BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

NEURAL NETWORKS AT THE LEVEL OF NETWORK PACKETS AND FLOWS

NEURONOVÉ SÍTĚ NA ÚROVNI SÍŤOVÝCH PAKETŮ A TOKŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **PETR URBÁNEK**
AUTOR PRÁCE

SUPERVISOR **Ing. DANIEL POLIAKOV,**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

156741

Institut:	Department of Information Systems (DIFS)
Student:	**Urbánek Petr, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Cybersecurity
Title:	**Neural Networks at the Level of Network Packets and Flows**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Get acquainted with methods for monitoring computer networks at the network flow level and with the open source implementation of ipfixprobe exporter.
2. Explore the possibilities of integrating neural networks over network flows, primarily using the models of the PyTorch library.
3. Propose an extension to the ipfixprobe exporter that will make it possible to deploy different neural network architectures directly when exporting network flows.
4. Implement the proposed extension.
5. Test and evaluate the extension. Focus especially on performance tests and limits depending on the size of the delivered network.
6. Discuss achieved results.

Literature:

- Hofstede, R.; Čeleda, P.; et al. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. IEEE Communications Surveys & Tutorials, volume 16, no. 4, 2014: pp. 2037–2064, doi:10.1109/COMST.2014.2321898.
- Stevens, E.; Antiga, L.; Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications.
- Balestriero, R.; et al. A cookbook of self-supervised learning. *arXiv preprint arXiv:2304.12210* (2023). https://doi.org/10.48550/arXiv.2304.12210

Requirements for the semestral defence:
The first two points of the assignment and at least some initial work on the third point.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Poliakov Daniel, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	25.10.2023

Abstract

This thesis addresses the integration of neural networks into network flow monitoring, particularly focusing on the ipfixprobe — an open-source network flow exporter developed by CESNET. The objective is to explore the potential of neural networks for classifying and extracting representations from network flows. The challenges of deploying such solutions in large-scale production environments are considered, with a specific emphasis on enhancing efficiency and effectiveness in dynamic technological landscapes.

Abstrakt

Tato diplomová práce se zabývá integrací neuronových sítí do monitorování toků v síti, zejména se zaměřením na ipfixprobe — open-source exportér IP toků sítí vyvinutý společností CESNET. Cílem je zkoumat potenciál neuronových sítí pro klasifikaci a extrakci reprezentací ze síťových toků. Jsou zde zvažovány výzvy spojené s nasazením takových řešení ve velkém měřítku v produkčních prostředích, s konkrétním důrazem na zlepšení efektivity a účinnosti v dynamickém technologickém prostředí.

Keywords

Computer networks, Network flow, Neural Networks, PyTorch, ipfixprobe

Klíčová slova

Počítačové sítě, Síťové toky, Neuronové sítě, PyTorch, ipfixprobe

Reference

URBÁNEK, Petr. *Neural Networks at the Level of Network Packets and Flows.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Daniel Poliakov,

Rozšířený abstrakt

V neustále se vyvíjejícím prostředí moderních technologií je pro úspěch podniků a organizací nejdůležitější efektivita a odolnost síťových infrastruktur. S rozšiřováním připojených zařízení a systémů je udržování optimálního zabezpečení a výkonu sítě stále složitější. Tradiční přístupy k analýze sítí, které se soustředí především na detekci anomálií a analýzu komunikace, se mění v sofistikované disciplíny, které mají zásadní význam pro integritu a funkčnost systému.

Tato práce se zabývá integrací neuronových sítí do monitorování síťových toků, se zvláštním zaměřením na ipfixprobe, open-source exportér síťových toků vyvinutý sdružením CESNET. Hlavním cílem je prozkoumat potenciál neuronových sítí při analýze síťových toků a řešit výzvy spojené s nasazením takových řešení ve velkých sítích.

Úvodní část uvádí do kontextu význam síťové infrastruktury v dnešním digitálním ekosystému a zdůrazňuje rostoucí složitost udržování bezpečnosti a výkonnosti při rozšiřující se konektivitě. Zdůrazňuje nutnost využití technik strojového učení pro efektivní monitorování sítě, zejména ve scénářích šifrované komunikace, kde tradiční metody selhávají. Dále se snaží přiblížit teoretické znalosti nutné k vývoji pluginu a aplikaci strojového učení.

Práce vytváří most mezi modelem neuronové sítě a flow eportérem ipfixprobe od sdružení CESNET. Ipfixprobe podporuje architekturu zásuvných modulů, a proto se práce zaměřuje na vytvoření takového modulu a jeho intergaci. Plugin musí být napsán v C++, ale pro vývojáře a uživatele ipfixprobe je pohodlnější používat řešení PyTorch pro neuronové sítě, který je ale v základu dělaný pro práci v jazyce python. Aby se developeři nemuseli učit jak psát neuronové sítě a jiné metodiky strojového učení v C++ a mohli použít podobné postupy jako při práci v pythonu, využívá TorchScript a C++ frontend API pro komunikaci mezi modelem a zásuvným modulem. Tato architektura umožňuje zaměňovat modely neuronové sítě bez nutnosti úprav nebo opětovné kompilace aplikace ipfixprobe.

Následující kapitoly poskytují komplexní zkoumání monitorování síťových toků a objasňují základní koncepty a metodiky nezbytné pro praktickou analýzu sítí. Navrhovaná architektura integruje neuronové sítě do systému ipfixprobe a nabízí vhled do procesu vývoje a technických složitostí. Kapitola o implementaci popisuje kroky provedené při realizaci zásuvného modulu, od importu knihoven až po analýzu výkonnostních metrik. Dále analyzuje funkčnost a rychlost učení a rychlost inference v použití zásuvného modulu s modelem neuronové sítě. Na závěr jsou popsané dosažené výsledky, nedostatky a možnosti budoucího rozzíšení.

Neural Networks at the Level of Network Packets and Flows

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Daniel Poliakov. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Petr Urbánek
May 16, 2024

Acknowledgements

I would like to express sincere gratitude to Ing. Daniel Poliakov, for his guidance and enormous patience and great feedback. Also, I thank Ing. Miloslav Zadrazil for keeping me motivated while finishing this thesis.

Contents

List of Figures

Chapter 1

Introduction

In the current dynamic world of modern technology, where digital connectivity can decide a business's success or failure, the resistance and efficiency of network infrastructure are essential. With the increasing number of devices and their connections to the internet and other systems, the complexity of maintaining optimal network security and performance is increasing. Traditionally, network analysis is based on anomaly detection and communication analysis, once a simple task of catching undesired communication in the form of attacks or the overall system health evolves into a new and complex discipline crucial for the preservation and smooth running of the system.

The majority of network traffic works on top of TCP and UDP transport protocols. Communication from one device to another can be categorized into individual IP flows 2.3, which are then browsed to find anomalies. Usually, we want to protect our data and the data we send. Therefore, the usage of encrypted communication has become common practice. However, this brings new challenges to network monitoring as we cannot ascertain the specific content of the transmitted data. We can only orient ourselves in such situations by examining headers, sizes, or packet arrival times.

Moreover, here comes the question of leveraging machine learning techniques for effective network monitoring. Could we utilize machine learning to help discern patterns, detect anomalies, and ensure the security and performance of encrypted network communication?

The main objective of this thesis is to explore the possibilities of neural networks for the classification or feature extraction of network flows. The main advantage is the possibility of using more information inside a packet, although this brings overhead to network monitoring tooling. Other challenges include determining how the network should be trained, what data it should contain, or how extensive the neural network itself should be.

Furthermore, the main problem is if it is possible to integrate it into existing systems and how. This thesis delves into dynamic network monitoring, exploring traditional approaches, and tries to use artificial intelligence (AI) to simplify these tasks.

Some papers have researched and demonstrated the usefulness of neural networks over packets [14, 24]. However, they are still being prepared to be used in a production environment, as the tooling must be deployed on fast and extensive networks. This work focuses on the exporter of ipfixprobe, an open-source exporter of network flows deployed over networks in CESNET. It is architectured for pluggability, meaning we can use already implemented plugins or create and deploy new ones with different functionalities [4].

The following chapters comprehensively explore network flow monitoring, delving into fundamental concepts and methodologies essential for practical network analysis. The initial section provides a basic description of network flow monitoring, elucidating key components

such as packet capture, IP flow, and IPFIX (IP Flow Information Export). The discourse extends to the intricacies of flow exporters, unraveling architecture with a specific focus on the ipfixprobe developed by CESNET. Next, the focus turns towards incorporating neural networks, providing readers with a profound introduction to these formidable computational models and their indelible impact on network monitoring. The proposed architecture is described in the chapter proposal, which describes integrating neural networks into ipfixprobe and weaving together insights from the preceding discussions. The implementation chapter describes what had to be done to achieve the final plugin, from importing libraries to the ipfixprobe building process to describing the main methods. The basic functionality and performance were analyzed in the evaluation chapter, figuring out the current problems and limits of the newly developed plugin. Finally, the conclusion chapter summarizes the work done and some ideas for future improvements.

Chapter 2

Network Flow Monitoring

Network monitoring can be divided into two categories: active and passive. The passive approach only observes the traffic without modifying it; the active approach, on the other hand, manipulates the packets on the link [29].

Packet capture is one of the passive approach options, which generally provides the most insight. However, this method may be challenging to implement in extensive high-speed networks, as it creates significant overhead and requires special hardware and infrastructure [27].

Much more suited for high-speed networks is another passive approach called flow export. This method aggregates packets into flows further described below 2.3 and stores them for later analysis. As a lot less data is stored and analyzed, it is more suitable for use in data centers like networks [27].

This chapter will primarily discuss the passive approaches, mainly IP Flow 2.3.

2.1 Packet Capture

Packet capture serves three different purposes without a specific hierarchy:

1. It provides an interactive approach to network monitoring.
2. It functions as a file containing the route of packets.
3. It involves the active capturing of packets from the network.

Captured data can be saved to a file or read directly in real-time using a network traffic analyzer such as Wireshark or others [29].

The captured network traffic serves as a basis for comprehensive network monitoring. It can be saved to a file for later analysis, either as a temporary file during the monitoring process or explicitly for future use. The captured data remains true to the original traffic, and the packet capture process can be both manual and automatic. Capturing network traffic can be achieved through two primary methods: in-line mode and mirroring mode, both essential for effective network monitoring [29].

2.1.1 Mirroring Mode

Mirroring mode involves packet forwarding devices mirroring packets from one or more ports to another port where a capture device is connected. Also known as port mirroring,

port monitoring, or Switched Port Analyzer (SPAN) session. This method requires a configuration change in the forwarding device but does not incur additional costs in the form of hardware or software. However, mirroring may introduce delays or mess with the content of packets. It is crucial to select a mirror port with sufficient bandwidth, ideally twice that of the monitored port, for full-duplex captures [9].

2.1.2 In-Line Mode

In the in-line mode, the capture device is directly linked to the monitored connection between two hosts. This involves additional hardware, such as bridging hosts or network test access points (taps), designed to duplicate all passing traffic without introducing delays or altering data [9].

Network taps employ passive splitting for networks, ensuring continuous traffic flow even if the tap experiences issues. After installation, capture devices can be connected or disconnected without affecting the monitored link [9].

2.2 Deep Packet Inspection

Deep Packet Inspection (DPI), also known as complete packet inspection or Information extraction (IX), is a method for filtering computer network traffic packet by packet. It examines the header and packet payload as it traverses an inspection point within the network. This examination aims to identify protocol non-compliance, viruses, spam, intrusions, or specific criteria, determining whether the packet should be allowed to pass through or be redirected to a different destination. Contrary to a common misconception, DPI does not solely focus on payload parts but extends to various fundamental functions, including recognition and action based on the recognition results. Recognition involves identifying hidden packet characteristics, such as application protocols, viruses, worms, or specific format data. At the same time, actions are triggered based on the recognition results, ranging from logging for network analysis to rigorous discarding in security applications [31].

The applications of DPI are diverse and complex, making classification challenging due to the combination of recognition methods and actions across various levels. Some typical DPI applications include network visibility, user profiling, network security, copyright policing, censorship, or content regulation. These applications rely on recognizing and identifying characteristics such as protocols, applications, URLs, text strings, particular format data, viruses, malware, and cyber-intrusions [31].

