0.000%	0.000%
XSS	14	0.000%	0.000%	0.000%	0.000%
Benign	10	99.374%	99.226%	99.320%	99.231%
Attack	!10	98.640%	98.746%	98.718%	98.679%
Overall	ALL	99.189%	99.105%	99.168%	99.091%

Table A.2: Per category accuracy analysis of experiments 3.2.1-6 with transformer encoder model finetuned with 1% of dataset CIC-IDS2017.

		3.3.1	3.3.2	3.3.3	3.3.4
Class	#	NONE	MASK	OBSCURE	AUTO
Botnet ARES	0	97.333%	97.368%	97.368%	100.000%
FTP-Patator	1	73.077%	61.538%	38.462%	88.462%
SSH-Patator	2	98.431%	86.667%	94.466%	98.828%
DDoS LOIT	3	99.679%	98.121%	99.251%	99.808%
DoS GoldenEye	4	90.566%	90.946%	96.221%	93.631%
DoS Hulk	5	66.554%	64.833%	66.073%	76.947%
DoS Slowhttptest	6	92.271%	96.659%	96.394%	95.433%
DoS slowloris	7	90.181%	96.114%	90.439%	90.415%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%
Infiltration	9	90.164%	92.623%	91.751%	63.775%
Benign	10	96.633%	94.725%	96.928%	95.581%
PortScan - Firewall off	11	92.416%	99.773%	99.123%	89.737%
PortScan - Firewall on	12	94.737%	83.333%	88.889%	76.316%
SQL Injection	13	100.000%	0.000%	0.000%	0.000%
XSS	14	29.412%	100.000%	4.478%	73.529%
Benign	10	96.633%	94.725%	96.928%	95.581%
Attack	!10	82.865%	84.338%	84.675%	82.987%
Overall	ALL	93.154%	92.099%	93.832%	92.398%

Table A.3: Per category accuracy analysis of experiments 3.3.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset CIC-IDS2017.

		3.1.1	3.1.2	3.1.3	3.1.4
Class	#	NONE	MASK	OBSCURE	AUTO
Botnet ARES	0	93.333%	94.595%	3.947%	1.316%
FTP-Patator	1	44.000%	3.846%	0.000%	3.846%
SSH-Patator	2	96.850%	94.902%	0.000%	92.126%
DDoS LOIT	3	99.915%	99.968%	99.947%	99.979%
DoS GoldenEye	4	99.325%	98.654%	91.757%	98.922%
DoS Hulk	5	99.996%	99.991%	99.909%	99.991%
DoS Slowhttptest	6	98.804%	98.801%	95.943%	98.558%
DoS slowloris	7	99.742%	97.172%	96.382%	99.741%
Heartbleed	8	100.000%	100.000%	100.000%	100.000%
Infiltration	9	93.066%	93.237%	91.982%	92.937%
Benign	10	99.743%	99.690%	98.994%	99.613%
PortScan - Firewall off	11	98.889%	98.980%	99.704%	98.946%
PortScan - Firewall on	12	73.684%	81.579%	76.316%	86.486%
SQL Injection	13	0.000%	100.000%	0.000%	100.000%
XSS	14	0.000%	0.000%	0.000%	0.000%
Benign	10	99.743%	99.690%	98.994%	99.613%
Attack	!10	98.576%	98.584%	97.927%	98.427%
Overall	ALL	99.448%	99.411%	98.724%	99.313%

Table A.4: Per category accuracy analysis of experiments 3.1.1-6 with transformer encoder model finetuned with 10% of dataset CIC-IDS2017.

