Towards Machine Learning Inference in the Data Plane

Jonatan Langlet

The Faculty of Health, Science and Technology

Department of Mathematics and Computer Science

C-Dissertation 15 HP

Advisor: Andreas Kassler

Examiner: Bestoun Al-Beywanee

Date: 14 June 2019

Towards Machine Learning Inference in the Data Plane

Jonatan Langlet

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Jonatan Langlet

Approved, June 14, 2019

Advisor: Dr. Andreas Kassler

Examiner: Dr. Bestoun Al-Beywanee

Abstract

Recently, machine learning has been considered an important tool for various networking related use cases such as intrusion detection, flow classification, etc. Traditionally, machine learning based classification algorithms run on dedicated machines that are outside of the fast path, e.g. on Deep Packet Inspection boxes, etc. This imposes additional latency in order to detect threats or classify the flows.

With the recent advance of programmable data planes, implementing advanced functionality directly in the fast path is now a possibility. In this thesis, we propose to implement Artificial Neural Network inference together with flow metadata extraction directly in the data plane of P4 programmable switches, routers, or Network Interface Cards (NICs).

We design a P4 pipeline, optimize the memory and computational operations for our data plane target, a programmable NIC with Micro-C external support. The results show that neural networks of a reasonable size (i.e. 3 hidden layers with 30 neurons each) can process flows totaling over a million packets per second, while the packet latency impact from extracting a total of 46 features is $1.85\mu s$.

Acknowledgements

First, I would like to thank my supervisor ***Dr. Andreas Kassler*** for showing great interest in this project, and inspiring me for a future career in research.
I would also like to thank ***Jonathan Vestin*** for helping me with the hardware installation, and making time in his busy schedule for technical discussions and general advice during the development process.
Lastly, I would like to show my appreciation for *Jonathan Magnusson*, *Simon Sundberg*, and *Daniel Larsson* who made the lab a fun environment to work in.

Contents

List of Figures

List of Tables

1 Introduction

Machine learning (ML) is a field within computer science which has gained a lot of traction in recent years, introducing new techniques and applications at a rapid pace.
An important property of ML is its ability to quickly and efficiently analyze large amounts of data, as is the case when analyzing network traffic. This could mean a data flow with millions of packets per second without knowing exactly how to manually specify the classification rules, which is a task where ML shines with its ability to perform inference on massive amounts of data, and a classification accuracy often surpassing that of humans [7, 32].

1.1 Motivation

There are many different kinds of ML algorithms with varying degrees of effectiveness depending on the application. Among these are *Support-vector machines (SVMs)* [15], *Random forests* [9], and *Artificial Neural Networks (ANNs)* [25]. These ANNs have proven to deliver great results when trained on network packet traces to perform network intrusion detection [10, 23, 31], flow classification [27, 16, 2], etc., and could therefore be a great inclusion in the data plane where these classifications could be used for influencing packet forwarding decisions.
Recently, there has been significant research effort in making the data plane more programmable. Indeed with the recent advancements in programmable data plane architectures, programming languages such as P4 together with compiler support, the data plane has evolved from a fixed function forwarding pipeline, to a flexible data plane that can be used to implement advanced operations on packets at line rate.
Machine learning based classification algorithms are traditionally implemented outside of the fast data path, e.g. on external Deep Packet Inspection Boxes [24, 4], which imposes additional latency on flow classification. An inclusion of machine learning inference in the

data plane could result in faster reaction times when identifying specific kinds of traffic flows, blocking malicious traffic, etc. Consequently, the question that we ask in this thesis is if recent programmable data planes can be used to implement machine learning inference directly in the fast data path, and if so what sort of performance impact can be expected on the network traffic? This thesis evaluates the feasibility of implementing inference using a specific kind of ANN called a *feedforward neural network* directly in the data plane, as these are some of the least computationally expensive neural networks available and require a constant number of operations for every possible input, based only on the model complexity. Given this property, a feedforward neural network should be able to perform reliably in a real time environment as is the case with packet forwarding.

1.2 Contribution

A framework has been developed for performing neural network inference on extracted traffic flow metadata in a P4 programmable switch, router, or network interface card with Micro-C external support, and the performance impact of various complexities have been evaluated. The results using a programmable data plane show that ANNs of a reasonable size (i.e. 3 hidden layers with 30 neurons each) can process flows totaling over a million packets per second while performing inference on every 20:th packet, with an additional packet latency of 1.85μs for extracting a total of 46 features.
Training is not a part of this work, the focus is instead of implementing inference using pre-trained neural network configurations. The neural networks are trained offline, and these parameters are then loaded onto the network appliance to perform inference on the extracted flow metadata. The neural network output can then be used when determining forwarding rules for the traffic flows.

1.3 Roadmap

Section 2 contains background information, where 2.1 and 2.2 is a brief overview of the P4 programming language and Micro-C respectively. 2.3 explains the theory behind the machine learning algorithm which has been implemented, and 2.4 describes the hardware architecture where the machine learning algorithm has been implemented. Section 3 is describing the implementation in more detail, where 3.1 is a brief overview, 3.2 describes the workaround used for performing floating point calculations in this architecture, 3.3 briefly explains the physical setup of the test environment, 3.4 is describing how race conditions due to the high level of parallelization was avoided, 3.5 explains how metadata about the network traffic flows was recorded, 3.6 introduces an interesting cloning method that could be used to decrease packet latencies, and 3.7 explains how the neural network is stored in memory. Section 4 contains the resulting measurements showing the performance impact of various model configurations on the packet throughput in the switch. Section 5 contains a brief discussion where the feasibility of this implementation is evaluated based on the results.

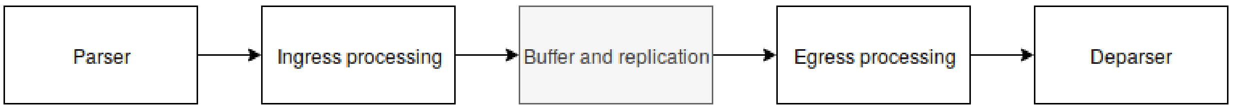

Figure 2.1: A simplified view of the PSA pipeline displaying four programmable blocks that every incoming packet is passing through in-order.

2 Background

2.1 The P4 programming language

P4 is an open source programming language developed by *The P4 Language Consortium* that is designed to be used for *data plane programming*, which can include extra functionality during the forwarding of incoming network packets [3]. Because of this, there is a lot of built-in support for parsing network packets and specifying the forwarding rules for these. The language is also designed to be target-independent, which is accomplished by using a P4 compiler that is aimed at the target architecture. An example of such a target architecture is the NFP-4000 network flow processor, which is presented in Section 2.4.

The P4 language includes tools for packet parsing and reassembly. There are also built-in methods to match these parsed values against lookup tables, where the user can specify matching-rules that are used to perform specific user-defined actions. These actions behave a lot like traditional functions, and can often be used as such.

To develop P4 firmware targeting packet switches, a library called *Portable Switch Architecture (PSA)* can be used [6]. To simplify development, PSA introduces additional pre-defined functionality to P4 including checksum calculation, packet cloning, and a packet processing pipeline.

Every incoming network packet goes through a pre-defined sequence of programmable blocks, as seen in Figure 2.1. First is the *parser*, which is where packet data can be extracted and categorized. A typical use is to parse the IP headers to retrieve the IP-addresses. These values that are extracted can then be used further down in the packet processing pipeline.

When the parser is finished, the packet enters the programmable *ingress block* for further processing. The main goal of this block is to specify the forwarding rules to determine the egress port for the packet. It is also possible to perform packet cloning to either the egress block or back again to ingress. After ingress processing, the packet is stored in a buffer waiting to be entered into the egress processing.

This *egress block* doesn't have a specific mission, but can be used for general processing of the packet. For example, this is where neural network inference could be performed.

Lastly there is the *deparser*, which might be the least interesting block. This is where the packet is prepared for emission.

One of the primary limitations of P4 is the lack of native floating-point calculations. An example of how this functionality can be implemented is described in Section 3.2. Variable-length loops are also not implemented by design, the reason being that these can result in inconsistent packet processing times. This can be circumvented by linking in external *Micro-C code* (see Section 2.2), at the cost of making the code target dependent.

Using these Micro-C additions, a P4 firmware can be expanded to do a lot of complex computations during the processing of the network packets passing through a switch. For example, it should be possible to perform real time neural network inference on the traffic flows passing through, which is the focus of this thesis. A problem with including Micro-C code snippets in P4 is that these are target dependent, and could require modifications when migrating to another hardware architecture. For cases when P4 is compiled to VHDL, as is the case when programming an FPGA [18], linking in Micro-C is not even a possibility as the hardware has no way of executing these instructions.

2.2 Micro-C

Micro-C is a slimmed down version of C that has been developed by Netronome for developing firmware targeting their network flow processor architecture. [30]

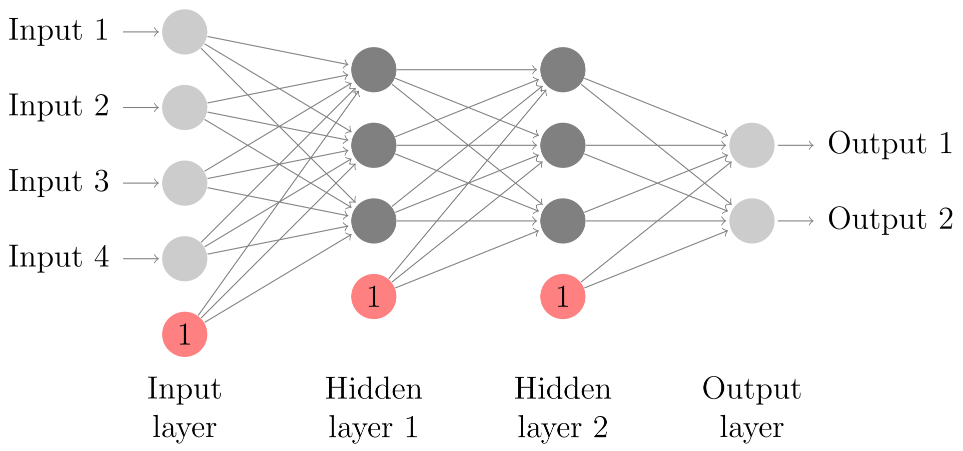

Figure 2.2: Visualisation of the neurons and connections in a feedforward neural network with two hidden layers. The grey nodes are neurons, and the red nodes with a '1' are biases

This language is based on C89, but without support of floating point variables and dynamic memory among other things. Due to the lack of stack memory, recursion and variable length function lists are also not allowed. Most standard C libraries are not included in Micro-C, and can't easily be modified for compatibility because of the limitations previously mentioned.