Network security is a critical DPI application, initially developed for intrusion detection, addressing threats from external attackers and internal insiders. DPI-equipped network operators can detect known malware. DPI for network security combines early intrusion detection systems (IDS) with later intrusion prevention systems (IPS) to detect and prevent potential threats. Additionally, DPI plays a crucial role in data loss prevention (DLP), aiming to prevent sensitive data from leaving the intranet [31].

2.3 IP Flow

IP flow refers to a set of IP packets observed passing through a network during a specific time interval. It is a fundamental concept in networking and plays a crucial role in various network management and security applications [9].

All packets belonging to a particular flow share common properties known as flow keys, which include source and destination IP addresses, source and destination port numbers, IP protocol, and possibly other relevant attributes. Most commonly, the flow is described as a 5-way tuple of values in IP header [27]:

$$(ip\_src, ip\_dst, port\_src, port\_dst, proto) \tag{2.1}$$

It is essential to clarify that TCP connection does not equal IP flow, as flow exists when TCP protocol is not used. Any traffic sent between the source address and port to the destination port and address is IP flow. Also, it has no size restrictions, so even if only a single packet is sent over this combination, it is considered flow [27].

Flow observation does not store payloads in its entirety, so the amount of data for analysis is significantly reduced, which is why it is faster than other approaches on the same hardware. It is also less inclined to influence of private (encrypted) traffic because usually the payload is what is encrypted [29].

2.4 Architecture

The architecture of IPFIX can be seen in the Figure 2.1. It encompasses metering processes (MPs), exporting processes (EPs), and collecting processes (CPs), enabling the generation, export, and reception of flows, respectively. In large multilayer deployments, intermediate processes (ImPs) were introduced to modify flows for aggregation, correlation, or anonymization [30].

IP flow accounting consists of two main processes: flow exporting and flow collection, performed by the flow exporter and collector.

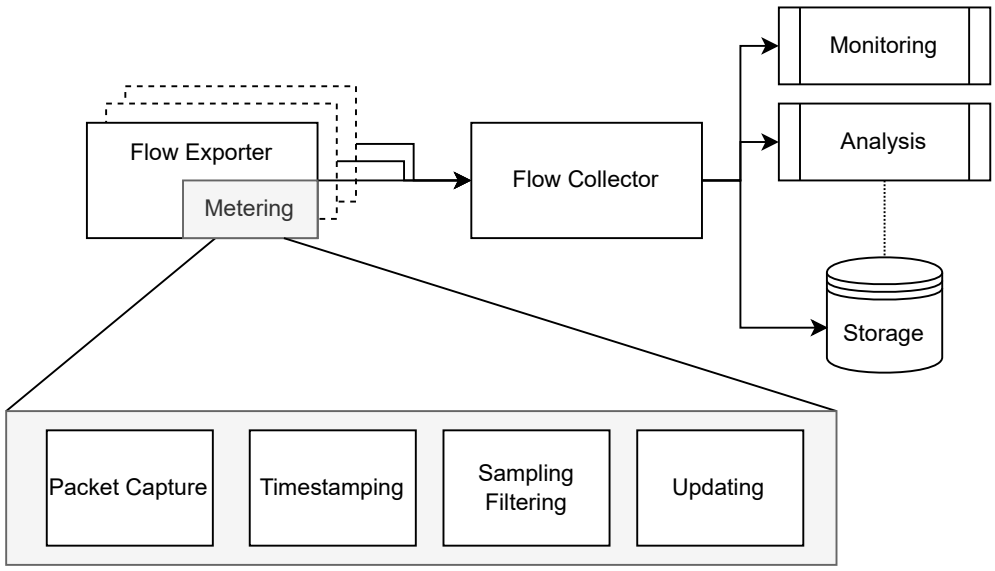

Figure 2.1: Flow monitoring architecture [27].

2.4.1 Flow Exporter

A flow exporter, commonly known as an observation point, is responsible for generating flow records from observed network traffic. The flow exporter extracts packet headers from each packet passing through the monitored interface, marking each header with a timestamp indicating when it was captured. These headers undergo further processing, including sampling or filtering, before being updated in a flow cache. A flow entry in the cache is created or updated for each incoming packet, and when a flow record expires, it is sent to the flow collector. Different criteria, such as idle time, active timeout, or the detection of specific flags in a TCP flow, determine when a flow is considered expired [27].

2.4.2 Flow Collector

The counterpart to flow export is flow collection, which involves retrieving flow records generated by flow exporters and storing them in a format suitable for subsequent analysis. A dedicated component known as the flow collector is responsible for this task. The flow collector collects and organizes flow records, providing an extensive view of network usage. It ensures that the valuable insights obtained from flow-based analysis are effectively utilized for various purposes, such as performance monitoring, anomaly detection, or intrusion prevention [9, 27].

2.5 IPFIX Protocol

In 1996, flow export technology called NetFlow was patented by CISCO, initially invented for flow switching. The first publicly available version was NetFlow v5 in 2002, which was later replaced with NetFlow v9. It incorporated adaptable data formats using templates and introduced support for IPv6, Virtual Local Area Networks (VLANs), and Multiprotocol Label Switching (MPLS), among various other features. The versatility introduced by NetFlow v9 has paved the way for recent advancements, allowing for greater flexibility in flow definitions and representation, and later became the basis for the IPFIX protocol [9].

The IP Flow Information eXport (IPFIX) protocol stands as a pivotal development in network management, addressing the growing need for detailed network flow data. Emerging over the past decade, IPFIX represents a standardized approach to exporting flow information from devices like routers and dedicated probes. The protocol was conceived to offer a finer-grained perspective on network traffic compared to conventional methods, such as Simple Network Management Protocol (SNMP) queries, allowing for scalability in large networks [30].

The protocol relies on a message format comprising data framing, encoding, and an information model. Key to its flexibility is the use of templates, which describe the format of data records. IPFIX supports the use of reduced-length encoding for efficient export of network flow data. In IPFIX, the flow keys can be flexibly defined, not only as a 5-way tuple but even all common properties from the whole packets in the flow [6, 30].

2.6 Extending IP Flows

IPFIX protocol, as well as NetFlow v9 and others, supports template set architecture, which enables extending the information inside of flow records. However, extending IP flows is

not unified; each protocol and flow exporter may extend them differently. Properties of the IPFIX information model are described in RFC 5102 [5].

The task of the template set is to describe the structure of data records. The data records' values are transmitted separately to avoid unnecessary and repeated transmission of this description. Template sets are typically transmitted only at certain time intervals, thus significantly reducing the size of the transmitted data. Template sets have their ID set to the value 2. Their body contains a sequence of template records defining the structure of records in data sets. A template record consists of a header containing the template ID and the number of data fields in the template, followed by the definitions of individual fields. The field definition contains only a numerical identifier (Information Element ID) and its length. The meaning of the field is determined by its ID, which refers to the table of elements of the information model [25].

The body of the data set is a sequence of data records whose structure is specified by a template with the same ID as the ID of this set. Data records now consist only of the values of the fields themselves, the meaning of which is determined by the field definition in the corresponding template [25].

Chapter 3

CESNET ipfixprobe

Ipfixprobe is a flow exporter 2.4.1 created and maintained by CESNET. It is an open-source tool written in C++, which offers the functionality of creating bidirectional flows from incoming packets in real-time or from captured traffic. This functionality can be extended by activating plugins or implementing new ones [4, 11].

Ipfixprobe software is not dependent on an operation system or hardware platform, which makes it possible to use different environments, e.g., integration into active network elements or annotating data on a workstation. Statistics can be created offline based on earlier captured traffic in the pcap file, or it can process the traffic on the input network interface in real time [11].

The exporter configuration is provided through the command line, as the program has no user interface. One part of the configuration happens in compilation (the possibility to add chosen plugins), and the other in executing the exporter with specific parameters. These parameters ensure the activation of plugins and handing over of the necessary values [4, 11].

```
./ipfixprobe -i 'raw;ifc=eth0' -o 'text'
```

Listing 3.1: Capturing traffic from interface 'eth0' using raw sockets and printing resulting flows to console [4].

3.1 Architecture

The architecture of ipfixprobe developed by CESNET 3.1 is designed to efficiently capture and analyze IP flow data for network monitoring and analysis purposes [11].

After receiving the packet from the pcap file or input interface, the key flow elements are extracted, and the packet is saved into the flow cache. Record of type bitflow is either created if it is the first packet of the flow or updated. The routines of activated process plugins are called to aggregate and extract properties of followed IP flows. The routines differ for the update or creation of the flow. A particular service routine is called for each plugin at the time of record export. Finally, the output plugin, connecting to the collector, sends the exported flows through the output interface [11].

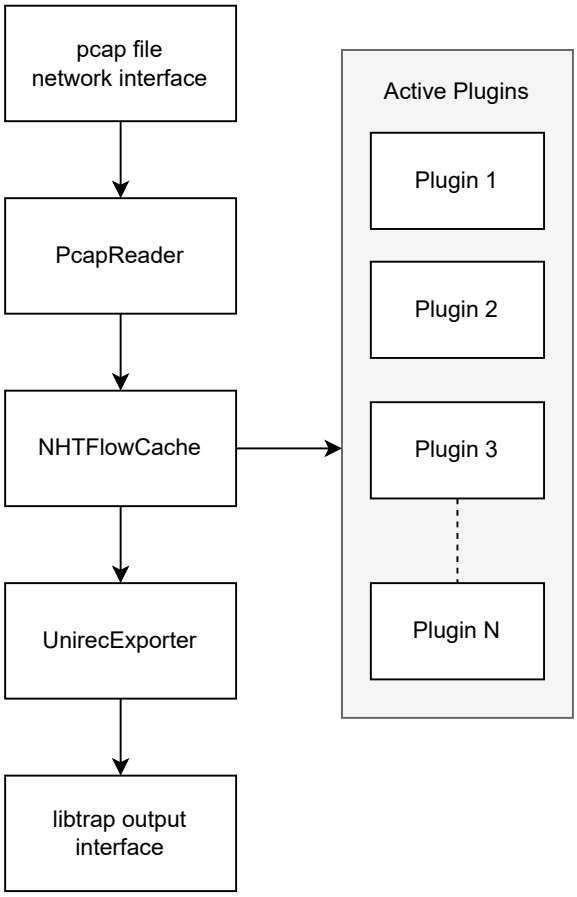

Figure 3.1: Simplified function diagram of ipfixprobe [4].

3.2 Plugins

Plugins affect incoming traffic (input plugins), e.g., pcap files or sockets of OS, data processing (processing plugins), and the final export of the processed data (output plugins). Common plugins provide parsers and aggregators of application protocols like DNS, HTTP, SIP, NTP, and SMTP. Also, there is a possibility to create a new plugin; for easier access to this functionality is provided a script, create_plugin.sh, which creates source and header files with the standard structure used throughout already implemented plugins [11, 26].

Plugins can be divided into three categories [4, 26]:

- **input plugins** facilitate the provision of input data in the appropriate format to processing plugins.
- **processing plugins** handle tasks such as protocol recognition and statistics collection.
- **output plugins** are responsible for exporting the processed data, or flows, for subsequent processing

This thesis focuses mainly on processing plugins, encompassing several phases:

- Metadata initialization
- Metadata enriching
- Flow export

The metadata initialization involves allocating memory for the plugin and flows' essential metadata, followed by its initialization. Metadata enrichment unfolds in two stages: pre-update and post-update. The pre-update stage is activated upon receiving a new packet that has not yet undergone processing by the internal ipfixprobe flow metadata enricher—conversely, the post-update stage triggers once the internal flow metadata has been enriched. The last stage is flow export, where the results from the flow processing are aggregated and exported. The sequence in which plugins are processed is dictated by their order in the initialization phase [4, 26].

Chapter 4

Neural Networks

In an era dominated by interconnected systems and digital communication, the significance of robust network security measures cannot be overstated. As organizations strive to safeguard their digital assets and sensitive information, monitoring IP flows has emerged as a crucial aspect of network security. Traditional methods have proven effective to a certain extent, yet the evolving landscape of cyber threats demands innovative and adaptive solutions [9, 24].

This chapter delves into the realm of neural networks 4.1, a powerful paradigm in machine learning, and explores their application in IP flow monitoring.