		1.4.1	3.4.1	3.5.1	3.6.1
Class	#	NONE	NONE	NONE	NONE
Analysis	0	22.642%	25.000%	69.811%	75.000%
Backdoors	1	87.500%	100.000%	76.923%	72.500%
DoS	2	87.595%	87.468%	85.751%	67.595%
Exploits	3	93.370%	93.899%	90.283%	66.793%
Fuzzers	4	46.258%	73.537%	67.714%	43.258%
Generic	5	93.224%	96.028%	90.187%	67.849%
Normal	6	99.457%	98.928%	98.845%	99.106%
Reconnaissance	7	94.510%	97.561%	94.458%	55.228%
Shellcode	8	96.129%	98.710%	96.711%	89.032%
Worms	9	94.737%	94.737%	100.000%	89.474%
Benign	6	99.457%	98.928%	98.845%	99.106%
Attack	!6	79.300%	88.111%	84.212%	58.910%
Overall	ALL	98.743%	98.545%	98.328%	97.684%

Table A.5: Per category accuracy analysis of experiments 1.4.1, 3.4.1, 3.5.1 and 3.6.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset UNSW-NB15.

		3.4.1	3.4.2	3.4.3	3.4.4
Class	#	NONE	MASK	OBSCURE	AUTO
Analysis	0	25.000%	19.231%	100.000%	19.231%
Backdoors	1	100.000%	7.500%	85.000%	70.000%
DoS	2	87.468%	18.066%	97.222%	72.222%
Exploits	3	93.899%	14.340%	98.383%	87.880%
Fuzzers	4	73.537%	31.864%	99.517%	63.628%
Generic	5	96.028%	30.374%	98.829%	90.845%
Normal	6	98.928%	99.326%	98.157%	99.151%
Reconnaissance	7	97.561%	42.617%	99.916%	97.315%
Shellcode	8	98.710%	52.903%	100.000%	99.355%
Worms	9	94.737%	15.789%	100.000%	89.474%
Benign	6	98.928%	99.326%	98.157%	99.151%
Attack	!6	88.111%	25.951%	98.897%	81.500%
Overall	ALL	98.545%	96.728%	98.184%	98.527%

Table A.6: Per category accuracy analysis of experiments 3.4.1-6 with transformer encoder model finetuned with 10% of dataset UNSW-NB15.

		3.5.1	3.5.2	3.5.3	3.5.4
Class	#	NONE	MASK	OBSCURE	AUTO
Analysis	0	69.811%	96.296%	59.259%	29.630%
Backdoors	1	76.923%	85.000%	76.923%	61.538%
DoS	2	85.751%	97.710%	81.218%	73.858%
Exploits	3	90.283%	97.619%	89.099%	87.483%
Fuzzers	4	67.714%	98.936%	77.183%	68.837%
Generic	5	90.187%	96.503%	88.028%	90.376%
Normal	6	98.845%	98.162%	98.722%	98.927%
Reconnaissance	7	94.458%	99.916%	93.182%	92.593%
Shellcode	8	96.711%	98.701%	87.662%	98.052%
Worms	9	100.000%	100.000%	88.889%	100.000%
Benign	6	98.845%	98.162%	98.722%	98.927%
Attack	!6	84.212%	98.260%	85.553%	82.106%
Overall	ALL	98.328%	98.165%	98.256%	98.331%

Table A.7: Per category accuracy analysis of experiments 3.5.1-6 with transformer encoder model finetuned with 1% of dataset UNSW-NB15.

		3.6.1	3.6.2	3.6.3	3.6.4
Class	#	NONE	MASK	OBSCURE	AUTO
Analysis	0	75.000%	0.000%	77.358%	100.000%
Backdoors	1	72.500%	0.000%	75.000%	82.500%
DoS	2	67.595%	0.000%	67.513%	91.349%
Exploits	3	66.793%	0.000%	57.846%	92.509%
Fuzzers	4	43.258%	0.000%	75.676%	88.389%
Generic	5	67.849%	0.000%	61.321%	95.093%
Normal	6	99.106%	100.000%	98.482%	98.333%
Reconnaissance	7	55.228%	0.000%	86.207%	73.249%
Shellcode	8	89.032%	0.000%	99.355%	94.839%
Worms	9	89.474%	0.000%	78.947%	100.000%
Benign	6	99.106%	100.000%	98.482%	98.333%
Attack	!6	58.910%	0.000%	69.498%	88.334%
Overall	ALL	97.684%	96.471%	97.456%	97.979%

Table A.8: Per category accuracy analysis of experiments 3.6.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset UNSW-NB15.

