A Machine Learning Framework To Implement Network Function On Programmable Data Plane

MS (Research) Thesis

By
Sourabh Singh



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

MAY 2025

A Machine Learning Framework To Implement Network Function On Programmable Data Plane

A THESIS

Submitted in fulfillment of the
requirements for the award of the degree
of

Master of Science (Research)

by

Sourabh Singh



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

MAY 2025

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **A Machine Learning Framework To Implement Network Function On Programmable Data Plane** in the fulfilment of the requirements for the award of the degree of **MASTER OF SCIENCE (RESEARCH)** and submitted in the **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from July, 2023 to May, 2025 under the supervision of **Dr Aniruddha Singh Kushwaha.** The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.



30-07-25

Signature of the student with date

(Sourabh Singh)

This is to certify that the above statement made by the candidate is correct to the best of my/own knowledge.



30/07/25

Signature of the Supervisor of MS(Research) thesis with date

Dr Aniruddha Singh Kushwaha

Sourabh Singh has successfully given his/her MS (Research) Oral Examination held on 21-05-26



29-May-2026

Signature(s) of Thesis Supervisor with date

ACKNOWLEDGEMENTS

I want to sincerely thank everyone who helped me with my MS (Research) thesis, "A Machine Learning Framework to Implement Network Function on Programmable Data Plane."

I want to start by expressing my sincere gratitude to the Indian Institute of Technology Indore for giving me the chance, the tools, and the ideal setting in which to conduct this study.

I would like to express my gratitude to Dr. Aniruddha Singh Kushwaha, my thesis advisor, for his unwavering support, encouragement, and direction. His profound understanding, tolerance, and faith in my skills were crucial in determining the course and outcome of this work.

In addition, I want to express my gratitude to my lab partners for their cooperation, insightful conversations, and the welcoming environment they created. The time spent working alongside them was enriching and motivating.

Lastly, I want to thank my wonderful and encouraging family. I am grateful to my parents for their unwavering support and belief in me. They helped me at every stage of my life. Without the combined efforts of everyone listed above, this thesis would not have been feasible. Many thanks to all of you.

Sourabh Singh

To everyone who believed in me

Abstract

A novel machine learning framework is proposed in this thesis to enable network function deployment on the programmable data plane. Using a modular and unified approach, the framework finds a minimum set of reusable primitives able to support a wide class of ML models. This work is motivated by the growing demand for intelligent and programmable networks by fusing the machine learning capabilities directly into the network data plane. However, the present network infrastructure lacks hardware architecture and a framework to allow effective development and deployment of ML-based network functions on the hardware. To address this gap, we proposed a modular hardware framework that uses basic building blocks to construct a developed ML model on the hardware. These basic building blocks are known as primitives. In this work, we have illustrated the working of our framework for an ANN and Tree-based model, which are used to build and deploy a network function over the data plane. Furthermore, to reduce computational overhead, the framework incorporates different model approximation strategies. These approximations help to maintain a balance between accuracy and performance, ensuring feasibility in high-speed, resource-constrained network environments. Additionally, the framework is built to be scalable and programmable, such that a myriad of ML architectures can be supported as the framework evolves. This approach not only enhances real-time packet processing capabilities but also opens various new directions for intelligent in-network decision-making and autonomous network management. Overall, this work lays a foundational step toward enabling scalable, ML-driven, real-time processing in next-generation programmable network devices.

Publications

1. **S. Singh**, K. Kanhaiya, P. Magadum, R. Patel, A. S. Kushwaha and "Enabling Seamless Integration of ML-Based Network Functions into the Network Data-plane," IEEE Networking Letters (**Accepted**)

2. **S. Singh**, K. Kanhaiya, P. Magadum, A. S. Kushwaha and A. Mondal, 2024, December. A Primitive Based Machine Learning Framework for the Network Data-plane. In 2024 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS) (pp. 1-6). IEEE.

Table of Contents

Abstract **ii**

List of Figures **vii**

List of Symbols **viii**

List of Abbreviations **x**

1 Introduction **1**

2 Literature Survey **4**

2.1 Foundation 4

2.1.1 SDN 4

2.1.2 Data-Plane Programmability 5

2.1.3 Machine-Learning Programmability Over SDN 5

2.2 Machine Learning in the Control Plane 6

2.3 Machine Learning in the Data Plane 8

2.4 In-Network Computation for Distributed ML 9

2.5 Model Compression and Approximation 10

2.6 Summary 11

3 Proposed Method **12**

3.1 ML Framework For Programmable Data Plane 12

3.1.1 Software Framework 12

3.1.2 Hardware Framework 13

3.2 Design Methodology 14
3.2.1 Artificial Neural Network (ANN) Model 14
3.2.2 Tree-Based Model 20
3.3 Summary 23

4 Experiments and Results **24**
4.1 Experimental Setup 24
4.2 Dataset & Model Setup 25
4.3 Performance Evaluation 25
4.3.1 Results on ANN-based model 25
4.3.2 Results on Tree-based model 28

5 Conclusions and Future Scope **31**

List of Figures

2.1 Standard SDN architecture [12]. 5
2.2 PISA architecture overview [6]. 6

3.1 Proposed machine learning framework overview. 12
3.2 High-level design on programmable data plane. 13
3.3 Illustration of MLP baseline and hardware-level mapping strategy . . . 15
3.4 Multistage model transformation 16
3.5 Transformation of a Decision Tree into a Table-Based Primitive Representation. 21

4.1 Relationship between pruning and model efficiency 26
4.2 Relationship between latency and resource with feature size and neuron 27
4.3 Accuracy comparison between the baseline model and the primitive-based mapping. 28
4.4 Processing cycles required for inference. 29
4.5 Comparison of memory index jumps during decision path traversal. . . 30

List of Symbols and Notations

$M_{\mathbf{orig}}$	Original (unpruned) neural-network model.
$M_{\mathbf{pruned}}$	Model obtained after global neuron pruning.
M_i	i-th candidate model in $\mathcal{M}$.
M^*	Model selected by the score-and-cost search (or **None** if no model qualifies).
λ_1	Weight for the ℓ_2-norm term in a neuron's importance score.
λ_2	Weight for the activation-variance term in a neuron's importance score.
ρ	Global pruning ratio (fraction of neurons removed).
L	Total number of layers in the network.
N_l	Number of neurons in layer l ($1 \leqslant l \leqslant L$).
l	Layer index; l' denotes a dummy layer index in set operations.
j	Neuron index within layer l.
W_j^l	Weight vector of neuron j in layer l.
b_j^l	Bias term of neuron j in layer l.
a_j^l	Activation values of neuron j in layer l over the training set.
$\mathcal{N}$	Set of all neurons with their parameters $\{(l, j, W_j^l, b_j^l, a_j^l)\}$.
$\mathcal{N}'$	Subset of $\mathcal{N}$ retained after pruning.