2.3 Feedforward neural networks

Neural networks are one of the most powerful machine learning algorithms currently available, and are heavily utilized in a variety of fields to perform inference on large amounts of data. This technique has proven to deliver great results when performing inference on network traffic [26, 11].

One of the simplest neural networks to implement are the *feedforward neural networks* [14]. These work by arranging *neurons* into different layers, where each layer is connected to the two adjacent ones as seen in Figure 2.2.

What we here refer to as a neuron is a node that can hold a single value, which is the neuron's activation level. This value is calculated based on the activations of every neuron in the previous layer, and how strong the connection is between those neurons and this one according to the following equation:

$$a_i^{(L)} = \sum_{k=1}^{N_{L-1}} a_k^{(L-1)} w_{i,k}^{(L)}$$

where

$a_i^{(L)}$: the activation of neuron i in layer L

$w_{i,k}^{(L)}$: the weight between neuron $a_i^{(L)}$ and $a_k^{(L-1)}$

N_L: the number of neurons in layer L

It is often not optimal for every neuron to have the same base activation level, so a bias is included in each calculation like so:

$$a_i^{(L)} = \sum_{k=1}^{N_{L-1}} a_k^{(L-1)} w_{i,k}^{(L)} + b_i^{(L)}$$

where

$b_i^{(L)}$: the bias attached to neuron $a_i^{(L)}$

These biases can be thought of as weights from a neuron with a constant activation level of 1.

An activation function ϕ is applied to the calculation, which will allow the neural network to learn more complex patterns with a smaller number of neurons [8].

$$a_i^{(L)} = \phi(\sum_{k=1}^{N_{L-1}} a_k^{(L-1)} w_{i,k}^{(L)} + b_i^{(L)})$$

There are numerous different activation functions to choose from such as *Sigmoid* [29, 12], *tanh* [28, 12], and *rectified linear units* (*ReLU*). Sigmoid ($\frac{1}{1+e^{-x}}$) used to be the most

widely adopted activation function for a long time, but has in recent years been almost universally replaced by one of the ReLU variants [5, 7, 22]. Most ReLU functions do not require calculation of any exponential functions, and could therefore be implemented in those architectures that do not have any native support for handling these.
The simplest ReLU is seen in Equation (2.1). These would be inexpensive to implement, but has some inherent problems such as dying neurons occurring when the neuron is stuck at an activation level of 0.

$$ReLU(x) = \begin{cases} x, & x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Because of this, *Leaky ReLU* [17] is sometimes used instead, and is calculated as seen in Equation (2.2). This function introduces a small slope for negative activations, which will make sure that the backpropagation algorithm can keep changing these parameters even in those cases when basic ReLU would have a dead neuron.

$$LReLU(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases} \tag{2.2}$$

where

$0 < \alpha < 1$

An alternative ReLU, which is especially useful in cases when a predictable maximum activation value is wanted, is the *ReLU-6* [13]. This activation function behaves like standard ReLU, except that it is also capped at an arbitrary maximum value (see Equation (2.3)).

$$ReLU6(x) = \begin{cases} 6, & x \geq 6 \\ x, & 0 < x < 6 \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

The input-layer neurons do not use an activation function at all. Instead, the activation levels in these input-neurons contain the scalar values of the features that the neural network will perform inference upon. These activations will then propagate through the neural network, until the resulting output is presented as activations in the last layer (i.e. the *output layer*).

For the neural network to do anything meaningful, every weight and bias has to be fine-tuned to produce a proper output. One way to tune these parameters is by doing a gradient descent of the *loss function*, which in the case of supervised machine learning is usually the squared difference between the expected output $\vec{y}$ and the produced output $\vec{a}$ from the neural network:

$$L(...) = \sum_{k=1}^{N_{out}} (a_k - y_k)^2$$

The neural network is then trained to perform as intended by doing a gradient descent on this loss function. The algorithm used for performing this gradient descent is called *backpropagation*, but is out of the scope of this thesis.

2.4 NFP-4000 architecture

The programmable network card which was used in this project is the Netronome Agilio CX 2x40 SmartNIC, which is based on *Network Flow Processors (NFP's)* [19, 20] clocked at 800MHz. This architecture is utilizing a lot of parallel processing to achieve a high packet processing performance.

2.4.1 Overview

The network cards have 60 flow processing cores called *Micro Engines (ME's)*, with 8 threads each. These ME's have a small amount of local memory, and run in parallel with every other ME. It is essential to keep this parallelization in mind during development to

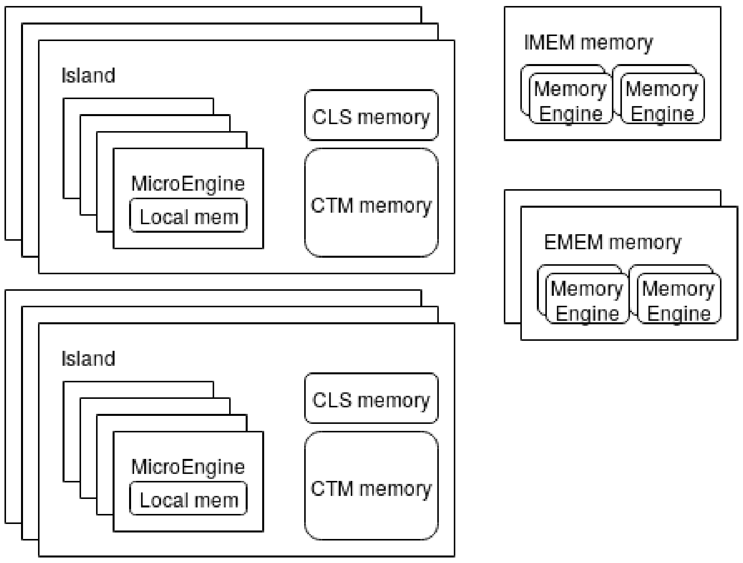

Figure 2.3: A simplified view of the components in the NFP-4000 architecture showing the locations of the various memory regions

avoid problems with race conditions.

An ME can have no more than one thread executing code at any given time. Each thread is working with its own set of CPU registers, which means that a context switch between threads is really quick, taking no more than 2 cycles to complete. The threads are non-preemptive, and have to explicitly tell the CPU that they are done executing for another context to take control.

These ME's are grouped together into different islands, where each island has some memory that can be shared between its ME's. See Figure 2.3 for a rough visualization of how components are grouped together.

2.4.2 Memory regions

There are multiple memory regions available to the programmer, each with its own capacity and access times. Picking an optimal memory region for data storage is important when developing firmware for this architecture, since there is a significant latency difference between regions.

The fastest memory, apart from CPU registers, is the ME's *local memory (LMEM)*, which is shared by all threads in an ME. This region has a maximum capacity of just 4 kilobytes, yet has an impressive latency of 1-3 cycles.

Then there is the *Cluster Local Scratch (CLS)*. This is a region which is shared within an island, and is accessible by an ME over the *Command Push Pull bus (CPP)*. It is possible for an ME in another island to access the CLS, but this is much slower than accessing the CLS from within the island.
The CLS has a maximum capacity of 64 kilobytes, with a latency of about 20-50 cycles.
It is also worth noting that the CLS is the only memory accessible through the CPP that processes queries in-order.

The third region is the *Cluster Target Memory (CTM)*. This region is shared by all ME's in an island, and is accessible through the CPP bus.
The CTM has a maximum capacity of 256 kilobytes, with a latency of about 50-100 cycles.

Another region is the *Internal Memory (IMEM)*, which is the first region that is not bound to any one island, but is instead shared by all ME's on the card through the CPP bus.
There is only one IMEM available, with a maximum capacity of 4 megabytes. The latency of the IMEM is about 150-250 cycles.

Finally there is the *External Memory (EMEM)* which, like the IMEM, is shared by all ME's through the CPP bus.
There are two EMEM areas available, each with 3 megabytes of built-in cache memory.

Data allocated here is stored in external DRAM, which in the case of the Netronome Agilio CX 2x40GbE cards contains 2GB memory storage.
The EMEM has a latency of about 150-500 cycles, making it the slowest memory region available. But because of the possibility to expand the EMEM memory, it could be a necessity to utilize it when storing large amounts of data in memory.

2.4.3 Memory accesses

The CPP bus is used when an ME wants to access the CLS, CTM, IMEM, or EMEM regions. These memories have a built-in dispatcher thread that handles incoming queries, and delegates work to internal threads that handle the memory.
When an ME thread wants to read one of these memories, it sends an instruction through the CPP bus and yields to the CPU until the read is complete, allowing another thread to continue its work while the memory is being read. When the memory is sent back, the ME changes the state of the thread to ready, indicating that it is ready to continue execution.
A write request works similarly to read, except that the thread doesn't yield while the query is being processed. Instead, the thread sends an instruction and lets the memory workers handle it on their end.

3 Design & Implementation

The feature collection and neural network inference is implemented on a Netronome NFP-4000 Flow Processor. This architecture has some limitations that makes the implementation tricky, which will be explained in more detail during this chapter.

3.1 Overview

Every incoming packet is first processed by the parser, where everything down to and including the first few bytes of the application payload is extracted[1]. After the packet has been parsed, P4 table rules are used to determine egress port for the packet (i.e. what physical port to send the packet through after its processing is complete). Immediately after that, the values that were extracted by the parser are used to update flow metadata to include the current packet. Next, the packet enters egress processing. Here, the flow metadata is used to determine if it is time for the flow to be analyzed by the neural network. This is triggered by every 20:th flow packet during all of the measurements, although additional packets in this flow might be included in flow metadata before the 20:th packet has time to enter egress processing. How many packets should get through before performing a neural network inference is a hyperparameter which has to be tuned to find an optimum for each application.
A visualization of the packet pipeline for this implementation can be seen in Figure 3.1.