List of Figures

2.1 Depiction of an unrolled RNN with $x^{(t)}$ being the input sequence, $\hat{y}^{(t)}$ the output sequence, and $h^{(t)}$ the internal state of the RNN after each processing stage. 9
2.2 One LSTM memory cell [Lip15] 10
2.3 Self attention layer of Transformer by [VSP$^+$17b] 11
2.4 Transformer encoder model as proposed by [VSP$^+$17b] 12
2.5 Visualization of an auto encoder. The input is encoded and subsequently decoded yielding and approximate reconstruction of the image [BKG20] . 13

3.1 Our design decisions (blue) and alternatives (green) based on the data source taxonomy proposed by Hongyu Liu et al. [LL19]. 18
3.2 Our design decisions (blue) and alternatives (green) based on the detection method taxonomy proposed by Hongyu Liu et al. [LL19]. 18
3.3 Composite model for input reconstruction and future prediction [SMS15] 24
3.4 Data flow in a three layered LSTM network. 25
3.5 Layer-wise pre-training of LSTM-SAE model. [SK19b] 26

4.1 All steps performed in dataset preprocessing to yield pre-training, training and validation splits. 35
4.2 Depiction of the LSTM model. 36
4.3 Depiction of the transformer encoder model used for classification. 37

5.1 Depiction of data flow, input and output of the model during pre-training on the identity function proxy task (ID). 43
5.2 Depiction of data flow, input and output of the model during pre-training on the prediction proxy task (PREDICT). 44
5.3 Depiction of data flow, input and output of the LSTM model during pre-training on the *Mask Packet* proxy task (MASK). 45
5.4 Depiction of data flow, input and output of the LSTM model during pre-training on the *Obscure Feature* proxy task (OBSCURE). 46
5.5 Depiction of data flow, input and output of the LSTM model during pre-training on *Auto Encoder* proxy task (AUTO). The fully connected layer is omitted from the graphic. 47

5.6 Depiction of data flow, input and output of the LSTM model during pre-training with the *Composite* proxy task (COMPOSITE). The fully connected layer is omitted from the graphic. 49
5.7 Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Obscure Feature* proxy task (OBSCURE). . . 51
5.8 Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Auto Encoder* proxy task (AUTO). 52
5.9 Depiction of data flow, input and output of the transformer encoder model during pre-training with the *Mask Packet* proxy task (MASK). 53

6.1 Plot of mean training loss per epoch during fine-tuning on the LSTM model with specialized subset CIC17_10. 64
6.2 Plot of mean validation loss per epoch during fine-tuning on the LSTM model with specialized subset CIC17_10. 64
6.3 Neuron comparison plot of the average neuron activation level of the latest stage of the LSTM model after processing all flows of category *PortScan - Firewall off* in dataset CIC-IDS2017. The model was pre-trained with the ID proxy task and afterwards fine-tuning with specialized subset CIC17_10. 75
6.4 Neuron comparison plot of the average neuron activation level of the latest stage of the LSTM model after processing all flows of category *Infiltration* in dataset CIC-IDS2017. The model was pre-trained with the ID proxy task and afterwards fine-tuning with specialized subset CIC17_10. 75
6.5 Partial dependency plot between feature *packet length* and classification of *SSH Patator* attacks in the CIC-IDS2017 dataset. The histogram describes the distribution of occurring values for feature *packet length* in flows of type *SSH Patator*. 76
6.6 Partial dependency plot between feature *source port* and classification of *SSH Patator* attacks in the CIC-IDS2017 dataset. The histogram describes the distribution of occurring values for feature *source port* in flows of type *SSH Patator*. 77
6.7 Decision tree yielded from a DTC fitted to the a subset of the CIC-IDS2017 dataset (99.329% benign samples, 0.671% attack samples) containing only benign flows and attack flows of category *SSH-Patator*. The DTC achieved an accuracy of 99.781%. 82