$\phi_{l,j}$	Importance score of neuron (l, j), $\phi_{l,j} = \lambda_1 \lVert W_j^l \rVert_2 + \lambda_2 \operatorname{Var}(a_j^l)$.
$\lVert W \rVert_2$	Euclidean (ℓ_2) norm of a weight vector.
$\operatorname{Var}(a)$	Sample variance of a neuron's activations.
S_i	Predictive score/accuracy of model M_i (e.g., F1, AUC).
S_{th}	Minimum acceptable score threshold.
$\mathcal{M}_f$	Subset of $\mathcal{M}$ whose models meet S_{th}.
R_i	Triple of resource counts for model M_i: $R_i = (\#\text{Reg}_i, \#\text{Mul}_i, \#\text{Add}_i)$.
$\#\mathbf{Reg}_i$	Number of register (memory) units required by M_i.
$\#\mathbf{Mul}_i$	Number of multipliers/DSPs required by M_i.
$\#\mathbf{Add}_i$	Number of adders (ALU additions) required by M_i.
α	Weight assigned to register usage in total hardware cost.
β	Weight assigned to multiplier usage.
γ	Weight assigned to adder usage.
$\text{Score}(\cdot)$	Function that evaluates a model's predictive performance.

List of Abbreviations

SDN	Software Defined Networking
API	Application Programming Interface
PISA	Protocol-Independent Switch Architecture
P4	Programming Protocol-Independent Packet Processors
FPGA	Field Programmable Gate Array
ML	Machine Learning
DDoS	Distributed Denial of Service
ANN	Artificial Neural Network
QoS	Quality Of Service
GNN	Graph Neural Network
CNN	Convolutional Neural Network
ISP	Internet Service Provider
TE	Traffic Engineering
IDS	Intrusion Detection System
WAN	Wide Area Network
RGNN	Recurrent Graph Neural Network

HP	Hardware Primitive
SP	Software Primitive
MSB	Most Significant Bit
DRL	Deep Reinforcement Learning
ASIC	Application-Specific Integrated Circuit
NetFPGA	Network Field-Programmable Gate Array
BMV2	Behavioral Model version 2
LSTM	Long Short-Term Memory
CIC_IDS	Canadian Institute for Cybersecurity
UNSW_NB15	University of New South Wales Network-Based Dataset
TON_IoT	Telemetry and Operational Network Internet of Things Dataset
NSL_KDD	Network Security Laboratory Knowledge Discovery in Databases
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
DSP	Digital Signal Processing Unit
ALU	Arithmetic Logic Unit
REST	Representational State Transfer
SIMD	Single Instruction, Multiple Data
LUT	Look-Up Table
FF	Flip-Flop

Chapter 1

Introduction

Software-Defined Networking (SDN) reshapes network design by decoupling the control and data planes and centralising decision-making [5]. A logically central controller keeps a global view of a network that includes topology, traffic, and policy. The controller uses this information in programming switches through southbound APIs such as OpenFlow [11]. This separation accelerates the rollout of new algorithms and policies without touching existing hardware. While control-plane logic is easily updated in software, the data plane long remained fixed-function, offering little room for innovation. The Protocol-Independent Switch Architecture (PISA)[1] and the P4 language[2] changed this landscape by enabling operators to program packet parsing and processing on specialised hardware—such as FPGAs and Barefoot Tofino ASICs, bringing true programmability to the data plane.

Machine-learning techniques are now woven into nearly every domain, and SDN's control plane is no exception. Researchers have already applied ML to a wide range of controller-side tasks—for example, deep-reinforcement-learning agents that steer traffic to balance load and minimise latency [15], supervised classifiers that detect intrusions and volumetric DDoS attacks in real time [7], graph-neural-network models that predict link failures and trigger proactive rerouting [4], and time-series forecasters that anticipate traffic surges and adjust resource allocations accordingly [9]. These successes highlight the power of software-based intelligence. However, machine-learning deployments *inside* the data plane remain rare and narrowly focused. The few existing

efforts concentrate almost exclusively on two tasks, like heavy flow prediction, which flags elephant flows so that they can be metered or routed early [23], and anomaly or attack detection, the P4-based adversarial-traffic filter [16]. Both systems implement decision-tree classifiers in customised hardware pipelines; inference reduces to a handful of table look-ups, eliminating the need for real-time arithmetic and allowing line-rate execution without general-purpose compute. As a result, today's in-switch ML solutions are very task-specific and representative of the wider ML toolkit.

To embed true intelligence in the network, the existing infrastructure must evolve to automate decision-making, fault tolerance, intrusion prevention, and even hardware reconfiguration. Our analysis reveals a critical gap: current SDN/P4 architectures lack a *unified, programmable ML framework* that spans *both* the control and data planes. This work takes a first step toward bridging this gap. The contribution on this work are as following:

- **A ML framework to deploy network function in data plane.** We introduce a novel SDN-compatible programmable framework that allows operators to deploy machine-learning functions in the network *and* data plane without changing the underlying hardware. This framework works on the basic building blocks, which can be arranged to built a developed model. These building blocks are known as primitives.

- **Primitive identification, definition and database.** We identify and formally define the minimal set of primitives required to build ML models. We illustrate the process by building artificial neural network (ANN) and decision-tree-based models within programmable pipelines.

- **Optimisation** Further, a simulation toolchain benchmarks each primitive and explores approximation knobs. This process helps in obtaining resource-efficient configurations that meet line-rate and latency constraints.

To the best of our knowledge, this work is the first attempt to provide a unified platform for deploying machine-learning models across both the control plane and the

data planes. This thesis presents the framework's preliminary design and validates its feasibility. Although the framework cannot yet host every possible ML architecture, it is expressly built to evolve. By gradually integrating new computational primitives, the platform will support an expanding range of applications.

Chapter 2

Literature Survey

This chapter surveys the body of work that motivates the primitive–based machine-learning (ML) framework proposed in this thesis. We begin with foundational enablers, then review ML on the SDN controller and in the data plane, and finally discuss related advances in in-network computation and model compression.