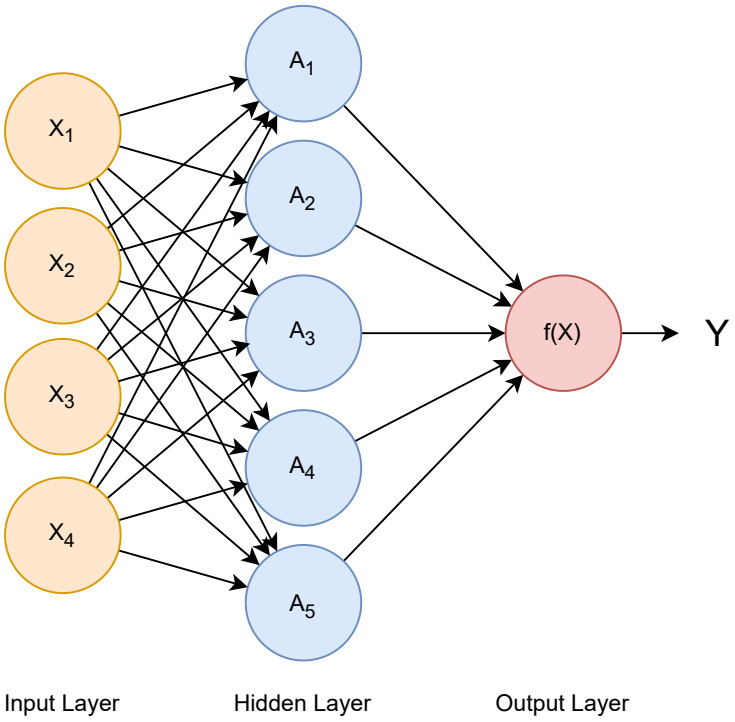

Figure 4.1: FFNN with single hidden layer [10].

4.1 Introduction to Neural Networks

To comprehend the role of neural networks in monitoring IP flows, it is imperative first to grasp the fundamental principles underlying these artificial intelligence systems.

This section provides a concise overview of neural networks, their structure, and the mechanisms through which they learn from data. Key concepts such as neurons and activation functions will be clarified, setting the stage for a deeper exploration of their application in the context of network security.

4.1.1 Basic Terms

Neural networks, the backbone of modern artificial intelligence, are intricate systems designed to mimic the complex information processing of the human brain. To effectively navigate the world of neural networks, it is essential to familiarize oneself with a set of fundamental terms and concepts that form the building blocks of the field.

Neuron

Neurons, in the field of deep learning, serve as essential computational units within artificial neural networks. They draw inspiration from biological neurons; these units are arranged into layers, constructing intricate networks capable of solving and mastering patterns [3].

Layer

Layers are fundamental building blocks within neural networks. Feed-forward neural networks are structured hierarchically, each layer serving a specific purpose. It can be categorized into three types: input, hidden, and output [10].

The input layer receives raw data, which is then processed and transformed through one or more hidden layers. Each hidden layer extracts and learns features from the input data, contributing to the network's ability to discern patterns. The final layer, known as the output layer, produces the network's predictions or classifications [10].

Feed-Forward Neural Network

Feed-Forward Neural Network (FFNN) embody a fundamental architecture. These networks, comprising layers of interconnected nodes, facilitate unidirectional information flow from the input layer, traversing hidden layers, to the output layer. The sequential progression of information within FFNNs makes them particularly adept at tasks like pattern recognition and classification [3].

Optimizer

Optimizers play a crucial role in training deep learning models by minimizing the error or loss function. In deep neural networks, the optimization process involves adjusting the weights and biases of the NN to enhance its performance. Various optimization algorithms, such as stochastic gradient descent (SGD) and Adam, differ in their approaches to updating parameters, aiming to converge towards the optimal configuration. The choice of an optimizer can significantly impact the convergence speed and final performance of a deep learning model [3].

Weights

Weights are critical parameters in neural network architecture. They determine the impact of each feature on the prediction outcome. A positive weight implies that increasing the feature's value will increase the prediction, while a negative weight suggests the opposite effect. These weights are adjusted during training to minimize the difference between the predicted output and the target. The magnitude of the weight indicates the strength of its influence on the prediction, with larger magnitudes corresponding to more significant effects. If a feature's weight is zero, it does not impact the prediction [3].

Bias

Biases are crucial components working in conjunction with weights to influence the output of each node. Unlike weights, biases are independent of input and serve to shift the overall activation of a neuron. In training, biases are adjusted alongside weights to minimize the overall error [3].

Gradient

Gradients represent the partial derivatives of the error function concerning the model parameters, such as weights and biases. The gradient provides crucial information about the direction and rate at which the error function changes concerning parameter adjustments. In the training phase, optimization algorithms utilize gradients to update the model's parameters [10].

$$R(X^m) = \frac{\partial R(X)}{\partial X} \tag{4.1}$$

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a standard optimization algorithm in deep learning. It is a variant of the traditional gradient descent that processes and updates parameters based on a randomly selected subset or individual data point from the training set rather than the entire dataset. This randomness introduces a noise level and offers computational efficiency, making it particularly well-suited for large datasets [3].

Adam

Adam is a frequently employed optimization algorithm in deep learning. It merges ideas from momentum and RMSprop optimization. Adam adjusts learning rates for individual parameters by computing both the gradients' mean (first-order moment) and uncentered variance (second-order moment). These moments are then used to update the parameters, providing advantages such as faster convergence and robustness to noisy gradients [3].

Minibatch

Rather than updating model weights based on the entire training dataset or individual data points, a small subset or minibatch of input data is used each time a gradient step is computed (each iteration). This approach offers computational efficiency, as it harnesses parallel processing and proves especially advantageous when dealing with large datasets [10].

Training

The training process refers to iteratively adjusting model parameters, such as weights and biases, to minimize the difference between predicted and actual outputs. This optimization is typically achieved by using a labeled dataset and a chosen loss function that quantifies the model's performance. The training phase involves forward and backward passes through the network, where input data is fed forward to produce predictions, and then gradients are computed backward to update the model parameters [3].

Inference

Inference involves utilizing a trained model to make predictions or classifications on new, unseen data. Once the model has undergone training, it becomes ready for deployment in practical scenarios. During inference, input data is fed into the trained model, and predictions are generated based on the learned parameters. The effectiveness and accuracy of the inference process are of the highest importance for the model's practical applicability [3].

Normalization

Data normalization is a preprocessing step in training neural networks. Normalization can be applied to the input features or the activations at different network layers. It involves scaling and centering the input data so that it has a mean of zero and a standard deviation of one. Normalization helps improve the convergence of gradient-based optimization algorithms and ensures that the features are on a similar scale, preventing some features from dominating others during training [7].

4.2 Classification and Embedding

Neural networks are vital in deep learning and can be used for prediction, classification, natural language processing, or image and speech recognition. This thesis mainly focuses on classification and feature extraction.

4.2.1 Training

Training routines are vital when creating a neural network because they are the primary mechanism through which these powerful models acquire the knowledge and skills needed to perform valuable tasks in diverse domains. During training, NN is given a large amount of training data, and it iteratively changes its parameters like weights and bias [3].

Mathematically, if the inputs are denoted as $x = [x_1, x_2, ...x_n]$ and the weights as $w = [w_1, w_2, ...w_n]$ then the weighted sum (logit) of inputs can be expressed as 4.2.

$$z = \sum_{i=0}^{n} (w_i \cdot x_i) \tag{4.2}$$

This weighted sum is then passed through an activation function f, which introduces non-linearity into the model and produces the output $y = f(z)$. The output can then be transmitted to other neurons in the network [3].

However, the weights are assigned using the initializing method, which will most likely differ from the weights we aim for. Training aims to adjust the weights and other parameters as optimally as possible, which tries to minimize errors made in learning examples. It

typically uses the loss function to compare the error function of current data and new example [3].

4.2.2 Loss Function

The loss function is a feature that measures the disparity between predicted and actual values. Acting as the optimization criterion during training, it directs the model toward parameter adjustments to minimize the error. The choice of a suitable loss function is critical, affecting the effectiveness of the network in learning and generalizing from the data [10].

4.3 Classification

Classification refers to assigning predefined labels or categories to input data based on the patterns and features learned by a model during training. It is a fundamental application with the goal of mapping input instances to discrete classes. The output layer of NN often employs activation functions like softmax to generate probability distributions over classes. This task's most commonly used loss function is cross entropy [10].

4.4 Extraction of Representations

Representation extraction transforms raw input data into a more suitable format for the learning algorithms to understand and process effectively. The goal is to capture the underlying structure and patterns present in the data to facilitate learning and generalization to new, unseen data [7, 13].

In deep learning, extracting representations has become more automated and hierarchical. Deep learning algorithms learn to extract increasingly abstract data representations through multiple layers of non-linear transformations. These hierarchical representations capture complex patterns in the data, making them suitable for tasks such as classification, prediction, and data analysis [13].

4.4.1 Supervised Learning

Supervised learning is a type of machine learning where algorithms are trained on a dataset containing both input features and corresponding output labels or targets. In this paradigm, the algorithm learns to map input data to the correct output based on the provided examples [7].

The term „supervised“ reflects the idea that the algorithm's learning process is guided by a teacher or instructor, represented by the labeled data. Unlike unsupervised learning, where the algorithm must derive patterns and structures from unlabeled data alone, supervised learning algorithms benefit from having this labeled guidance to learn the underlying relationships between the input features and the output labels [7].

4.4.2 Self-Supervised Learning

Unlike traditional supervised learning, which relies on labeled data for training, Self-supervised learning (SSL) thrives on utilizing vast amounts of unlabeled data. At its core,

SSL defines a unique approach where models are trained to generate descriptive and intelligible representations through pretext tasks based on unlabeled inputs [1].

Contrastive Loss and Contrastive Learning

Contrastive learning is a machine learning paradigm that leverages contrastive loss as a fundamental learning objective. The primary goal of contrastive learning is to train a model to differentiate between similar and dissimilar instances in the input data space [1].

Contrastive loss is applied to train a neural network to predict whether or not two inputs are from the same class. This is achieved by making the embeddings of similar inputs close to each other in the feature space while embeddings of dissimilar inputs are pushed further apart. Since the data used for training is without labels, positive pairs are formed by creating variants of a single input using known semantic preserving transformations. These positive pairs are contrasted with negative examples to reinforce the learning objective [1].

The margin parameter (often denoted as 'm') plays a crucial role in contrastive loss, imposing a constraint that the distance between examples from different classes should be larger than the margin 'm.' This ensures a clear distinction between positive and negative pairs. The Triplet loss, another related concept, shares a similar spirit but is composed of triplets—consisting of a query, a positive example, and a negative example. One of the most used methods is SimCLR [1].

4.5 Collaborative Learning

The Google AI research team introduced collaborative learning (CL) in 2016. It facilitates training a unified machine learning model by multiple parties, each leveraging their local data samples without the need for data exchange. Unlike Distributed Learning (DL), where a single joint model is trained across multiple devices without privacy concerns, CL prioritizes data privacy by keeping local datasets confidential. This approach allows for enhancing a global model managed by a centralized entity, utilizing insights collected from diverse datasets without compromising individual data privacy [8].

In practice, collaborative learning operates within a framework where multiple computing parties contribute their local datasets and models to refine a global model orchestrated by a centralized entity. Each computing party retains control over its local dataset, typically tailored to its specific environment (e.g., IoT device data or network traffic). Through collaborative efforts, CL aims to iteratively improve participating parties' global and local models, thus advancing machine learning capabilities while safeguarding data privacy in an increasingly interconnected world [8].

Chapter 5

Proposal

This thesis concentrates on using ipfixprobe made by CESNET, which is deployed and, as such, testable on fast networks of CESNET with multiple 100Gbit/s peering and transit links. Because of the architecture of this flow exporter, it is possible to easily extend the functionality by creating a new plugin [2].

Overall, the concept, shown in Figure 5.1, aims to use PyTorch libraries and APIs to implement neural networks in ipfixprobe. The probe is implemented in C++. However, users are most accustomed to Python. This language and its related libraries are also considered for its popularity in machine learning and neural networks research, and the scientific group in CESNET is most comfortable working with it. On account of this, the concept aims to connect Python neural network models with routines written in C++.