3.2 Floating point calculations

The NFP-4000 architecture, which Netronome Agilio CX SmartNICs are based upon, doesn't have any native support for handling floating point calculations. Because of the extensive use of floating point multiplications in neural networks, it is essential to find a

[1]The first few bytes of the application payload is parsed so that application headers could be retrieved. These can be used to verify application protocol without just relying on transport port. For example, port 80 by itself doesn't guarantee HTTP traffic

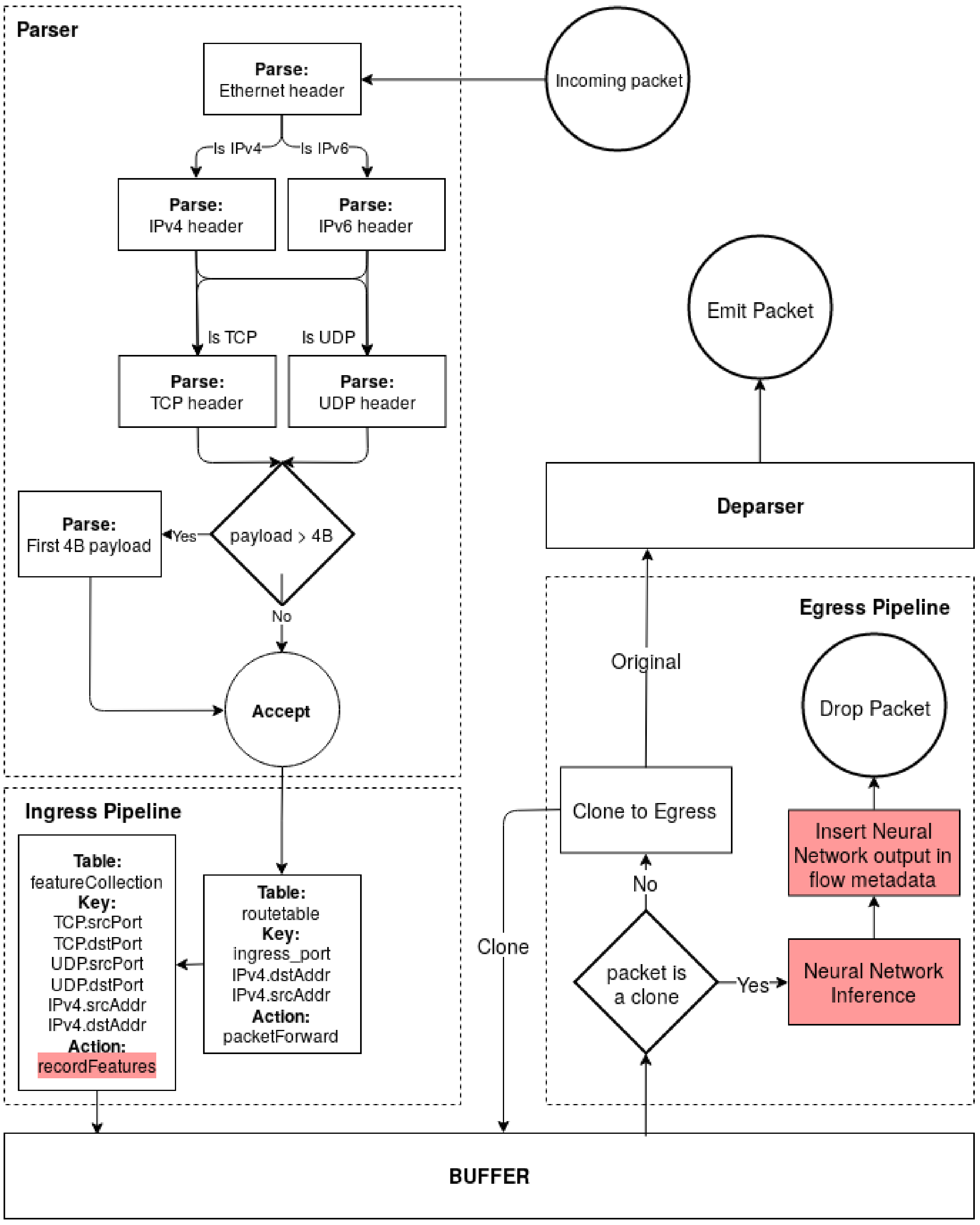

Figure 3.1: Visualization of the P4 pipeline for incoming packets in this implementation. The red blocks are Micro-C functions, while the white are pure P4.

working representation of these.

The solution implemented in this thesis is using arithmetic bit-shifting of 64-bit integers to represent floating point values. A shift of 30 will be used during this section as an example, but this shift has to be changed to ensure that the largest expected values can be represented without an integer overflow occurring. Using a bit-shifting of 30 means that a floating point value of x is represented in memory by an integer with the value $x2^{30}$.

The smallest non-zero unit that can be stored in an integer is 1. This means that the smallest unit x which can be represent using this method is:

$$x2^{30} = 1 \Leftrightarrow x = 2^{-30} \approx 9.3 \times 10^{-10} \tag{3.1}$$

A floating point multiplication can be done by multiplying two of these representations and then bit-shifting the product. This is more clearly demonstrated with the following equation:

$$x2^{30}y2^{30} = xy2^{60} \tag{3.2}$$

where simply multiplying the product with 2^{-30} (i.e. bit-shifting 30 positions to the right) results in the correct answer. This step in the floating point multiplication seen in Equation (3.2) results in a pretty small maximum value that can be handled using this implementation. The largest value that can be stored in a signed 64-bit integer is $2^{63} - 1$, which means that the largest values that can appear during multiplication can't exceed this maximum[2]. The largest value that has to be stored during these multiplications are the ones from Equation (3.2), which means that the following inequality has to be met for floating point multiplications to work as expected:

$$xy2^{60} < 2^{63} - 1 \Rightarrow xy < 8 \tag{3.3}$$

[2]A workaround that could solve this might be using datatypes larger than 64 bits, but because floating point multiplications are such a big part of the computational time of the neural network, we try to avoid working with anything that can't fit in the CPU registers

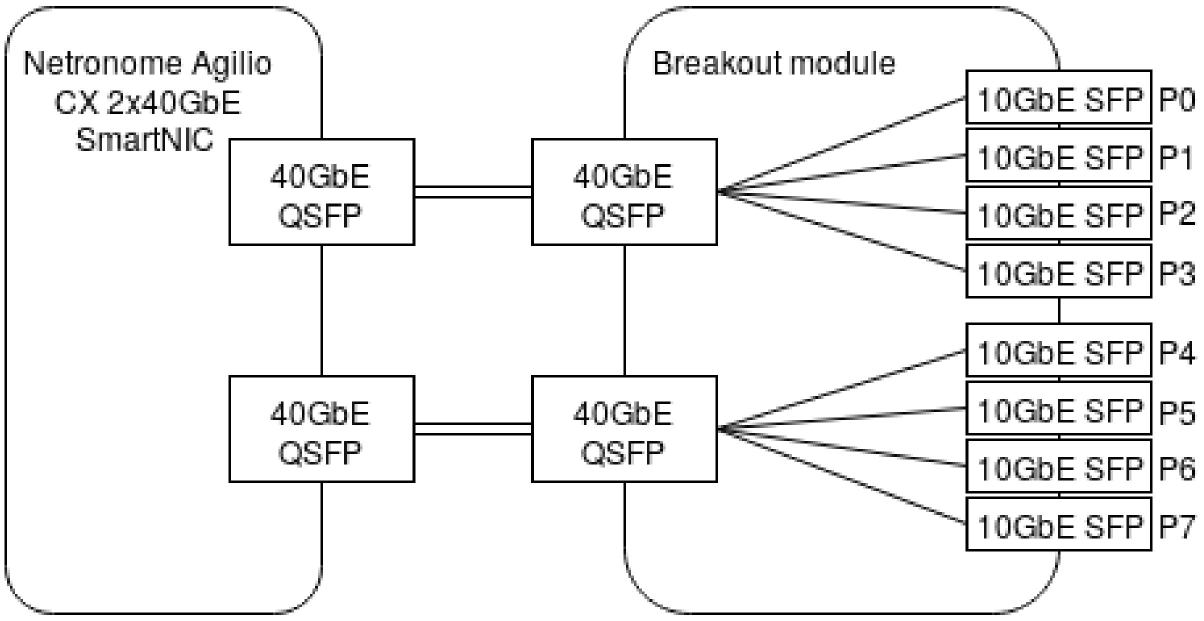

Figure 3.2: Visualizing the breakout module splitting the two ports into eight

As described in Section 2.3, some variation of ReLU-6 can be used to keep the maximum values down, at the same time as the bit-shifting is kept low as to enable handling of larger values without integer overflows.

3.3 Physical setup

The system was implemented in a Netronome Agilio CX 2x40GbE SmartNIC, equipped with a breakout module splitting each of the 40GbE ports into 4x10GbE port as seen in Figure 3.2. These split ports are then referred to as P0,P1,P2 and so on during firmware development.

3.3.1 Measuring processing time in SmartNIC

To measure the packet latencies, a NetFPGA SUME [33] running *Open Source Network Tester (OSNT)* [1] was plugged into two of the SmartNIC ports as seen in Figure 3.3. The OSNT sent traffic through one of these ports, and the SmartNIC performed its computa-

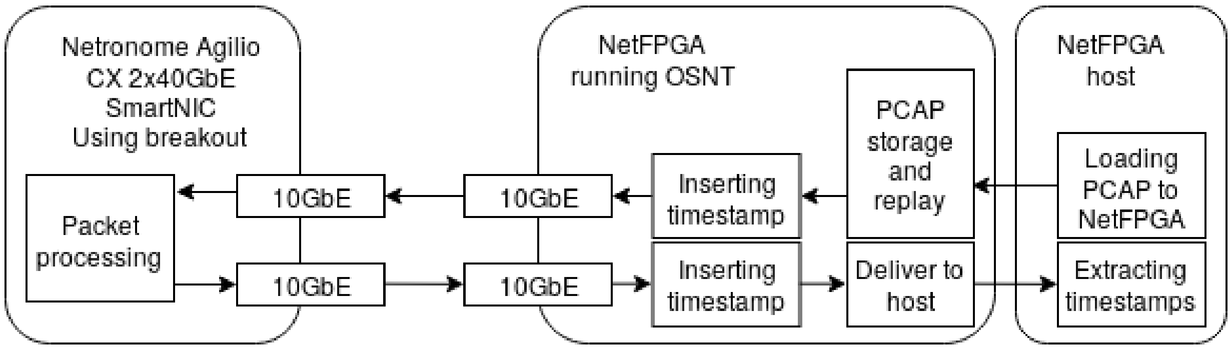

Figure 3.3: Visualization on how OSNT is used to calculate packet latency using one 10GbE port for sending traffic and another 10GbE for receiving those same packets. Timestamps are inserted in the network packets

tions and forwarded the packet back to OSNT through the other port.