2.1 Foundation

2.1.1 SDN

Software-Defined Networking (SDN) decouples the control and data planes, transforming switches to fast packet forwarders while moving all decision logic to a logically central controller. The controller provides a unified, programmable view of network topology, performance metrics, and policy enforcement, and offers sophisticated north-bound APIs tailored for applications like traffic engineering, security orchestration, and telemetry analytics. South-bound protocols—most prominently OpenFlow [11] and the vendor-neutral P4Runtime—translate these intents into match-action rules installed on every switch. This architecture accelerates feature roll-outs, enables network-wide optimisation, and lowers operational cost through automated policy enforcement. Its logical centrality also provides the data breadth needed to power machine-learning driven control decisions.

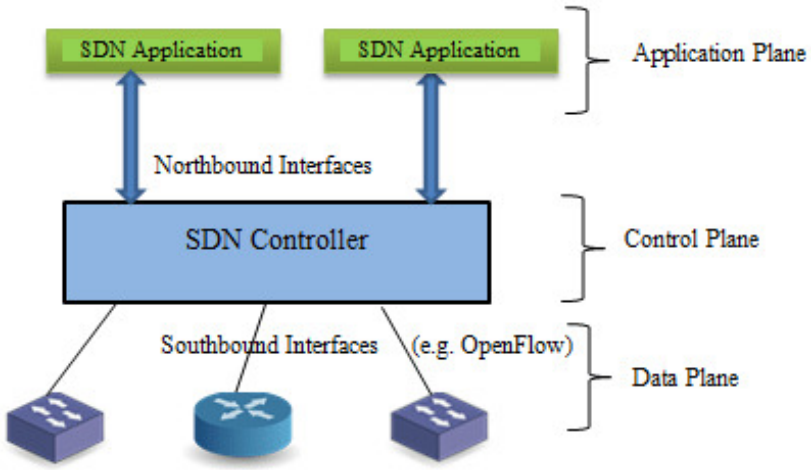


Figure 2.1: Standard SDN architecture [12].

2.1.2 Data-Plane Programmability

While SDN enabled flexible control-plane logic, traditional data planes remained fixed-function with limiting innovation. The emergence of the Protocol-Independent Switch Architecture (PISA) and the P4 language changed this paradigm by allowing the data plane itself to be programmed. P4 defines packet processing via programmable blocks—parsers, match–action tables, and deparsers—that can be mapped to both software (e.g., BMv2) and hardware (e.g., Tofino, NetFPGA) targets. Programmability empowers tailored packet-forwarding logic, built-in network telemetry, and flexible pipelines that remain agnostic to specific protocols. Importantly, it opens the door to deploying domain-specific logic—including machine learning inference—directly within the forwarding path, under strict resource and latency constraints.

2.1.3 Machine-Learning Programmability Over SDN

Researchers have investigated the integration of machine learning (ML) throughout the SDN stack, uses the enhanced flexibility of the control plane and the adaptability of the programmable data plane. Initial works were concentrated on intelligence at the controller level, where comprehensive visibility and latency limitations facilitate intricate models for functions such as traffic forecasting, intrusion detection, and Quality

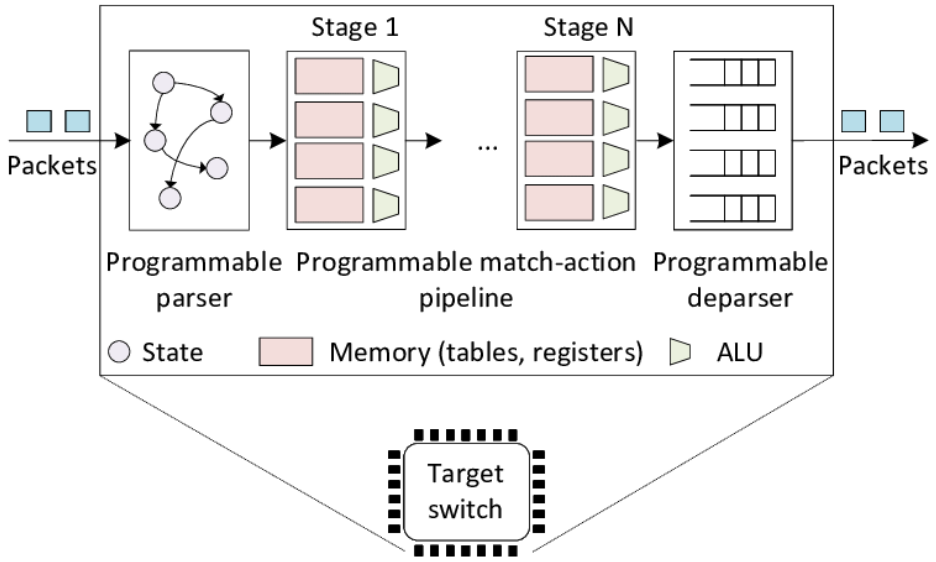


Figure 2.2: PISA architecture overview [6].

of Service enforcement. Recent advancements in P4-enabled hardware have facilitated lightweight machine learning inference directly within the data plane, but constrained by stringent memory and processor limitations. This growth necessitates hardware-aware, modular machine learning abstractions. This thesis presents a framework that further this trajectory by incorporating primitive-based machine learning components applicable in both domains.

2.2 Machine Learning in the Control Plane

Extensive efforts have been made to apply artificial intelligence and machine learning techniques to traffic classification, routing, anomaly detection and policy based shaping within the network control plane.

EfficientTE [15] tackles WAN traffic engineering by pairing a Graph Neural Network, which learns spatial traffic patterns, with a Deep-RL agent that continuously selects low-congestion routes. This closed loop adapts in seconds to traffic bursts or link failures and, in published WAN traces, trims average end-to-end latency by about 30% . The work illustrates how an SDN controller can replace static LP heuristics with learning-driven, near-real-time optimisation.

RouteNet [17] RouteNet introduces a graph-neural-network (GNN) approach to network modelling that captures how topology, routing policy, and traffic load jointly shape performance. By treating the network as a graph and passing messages between links and flows, the model learns to predict per-source/destination delay distributions—including mean delay, jitter—and packet-loss probabilities, all at a fraction of the cost of packet-level simulation. Crucially, its GNN architecture lets it generalise to unseen topologies, routing schemes, and traffic intensities: in tests, its worst-case mean-relative error remained below 16%. These accurate, low-overhead KPI estimates enable practical "what-if" analyses for tasks such as routing optimisation and capacity planning in self-driving SDN environments.