Considering these requirements, this thesis aims to create a plugin that enables machine learning in ipfixprobe using PyTorch-like technologies. It would utilize PyTorch C++ frontend API and let users define a model in Python. It would also use another PyTorch API — TorchScript — to export and load the model into it. The user would define a model in Python, through a parameter, let the plugin know the location of the defined model, and the probe will start in inference 5.2 or training 5.3 mode.

5.1 PyTorch Model

PyTorch is a popular open-source machine learning library used primarily for developing deep learning models. It offers a flexible and dynamic approach to building neural networks, making it particularly favored by researchers and practitioners [23].

This proposal aims to create a model in PyTorch and use it inside the C++ plugin of ipfixprobe using TorchScript and C++ frontend API.

5.1.1 Functionality

PyTorch provides many functionalities for creating neural network models and multiple APIs for integrating these models into different languages like Java or C++ [23].

Tensor Operations

At the core of PyTorch lies its tensor computation library, similar to NumPy but bolstered with GPU acceleration capabilities. Tensors, multidimensional arrays, serve as the foundational data structures for constructing neural networks. PyTorch tensors are highly

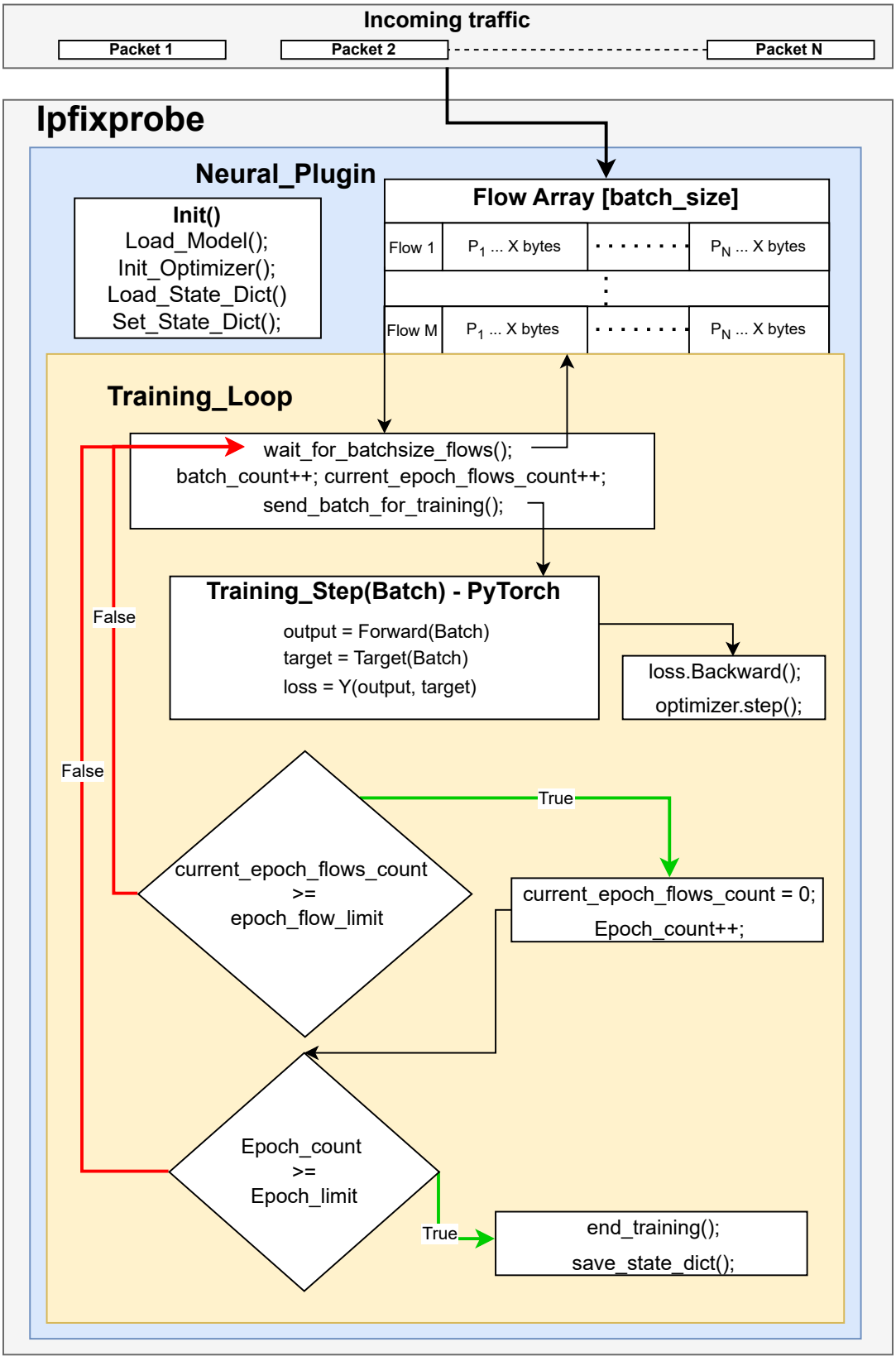

Figure 5.1: Proposed architecture.

efficient and can be easily manipulated through different functions for various mathematical operations [23].

Autograd

Autograd is a reverse automatic differentiation engine, a defining feature of PyTorch, that meticulously records a computational graph as operations are executed. This mechanism enables the smooth computation of gradients, which is crucial for training neural networks via backpropagation. PyTorch autonomously derives gradients with input variables by dynamically tracking operations on tensors, thereby facilitating the optimization process [23].

Internally, autograd represents this graph as a network of function objects akin to expressions, each capturing a specific operation. During the forward pass, as computations unfold, autograd concurrently constructs this graph while performing requested operations [23].

Torch.nn Module

Building neural network architecture is provided by torch.nn class. It contains multiple base building blocks like linear, recurrent, or normalization layers through subclassing nn.Module, users can define custom layers, models, and operations. This encapsulation organizes model parameters and facilitates forward propagation and parameter optimization [23].

Forward Propagation

The process of propagating input data through the neural network to yield predictions is realized through PyTorch's forward method. During this phase, input tensors traverse the network's layers, culminating in the computation of intermediate outputs. PyTorch's dynamic nature allows for seamless adaptation of network architectures to diverse input data [23].

Backward Propagation

PyTorch integrates backpropagation through its Autograd engine. Upon invocation of the backward method, gradients of the loss function concerning model parameters are computed recursively. Leveraging the chain rule, PyTorch efficiently updates model parameters, optimizing the network's performance [23].

Optimization Algorithms

PyTorch offers multiple optimization algorithms via the torch.optim class. From classic techniques like Stochastic Gradient Descent (SGD) to state-of-the-art methods like Adam and RMSprop, users can select and customize optimization strategies tailored to their specific applications. These algorithms iteratively refine model parameters to minimize the loss function [23].

5.1.2 Proof of Concept

Even though the approach is very similar to starting with neural networks in PyTorch, a combination of different APIs and languages is challenging. The routine for training must be split into two parts, one in the ipfixprobe implementation and one in the model itself,

which requires the definition of special functions in the model. Extra functions, except for forward(), must have implemented decorator @torch.jit.export so the TorchScript knows that they will be called from outside.

The model needs to have defined two methods, forward() and training_step(). These methods are used the same way as in classic PyTorch implementation of neural networks. An example of the PyTorch model can be seen below 5.1.

Both are defined in Python but are called through API from C++ source code. The Forward() method is responsible for defining the forward pass of the neural network. Method training_step() calculates loss over the current batch of incoming data. The loss is then backpropagated inside the C++ codebase using the backward() method during the training loop, which updates the neural network's parameters (weights and biases) based on the computed loss and optimizer used.

```
import torch
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(30, 1)

    def forward(self, x):
        return self.linear(x)

    @torch.jit.export
    def training_step(self, batch) -> torch.Tensor:
        # Forward pass
        output = self.forward(batch)

        # Custom training step logic
        targ = torch.mean(batch, dim=-1, keepdim=True)
        loss = torch.nn.functional.mse_loss(output, targ)

        return loss

model = Model()
# Export the model to TorchScript
scripted_model = torch.jit.script(model)
scripted_model.save("scripted_model.pth")
```

Listing 5.1: Example of PyTorch model serialization.

5.1.3 Model Export

The model is defined in Python and can use third-party libraries like PyTorch. To export the model, the user must install Python with PyTorch extensions, compile it, and execute it. Afterward, it can be exported into a new representation loadable into C++ code, TorchScript, and serialized in non-human readable form into a file using methods like a torch.jit.script(module) and torch.jit.save(script_module, FILEPATH.pt) [22].

Most commonly, the model does not need to be exported in its entirety, but its parameters, like weights of defined layers and bias, are not. This is in PyTorch achieved through state_dict, which represents the current state of the model [28].

Unfortunately, C++ frontend API does not have this export functionality, and as such, it will have to be implemented in a different way as multiple issues are raised [15, 17, 18, 19]. However, based on the PyTorch forums, there is a workaround to save parameters as tensor and export it in the same way as model, torch::save() [16].

TorchScript

TorchScript is a tool to create serializable and optimizable models defined in PyTorch. This makes it possible to load the model into a process without any Python dependency. As the use of Python programs in a production environment might be disadvantageous and slow, this is a way to provide a less performance-burdensome solution with the advantage of PyTorch familiar processes and tooling [22].

5.1.4 Model Import

The PyTorch model is imported into the C++ frontend API by loading the serialized model from TorchScript. The model imported this way contains default parameters, which is fine for the first training of the model. However, if users want to achieve collaborative training 4.5 or want to continue training on the already trained model, the state of the previous model (state_dict) must also be imported. To enable this functionality, the parameters are to be exported into the file as tensor and imported in the same manner as the model [16, 22].

5.1.5 Pros & Cons

Using TorchScript to load PyTorch models into a C++ frontend comes with several advantages and drawbacks. On the positive side, one of the major pros is its ease of use. TorchScript simplifies the process of integrating PyTorch models into C++ applications, making it accessible even for developers who may not be experts in both environments.

Additionally, the considerable community support behind Torch and its extensive documentation further contribute to the module's appeal. Leveraging a large community can be advantageous, as it means a wealth of resources, including tutorials, forums, and pre-existing solutions, making troubleshooting and development more straightforward.

Another advantage is the module's adaptability, as it is not constrained to a single model. This flexibility allows for easy substitution of models without significant code changes, enhancing the module's versatility and scalability.

However, there are some drawbacks to consider. One notable con is the overhead associated with creating a model using TorchScript. The process involves defining special functions and multiple steps, including creating the PyTorch model and exporting it to TorchScript. This additional complexity may increase the chances of errors during the model integration process.

Furthermore, there is a trade-off in terms of visibility into the model. While the module allows for seamless integration, the abstraction introduced by TorchScript may result in reduced transparency into the model's inner workings.

5.2 Model Deployment and Inference

Ipfixprobe is written in C++, and as such, the model cannot be written in PyTorch and must be loaded into it in different ways. Earlier, we discussed the deserialization of the model into TorchScript; now, it can be loaded using a similar function, torch.jit.load(FILEPATH).

The model shall be loaded when initializing the plugin so that the ipfixprobe works with the same model for the whole run. The data gained through capturing the traffic are sanitized in methods post_create() and post_update(), so the model is not working with IP addresses, for example. When the flow ends, pre_export() is called, in which the bytes gained are converted into tensors, and method forward(), defined in the PyTorch model, is called for inference. The result of this operation is again turned into a tensor. Afterward, the chosen data based on the record template are exported.

5.3 Training Alternatives

There exist several possibilities for how to train the model. The considered ones will be explained below, along with their pros and cons. The final proposed method is training the model inside ipfixprobe 5.3.4 as it looks most versatile and usable.

5.3.1 Trained Model

The easiest method would be to have an already trained model, which would then be used only for inference of incoming traffic. However, this raises the question of what data the model should be trained on. It is best to train the model on expected traffic on the network.

5.3.2 Training outside of ipfixprobe

This is a scenario where the model still needs to be trained. The model is loaded and inference mode is started on incoming data, but it also dumps incoming traffic to different locations where a new iteration of the current model is created and trained on this data. After a certain amount of data is dumped or time, the model is reloaded with the newly trained, which dumps the data, too, and the loop starts again 5.2. After a certain threshold (or never), the model would be reloaded for the last time, and it would operate only in inference mode.