To measure latencies, the OSNT was configured to send custom UDP packets with a payload containing an arbitrary string of characters. OSNT is inserting a timestamp in this payload, with a granularity on 6.25ns, right before transmission. When it receives these same packets in the other port, it just has to calculate the difference between the packet timestamps to know the total latency.

This latency is a bit larger than the actual processing latency in the SmartNIC, because it takes a non-zero time for the transceivers to send and receive the packets[3]. This delay can easily be measured by connecting a fiber directly between the transceivers of the OSNT card without a SmartNIC in the middle. Simply subtracting this value from the measurements should result in a decent approximation of the actual latency impact of the SmartNIC.

3.4 Handling race conditions

Because of the high level of parallelization, it is essential to keep the risk of race conditions in mind during development to ensure that the system performs as expected [21].

[3]There is also a tiny propagation delay in the fiber cables

An example of when race conditions have to be accounted for is when the firmware is counting total number of packets that has been seen in a specific traffic flow[4]. If there are two or more packets belonging to the same flow being processed in parallel inside of different ME's, and they are both at the part where they should increment the same packet counter, one of these packets might not be included in the counter unless there is some safeguard implemented.

3.4.1 Mutex locks

One method to avoid race conditions is using mutex locks. These locks can be utilized to make sure that there can only be one worker inside of a critical section at any given time. This has an obvious performance impact since they prevent all but one packet from doing these calculations at any given time.
Mutex locks are not used in this implementation, *atomic functions* are used instead.

3.4.2 Atomic functions

Atomic functions can sometimes be used as an alternative to mutex locks. These are functions where other threads see them happen instantaneously, and therefore eliminating the risk of a thread being half-done when another is entering a critical section. Only very short and simple functions can be written atomically.
Atomic functions are heavily utilized in this implementation to avoid race conditions while at the same time keeping the performance high. As an example, atomic functions are used in the case of recording the total amount of data seen in a flow as seen below:

```
//get packet size
pktsize_t pktSize = pif_plugin_meta_get__standard_metadata__packet_length(
    headers);
```

[4]The total number of packet seen in a flow is the base for various other features that the neural network will perform inference upon. An example is the mean packet size, which is the total amount of data divided by the total number of packets

```
mem_add64_imm( pktSize , (__mem40 void*)&flowMetadata[flowIndex].
    totalFlowData );

```

3.5 Feature collection

A big part of the implementation is the collection of features for the neural network to perform inference on. In this implementation, feature collection happens during ingress processing on a per-flow basis where metadata is extracted from the network packets passing through the switch. Most of the metadata is recorded separately for each direction of the packet flow, meaning that for example the total number of flow packets is split into the forward and backwards direction.

There are both recorded raw metadata, and derived metadata which is based on recorded values. The full list of stored flow metadata is presented in Table 3.1.

Table 3.1: A full list of stored flow metadata. The source in the first packet is called flow client, while the destination is flow server

Name	Description
clientAddr	Source IP-address in first flow packet
serverAddr	Destination IP-address in first flow packet
clientPort	Source port in first flow packet
serverPort	Destination port in first flow packet
clientLowerIP	1 if clientIP < serverIP
initTime	Time when first flow packet was processed
lastTime	Time when latest flow packet was processed
pktCount	Total number of packets in flow
pktCount_forward	Number of packets from client to server
pktCount_backward	Number of packets from server to client

Continued on next page

Table 3.1 – continued from previous page

Name	Description
totalFlowData	Total data volume in flow
totalFlowData_forward	Data volume from client to server
totalFlowData_backward	Data volume from server to client
tproto_tcp	If TCP traffic
tproto_udp	If UDP traffic
maxPktSize	Size of largest packet in flow
maxPktSize_forward	Size of largest packet from client to server
maxPktSize_backward	Size of largest packet from server to client
minPktSize	Size of smallest packet in flow
minPktSize_forward	Size of smallest packet from client to server
minPktSize_backward	Size of smallest packet from server to client
aProto_http	Is this HTTP traffic?
aProto_ssh	Is this SSH traffic?
aProto_ftp	Is this FTP traffic?
lastSeqnum_forward	The last TCP sequence number from client to server
lastSeqnum_backward	The last TCP sequence number from server to client
synTime	Time when SYN-packet was processed
synackTime	Time when SYNACK-packet was processed
tcpSynSynackTime	Time between SYN and SYNACK
tcpSynackAckTime	Time between SYNACK and ACK
tcpSetupRTT	RTT during TCP handshake (see fig 3.4)
flowDuration	Time between first and last packet in flow
load_forward	Traffic load from client to server (Bytes/sec)
load_backward	Traffic load from server to client (Bytes/sec)
ttl_forward	TTL of last packet from client to server
ttl_backward	TTL of last packet from server to client

Continued on next page

Table 3.1 – continued from previous page

Name	Description
retran_forward	Number of retransmissions from client
retrans_backward	Number of retransmissions from server
interPktGap_forward	Interpacket arrival time from client to server
interPktGap_backward	Interpacket arrival time from server to client
tcpWindow_source	Advertised TCP window from client
tcpWindow_dest	Advertised TCP window from server
ISN_forward	TCP sequence number in first packet from client
ISN_backward	TCP sequence number in first packet from server
meanPktSize_forward	Mean packet size from client to server
meanPktSize_backward	Mean packet size from server to client

3.5.1 Identify traffic flow

A fundamental part of flow analysis is to identify what flow an incoming packet is part of. This is accomplished by calculating a simple hash, which is based on the source/destination IP addresses and ports. The flow metadata can then be stored in a hash table using this calculated hash as index.
To make sure that the hashes of packets in both direction point to the same entry in the hash table, the hash function is based on the sum of the source/destination addresses and ports as seen below:

$$h(p_s, p_d, a_s, a_d) = (31(a_s + a_d) + p_s + p_d)\%T_{size} \qquad (3.4)$$

where

p_s, p_d: the source and destination ports

a_s, a_d: the source and destination IP addresses

T_{size}: the maximum number of entries in hash table

This hash function in Equation (3.4) is not cryptographic, and should be changed before being used in a real-world scenario so that an attacker could not mess with recordings of per-flow metadata. A simple non-cryptographic hash function was used while testing because of time restrictions during implementation, but because of the modular design it should be trivial to replace.

3.5.2 Time based features

There is a timestamp that can be retrieved during packet processing, which is a simple counter in memory that is incremented every 16:th clock cycle. For this to be compatible with the timestamps that are used while training a neural network, these values have to be converted into a second-based format.
To get a base value that can be used for this conversion, timestamps were collected with 100 seconds in between[5]. A function was then created which can be used to convert from the timestamp value into microseconds, which has proven to result in valid timestamps differences during testing.

3.5.3 Round trip time

The *round-trip time (RTT)* is a feature which is extracted for TCP traffic. This is calculated only once during the first 3 packets in a flow, and takes into account the fact that this implementation can be placed anywhere in between the client and server as seen in Figure 3.4. It is assumed that its relative position is unchanged during the handshake process.
For every incoming packet in a flow, a timestamp of the last flow packet is updated with

[5]The timestamp had increased by 5,074,521,076 while running for 100 seconds

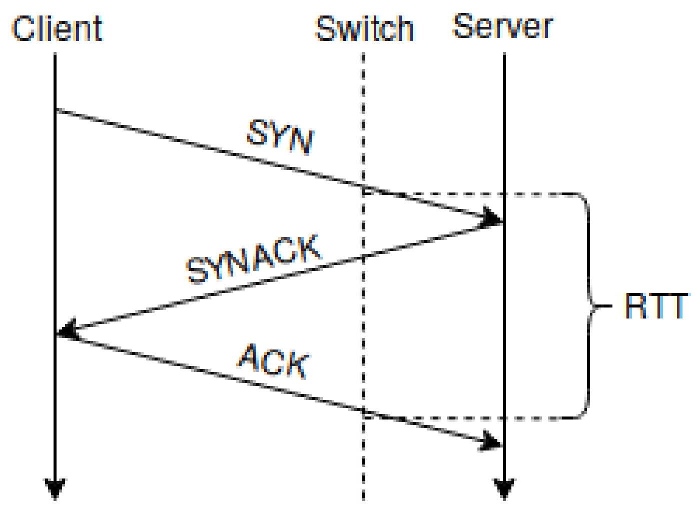

Figure 3.4: A visualization of the RTT calculation from switch's point of view

the current time. This is the last thing that happens during the ingress processing to ensure that the timestamp of the previous packet is still accessible.
The time between SYN and SYNACK can easily be recorded and stored because these packets arrive to the switch in-order, meaning that when a SYNACK is detected, the SYN-SYNACK duration is the difference between the current time and the last stored timestamp in the flow. The same logic applies to SYNACK-ACK time difference. Total RTT is then the sum of these two time differences.

3.5.4 Detecting packet retransmissions

The TCP sequence numbers of incoming packets is used to identify retransmissions for TCP traffic flows. For every incoming packet, the sequence number is compared to the sequence number of the last packet. If the sequence number isn't larger than the last one, the packet is assumed to be a retransmission. Below is a code snipped showing how this is implemented in this thesis.

```
if( thisSeqnum <= flowMetadata[flowIndex].lastSeqnum_forward )
{
  //if this is a retransmission -> increment counter
  mem_incr32( (__mem40 void*)&flowMetadata[flowIndex].retrans_forward );
}
else
{
  //not retransmission -> update last seqnum with this one
  flowMetadata[flowIndex].lastSeqnum_forward = thisSeqnum;
}

```

However, this implementation has some problems. It does not take into account the rollover effect, which is when the sequence numbers reach their maximum value of $2^{32} - 1$ and rolls back from 0. These events will be falsely counted as a retransmitted packet because the sequence number is lower than the previous one. Another problem is that packets in reality can arrive at a switch out of order, which this implementation can not handle.
Because of these reasons, another method of detecting retransmissions has to be implemented for this to work in practice. During the experiments, a packet stream can be designed in such a way that every packet arrives to the switch in-order and without reaching the maximum value.