List of Tables

3.1 Evaluation metrics of DL models with the five data sets in binary classification. [SYZ20] 20
3.2 Evaluation metrics of DL models with the five data sets in binary classification. P denotes precision, R recall and F1-M the F1 measure. [SYZ20] 20
3.3 Model performance comparison for CIC-IDS2017 dataset [TICE19] 21
3.4 Summary of results on Action Recognition [SMS15] 24
3.5 The results of DLSTM and LSTM-SAE using data set 1 [SK19b] 27
3.6 The results of DLSTM and LSTM-SAE using data set 2 [SK19b] 27

4.1 UNSW-NB15 dataset record distribution by attack category [MS15]. . . . 30
4.2 CIC-IDS2017 dataset record distribution by attack category. [PB18]. 31
4.3 Subset **CIC17_10** devised for CIC-IDS2017 to include a minimal amount of records amounting to approximately 0.023% of the total dataset. 32
4.4 Subset **UNSW15_10** devised for UNSW-NB15 to include a minimal amount of records amounting to approximately 0.11% of the total dataset. 32
4.5 Packet features [PB18]. 34

5.1 List of baseline training runs used for comparison with results from pre-trained models. 40
5.2 Devised proxy tasks for pre-training of DL models. 41
5.3 Training and pre-training configurations for LSTM model with different proxy tasks. 42
5.4 Training and pre-training configurations for transformer model with different proxy tasks. 50

6.1 Experiments 1.1.1, 2.1.1, 2.2.1 and 2.3.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset CIC-IDS2017. 56
6.2 Per category accuracy analysis of experiments 1.1.1, 2.1.1, 2.2.1 and 2.3.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset CIC-IDS2017. 57
6.3 Experiments 1.2.1, 2.4.1, 2.5.1 and 2.6.1 with LSTM model trained in a purely supervised fashion on different percentages of data for UNSW-NB15. . . . 58

6.4 Per category accuracy analysis of experiments 1.2.1, 2.4.1, 2.5.1 and 2.6.1 with LSTM model trained in a purely supervised fashion on different percentages of data from dataset UNSW-NB15. 58
6.5 Experiments 2.1.1-6 with LSTM model finetuned with 10% of dataset CIC-IDS2017. 59
6.6 Per category accuracy analysis of experiments 2.1.1-6 with LSTM model finetuned with 10% of dataset CIC-IDS2017. 60
6.7 Experiments 2.2.1-6 with LSTM model finetuned with 1% of dataset CIC-IDS2017. 61
6.8 Per category accuracy analysis of experiments 2.2.1-6 with LSTM model finetuned with 1% of dataset CIC-IDS2017. 61
6.9 Experiments 2.3.1-6 with LSTM model finetuned with subset CIC17_10 of dataset CIC-IDS2017. 62
6.10 Per category accuracy analysis of experiments 2.3.1-6 with LSTM model finetuned with subset CIC17_10 of dataset CIC-IDS2017. 63
6.11 Experiments 2.4.1-6 with LSTM model finetuned with 10% of dataset UNSW-NB15. 65
6.12 Per category accuracy analysis of experiments 2.4.1-6 with LSTM model finetuned with 10% of dataset UNSW-NB15. 66
6.13 Experiments 2.5.1-6 with LSTM model finetuned with 1% of dataset UNSW-NB15. 66
6.14 Per category accuracy analysis of experiments 2.5.1-6 with LSTM model finetuned with 1% of dataset UNSW-NB15. 67
6.15 Experiments 2.6.1-6 with LSTM model finetuned with subset UNSW15_10 of dataset UNSW-NB15. 67
6.16 Per category accuracy analysis of experiments 2.6.1-6 with LSTM model finetuned with subset CIC17_10 of dataset UNSW-NB15. 68
6.17 Experiments 1.3.1, 3.1.1, 3.2.1 and 3.3.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset CIC-IDS2017. 69
6.18 Experiments 1.4.1, 3.4.1, 3.5.1 and 3.6.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset UNSW-NB15. 70
6.19 Experiments 3.1.1-6 with transformer encoder model finetuned with 10% of dataset CIC-IDS2017. 70
6.20 Experiments 3.2.1-6 with transformer encoder model finetuned with 1% of dataset CIC-IDS2017. 71
6.21 Experiments 3.3.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset CIC-IDS2017. 71
6.22 Experiments 3.4.1-6 with transformer encoder model finetuned with 10% of dataset UNSW-NB15. 72
6.23 Experiments 3.5.1-6 with transformer encoder model finetuned with 1% of dataset UNSW-NB15. 73