DL-IDS-SDN [10] addresses the single-controller vulnerability of Software-Defined Networks by incorporating a multi-stage, deep-learning intrusion-detection pipeline directly within the control plane. A Deep Convolutional GAN (DCGAN) improves minority attack classes, addressing the substantial data imbalance that negatively impacts classifier recall. Thereafter, a lightweight CenterNet module extracts spatial-temporal flow patterns from enhanced datasets. The features are employed by a ResNet-152V2 backbone, with its weights fine-tuned via the Slime Mold Algorithm (SMA) to improve convergence and reduce overfitting. The model is subsequently trained and assessed utilizing InSDN and Edge-IoT traces, continuously exceeding baseline CNN and LSTM IDS by substantial margins. Upon detecting malicious traffic, an integrated defense module enforces OpenFlow rules that either terminate or restricts the offending flows, restoring normal communication within milliseconds. Extensive studies confirm the system's robustness, enhancing detection accuracy and rapid remediation across many attack scenarios, including DDoS, spoofing, and API exploitation.

DeepSTN-QoS [24] focuses on anticipatory QoS in SDN by *forecasting* near-term (5–10 min) traffic loads instead of reacting after congestion appears. It combines 3-D convolutions, which capture how bursts ripple across adjacent links, with gated recurrent units that track short-term temporal trends. The model's per-link load predictions feed a lightweight optimiser in the controller that pre-emptively adjusts

queue weights and meters. Tested on a three-month ISP trace, DeepSTN-QoS cut prediction error to $< 4\%$ of link capacity and, when closed-looped, reduced peak-time packet loss by 40% while boosting throughput by 18%. In short, it turns QoS from a reactive to a proactive control-plane function.

2.3 Machine Learning in the Data Plane

This subsection highlights the works done towards the application of AI/ML model in network data plane.

pHeavy [23] introduces a decision-tree-based system for detecting heavy flows (a.k.a. elephant flows) directly in the data plane. By translating the trained decision tree into P4 match–action rules, the design executes entirely at line rate on both BMv2 and Tofino platforms. The system achieves over 97% detection accuracy while using less than 1% of chip memory. This work demonstrates the feasibility of deploying ML-based classifiers in programmable switches and shows that decision-tree logic can be restructured to comply with the stateless and deterministic execution model of PISA targets. It also emphasizes accuracy-resource trade-offs for on-chip inference in network measurement tasks.

P4-NIDS [3] presents a lightweight intrusion detection system that implements the decision tree algorithm as a chain of match–action stages within a P4 program. Each internal node of the decision tree corresponds to a table that evaluates one feature, effectively enabling feature-wise flow classification in hardware. This pipeline is tailored to maintain microsecond-level latency, making it suitable for real-time threat detection directly within the data plan. This work already being tested on multiple public intrusion datasets, the implementation shows competitive detection rates compared to software-based systems, while remaining deployable on production P4 switches. This work underscores the potential of deterministic tree traversal for in-network security.

Wang *et al.* [22] presents a packet-classification scheme that embeds an entire random-forest ensemble directly within the match–action pipeline of a P4-programmable hardware switch. Their design translates each decision tree into a

sequence of table look-ups, enabling more than 100 trees to operate concurrently without exceeding pipeline depth. Prototype measurements on a 100 Gbps Tofino switch confirm wire-speed throughput and deterministic per-packet latency. The authors also supply closed-form expressions for the exact number of match–action rules required, providing clear sizing guidelines for future deployments. This work demonstrates that sophisticated machine-learning classifiers can be realised in the data plane without sacrificing line-rate performance.

These works collectively demonstrate that decision-tree-based machine learning models can be implemented in data plane hardware within stringent latency and memory limitations. Nevertheless, each solution is intricately linked to a particular task and tree configuration. The reusability and extensibility across models are constrained, highlighting the necessity for a more modular and portable abstraction layer for machine learning in the data plane. This thesis presents a framework that fulfills this requirement by breaking down intricate models into hardware-compatible primitives that are applicable across various tasks and platforms.

2.4 In-Network Computation for Distributed ML

SwitchML [18] pioneers in-network acceleration for distributed machine learning by offloading the All-Reduce operation–commonly used for synchronising gradients in data-parallel training, into programmable switches. Rather than sending raw gradients between nodes, SwitchML performs integer-based aggregation directly in the dataplane, significantly reducing the volume of inter-server communication. This yields up to a 5× speedup in training convergence time, without modifying the ML model or training loop.

Building on SwitchML's insight, Q-Switch—realised by Li *et al.* as THC [8]—adds two key refinements: (i) *non-uniform gradient quantisation* to shrink packet sizes even further, and (ii) an *error-feedback accumulator* that injects the quantisation residue back into subsequent updates. These mechanisms let the switch operate on 4-bit or 8-bit integers while preserving the convergence behaviour of 32-bit training. Evaluations

on ResNet-50 show that THC achieves similar accuracy to full-precision baselines yet consumes markedly less bandwidth; combined with in-switch aggregation, this translates into an additional 1.4× reduction in time-to-accuracy on a 100Gbps Tofino switch..

2.5 Model Compression and Approximation

Deploying machine learning models on hardware-constrained platforms such as ASICs, FPGAs, or programmable switches necessitates aggressive model compression without sacrificing critical performance. Three classes of techniques are particularly effective for this purpose:

Structured pruning removes entire neurons, filters, or channels rather than individual weights. This not only reduces memory usage but also retains regular computation patterns that are friendly to SIMD and pipelined hardware architectures (e.g., RL-Pruner [21]).

Quantisation compresses model parameters by converting floating-point data to low-bitwidth integers (e.g., 4-bit or 8-bit), hence diminishing memory bandwidth and facilitating the utilization of integer-only arithmetic pipelines.

Low-rank factorization and sparsity breakdown extensive weight matrices into smaller components or eliminate superfluous values, hence diminishing the requisite multiplications and utilization of DSP units [19].

These methods form the foundation of the optimization stage in the proposed primitive-based ML framework, ensuring feasibility under strict performance, memory, and latency constraints.