3.5.5 Counting number of packets

To record the number of packets in each flow, an atomic incremental function is used which can increase a variable value by 1 in a single clock cycle. This means that there is no need for a mutex lock, which would slow down the system.

3.5.6 Traffic load

One of the features required by the ANN is the traffic load going back and forth between the source and destination. This is recorded as total bytes sent per second in each direction of a traffic flow.

For every processed packet, a specific function is called depending on the direction of the packet. These functions use an atomic add function, which is increasing a variable containing the total data volume sent in the current direction by the size of the current packet. This function has to either be atomic, or use a mutex lock to prevent race conditions between different workers handling different packets belonging to the same flow.

This atomic function is handling 64-bit memory areas in a strange way, swapping the highest and lowest 32 bits. To get around this, the value is being retrieved using a getter function which is swapping the bits to their correct positions, and by doing so returning the actual value being represented there.

```
uint64_t getTotalFlowData_forward(index_t flowIndex)
{
uint64_t result = flowMetadata[flowIndex].totalFlowData_forward;
return ( (result & 0x00000000ffffffff) << 32 ) | ( (result & 0
    xffffffff00000000) >> 32 );
}
```

This total data volume can then be divided by the total number of packets to get the traffic load, which can then be used as a feature for the ANN.

3.6 Improving performance by emitting packets before analysis

The neural network is waiting to perform inference until there is enough metadata collected. There would therefore be latency spikes for those packets which are triggering an inference if these calculations are performed in the processing pipeline before packet emission. A solution to this problem is to create a packet clone, and immediately emitting the original packet before performing the inference (this is visualized in Figure 3.5).

A problem with this approach is that this creates extra work in egress, which can result in packets spending more time in queue to enter egress processing than they would do without this cloning method. This would likely only be a problem during high load on the

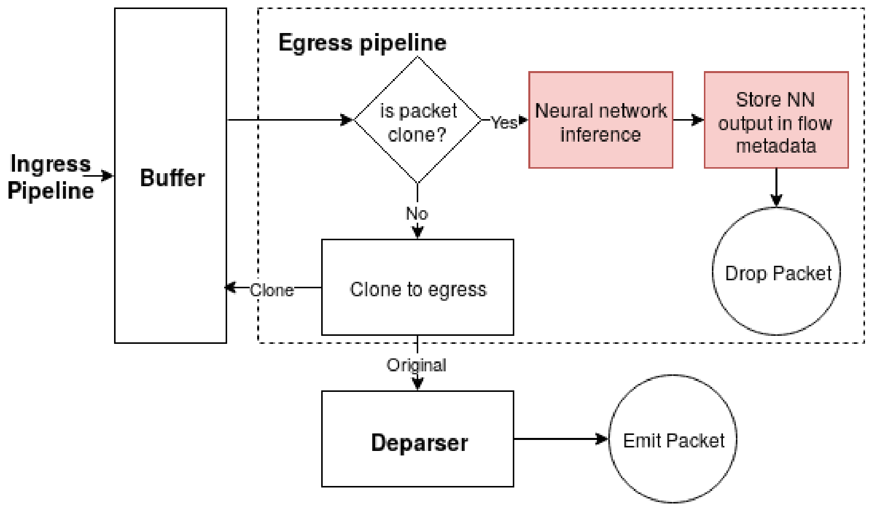

Figure 3.5: Simplified visualization on the egress pipeline explaining the cloning technique. Red block includes Micro-C functions, while white are pure P4

network card when there are no any idle egress workers. The performance impact of this technique has been measured for various neural network complexities, and is presented in Section 4.2.2.

Unfortunately, this technique does not work when storing the neuron activations in CLS memory for unknown reasons. Attempting this results in a service crash on the host machine.

3.7 Specifying neural network configuration

3.7.1 Specifying neural network size

The NFP architecture lacks dynamic memory allocations, which means that every variable has to be declared at compile time. Because of this, the number of hidden layers and neurons per layer are specified using preprocessing directives. This means that the firmware has to be recompiled when the ANN size has to be changed.

Two-dimensional arrays of signed 64-bit integers are used to store the neural network weights. One-dimensional arrays of signed 64-bit integers are used to store neural network biases and activation levels. Preprocessor directives determine how many arrays should be declared, and the sizes of these base on the specified number of neurons and hidden layers. Below is a Micro-C code snippet showing what the pre-processor is generating for neural networks with two hidden layers:

```
//Weights between layers
ann_weight_t weights_I_H1[INPUT_NNODES][HIDDEN1_NNODES];
ann_weight_t weights_H1_H2[HIDDEN1_NNODES][HIDDEN2_NNODES];
ann_weight_t weights_H_O[HIDDEN2_NNODES][OUTPUT_NNODES];
//Biases
ann_weight_t bias_I_H1[HIDDEN1_NNODES];
ann_weight_t bias_H1_H2[HIDDEN2_NNODES];
ann_weight_t bias_H_O[OUTPUT_NNODES];
//Neuron activations
ann_activation_t activations_I[INPUT_NNODES];
ann_activation_t activations_H1[HIDDEN1_NNODES];
ann_activation_t activations_H2[HIDDEN2_NNODES];
ann_activation_t activations_O[OUTPUT_NNODES];

```

3.7.2 Loading parameters into firmware

The weights and biases that are calculated during the training of the neural network have to be loaded into the ANN implementation. Because of the fact that every island has their own isolated CLS and CTM regions, these all have to be updated separately. This is done first thing when the firmware has loaded, telling every ME to execute a certain function which is populating their memory region with the weights and biases.

For this implementation, a simple python script was created which generates Micro-C code that is defining these values during compilation and includes this in the ME instruction store. This is not a flexible solution, seeing as the firmware has to be recompiled every time the model has been retrained, however, it is a straight forward approach which should not impact the runtime performance of the implementation. Another technique is required to load more complex models that do not fit in an ME code store.

An example of such a technique could be to use registers for communication with the host machine to update one parameter value at a time. Alternatively, if the host machine is supporting virtual functions it should be possible to send a custom network packet, containing weights and biases as packet payload, through the PCI slot that the SmartNIC is connected through. This packet could then be identified during processing either by a custom flag, or by the fact that it came in through a virtual interface from the host. This solution would allow quick modifications of the neural network weights, for example to automatically update the model when an updated version has been developed, without having to recompile the firmware[6].

[6] On-the-fly updates would only work in the case when the updated neural network model has the same structure, but with new values for weights and biases. New structures still require a recompilation to allocate these arrays in memory

4 Results

4.1 Memory latency

A simple test has been performed to get an idea of the latency for each memory region. This has been done for read- and write-operations separately.
100 TCP packets with a 100 byte payload was sent through the SmartNIC with an *inter-packet gap (IPG)* of 10ms during each test. The SmartNIC firmware has been programmed to perform either a read- or a write operation a specified number of times before the packet is emitted. During this test, the operation was performed [1,1000,10000,100000,1000000] times for each memory region to ensure that the results are linearly increasing with the number of memory operations performed. The latency when performing 1 read/write was subtracted from the results to isolate the impact of just the additional memory operations, and the latency when performing 1000000 memory operations was used to get an average latency for read/write-operations to each of the memory regions.
The firmware running on the SmartNIC is executing the following Micro-C code during ingress processing:

```c
volatile __declspec(local_mem) uint32_t lmem_counter;
volatile __declspec(cls) uint32_t cls_counter;
volatile __declspec(cls4) uint32_t cls4_counter;
volatile __declspec(ctm) uint32_t ctm_counter;
volatile __declspec(ctm4) uint32_t ctm4_counter;
volatile __declspec(imem) uint32_t imem_counter;
volatile __declspec(emem) uint32_t emem_counter;
int pif_plugin_memtest(EXTRACTED_HEADERS_T *headers, ACTION_DATA_T *
    action_data)
{
  /*
  //Measure write latency
  volatile uint32_t i;
  for(i = 0; i < MEMACCESS_NUM; i++)
    lmem_counter = i;
  */
  //Measure read latency
  volatile uint32_t i, j;
  for(i = 0; i < MEMACCESS_NUM; i++)
    j = lmem_counter;

  return PIF_PLUGIN_RETURN_FORWARD;
}

```

Table 4.1: Measured average delay for read/write-operations to specified memory regions. Standard deviation in parenthesis

Region	**Read delay (ns)**	**Write delay (ns)**
LMEM	13 (0.0)	15 (0.0)
CLS (same island)	51 (0.4)	43 (0.4)
CTM (same island)	74 (0.5)	53 (0.5)
CLS (another island)	112 (7.0)	106 (6.3)
CTM (another island)	132 (4.3)	103 (5.0)
IMEM	136 (9.2)	115 (7.9)
EMEM	137 (9.5)	102 (8.3)

The results in Table 4.1 show that it is important to place the neural network parameters in a fast memory region for its inference to have as low latency impact as possible on the network traffic.

As explained in Section 2.4.3, memory-operations to regions other than LMEM go through the CPP bus. The way this works for write-operations is for the worker thread to delegate these operations to the memory workers and immediately keep working, while read-operations require the thread to wait for the results to get delivered before continuing. The data shows that write-operations are faster than read-operations for every memory region that is accessed through this CPP bus.

4.2 Neural network inference delay

Because of how memory-heavy this ANN implementation is, it is important to pick an optimal memory region when storing the model.

A total of 62 different combinations of neural network sizes and memory regions have been measured (see Table B.1). During these measurements, empty TCP packets was being sent from the OSNT with an IPG of 0. Every feature was recorded. Every neural network that has been evaluated had 5 neurons in the input as well as in the output layer, with a variable number and size of hidden layers. All of these measurements were recorded while the OSNT sent traffic through one 10GbE port, and received those same packets back through another 10GbE port.