This solution would require a lot of space to store incoming traffic. The model would have to implement new functions like GetNewBatch(), which would acquire the data from storage. Also, each reload of the model could cause some downtime and would undoubtedly be demanding performance-wise.

5.3.3 Training inside ipfixprobe

This approach lets the traffic be inference and the neural network be trained in real-time without extra storage for capturing the traffic. At the start, the inference mode of the plugin would be disabled, and the NN would be only iteratively trained based on incoming data. After X iterations, the inference mode would start, but NN could still keep training and updating without any further disadvantages in performance.

There would be no need for extra storage; the NN could start fresh without any prior training and would train depending on the exact traffic it is used on.

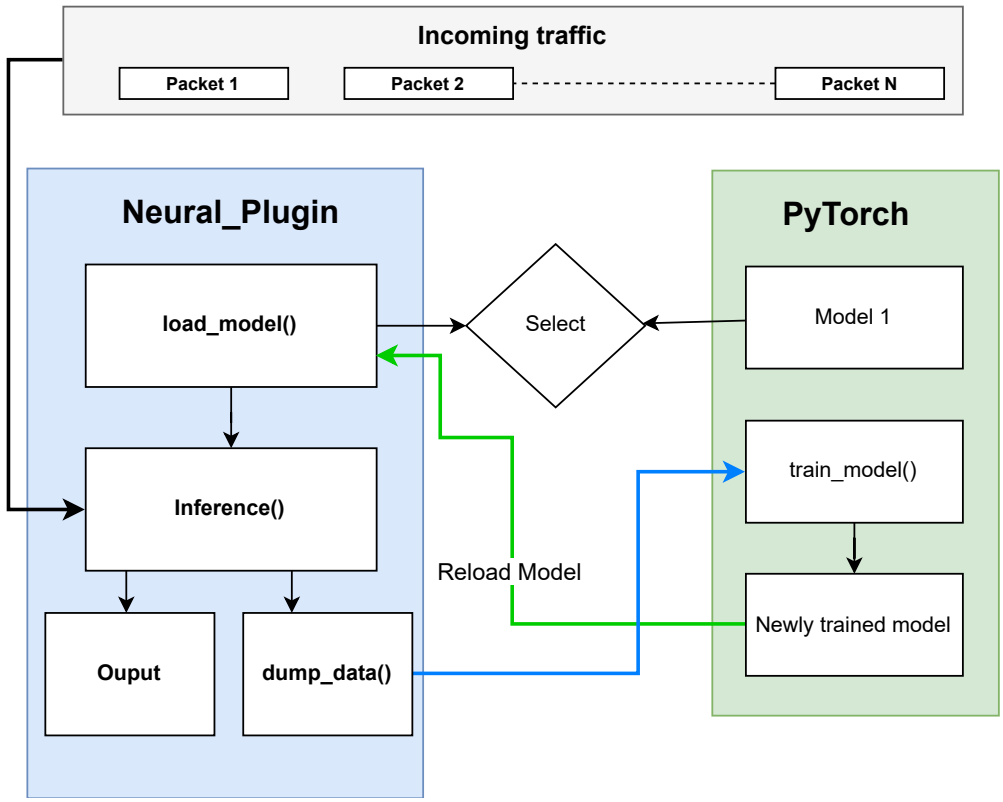

Figure 5.2: Scenario with dumping data to train in PyTorch.

5.3.4 Training and Inference in Separate Runs of Ipfixprobe

The approach chosen in this thesis is a combination of the possibilities above. The data on which the model is trained are provided by the ipfixprobe. There is no labeling of the data. The model is trained using self-supervised learning. However, this can be easily changed inside training_step() in the PyTorch model. In the first run of the ipfixprobe, the option for training is used with the chosen batch size, epoch size, number of epochs, and learning rate. After the plugin reaches the limit of the epochs, the model parameters are exported similarly to state_dict of PyTorch, and the plugin ceases its work.

In the next run of the ipfixprobe plugin, the options for inference, location of model, and parameters are set. First, the model is loaded with parameters based on the option, and the train() method is omitted. After receiving enough flows and packets, the inference is run using the forward() method of the model and the output is printed on standard output.

In this approach, the plugin must be run twice. However, it brings the possibility to train the model multiple times, even on different devices 4.5. It does not have any additional performance bumps and can be utilized in a much simpler manner.

Chapter 6

Implementation

The main goal of this thesis is to create a functional proof-of-concept, letting users define neural network models in Python and loading it into the C++ plugin of ipfixprobe. As mentioned before, ipfixprobe is written in C++, which introduces better performance and platform independence at the cost of new challenges when implementing PyTorch-like routines. It is programmed using standard gnu++11. As such, the first thing to do was to add libraries supporting PyTorch-related work, like libtorch, libtorch_cpu, libc10, and tomlc99. These libraries required the standard to be updated to gnu++17.

6.1 Environment Preparation

As mentioned before, ipfixprobe is written in C++ and, as such, must be compiled into some executable binary with all of the needed dependencies. This can be achieved by downloading the repository and following these commands:

```
cd ipfixprobe
autoreconf -i
./configure
make
sudo make install
```

After installation is completed, it is possible to use ipfixprobe with the chosen plugins.

6.1.1 Libraries

To be able to use the C++ frontend API of PyTorch in ipfixprobe, its libraries are required to be added into the ipfixprobe. The official API distribution offers the libraries in the format of .so for different purposes, like CPU or GPU usage. This example utilizes only CPU-based operations [21].

Unfortunately, as ipfixprobe runs on the C++ standard C++11, it needed to be updated because of these libraries to C++14 or newer [20]. As such, updating the ipfixprobe building process was required.

Building Process

Ipfixprobe uses Autoconf and Automake to manage the building process. They are integral components of the Autotools suite, designed to streamline the configuration and build

process of software projects, particularly those adhering to GNU standards. Automake facilitates the automatic generation of Makefile.ins from Makefile.am files, simplifying the management of Makefiles by defining variables and occasionally rules. These generated Makefile.ins files adhere to the intricate GNU Makefile standards, alleviating the maintenance burden from individual GNU maintainers and shifting it to Automake maintainers. Each directory typically contains one Makefile.am file, processed by Automake to produce a Makefile.in [12].

While Automake imposes certain constraints on projects, such as requiring Autoconf, it ensures compliance with GNU standards without necessitating Perl for build processes. Autoconf, another component of Autotools, works with Automake to automate the configuration process. It generates configuration scripts based on configure.ac files, which are typically included alongside Makefile.in and Makefile.am files in software packages [12].

The standard was updated to gnu++17, and libraries were added like this:

```
1  AC_ARG_WITH([libtorch],
2  AS_HELP_STRING([--with-libtorch],
3        [Path to LibTorch installation directory]))
4  if test "x$with_libtorch" != "xno"; then
5    AC_MSG_CHECKING([for LibTorch])
6    if test -f "$with_libtorch/lib/libtorch.so"; then
7      AC_MSG_RESULT([yes])
8
9      CPPFLAGS="$CPPFLAGS -I $with_libtorch/include
10     -I $with_libtorch/include/torch/csrc/api/include
11     -I thirdparty -I thirdparty/tomlc99 -D_GLIBCXX_USE_CXX11_ABI=0 "
12
13     LIBS="$LIBS -Wl,-rpath, $with_libtorch/lib
14     -Wl,--no-as-needed, "$with_libtorch/lib/libtorch_cpu.so"
15     -Wl,--no-as-needed, "$with_libtorch/lib/libtorch.so"
16     -Wl,--as-needed $with_libtorch/lib/libc10.so"
17   else
18     AC_MSG_ERROR([LibTorch not found in the specified directory])
19   fi
20 else
21   AC_MSG_ERROR([LibTorch is required.
22        Use --with-libtorch option to specify its location.])
23 fi
```

Listing 6.1: Add torch libraries to build.

6.2 Neural Network Plugin

To enable the use of Python-like routines in ipfixprobe, it was necessary to divide the work into two codebases: create a plugin inside ipfixprobe and define a PyTorch model inside a Python file. This enables users to work with familiar PyTorch tooling and use it inside the C++ code of ipfixprobe. This also aims to let users change only PyTorch models for different tasks without the need to update the code of the plugin or even compile the ipfixprobe again.

6.2.1 Python Model

For easier use by ipfixprobe users, the neural network model is written in Python using PyTorch for its neural network operations and TorchScript for the model export. The model used for the final experiments looks like this:

The model looks very similar to standard models created with PyTorch. Although there are some differences shown by commented code, The initialization of the model defines a single layer of the neural network of size: $packets\_count \times content\_size$. This layer is then used inside the forward() method.

The model defines a method called training_step(self, batch), which simulates an iteration of standard training in PyTorch. It is necessary as this method is called in the training loop inside of the newly created plugin. This method returns calculated loss based on the incoming batch using mean squared error in comparison with output from the torch function for calculating the mean.

Finally, the model is exported using TorchScript API torch.jit.script(MODEL) and save(PATH) methods into the serialized format and ready to be loaded into ipfixprobe.

6.2.2 Ipfixprobe Plugin

Neural plugin represents the C++ part of the codebase. The main features are loading the model, creating a bridge between the PyTorch model and ipfixprobe, and processing data.

It contains fields defining how long the training should be, with what learning rate, how many flows create a batch, how many packets from the flow, or how much of the payload the model should process. Methods used for the work with model are 6.1:

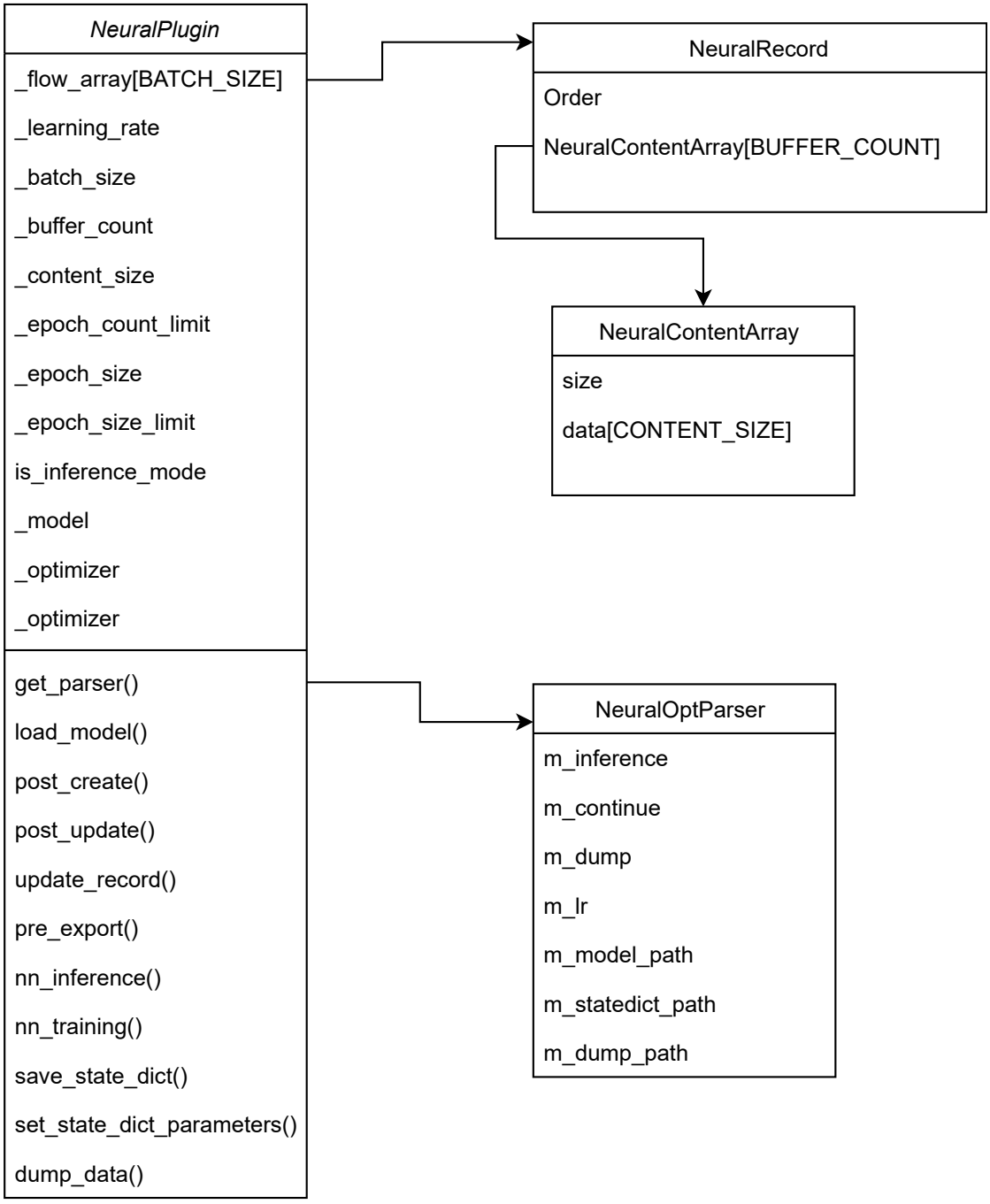

Figure 6.1: Class diagram of Neural plugin.