6.24 Experiments 3.6.1-6 with transformer encoder model finetuned with subset UNSW15_10 of dataset UNSW-NB15. 74
6.25 Results of a DTCs with max. depth 20 fitted to different amounts of data of the CIC-IDS-2017 dataset. 78
6.26 Results of a DTCs with max. depth 16 fitted to different amounts of data of the UNSW-NB15 dataset. 78
6.27 Performance of DTC for binary classification fitted on 90% of data from the respective dataset with different maximum depth values. Accuracy was calculated on the remaining 10% of data not used for fitting. 79
6.28 Comparison between model performances without pre-training for 90%, 10% and 1% of training data with random seed 500 and stratified sampling. DTC performance is only partly comparable as it operates on packets and not flows. 80
6.29 Results of the DTC discerning between benign packets and packets of a certain attack type of dataset CIC-IDS-2017 81
6.30 Results of the DTC discerning between benign packets and packets of a certain attack type of dataset UNSW-NB15 81
6.31 Normalized Gini importances of features resulting from a DTC fitted on 90% of data from the respective dataset. Highest values are marked bold. . . . 83
6.32 Normalized Gini importances of features for classification of attack categories as decerned by a DTC of max. depth 5 fitted on 90% of dataset CIC-IDS-2017. The highest value is marked bold. 84
6.33 Normalized Gini importances of features for classification of attack categories as decerned by a DTC of max. depth 5 fitted on 90% of dataset UNSW-NB15. The highest value is marked bold. 84

7.1 Absolute differences of validation accuracy between differently pre-trained LSTM model and the same model without pre-training. The highest value of each row is marked bold. 87
7.2 Absolute differences of validation accuracy between differently pre-trained transformer model and the same model without pre-training. The highest value of each row is marked bold. 87
7.3 Table of comparisons whether accuracy improved for pre-trained LSTM models when compared to supervised only trained baseline experiments. 90
7.4 Table of comparisons whether accuracy improved for pre-trained transformer models when compared to supervised only trained baseline experiments. . 90

A.1 Per category accuracy analysis of experiments 1.3.1, 3.1.1, 3.2.1 and 3.3.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset CIC-IDS2017. 96
A.2 Per category accuracy analysis of experiments 3.2.1-6 with transformer encoder model finetuned with 1% of dataset CIC-IDS2017. 97
A.3 Per category accuracy analysis of experiments 3.3.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset CIC-IDS2017. 98

A.4 Per category accuracy analysis of experiments 3.1.1-6 with transformer encoder model finetuned with 10% of dataset CIC-IDS2017. 99
A.5 Per category accuracy analysis of experiments 1.4.1, 3.4.1, 3.5.1 and 3.6.1 with transformer encoder model trained in a purely supervised fashion on different amounts of data from dataset UNSW-NB15. 100
A.6 Per category accuracy analysis of experiments 3.4.1-6 with transformer encoder model finetuned with 10% of dataset UNSW-NB15. 100
A.7 Per category accuracy analysis of experiments 3.5.1-6 with transformer encoder model finetuned with 1% of dataset UNSW-NB15. 101
A.8 Per category accuracy analysis of experiments 3.6.1-6 with transformer encoder model finetuned with subset CIC17_10 of dataset UNSW-NB15. 101