The measurements are confirmed to be identical with experiments where 4x10GbE ports are used for receiving traffic, which is forwarded back through another set of 4x10GbE ports. See Table B.2 for these measurements. The throughput difference between 1x10GbE and 4x10GbE is explained by changes to the feature collection made since the older 1x10GbE measurements. The reason for not using all 8 ports during the experiments presented here is because this required a more advanced hardware setup which was not available until at the very end of the project.

4.2.1 Memory region impact on performance

There is a clear negative correlation between the neural network complexity and the packet throughput in the switch, as can be seen in Figure 4.1. The choice of memory region also has a clear performance impact, as seen by the fact that faster memory regions (as defined in Table 4.1) consistently deliver a higher packet processing rate.

It is also possible to store the weights and neural activations in separate memory regions instead of storing them together. This will allow more complex neural networks to be implemented before having to move the model outside of the faster in-island memory regions.

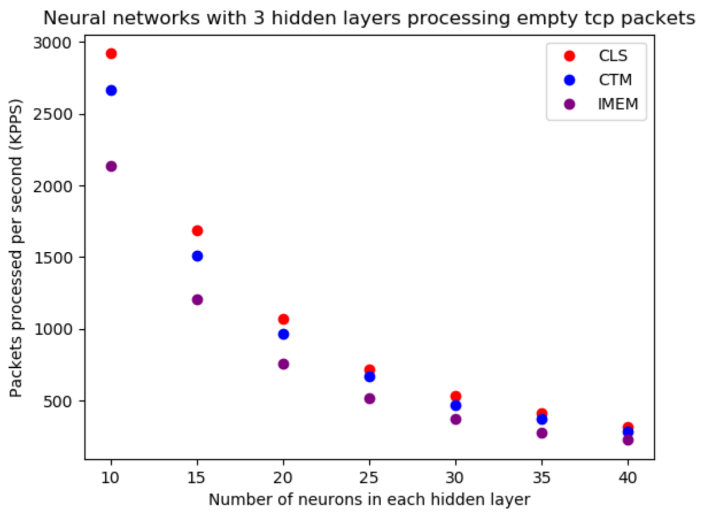

Figure 4.1: Performance impact for different neural network configurations. Comparing choices of memory region and network complexity.

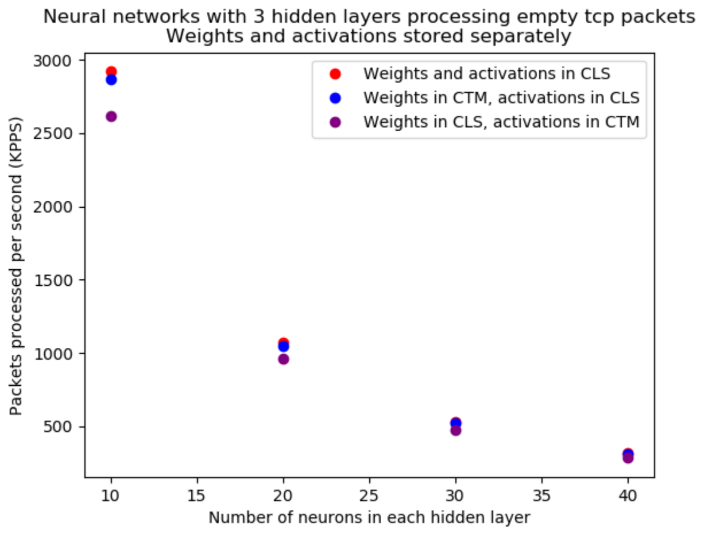

Figure 4.2: Neural network performance when the weights and activations are stored together compared with stored separately

Splitting the weights and activations into CTM and CLS have a slightly worse performance than storing them both in the faster CLS. But as Figure 4.2 shows, relocating only the weights to a slower memory region while keeping the activations unchanged has almost as good of a performance as storing the weights and activations together, and could therefore be acceptable when very complex neural networks that are too big to be stored in only CLS are needed.

4.2.2 Performing inference on packet clones

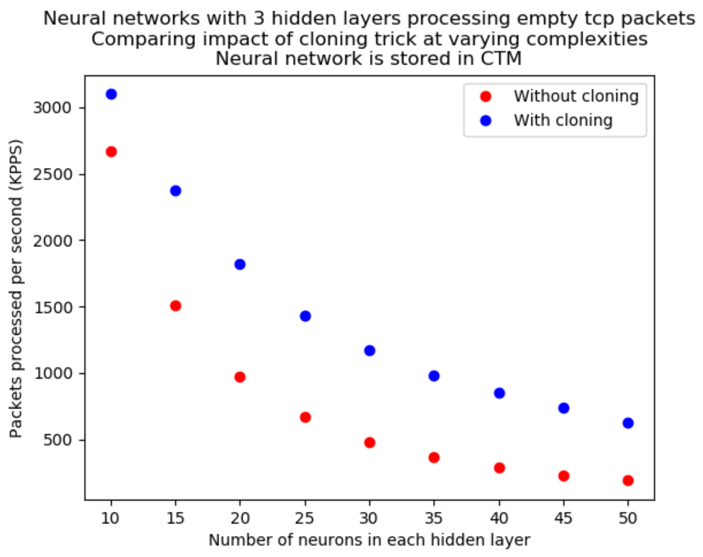

Figure 4.3: Performance impact when doing neural network inference on a clone after emitting the packet. 1x10Gbit port for in-traffic forwarded to another 10Gbit port during measurements

A possible technique to keep the performance high for more complex neural networks could be to perform the inference after having already emitted the original packet, as discussed in Section 3.6.

As Figure 4.3 shows, this technique results in a significant improvement on the packet processing rate, especially for complex neural networks where the packet processing rate is more that doubling[7]. Figure 4.4 shows the reduction in latency spikes while using the cloning technique. There is a small increase in the base latency when packet cloning is enabled. This increase comes from the fact that every single packet is being cloned even if

[7]As an example, three hidden layers with 40 neurons each go from a packet processing rate of 285KPPS to 850KPPS when enabling cloning as seen in table B.1

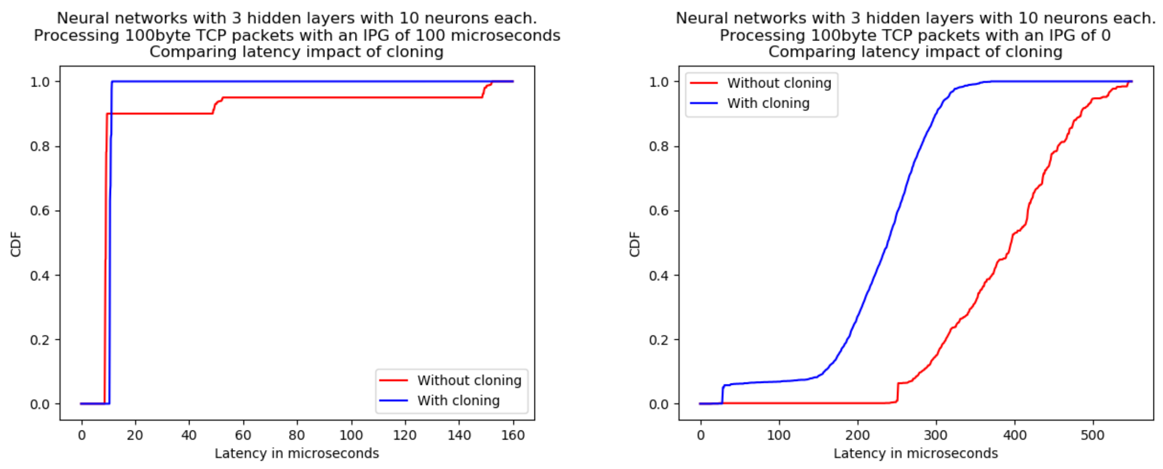

Figure 4.4: Cumulative distribution function for latency impact with/without cloning, showing the reduction in latency spikes when performing inference on packet clones. Left image is using an IPG of 100 microseconds, while the image on the right has an IPG of 0. 10000 packet latencies are included in each experiment

they would not trigger a neural network inference. This could easily be resolved by moving this check to before the cloning occurs, instead of after as is the case here.

Unfortunately, it is not possible to do this cloning technique while the memory activations are positioned in CLS for unknown reasons. Attempting this results in a service on the host machine crashing. A workaround for this problem has not yet been found.

4.3 Impact on data throughput

Two experiments measuring the maximum data throughput with and without metadata collection and neural network inference has been done, evaluating two different neural network complexities. The first neural network (ANN1) has 3 hidden layers with 10 neurons each, and the second neural network (ANN2) has 3 hidden layers with 20 neurons each. Both of these have 5 neurons in the input and output layers, are stored in CTM, and have cloning enabled. OSNT was used to generate the traffic through one 10GbE port, and receiving these packets back in another 10GbE port. A total of 46 features are being extracted from the traffic flows.

Table 4.2: Measuring total data throughput with and without metadata collection and neural network inference. Processing TCP packets with varying payload length

Payload	**Only parsing**	**+Metadata collection**	**+ANN1**	**+ANN2**
0 bytes	7.62Gbit/s	4.29Gbit/s	1.42Gbit/s	0.82Gbit/s
100 bytes	8.78Gbit/s	8.78Gbit/s	2.66Gbit/s	1.47Gbit/s
1000 bytes	9.80Gbit/s	9.80Gbit/s	9.80Gbit/s	9.80Gbit/s

These results presented in Table 4.2 show that neural network inference can be performed at line rate for larger packet sizes.

4.4 Memory size limiting the model complexity

Both the weights and activations for the neural network are in this implementation stored as signed 64-bit integers, which are used for representing their actual floating point values. The various memory regions are rather limited in their sizes, which adds an upper boundary on the maximum complexities of the neural network models which can be stored.
As discussed in Section 4.2.1, placing the neural network in a faster memory region has a significant impact on the performance. Unfortunately, these faster regions are also smaller in size, which means that a slower memory regions is required when working with complex models.
CLS is the fastest memory region, not counting the tiny local memory, and is recommended when possible. Placing the model here requires a copy to be placed in the CLS of every island, otherwise there will be a significant performance loss because of the cross-island memory accesses. The same is true when placing the model in CTM.
Table 4.3 shows that it is possible to implement complex neural network models in the cards without having to use memory regions outside of the islands. Of course, utilizing models as complex as these would result in a terrible packet forwarding performance, unless the inference is triggered very rarely. More of these measurements can be seen in the expanded Table B.3.