Options

The option parser is defined in the header file of the neural plugin and offers multiple options for customizing the plugin.

List of options:

- i - Switch for inference/training.
- m - File path to the scripted neural network model.

- s - File path to state_dict.
- c - Continue training with selected state_dict enabling the collaborative training 4.5.
- d - Dumps tensors of flow into multiple TorchScript files named based on the epoch.
- lr - Overwrite of default learning rate.

Initialization of the Model

In the initialization phase, the default values for the number of epochs, size of the epoch, learning rate, number of flows per batch, number of packets per flow, and size of packet taken into account are set, and the array for containing flow records is created. Afterward, options with which the plugin was started are parsed and applied, changing the plugin's behavior. Then, the PyTorch model is loaded through torch::load(), and if the 's' option was specified, the models' parameters are set based on the state_dict. If the state_dict option is not specified and the model is used for training, the state_dict is saved into the default location.

In the next step, the optimizer is initialized; in the example, it is Stochastic Gradient Descent, and the model parameters and learning rate are pushed to it.

State_dict

Methods like MODEL.save_state_dict() from PyTorch are not implemented in C++ frontend API; as such, it was necessary to create something similar. The newly implemented method respects the naming conventions of PyTorch so that users do not have to think about the functionality of the method.

The new method save_state_dict() takes the named parameters of the model, turns the value tensors into a single std::vector of tensors, and saves it into a single file specified by the 's' option using the torch::save() method. The contradictions to this method are load_state_dict() and set_state_dict_parameters().

Load_state_dict() loads the TorchScript file in the same manner as when loading the model, using the torch::load() method, and returns the loaded parameters as std::vector of tensors. This vector is then taken by set_state_dict_parameters(), which iterates through the named parameters of the model and sets their value to the values inside the vector.

This should simulate missing methods from PyTorch API MODEL.state_dict() and MODEL.load_state_dict() in C++ frontend API. There is one drawback: the parameters must be in the same order (layers of the neural model) when set as when they were exported, which is not a problem when the same model is used. However, if some extension of this plugin would be implemented in the future, it is essential to consider this.

Packet Processing

When a new packet arrives, ipfixprobe decides if it is part of the already seen flow or if it starts a new one. If it starts a new flow, the post_create() method is called, which creates a new NeuralRecord for holding information about this flow. If the flow had already been seen, method post_update() is called, which gets the already created NeuralRecord. Afterward, update_record() is invoked.

The update_record() method is the primary function for handling the packets of flows. Initially, it checks if it should skip this packet based on the number of packets already

contained inside NeuralRecord and the number of packets per flow allowed or if the training phase has already ended. Then, the packet's payload is trimmed to the value defined in the initialization phase by CONTENT_SIZE, meaning the maximum size of a single packet. The packet is added to the array of packets of the current flow. For copying the data of the packet and not updating the packet itself, the method std::copy() is used.

Communication with Neural Model

All the communication between the neural plugin and neural network model happens inside the method pre_export(), which is invoked on the current flow when it ends. If the model is already trained and another flow ends, nothing happens, and this traffic is skipped. Otherwise, the size of the flow array is checked to see if the batch is ready to be sent.

If the plugin is in inference mode, the method nn_inference() is called. It takes the current flow, creates a tensor from its NeuralRecord, and pushes it into the neural model through the method forward(), which returns the estimated value from the model.

In the training mode, the training loop starts, counting batches in the epochs and iteratively increasing epochs until the limit for training is hit. In the training loop, it calls method nn_training(). It creates the whole flow array as a batch and creates a tensor with normalized data from it in dimensions of batch size, number of packets, and size of the payload. Each byte is normalized by dividing it by its maximal possible value (255).

Then, the gradients are reset to none, and the tensor is pushed into the neural network model, which returns the calculated loss. This loss is then backpropagated, and the optimizer makes a step, ending the training loop's single iteration.

6.3 Workflow

Ipfixprobe receives numerous incoming packets, collecting them into flows. The plugin takes 30 packets per flow or less if the flow ends earlier. From each packet, 100 bytes are saved into an array. These bytes are then normalized for easier processing by NN. Each byte is divided by a maximum value of byte, 255. After having 30 normalized packets per flow with a total of 16 flows (batch size), they are pushed into the training routine or inference of the earlier defined neural model. This scenario is shown in the Figure 6.2.

In the training routine, the loss is calculated based on the chosen algorithm; for proof-of-concept, the mean squared error (mse_loss) defined in PyTorch is used. After all of the training epochs are over, the models' parameters are saved as the state_dict into the TorchScript file.

In inference, the data vectors are sent into the forward() method, which calculates the output based on the neural network specifications. Afterward, the data incoming from the neural network are processed and exported using the export() method of the ipfixprobe plugin.

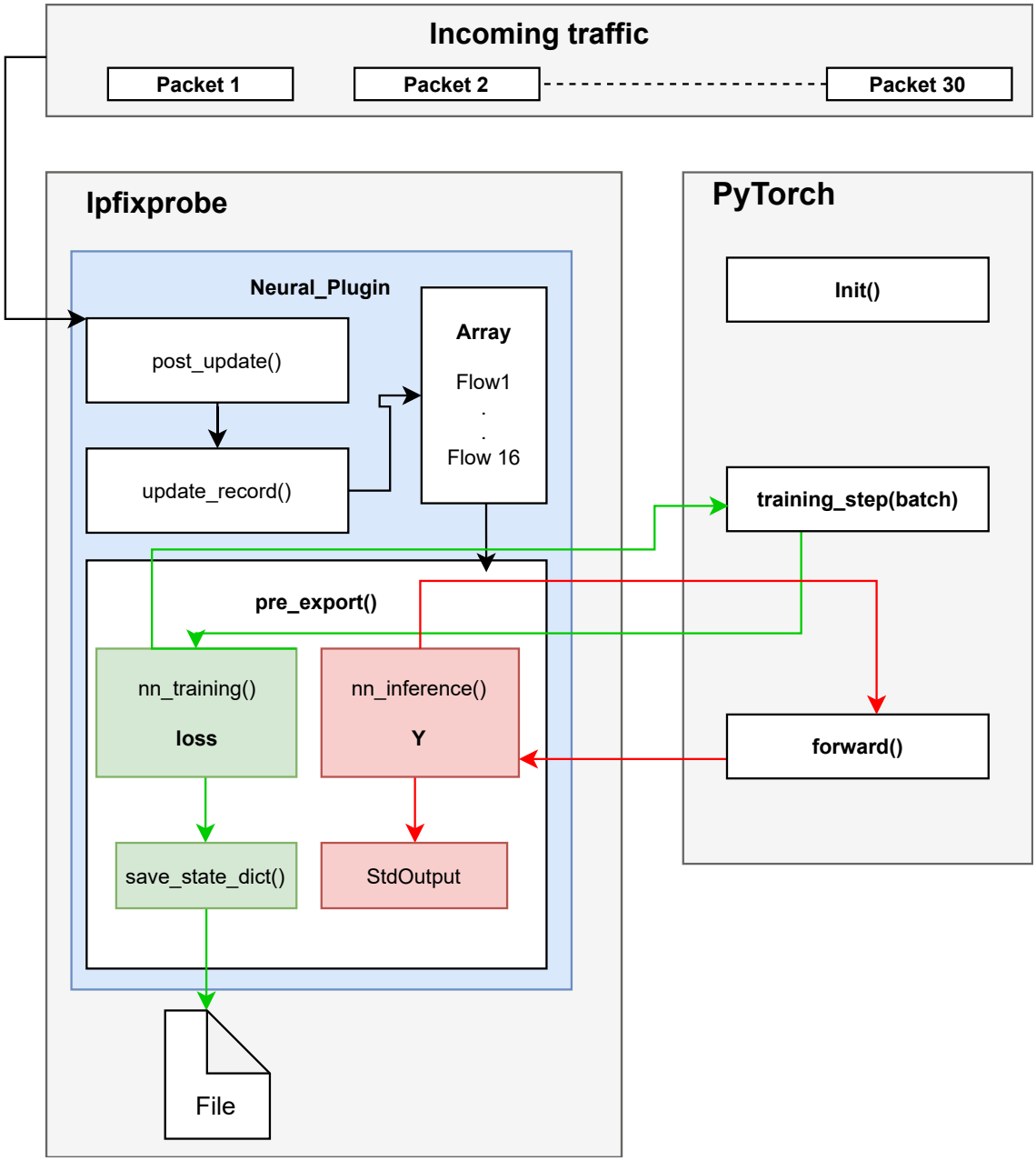

Figure 6.2: Simplified workflow of neural plugin.

6.4 Execution of the Plugin

The neural plugin extends Ipfixprobe, and as such, the plugin 6.2.2 is started by specifying specific options when starting ipfixprobe. In plugins, the (-p) option specifies the neural plugin as a string consisting of its name and options, each separated by a semicolon. The order of the options does not matter.

```
ipfixprobe -i "pcap;file=PCAP"
    -p "neural;m=scripted_model.pth;s=state_dict_values.pt;c;
        lr=0.1;d=FOLDER"
    -o "unirec;i=f:$out.trap;p=(neural)"
```

Listing 6.2: Starting neural plugin.

In the example above, the neural plugin is executed with model scripted_model.pth, state_dict file state_dict_values.pt, with the option to continue learning based on the loaded state_dict, the learning rate of 0.1, and the path to the folder for dumping tensors of the network traffic. For more options, please refer to the optionssection 6.2.2.

Chapter 7

Evaluation

In this chapter, the performance and effectiveness of the developed plugin, which utilizes the C++ frontend API of PyTorch and the PyTorch neural model scripted into TorchScript for ipfixprobe, are evaluated. Various datasets and performance metrics are used to assess its functionality, training process, and inference speed.

In the first section, what is being evaluated, the testing data used, and the baseline for testing are precisely established. Subsequently, the focus shifts to the training routine, particularly its loss convergence and the duration required to execute a single training routine. Following this, the processing time of the plugin is compared across different numbers of parameters. Finally, the performance difference between the neural plugin and the pstats plugin is assessed using a statistical approach.

7.1 Functionality Measurement

To evaluate the functionality of the newly created plugin, vital performance metrics were employed to assess its efficacy in handling network traffic analysis tasks. Primarily, emphasis was placed on measuring the processed flows per second, providing a quantitative measure of the plugin's efficiency in real-time data processing. Additionally, the loss incurred during the training phase of the neural network model was monitored. Lower loss values are indicative of better convergence and learning from the provided data, showcasing the plugin's effectiveness in capturing underlying patterns within the network traffic.

When conducting our experiments, an artificial task was adopted as the focal point of investigation. By selecting an artificial task, a controlled environment was created to facilitate a detailed analysis of specific aspects. This approach allowed the nuances of various techniques, architectures, and methodologies to be meticulously studied without the confounding factors present in real-world datasets.

It is noteworthy that while our experiments primarily focused on an artificial task, the architecture and design of the neural model within the plugin enable seamless interchangeability with various learning algorithms and neural models, broadening its utility beyond our specific investigation.