Acronyms

ANN	Artificial Neural Network
BCE	Binary Cross Entropy
BERT	Bidirectional Encoder Representations from Transformers
Bi-LSTM	Bidirectional Long Short-Term Memory
CEL	Cross Entropy Loss
CLF	Conditional Random Fields
CNN	Convolutional Neural Network
CNN-LSTM	Convolutional Neural Network Long Short-Term Memory
DL	Deep Learning
DLSTM	Deep Long Short-Term Memory
DNN	Deep Neural Network
DR	Detection Rate
DTC	Decision Tree Classifier
FAR	False Alarm Rate
FF	Feed Forward
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit

IAT	Interarrival Time
IDS	Intrusion Detection System
IoT	Internet of Things
LSTM	Long Short-Term Memory
LSTM-AE	LSTM-based Auto-Encoder
LSTM-SAE	LSTM-based Stacked Auto-Encoder
MAE	Mean Absolute Error
MAR	Missed Alarm Rate
ML	Machine Learning
MLM	Masked LM
MSE	Mean Squared Error
MTS	Multivariat Time Series
NID	Network Intrusion Detection
NIDS	Network Intrusion Detection System
NLP	Natural Language Processing
NN	Neural Network
NSP	Next Sentence Prediction
PD	Partial Dependence
PDP	Partial Dependence Plot
R2L	Remote to Local
ReLU	Rectified Linear Unit
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
SDA	Scaled Dot-Product Attention
SDSA	Scaled Dot-Product Self-Attention
SEL	Squared Error Loss
SGD	Stochastic Gradient Descent
SMAPE	Symmetric Mean Absolute Percentage Error
TN	True Negative
TP	True Positive
U2R	User to Root

Bibliography

[ASR21] Nikita Araslanov, Simone Schaub-Meyer, and Stefan Roth. Dense unsupervised learning for video segmentation. *CoRR*, abs/2111.06265, 2021.

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

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

[BLB$^+$13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[BMR$^+$20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[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.

[JEP$^+$21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, Aug 2021.

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

[KSCP21] Minguk Kang, Woohyeon Shim, Minsu Cho, and Jaesik Park. Rebooting ACGAN: auxiliary classifier gans with stable training. *CoRR*, abs/2111.01118, 2021.

[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.

[oT19] Vienna University of Technology. go-flows, 2019.

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

[PKBB19] Aditya Phadke, Mohit Kulkarni, Pranav Bhawalkar, and Rashmi Bhattad. A review of machine learning methodologies for network intrusion detection. In *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pages 272–275, 2019.

[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.

[Pow08] David Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation. *Mach. Learn. Technol.*, 2, 01 2008.

[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.

[RBCL07] Marc'Aurelio Ranzato, Y-Lan Boureau, Sumit Chopra, and Yann LeCun. A unified energy-based framework for unsupervised learning. In Marina Meila and Xiaotong Shen, editors, *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, volume 2 of *Proceedings of Machine Learning Research*, pages 371–379, San Juan, Puerto Rico, 21–24 Mar 2007. PMLR.

[RJL18] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *CoRR*, abs/1806.03822, 2018.

[RSR+19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

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

[SGSG19] Jenni A. M. Sidey-Gibbons and Chris J. Sidey-Gibbons. Machine learning in medicine: a practical introduction. *BMC Medical Research Methodology*, 19(1):64, Mar 2019.

[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.

[SPP$^+$19] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[SYS$^+$20] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702.e13, 2020.

[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.

[TICE19] Mengxuan Tan, Alfonso Iacovazzi, Ngai-Man Man Cheung, and Yuval Elovici. A neural attention model for real-time network intrusion detection. In *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, pages 291–299, 2019.

[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.