Table 4.3: Examples of memory usages while running various model complexities. Every model has 5 input neurons and 5 output neurons. CTM+CLS means that the model weights are placed in CTM, while activations are stored in CLS

Hidden layers	**Neurons per hidden**	**Mem placement**	**CLS**	**CTM**
2	55	All CLS	93.40%	50.00%
2	120	CTM+CLS	98.36%	86.44%
3	40	All CLS	96.75%	50.00%
3	80	CTM+CLS	86.44%	92.25%
4	30	All CLS	89.43%	50.00%
4	70	CTM+CLS	98.16%	97.87%

4.5 Impact of feature collection on packet latency

The total packet latencies while collecting various flow metadata has been measured. These values, presented in Table 4.4, were all recorded with direction and time functionality enabled, meaning that the firmware is identifying if an incoming packet is going towards the client or server, and has stored a timestamp in local memory for fast retrieval. The SmartNIC is only performing packet forwarding and metadata recording, neural network inference is disabled during these measurements.

The total number of packets in each flow is also recorded during every test, the reason being that this value is used to only perform certain actions for the first few packets in each flow. An average latency for 5000 packets is what is presented in the table, after already having sent a few flow packets to not include the flow initialization in this data.

Table 4.4: Measured packet latencies while collecting various features. See table 3.1 for a full list of extracted flow metadata

Features extracted	**Mean packet latency (ns)**	**Additional delay (ns)**
None	6471	N/A
Everything	8318	+1847
ttl_forward	6600	+129
lastPktTime	6631	+160
pktCount_forward	6501	+30
totalFlowData_forward	6515	+44
isn_forward	6508	+37
maxPktSize_forward	6637	+166
tcpWin_forward	6601	+130
minPktSize_forward	6633	+162
retrans_forward	6656	+185
firstPktTime	6472	+1
duration & firstPktTime	6762	+291
handshakeTimes	6631	+160

5 Conclusion

A framework for performing neural network inference on Netronome NFP-4000 flow processors has been developed and tested for various neural network complexities.

The results show that neural networks of a low-medium complexity can perform real time inference while keeping an acceptable packet processing rate. Very complex models are possible to implement, but these come at a high performance cost. Combining these complex models with the cloning trick discussed in Section 3.6, while only performing inference once per traffic flow could yield a good throughput in the switch.

The results presented in this thesis are all recorded when the neural network is performing inference for every 20:th packet, which might not translate well to real-world scenarios. A more realistic scenario might be to restrict inference to once per traffic flow, after some specified number of flow packets has been included in the flow metadata. This would most likely drastically improve packet throughput over all, since traffic flows with a lot of packets would only require neural network inference once instead of multiple times.

References

[1] Gianni Antichi et al. “OSNT: Open source network tester”. In: *IEEE Network Magazine* 28.5 (2014), pp. 6–12.

[2] Tom Auld, Andrew W Moore, and Stephen F Gull. “Bayesian neural networks for internet traffic classification”. In: *IEEE Transactions on neural networks* 18.1 (2007), pp. 223–239.

[3] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[4] Johan Garcia et al. “Towards Video Flow Classification at a Million Encrypted Flows Per Second”. In: *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. IEEE. 2018, pp. 358–365.

[5] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, 2011, pp. 315–323. URL: `http://proceedings.mlr.press/v15/glorot11a.html`.

[6] The P4.org Architecture Working Group. *P416 Portable Switch Architecture (PSA)*. 2019. URL: `https://p4.org/p4-spec/docs/PSA.html` (visited on 05/14/2019).

[7] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[8] Knut Hinkelmann. *Neural Networks*. University of Applied Sciences Northwestern Switzerland. URL: `http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay_1718:ke-11_neural_networks.pdf` (visited on 05/15/2019).

[9] Tin Kam Ho. “Random decision forests”. In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282.

[10] Elike Hodo et al. “Threat analysis of IoT networks using artificial neural network intrusion detection system”. In: *2016 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. 2016, pp. 1–6.

[11] Ahmad Javaid et al. “A deep learning approach for network intrusion detection system”. In: *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. ICST (Institute for Computer Sciences, Social-Informatics and . . . 2016, pp. 21–26.

[12] Bekir Karlik and A Vehbi Olgac. "Performance analysis of various activation functions in generalized MLP architectures of neural networks". In: *International Journal of Artificial Intelligence and Expert Systems* 1.4 (2011), pp. 111–122.

[13] Alex Krizhevsky and Geoff Hinton. "Convolutional deep belief networks on cifar-10". In: *Unpublished manuscript* 40.7 (2010).

[14] Antonino Laudani et al. "On Training Efficiency and Computational Costs of a Feed Forward Neural Network: A Review". In: *Intell. Neuroscience* 2015 (Jan. 2015), 83:83–83:83. ISSN: 1687-5265. DOI: `10.1155/2015/818243`. URL: `https://doi.org/10.1155/2015/818243`.

[15] scikit learn. *Support Vector Machines*. URL: `https://scikit-learn.org/stable/modules/svm.html` (visited on 05/14/2019).

[16] Mohammad Lotfollahi et al. "Deep packet: A novel approach for encrypted traffic classification using deep learning". In: *arXiv preprint arXiv:1709.02656* (2017).

[17] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[18] Netcope. *P4 to VHDL*. 2019. URL: `https://www.netcope.com/en/products/p4-to-vhdl` (visited on 05/14/2019).

[19] Netronome. *NFP-4000 Theory of Operation*. URL: `https://www.netronome.com/m/documents/WP_NFP4000_TOO.pdf` (visited on 05/14/2019).

[20] Netronome. *Programming Netronome Agilio® SmartNICs*. URL: `https://www.netronome.com/m/documents/WP_NFP_Programming_Model.pdf` (visited on 05/14/2019).

[21] Robert HB Netzer and Barton P Miller. "What are race conditions?: Some issues and formalizations". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pp. 74–88.

[22] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[23] Alan Saied, Richard E Overill, and Tomasz Radzik. "Detection of known and unknown DDoS attacks using Artificial Neural Networks". In: *Neurocomputing* 172 (2016), pp. 385–393.

[24] Cole Schlesinger, Michael Greenberg, and David Walker. "Concurrent NetCore: From policies to pipelines". In: *ACM SIGPLAN Notices*. Vol. 49. 9. ACM. 2014, pp. 11–24.

[25] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[26] Alex Shenfield, David Day, and Aladdin Ayesh. “Intelligent intrusion detection systems using artificial neural networks”. In: *ICT Express* 4.2 (2018). SI on Artificial Intelligence and Machine Learning, pp. 95 –99. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2018.04.003. URL: http://www.sciencedirect.com/science/article/pii/S2405959518300493.

[27] Zhanyi Wang. “The applications of deep learning on traffic identification”. In: *Black-Hat USA* 24 (2015).

[28] Eric W. Weisstein. *Hyperbolic Tangent*. MathWorld–A Wolfram Web Resource. URL: http://mathworld.wolfram.com/HyperbolicTangent.html (visited on 05/17/2019).

[29] Eric W. Weisstein. *Sigmoid Function*. MathWorld–A Wolfram Web Resource. URL: http://mathworld.wolfram.com/SigmoidFunction.html (visited on 05/17/2019).

[30] Stuart Wray. *The Joy of Micro-C*. Open-NFP. URL: https://open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf (visited on 06/11/2019).

[31] Chuanlong Yin et al. “A deep learning approach for intrusion detection using recurrent neural networks”. In: *Ieee Access* 5 (2017), pp. 21954–21961.

[32] Qian Yu et al. “Sketch-a-net that beats humans”. In: *arXiv preprint arXiv:1501.07873* (2015).

[33] Noa Zilberman et al. “NetFPGA SUME: Toward 100 Gbps as research commodity”. In: *IEEE micro* 34.5 (2014), pp. 32–41.

Appendix

A Setting up a host for the SmartNIC card

To use a Netronome SmartNIC, you need to install a host server for the card to be plugged into. This host is used to power, flash firmware, and configure the card.
We used Dell Optiplex 755 running Ubuntu Server 18.04 to host the cards.

A.1 Host hardware

We do not recommend using the same server models that we used for hosts. The PCI ports lack SR-IOV support, which seems to make it impossible to forward packets from the Netronome card to an interface on the host using virtual functions.
Make sure that your host has at least PCI Gen3 x8 (or x16) slots, ideally with SR-IOV support if you want to send packets between the card and host.

A.2 Host installation

The first step is to install an operating system.
At the time of writing, we would recommend using either Ubuntu 18.04 or CentOS 7. This is because the pre-compiled binaries you need are only available for Ubuntu and CentOS/Red Hat. This guide is for Ubuntu, but the setup for CentOS is very similar.

You need to install the NFP kernel module, like so:

```
$ wget https://deb.netronome.com/gpg/NetronomePublic.key
$ sudo apt-key add NetronomePublic.key
$ apt-get update
$ apt-get install nfp
```

You need to download *NFP SDK6 Run Time Environment* and *Board Support Package* from the Netronome support website, which in my case is 'nfp-sdk-p4-rte-6.1.0.1-preview-3214.ubuntu.x86_64.tar' and 'nfp-bsp-6000-b0_2018.06.29.1443-1_amd64.deb'.