2.6 Summary

The integration of Software-Defined Networking (SDN) with programmable data planes via P4 has revolutionized packet processing from fixed rule sets to entirely flexible pipelines. Although machine learning in the control plane has advanced–utilizing global state and reduced latency constraints for functions such as traffic engineering, anomaly detection, and QoS optimization, machine learning in data-plane is still limited to manually designed, task-specific classifiers. Nonetheless, the efficacy of systems such as SwitchML and Q-Switch demonstrates that even limited arithmetic capabilities in switches may produce significant machine learning acceleration. Moreover, advancements in model compression methods, such as pruning and quantization, illustrate that intricate models may be tailored for hardware-constrained settings. This thesis advances these developments by presenting a modular, primitive-based machine learning framework that is both portable and resource-efficient across software-defined networking controller and data-plane systems.

Chapter 3

Proposed Method

In this section, we present our proposed framework. This framework can be divided into:

- A Software Framework
- A Hardware Framework.

3.1 ML Framework For Programmable Data Plane

3.1.1 Software Framework

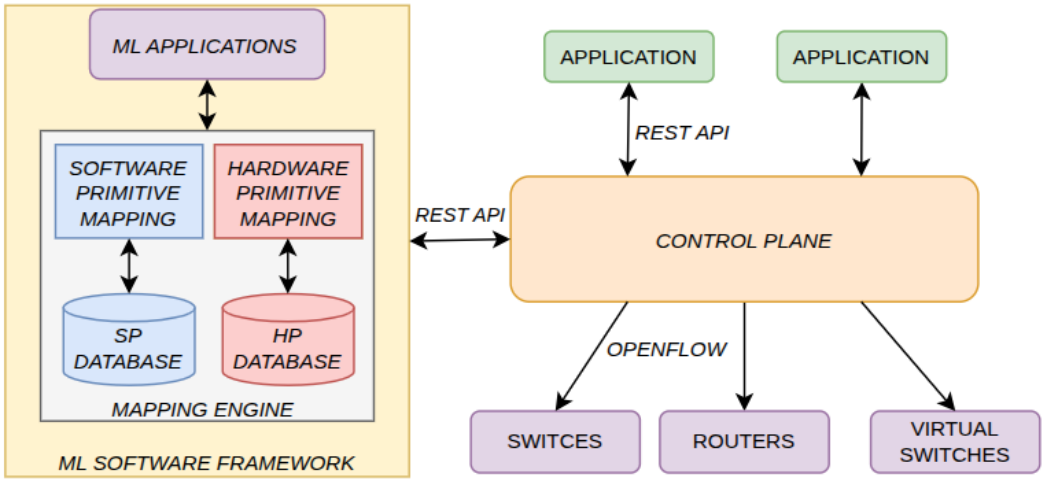


Figure 3.1: Proposed machine learning framework overview.

The software framework interfaces with the control plane through REST APIs and executes a machine learning (ML) application, as seen in Fig. 3.1, that employes libraries such as PyTorch and sklearn for ML model training and development. Subsequent to the generation of the ML model, the trained model is transmitted to the *Mapping Engine* for higher-level abstraction and mapping. The Mapping Engine consists of four components: the Software Primitive (SP) database, which manages the foundational elements employed in machine learning models; the Software Primitive Mapper, which integrates and maps these elements into the overall model; the Hardware Primitive (HP) database, which stores the hardware equivalents of the software elements; and the Hardware Primitive Mapper, which converts the software model to the hardware and generates configuration specifications for deployment. The comprehensive solution ensures efficient model development and conversion between software and hardware components.

3.1.2 Hardware Framework

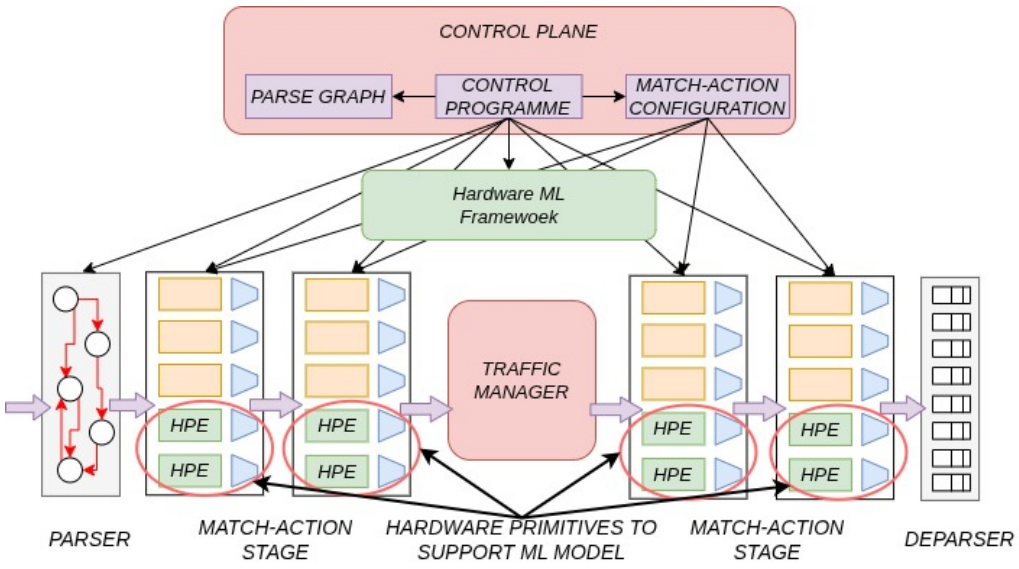


Figure 3.2: High-level design on programmable data plane.

The hardware framework employs network programmability to execute machine learning (ML) inference models on hardware. The modular hardware primitives (HPs) are designed for reprogramming, that enables us adapte with various ML models or

network functions. The hardware framework interfaces with the software framework through the SDN controller, acquire information on hardware primitive mappings and other details to implement the inference model on the data plane. Here the main aim of the framework is to synchronize the HP model with the pipeline stages of the data plane, such that it can utilize a PISA-like pipelined architecture that includes several stages at both the ingress and egress, as depicted in Fig. 3.2. The hardware primitive blocks are situated at various pipeline stages, and their arrangement either in cascade or parallel is determined by the software and hardware mappers to implement the ML model.

3.2 Design Methodology

To acquire a better grasp of our method, we start by investigating the system using two different machine learning architectures: Artificial Neural Networks (ANNs) and tree-based models. We describe the production of primitives, the fundamental building blocks of our framework, for each design. Then, we will explore model improvement and mapping methodologies to guarantee effective deployment and execution.