7.1.1 Datasets

For the basic functionality of connecting the PyTorch model with the ipfixprobe plugin, I chose firstly to use random tensors and calculate the mean. Afterward, the performance tests were performed on a 580 MB pcap file of network traffic from the CESNET network.

Random Tensor

The random data are used only to test the convergence of neural networks. The tensor is created inside the ipfixprobe neural plugin with the exact dimensions that the tensor of actual network traffic would have. The random tensor was created via the torch::rand() method, which creates a tensor with values between 0 and 1 in normal distribution [23].

Network Traffic

The pcap file from network traffic should simulate real-world network data and be used for both tests, loss convergence, performance tests, and comparison against the pstats plugin of ipfixprobe. It contained 11 620 flows with 1 067 579 packets, a total of 561 874 569 bytes.

7.1.2 Baseline

The evaluation considered primarily small neural network models, due to the capacity limitations of the ipfixprobe. To establish a baseline for performance evaluation, the plugin was tested using the random tensor dataset with fixed dimensions (64x30x100). All the tests performed in training mode used a learning rate of 0.01. The training was conducted on 5 epochs of size 1024 flows, each flow consisting of 30 packets. The considered payload size of packets was changed throughout different tests to ascertain the plugin's ability to handle different tensor sizes.

The chosen neural model consists of a single linear layer, meaning only a single operation (matrix multiplication) is performed. As the payload of the incoming packets increases, the size of the linear layer and the size of the neural model are affected by increasing the number of model parameters.

7.1.3 Training Measurements

The loss convergence during training is shown in Figure 7.1, depicting the loss value per batch. In a single training run, the loss decreases to about $3.62E-06$ with random values and to $2.81E-06$ with the network traffic by the last batch.

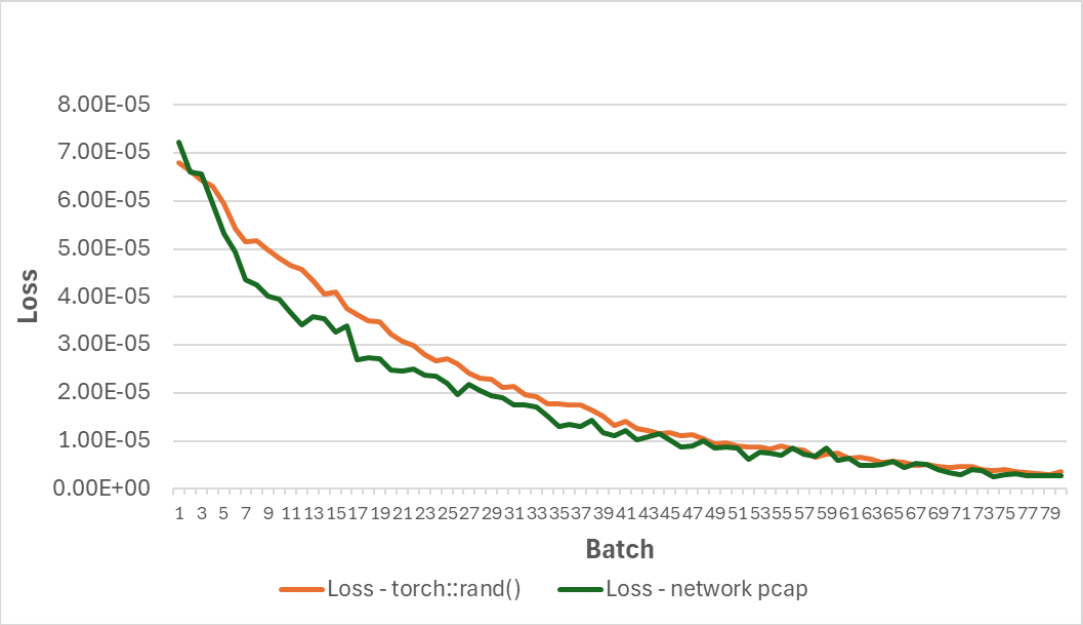

Figure 7.1: Calculated loss throughout a single run of training.

The training over network data took about 22.466 seconds, resulting in processing 227.899 flows (3.6 batches) per second. In reality, only 153 600 (5x1024x30) out of 1 067 579 packets (14.3876940%) were processed during training as the training ended in the fifth epoch, and all other flows were skipped. The training took about 3.687 seconds with random tensors, meaning 21.69 batches were processed per second. The comparison can be seen below 7.2.

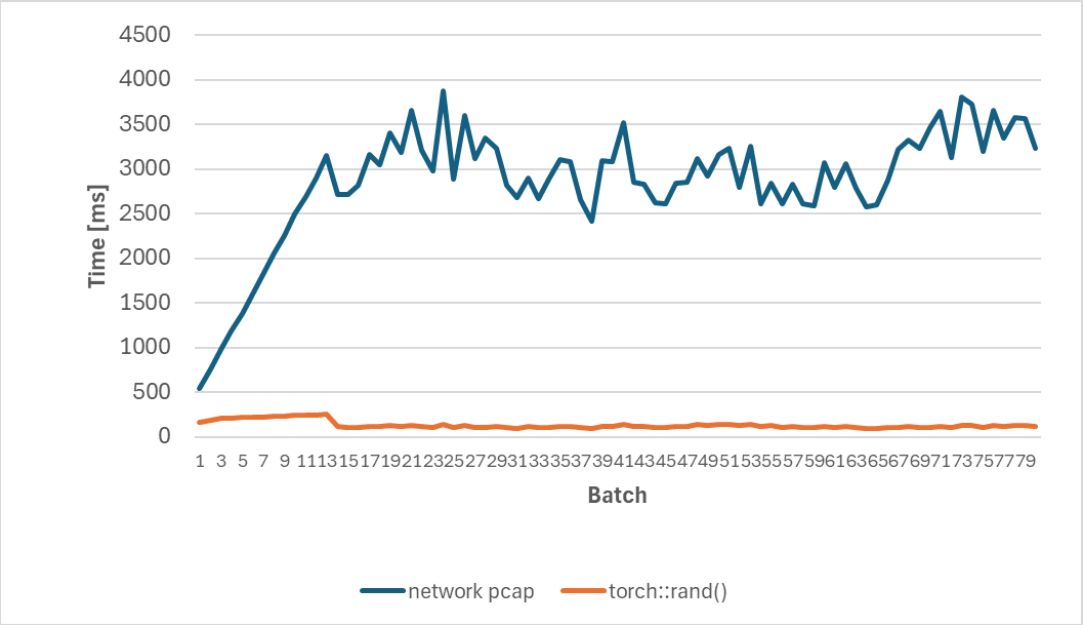

Figure 7.2: Processing time of batches in a single run of training.

The big difference in the time it took to train is most likely caused by a method for processing the packets called create_tensor_based_on_flow_array(), which takes incoming packets and pushes them into flow records.

7.1.4 Inference Measurements

In the first experiment to measure inference of neural plugin, tests were parameterized with the same baseline 7.1.2 random tensors against pcap with network traffic. In Figure 7.3 is shown the performance problem as in 7.2 with method create_tensor_based_on_flow_array(). This overhead is even more visible as the neural model does not receive data in batches, but each flow is sent as a separate tensor.

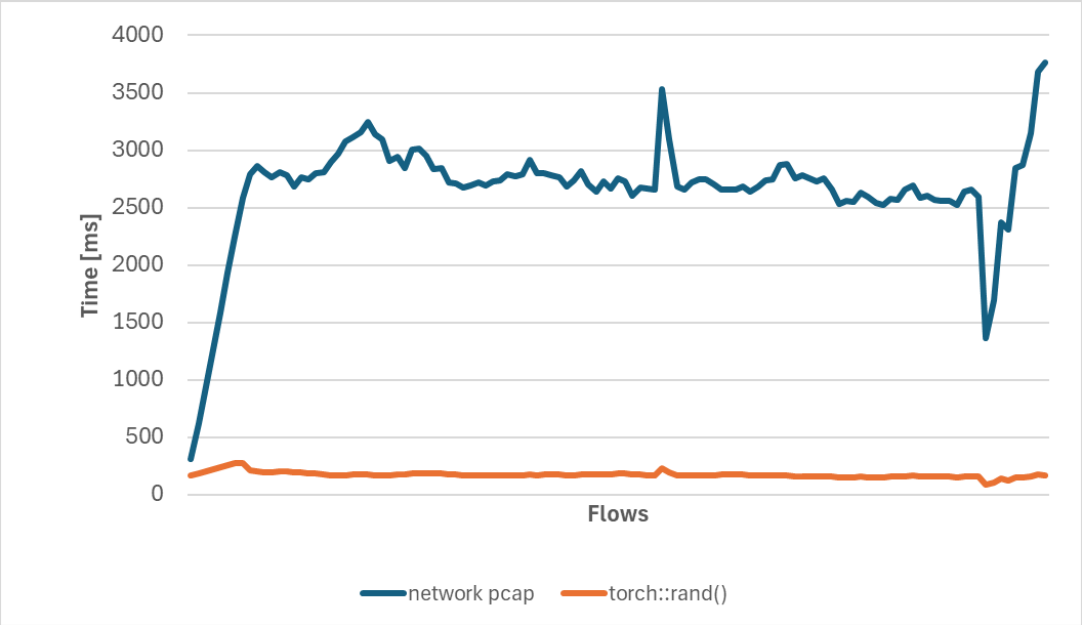

Figure 7.3: Comparison in inference on 30x100 tensors.

Next experiment consisted of increasing size of the inputs from 30 to 7500 and measuring how long it takes to process them. In Figure 7.4 is depicted processing time in the create_tensor_based_on_flow_array(), from 1.116 seconds for 30 bytes to 59.283 second for 7500 bytes.

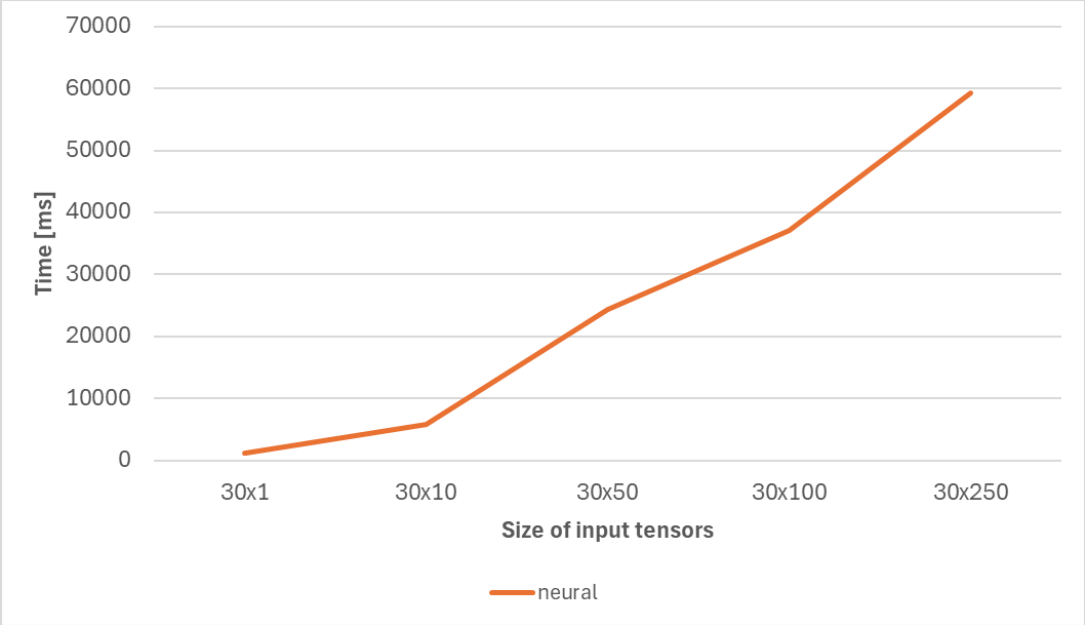

Figure 7.4: Processing time of neural plugin based on size of input tensors.

The following experiments preserved the baseline 7.1.2, but the payload size is changing. This causes the model's parameters to change throughout the experiments to show how the model's size affects the performance. The model uses a single linear layer, and its size in the experiments is based only on the number of packets and the number of bytes considered.

The charts below 7.5 show how the processing time of flows in inference grows as the number of model parameters increases from 30 to 7500 parameters.