Before you continue, there's some dependencies that are needed. They can be installed like so:

```
$ sudo apt-get install dkms build-essential libjansson4 libftdi1
```

Install the BSP:

```
$ sudo dpkg -i nfp-bsp-6000-b0_2018.06.29.1443-1_amd64.deb
```

Next, install the RTE and reboot:

```
$ tar xf nfp-sdk-p4-rte-6.1.0.1-preview-3214.ubuntu.x86_64.tar
$ cd nfp-sdk-6-rte-*
$ sudo ./sdk6_rte_install.sh install
$ sudo init 6
```

Every time that the host has rebooted or changed some configuration you have to reload the kernel module and restart services to be able to load firmware. It is easiest to create a script to automate this.
Here is the final version of the script we used:

```
#!bin/bash
echo "Unloading old firmware..."
sudo /opt/netronome/bin/nfp-nffw unload
sudo pkill pif_rte
echo "Stopping services..."
sudo systemctl stop nfp-sdk6-rte
sudo systemctl stop nfp-hwdbg-srv
echo "Reloading kernel modules..."
```

```
sudo depmod
sudo modprobe -r nfp
sudo modprobe nfp nfp_dev_cpp=1
sudo modprobe devlink
echo "Starting services..."
sudo systemctl restart nfp-sdk6-rte
sudo systemctl restart nfp-hwdbg-srv
```

Listing 1: setup.sh

A.2.1 Splitting the ports

The SmartNIC card has two 1x40G ports. We are splitting these into 4x10G virtual ports using a breakout module from 1x40G QSFP+ to 4x10G SFP.
You configure the physical port for splitter-mode like so:

```
$ sudo /opt/netronome/bin/nfp-media phy0=4x10G
$ sudo /opt/netronome/bin/nfp-media phy1=4x10G
```

The ordering is important. It is not possible to split phy1 without already having split phy0, but you can split phy0 without also splitting phy1.
We have encountered a lot of problems with splitting the ports, and would recommend you to reload the kernel modules every time you make a change to the splitting configuration:

```
$ sudo modprobe -r nfp
$ sudo depmod -a
$ sudo modprobe nfp nfp_dev_cpp=1
$ sudo modprobe devlink
```

A.3 Compiling and loading firmware from the host

It is possible to compile P4 and load the generated firmware from within the host system. You will need to download the SDK from the Netronome support website. The file that we downloaded was called 'nfp-sdk_6.1.0.1-preview-3243-2_amd64.deb'.

```
$ sudo dpkg -i nfp-sdk_6.1.0.1-preview-3243-2_amd64.deb
```

After installing the SDK, you have to add a missing symlink like so:

```
$ sudo ln -s /opt/netronome/p4/bin/p4c-bm2-ss \
 /opt/netronome/p4/libexec/
```

The host should now be ready to compile P4 firmware.

A.3.1 Compiling P4 code from Linux

To compile a P4_16 file called 'main.p4', you do the following:

```
$ /opt/netronome/p4/bin/nfp4build -o output/firmware.nffw \
 -l beryllium ..nfp4c_I /opt/netronome/p4/include/16/p4include/ \
  --nfp4c_p4_version 16 -4 main.p4
```

For P4_14, you do:

```
$ /opt/netronome/p4/bin/nfp4build -o output/firmware.nffw \
-l beryllium -4 main.p4
```

There should now be a directory './output/' that is populated with a bunch of files. The most important ones are 'firmware.nffw' which is the actual firmware, and 'pif_design.json'.

A.3.2 Loading firmware to card

In addition to the files created in Section A.3.1, you also need to include a configuration file 'config.p4cfg' with your rules and tables.

We ended up with the following script to load the firmware, design, and configuration to the card:

```bash
#!/bin/bash
DIR=$(pwd)
LOG_FILE_RUN=$DIR/out_load #output from this run
LOG_FILE_MAIN=/var/log/nfp-sdk6-rte.log
FIRMWARE_FILE=$DIR/output/firmware.nffw #path to firmware
DESIGN_FILE=$DIR/output/pif_design.json #path to design
CONFIG_FILE=$DIR/config.p4cfg #path to config

#restart services
sudo systemctl stop nfp-sdk6-rte
sudo systemctl stop nfp-hwdbg-srv

#unload old firmware
sudo /opt/netronome/bin/nfp-nffw unload
sudo pkill pif_rte

#load firmware and rules to card
pushd /opt/nfp_pif/bin/ > /dev/null
  sudo ./pif_rte -n 0 -p 20206 -I -z \
  -s /opt/nfp_pif/scripts/pif_ctl_nfd.sh \
  -f $FIRMWARE_FILE -d $DESIGN_FILE -c $CONFIG_FILE \
   --log_file $LOG_FILE_MAIN > $LOG_FILE_RUN &
popd > /dev/null
```

Listing 2: load_firmware.sh

After running the script, you can verify that the firmware loaded successfully by checking the log file located at '/var/log/nfp-sdk6-rte.log'.

B Raw data

B.1 Packet processing rate for various neural network configurations

The measurements in Table B.1 contains packet processing rates for the latest implementation of the neural network. Network traffic came in to the switch through one 10Gbit port, and was forwarded to another 10Gbit port. The neural network inference was triggered every 20:th incoming packet.
Table B.2 is the same as B.1, except the traffic is forwarded to/from 4x10Gbit ports instead.

Table B.1: Packet processing rate while running various neural network configurations. Input and output layers contain 5 neurons each. Triggered every 20:th packet. 1x10GbE for in-traffic and 1x10GbE out

Num hidden layers	Neurons per hidden	Weight region	Activ region	**KPPS**	Clone hack
2	10	CLS	CLS	**4570**	0
2	10	CTM	CTM	**4170**	0
2	10	CTM	CTM	**3980**	1
2	20	CTM	CTM	**2530**	1
2	20	CLS	CLS	**1960**	0
2	20	CTM	CTM	**1780**	0
2	30	CLS	CLS	**1010**	0
2	30	CTM	CTM	**920**	0
2	55	CLS	CLS	**345**	0
2	55	CTM	CLS	**335**	0

Continued on next page

Table B.1 – continued from previous page

Num hidden layers	Neurons per hidden	Weight region	Activ region	**KPPS**	Clone hack
2	80	CTM	CTM	**550**	1
2	80	CTM	CTM	**155**	0
2	120	CTM	CLS	**75**	0
3	10	CTM	CTM	**3100**	1
3	10	CLS	CLS	**2920**	0
3	10	CTM	CLS	**2870**	0
3	10	CTM	CTM	**2670**	0
3	10	CLS	CTM	**2620**	0
3	10	IMEM	iMEM	**2135**	0
3	15	CTM	CTM	**2380**	1
3	15	CLS	CLS	**1690**	0
3	15	CTM	CTM	**1510**	0
3	15	IMEM	IMEM	**1210**	0
3	20	CTM	CTM	**1820**	1
3	20	CLS	CLS	**1070**	0
3	20	CTM	CLS	**1045**	0
3	20	CTM	CTM	**970**	0
3	20	CLS	CTM	**965**	0
3	20	IMEM	IMEM	**760**	0
3	25	CTM	CTM	**1430**	1
3	25	CLS	CLS	**720**	0
3	25	CTM	CTM	**670**	0
3	25	IMEM	IMEM	**515**	0
3	30	CTM	CTM	**1175**	1
3	30	CLS	CLS	**530**	0
3	30	CTM	CLS	**520**	0

Continued on next page

Table B.1 – continued from previous page

Num hidden layers	Neurons per hidden	Weight region	Activ region	**KPPS**	Clone hack
3	30	CTM	CTM	**475**	0
3	30	CLS	CTM	**475**	0
3	30	IMEM	IMEM	**375**	0
3	35	CTM	CTM	**985**	1
3	35	CLS	CLS	**410**	0
3	35	CTM	CTM	**370**	0
3	35	IMEM	IMEM	**280**	0
3	40	CTM	CTM	**850**	1
3	40	CLS	CLS	**320**	0
3	40	CTM	CLS	**310**	0
3	40	CTM	CTM	**285**	0
3	40	CLS	CTM	**285**	0
3	40	IMEM	IMEM	**225**	0
3	45	CTM	CTM	**740**	1
3	45	CTM	CTM	**230**	0
3	50	CTM	CTM	**630**	1
3	50	CTM	CTM	**190**	0
3	80	CTM	CLS	**85**	0
4	10	CLS	CLS	**2330**	0
4	20	CLS	CLS	**760**	0
4	30	CLS	CLS	**380**	0
4	30	CTM	CLS	**370**	0
4	45	CTM	CTM	**570**	1
4	45	CTM	CTM	**165**	0
4	50	CTM	CLS	**140**	0
4	70	CTM	CLS	**60**	0

Table B.2: Packet processing rate while running various neural network configurations. Input and output layers contain 5 neurons each. Triggered every 20:th packet. 4x10GbE ports for in-traffic and 4x10GbE ports out

Num hidden layers	Neurons per hidden	Weight region	Activ region	**KPPS**	Clone hack
2	10	CTM	CTM	**3976**	1
2	10	CLS	CLS	**3920**	0
2	10	CTM	CTM	**3588**	0
2	30	CTM	CTM	**1660**	1
2	30	CTM	CTM	**820**	0
2	120	CTM	CLS	**68**	0
3	10	CTM	CTM	**3116**	1
3	10	CTM	CTM	**2264**	0
3	30	CTM	CTM	**1060**	1
3	30	CTM	CTM	**424**	0
4	45	CTM	CTM	**516**	1
4	45	CTM	CTM	**144**	0
4	70	CTM	CLS	**64**	0

B.2 Memory usage for various neural network configurations

The measurements in Table B.3 are collected for the latest version of the implementation.

Table B.3: Measured memory usage for different neural network configurations. Input and output layers contain 5 neurons each

Num hidden layers	Neurons per hidden	Weight region	Activ region	CLS usage	CTM usage
2	10	CLS	CLS	24.73%	50.00%
2	50	CLS	CLS	83.33%	
2	55	CLS	CLS	93.40%	
2	55	CTM	CLS	48.35%	61.26%
2	70	CTM	CLS	57.14%	67.54%
2	80	CTM	CTM		97.53%
2	90	CTM	CLS	68.86%	78.03%
2	120	CTM	CLS	98.36%	86.44%
3	30	CLS	CLS	69.29%	
3	40	CLS	CLS	96.75%	
3	40	CTM	CLS	51.28%	61.37%
3	50	CTM	CTM	21.25%	90.84%
3	60	CTM	CLS	68.86%	74.37%
3	80	CTM	CLS	86.44%	92.25%
4	20	CLS	CLS	57.69%	
4	30	CLS	CLS	89.43%	
4	30	CTM	CLS	51.28%	59.54%
4	50	CTM	CLS	74.72%	75.04%
4	60	CTM	CLS	86.44%	85.54%
4	70	CTM	CLS	98.16%	97.87%