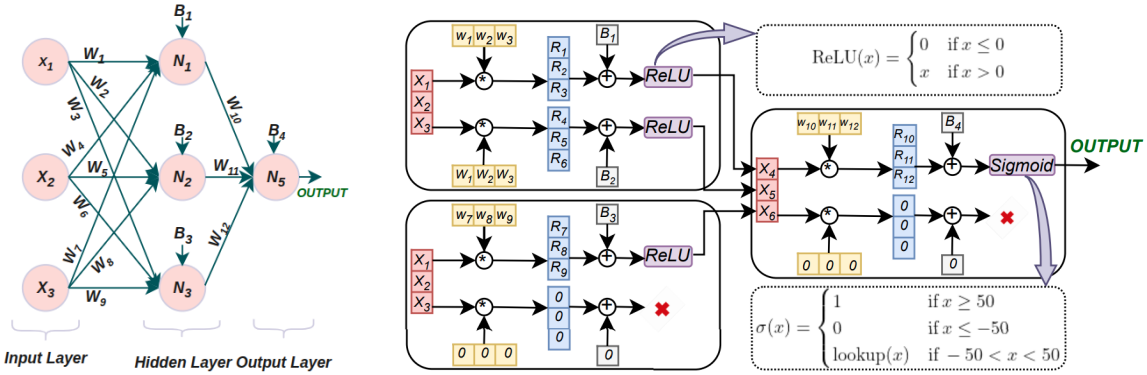
3.2.1 Artificial Neural Network (ANN) Model

In this section, we examine the construction of primitives for architectures based on artificial neural networks (ANN). We concentrate on the decomposition of neuron operations into hardware-compatible components, subsequently addressing methodologies for their efficient optimization and mapping.

3.2.1.1 Primitive Construction

Artificial neural networks (ANNs) are the building blocks of nearly all machine learning models, including Graph Neural Networks, Convolutional Neural Networks, and simple regression models. In this section, we will show how our framework works by building the ANN-based inference model.

As a foundational step, we begin by identifying and defining primitives for the ANNs. Fig. 3.3a illustrates a basic ANN model, from which two key primitives can be easily recognized: 1) Primitive for Neurons and 2) Primitive for the activation functions. Such that we can see mapped MLP as shown in fig. 3.3b



(a) Multilayer perceptron architecture

(b) Compatible hardware-aware mapping of multiLayer perceptron architecture

Figure 3.3: Illustration of MLP baseline and hardware-level mapping strategy

Primitive for Neuron Computation on neuron can be expressed using the following formula:

$$\text{Neuron Output} = \sum_{i=1}^{n} (I_i \cdot W_i) + \text{Bias}$$

where I_i are the input features, W_i are the weights, n is the number of features, and *Bias* is a constant parameter. The neuron output computation has two important operations: 1) Floating point multiplications– required to multiply input features with their respective weights and, 2) Floating point additions–to add all the multiplications and *Bias*. Every multiplication requires two registers for operands and one register for the result. Therefore, primitive complexity is dependent on the number of ALU operations and the primitive size n. We need to carefully choose the value of n. Since multiplications can be done in parallel but the addition of $n + 1$ operands will take $\lceil log_2(n + 1) \rceil$ number of steps, considering a two-input ALU.

Primitive for Activation Function Activation function is applied at the output of each neuron to introduce non-linearity in the model. The ReLU and Sigmoid are the most commonly used activation functions. Since activation functions can be expensive over hardware in terms of resources and time to compute. Therefore, we

adopt approximations to simplify their execution. For the sigmoid, primitive is created by using a lookup table for $x \in [-50, 50]$, where precomputed value are stored in the table. For example, $\sigma(5) \approx 0.993$ can be directly fetched from the table instead of computing it through a mathematical function, and values beyond this range $[-50, 50]$ are clipped to 0 or 1. ReLU primitive is implemented via a simple condition check: if the most significant bit (MSB) of x is 1, indicating a negative value, the output is 0; otherwise, x is returned e.g., $\text{ReLU}(-3) = 0$, $\text{ReLU}(4) = 4$. The softmax primitive involves exponentials and division; therefore, a two-level table approach is used. For example, to compute $\text{softmax}(z_1)$ where $z = [3.2, 2.0, 0.1]$, precomputed values of e^{z_i} from a coarse-grained table (e.g., $e^3 \approx 20.09$) are fine-tuned using a secondary table for precision (e.g., $e^{0.2}$), reducing runtime complexity while maintaining accuracy.

3.2.1.2 Model Optimization and Mapping

The multistage transformation shown in Fig. 3.4 represents a systematic process for converting a trained neural network into a hardware-optimized model. The entire pipeline is divided into three key stages:

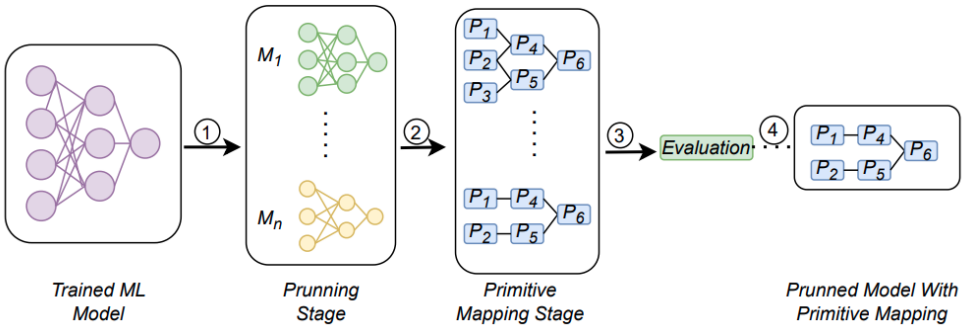


Figure 3.4: Multistage model transformation

Pruning Stage: The pruning stage serves as the initial step in simplifying the trained ML model by eliminating redundant or less impactful components. The overall pruning workflow is summarized in **Algorithm 1**, and consists of the following three steps:

i) Disaggregation of Neurons: The trained model M_{orig} is decomposed to extract each neuron n_j^l from the layer $l \in \{1, \ldots, L\}$ along with its weight W_j^l, bias b_j^l,

and activation a_j^l. Each neuron is stored as a tuple $(l, j, W_j^l, b_j^l, a_j^l)$ in a global set $\mathcal{N}$ for fine-grained and layer-agnostic analysis.