Figure 7.5: Processing time of flows in inference dependent on the number of parameters.

7.1.5 Comparison to pstats

The pstats plugin of ipfixprobe is designed to provide comprehensive statistical analysis of network traffic flows. It collects data from network packet headers and generates flow-level statistics [4].

The pstats plugin takes up to 30 packets and analyses their headers. For comparison, the plugin was executed with options shown below on the same dataset as baseline 7.1.2. The processing times of multiple runs were used to calculate the mean, which was then used as single pstats processing time, as it should be constant, equal to 3.227 seconds.

```
./ipfixprobe -i "pcap;file=PCAP" -p "pstats"
       -o "unirec;i=f:$out.trap;p=(pstats)"
```

Listing 7.1: Example of executing pstats plugin fo ipfixprobe.

The chart below depicts the comparison of the pstats and the neural plugin. At low number of parameters, the processing time is very similar to pstats. In an experiment where the neural model had 30 parameters, the times were almost identical; with 7500 parameters, the time increased by 53% to 4.892 seconds.

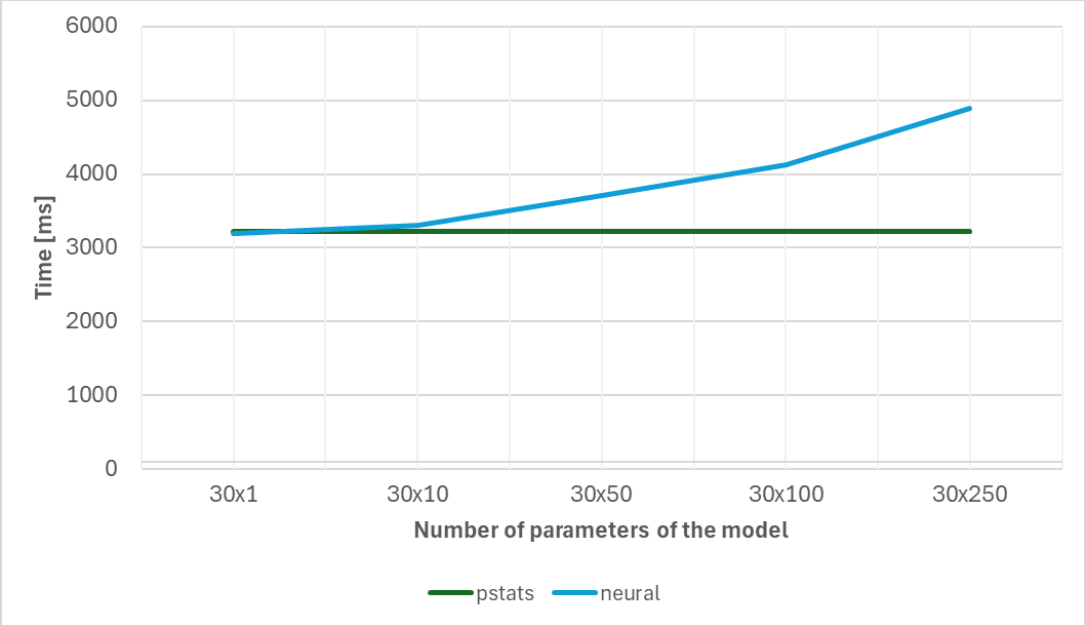

Figure 7.6: Processing time of neural plugin vs pstats based on a number of parameters.

7.2 Evaluation Results

In conclusion, the evaluation of the plugin demonstrated its effectiveness in handling both synthetic and real-world datasets. The training of the neural model was successful, the loss was converging, and even though only a single training routine was observed, the training can be enhanced by using the 'c' option of the neural plugin 6.2.2 to continue training with already trained parameters.

The inference also worked; however, the performance loss for preprocessing actual network traffic could be improved. Although we found a weak spot in data preprocessing, the proof-of-concept handles communication with neural model with only slight decrease of performance.

Chapter 8

Conclusion

This thesis focused on developing a new plugin into ipfixprobe created by CESNET. The plugin implements a wrapper around a neural model defined in PyTorch and the C++ codebase of ipfixprobe.

The proposed ipfixprobe plugin is implemented using different APIs provided by PyTorch and integrated into the C++ code of ipfixprobe. This enables monitoring flow-based network traffic using neural networks and, as such, detects complex patterns and anomalies in network traffic, providing a more intelligent and adaptive approach to network security.

The proof-of-concept utilizes an artificial task during training, requiring no additional external training data prior to deployment. However, this is solely dependent on the current routine defined in the PyTorch model, which can be easily replaced with different routines. It is possible to use previously trained state_dict (even from different devices if the model was the same) and achieve continuous or collaborative training 4.5.

At this phase, the plugin considers the first 30 incoming packets with 100 bytes from each packet per flow and uses 64 flows per batch. The training runs for 5 epochs of 1024 flows. This should provide enough data for training and comparison of results with different imported models.

The plugin was tested on randomly generated data as well as a network traffic dataset. In the evaluation, the training and inference processes are observed. In the training evaluation, the chapter focused mostly on the loss convergence and the model's ability to learn from the data. In inference, the focus switched to the analysis of the time the plugin and neural model need to process the traffic. In inference, the proof-of-concept suffered in performance because of the method used for packet processing. The processing time increased significantly by increasing the amount of the data to be stored and preprocessed by the plugin. This does not make the plugin unusable, but improvement shall be considered in future development.

Depending on further research and community interest, future development may focus on performance improvements, interchanging neural model capabilities, and easing the demand for users of this plugin. It was considered to add options to define epochs, batches, optimizers, and other parameters inside the PyTorch model and retrieve them as attributes of the model on the side of the plugin. Relieving the users from operating inside the C++ codebase and focusing only on improving the imported PyTorch model. The architecture of the plugin enables switching neural models and, as such, provides opportunities to test different models on the actual network traffic from single or multiple devices.

Bibliography

[1] Balestriero, R., Ibrahim, M., Sobal, V., Morcos, A., Shekhar, S. et al. *A Cookbook of Self-Supervised Learning*. 2023.

[2] Benes, T., Pesek, J. and Cejka, T. Look at my Network: An Insight into the ISP Backbone Traffic. In: *2023 19th International Conference on Network and Service Management (CNSM)*. 2023, p. 1–7. DOI: 10.23919/CNSM59352.2023.10327823.

[3] Buduma, N., Buduma, N. and Papa, J. *Fundamentals of deep learning*. „ O'Reilly Media, Inc.“, 2022.

[4] CESNET. *Ipfixprobe*. 2024. Accessed: 2024-01-20. Available at: https://github.com/CESNET/ipfixprobe.

[5] Claise, B., Quittek, J., Meyer, J., Bryant, S. and Aitken, P. *Information Model for IP Flow Information Export* [RFC 5102]. RFC Editor, january 2008. DOI: 10.17487/RFC5102. Available at: https://www.rfc-editor.org/info/rfc5102.

[6] Fatemipour, F. and Yaghmaee, M. H. Design and Implementation of a Monitoring System Based on IPFIX Protocol. In: *The Third Advanced International Conference on Telecommunications (AICT'07)*. 2007, p. 22–22. DOI: 10.1109/AICT.2007.18.

[7] Goodfellow, I., Bengio, Y. and Courville, A. *Deep learning*. Manning Publications, 2015.

[8] HEJCMANL, L. *COLLABORATIVE MACHINE LEARNING IN THE CONTEXT OF NETWORK SECURITY*. Brno, CZ, 2023. Masters thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/211926.

[9] Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R. et al. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*. 2014, vol. 16, no. 4, p. 2037–2064. DOI: 10.1109/COMST.2014.2321898.

[10] James, G., Witten, D., Hastie, T., Tibshirani, R. and Taylor, J. *An introduction to statistical learning: With applications in python*. Springer Nature, 2023.

[11] Jan, S. *Testing of Probes for Network Traffic Monitoring*. Brno, CZ, 2022. Masters thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://theses.cz/id/hv65ma/25032.pdf.

[12] MacKenzie, D., Tromey, T. and Duret Lutz, A. *GNU Automake*. 2012. Accessed: 2024-04-27. Available at: https://www.gnu.org/software/automake/manual/1.11.6/automake.pdf.

[13] Najafabadi, M. M., Villanustre, F., Khoshgoftaar, T. M., Seliya, N., Wald, R. et al. Deep learning applications and challenges in big data analytics. *Journal of big data*. Springer. 2015, vol. 2, p. 1–21.

[14] Oliveira, T. P., Barbar, J. S. and Soares, A. S. Computer network traffic prediction: a comparison between traditional and deep learning neural networks. *International Journal of Big Data Intelligence*. Inderscience Publishers (IEL). 2016, vol. 3, no. 1, p. 28–37.

[15] pytorch. *Module set_parameters*. 2018. Accessed: 2024-04-27. Available at: https://github.com/pytorch/pytorch/issues/13383.

[16] pytorch. *(libtorch) Save MNIST c++ example's trained model into a file, and load in from another c++ file to use for prediction?* 2019. Accessed: 2024-04-27. Available at: https://discuss.pytorch.org/t/libtorch-save-mnist-c-examples-trained-model-into-a-file-and-load-in-from-another-c-file-to-use-for-prediction/51681/11.

[17] pytorch. *Add load_state_dict and state_dict() in C++*. 2020. Accessed: 2024-04-27. Available at: https://github.com/pytorch/pytorch/issues/36577.

[18] pytorch. *Saving torchscript model in C++*. 2020. Accessed: 2024-04-27. Available at: https://github.com/pytorch/pytorch/issues/35464.

[19] pytorch. *Saving and loading model with libtorch C++*. 2023. Accessed: 2024-04-27. Available at: https://discuss.pytorch.org/t/saving-and-loading-model-with-libtorch-c/184482.

[20] PyTorch Contributors. *The C++ Frontend*. 2022. Accessed: 2024-04-21. Available at: https://pytorch.org/cppdocs/frontend.html.

[21] PyTorch Contributors. *Installing C++ Distributions of PyTorch*. 2022. Accessed: 2024-04-21. Available at: https://pytorch.org/cppdocs/installing.html.

[22] PyTorch Contributors. *PyTorch JIT Documentation*. 2022. Accessed: 2024-01-18. Available at: https://pytorch.org/docs/master/jit.html.

[23] PyTorch Contributors. *PyTorch documentation*. 2023. Accessed: 2024-04-27. Available at: https://pytorch.org/docs/stable/index.html.

[24] Radford, B. J., Apolonio, L. M., Trias, A. J. and Simpson, J. A. Network traffic anomaly detection using recurrent neural networks. *ArXiv preprint arXiv:1803.10769*. 2018.

[25] SEDLÁK, M. *FILTRACE A PROFILOVÁNÍ IP TOKŮ*. Brno, CZ, 2020. Bachelors thesis. Brno University of Technology, Faculty of Information Technology. Available at: http://hdl.handle.net/11012/191514.

[26] Šimek, Š. *Real-time Network Flow Control using Machine Learning and OVS.* Praha, CZ, 2023. Bachelors thesis. Czech Technical University in Prague, Faculty of Information Technology. Available at: https://netmon.fit.cvut.cz/theses/F8-BP-2023-Simek-Stepan-thesis.pdf.

[27] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A. et al. An Overview of IP Flow-Based Intrusion Detection. *IEEE Communications Surveys & Tutorials.* 2010, vol. 12, no. 3, p. 343–356. DOI: 10.1109/SURV.2010.032210.00054.

[28] Stevens, E., Antiga, L. and Viehmann, T. *Deep learning with PyTorch.* Manning Publications, 2020.

[29] Svoboda, J., Ghafir, I., Prenosil, V. et al. Network monitoring approaches: An overview. *Int J Adv Comput Netw Secur.* 2015, vol. 5, no. 2, p. 88–93.

[30] Trammell, B. and Boschi, E. An introduction to IP flow information export (IPFIX). *IEEE Communications Magazine.* 2011, vol. 49, no. 4, p. 89–95. DOI: 10.1109/MCOM.2011.5741152.

[31] Xu, C., Chen, S., Su, J., Yiu, S. M. and Hui, L. C. K. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials.* 2016, vol. 18, no. 4, p. 2991–3029. DOI: 10.1109/COMST.2016.2566669.