Algorithm 1 Pruning of Neurons

Require: M_{orig}, λ_1, λ_2, ρ
Ensure: M_{pruned}
$\mathcal{N} \leftarrow \varnothing$
for $l = 1$ to L **do**
  for $j = 1$ to N_l **do**
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{(l, j, W_j^l, b_j^l, a_j^l)\}$
  end for
end for
for all $(l, j, W, b, a) \in \mathcal{N}$ **do**
  $\phi_{l,j} \leftarrow \lambda_1 \cdot \|W\|_2 + \lambda_2 \cdot \text{Var}(a)$
end for
Sort $\mathcal{N}$ by $\phi_{l,j}$ ascending
$\mathcal{N}' \leftarrow$ top $(1 - \rho) \cdot |\mathcal{N}|$ neurons by score
for $l = 1$ to L **do**
  if $\{(l', j, W, b, a) \in \mathcal{N}' \mid l' = l\} = \varnothing$ **then**
    Reinsert $\arg\max_j \phi_{l,j}$ from layer l into $\mathcal{N}'$
  end if
end for
$M_{\text{pruned}} \leftarrow \text{Assemble}(\mathcal{N}')$
return M_{pruned}

ii) Identification of Neurons: Once the set $\mathcal{N}$ of neurons is prepared, each neuron is evaluated using a scoring function that captures its significance in the model. The importance score $\phi_{l,j}$ for neuron n_j^l is computed as:

$$\phi_{l,j} = \lambda_1 \cdot \|W_j^l\|_2 + \lambda_2 \cdot \text{Var}(a_j^l)$$

where $\|W_j^l\|_2$ is the ℓ_2 norm of the weight vector and $\text{Var}(a_j^l)$ is the variance of the neuron's activation output. The coefficients $\lambda_1, \lambda_2 \in \mathbb{R}$ are hyperparameters that control the balance between structural and functional contributions. Neurons with low $\phi_{l,j}$ scores indicate minimal impact on the overall result and are thus marked as candidates for pruning. However, to preserve essential information,

we ensure that at least one neuron from each layer l is retained, i.e.,

$$\forall\, l \in \{1, \ldots, L\},\ \exists\, n_j^l \in \mathcal{N}' \subset \mathcal{N}$$

where $\mathcal{N}'$ denotes the set of selected neurons after pruning.

iii) Segregation of Neurons: Based on the computed scores, the bottom $\rho \cdot |\mathcal{N}|$ neurons (for pruning ratio $\rho \in [0, 1]$) are removed, and the remaining neurons form the pruned set $\mathcal{N}'$. The selected neurons are then grouped back layer-wise to reconstruct the simplified model M_{pruned}, preserving the original structure as

$$M_{\text{pruned}} = \text{Assemble}(\mathcal{N}')$$

Primitive Mapping Stage: The primitive mapping stage is tasked with identifying a collection of software-based primitives and arranging them in a combination of cascade and parallel configuration to reproduce the same behavior of the original trimmed model. Each primitive relates to a fundamental operation, such as a neuron computation module and an activation function module. The execution of these primitives incorporates particular optimization strategies and various approximations to guarantee compatibility with hardware-constrained environments.

I) Optimization Consideration: Floating-point multiplication and accumulation are processes that need substantial resources. To reduce these expenses, we shift inputs, weights, and biases left by k bits before any computation, therefore scaling them by 2^k and enabling the use of accelerated integer arithmetic. A right shift of k bits restores the original scale while preserving precision after the majority of processing.

ii) **Approximation for Activation Function:** The activation primitive reuses the same approximation strategy introduced earlier in the *Primitive Construction* subsection (see Section 3.2.1.1).

Evaluation Stage: After the *primitive–optimization* step, it has generated a

diverse set of reduced model. In the *evaluation stage,* we benchmark each model from the reduced model set against two orthogonal objectives: *inference quality* and *hardware feasibility.*

Algorithm 2 Model selection based on score and hardware constraints

Require:
$\mathcal{M} = \{M_1, \ldots, M_n\}$ {candidate models}
S_{th} {minimum acceptable score}
α, β, γ {relative HW cost weights}
Ensure:
M^* {selected model (or **None**)}
$\mathcal{M}_f \leftarrow \varnothing$
for all $M_i \in \mathcal{M}$ **do**
  $S_i \leftarrow \text{Score}(M_i)$
  $R_i \leftarrow \{\#\text{Reg}_i, \#\text{Mul}_i, \#\text{Add}_i\}$
  if $S_i \geqslant S_{\text{th}}$ **then**
    $\mathcal{M}_f \leftarrow \mathcal{M}_f \cup \{M_i\}$
  end if
end for
if $\mathcal{M}_f = \varnothing$ **then**
  return None
end if
Compute cost and select:
$M^* \leftarrow \arg\min_{M_i \in \mathcal{M}_f} \left(\alpha\, \#\text{Reg}_i + \beta\, \#\text{Mul}_i + \gamma\, \#\text{Add}_i\right)$
return M^*

Note: The weights α, β, and γ can be extracted from synthesis reports (area, critical path, power) or set manually to meet design priorities.

Inference quality is captured by a scalar *score* S_i (e.g., accuracy, F_1, or AUROC), whereas hardware feasibility is measured by the triple $R_i = \{\#\text{Reg}_i, \#\text{Mul}_i, \#\text{Add}_i\}$ reporting the number of registers, multipliers, and adders required once the primitive candidate is mapped to the target pipeline. Designers specify a minimum acceptable score S_{th} together with the relative cost weights α, β, and γ that reflect technology-specific trade offs such as area, power, or latency. Algorithm 2 implements a greedy selector that discards any model failing the score threshold, then chooses the least-cost configuration among the remaining, feasible set. Because hardware budgets are often dominated by the resource constraints (e.g., DSP slices on an FPGA or ALU tiles on an ASIC), the linear cost formulation $C_i = \alpha\, \#\text{Reg}_i + \beta\, \#\text{Mul}_i + \gamma\, \#\text{Add}_i$

offers both transparency and tunability: increasing β immediately biases the selector toward multiplier-sparse designs, while reducing γ encourages deeper adder trees when timing slack is available. The procedure terminates in $\mathcal{O}(n)$ time and returns None if no candidate satisfies the quality floor, prompting the optimiser to relax pruning aggressiveness or retrain with higher precision.

3.2.2 Tree-Based Model

In this section, we examine the structural decomposition of decision tree models into hardware-friendly primitives. We propose a table-based mapping strategy that flattens the hierarchical tree structure into a linear, state-based lookup mechanism, enabling efficient execution within the constraints of programmable data plane stages.

3.2.2.1 Primitive Construction

Unlike the logical abstraction of Artificial Neural Networks (ANNs), traditional decision trees rely on sequential conditional checks. Existing in-network deployments, such as P4-NIDS, typically implement these trees as a fixed chain of match-action stages where each internal node corresponds to a specific hardware table. This architecture creates a linear dependency between tree depth and hardware stages, making the system rigid and difficult to scale without exceeding the physical pipeline limits of the switch.

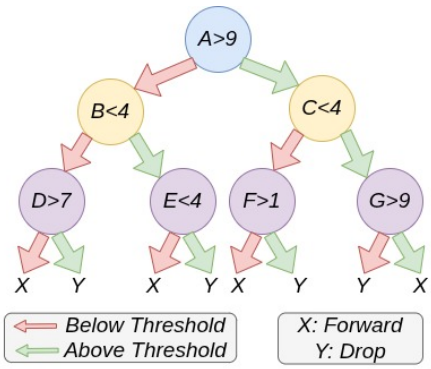
To overcome these constraints, we move away from complex nested control flows and instead adopt a modular state-machine approach. We define three core hardware-compatible primitives that encapsulate the decision logic required at each node, allowing us to represent the tree as a structured Mapped State Table (MST) as illustrated in Fig. 3.5b. This approach flattens the hierarchical tree into a linear lookup mechanism that decouples the model's logical depth from the physical hardware stages.

- **Primitive 1: Conditional Branch Tuple (CBT).** This primitive encodes the fundamental branching logic of a tree node. It is represented as a tuple $\mathcal{T} = (F, \theta, S_{next})$, where F represents the feature field to be examined (e.g.,

packet size), θ is the threshold value for the comparison, and S_{next} is the index of the next state if the condition evaluates to false. For example, in Fig. 3.5b, the entry $(A > 9, 2)$ indicates that if the condition $A > 9$ is false (i.e., $A \leqslant 9$), the execution flow transitions to State 2. Multiple CBTs can exist within a single row to represent sibling nodes or parallel checks.

- **Primitive 2: Default Transition Pointer (DTP).** Tree structures often require a fallback path when specific branching criteria are met (or in this inverse logic, when the "false" conditions are not triggered). The DTP, denoted as δ, serves as the unconditional next-hop index. In the context of our table mapping (Column 5 in Fig. 3.5b), this primitive dictates the flow of execution when all preceding Conditional Branch Tuples in the current state evaluate to true. This ensures a deterministic path through the decision logic, preventing pipeline stalls.

- **Primitive 3: Leaf Class Label (LCL).** This primitive represents the terminal state of the decision process. When a lookup points to a leaf node (e.g., Rows 5 and 6 in Fig. 3.5b), the standard branching logic is replaced by the LCL (denoted as `X` or `Y`). This value serves as the final classification result for the packet, triggering the associated action (e.g., allow, drop, or reroute) and concluding the inference pipeline.



(a) Original Decision Tree Structure

0	(A>9,2)	(C<4,4)	(G>9,6)	5	Stage 0
1	(B<4,3)	(E<4,6)	5	-	Stage 1
2	(D>7,6)	5	-	-	Stage 2
3	(F>1,6)	5	-	-	Stage 3
4	(F>1,6)	5	-	-	Stage 4
5	X	-	-	-	Stage 5
6	Y	-	-	-	Stage 6

(b) Mapped State Table (MST) Representation

Figure 3.5: Transformation of a Decision Tree into a Table-Based Primitive Representation.

Derived from the decision tree structure, we utilize multiple instances of three primitives: Conditional Branch Tuples (CBT), Default Transition Pointers (DTP), and Leaf Class Labels (LCL). These primitives offer flexibility in deployment and can be rearranged across the pipeline stages of the programmable data plane to optimize execution. A concrete representation of this mapping, where specific logical rows are assigned to hardware stages, is provided in Fig. 3.5b.

3.2.2.2 Primitive Mapping

The conversion from the raw table representation to an executable P4 pipeline involves the interaction between these three defined primitives. Rather than executing a deep chain of dependencies, our framework flattens the tree into a "Match-Action" paradigm compatible with RMT (Reconfigurable Match Tables) architectures.

State-Space Flattening and Lookup Logic: Each row in the Mapped State Table corresponds to a distinct logical state in the decision process. The execution logic proceeds as follows:

1. **State Indexing:** The system utilizes a metadata state variable to index the table. This variable tracks the current position in the decision tree, ensuring the pipeline fetches the correct row of primitives for the active node.

2. **Tuple Evaluation:** The system evaluates the Conditional Branch Tuples (CBT) in priority order. For a given state, the hardware compares feature values against stored thresholds. The first condition to evaluate as false triggers an immediate update to the state metadata using the index defined in S_{next}.

3. **Default Transition:** If all CBT conditions evaluate to true, the system utilizes the Default Transition Pointer (DTP) to determine the next state index, ensuring logical continuity.

Leaf Node Resolution: When the lookup process retrieves a row corresponding to a terminal state (as seen in Rows 5 and 6 of Fig. 3.5b), the branching logic is superseded by the Leaf Class Label (LCL). This label provides the final classification

result. By encoding the tree using these three specific primitives, we decouple the depth of the tree from the hardware pipeline stages. This allows for circular or folded pipeline execution if necessary, and significantly reduces the TCAM (Ternary Content-Addressable Memory) usage compared to expanding the tree.

3.3 Summary

The proposed optimization and mapping technique provides a singular approach for utilizing both artificial neural networks (ANNs) and decision trees within programmable data plane environments. We employ structural pruning, hardware-aware primitive generation, and approximation methods to reduce the arithmetic complexity of artificial neural network models while maintaining inference quality. In the decision trees, we employed logical minimization and staged decomposition to demonstrate our classification logic using a minimal amount of Boolean expressions that operate within the constraints of the hardware pipeline. Our methodology ensures compatibility with resource-constrained targets such as PISA-based architectures and FPGA platforms by transforming both model types into lightweight, modular primitives and systematically distributing their functionality throughout pipeline stages. This enhances the efficiency and comprehensibility of ML inference at line rate, while also facilitating the development of more complex models such as ensembles, hybrid pipelines, and model composition strategies for future applications.

Chapter 4

Experiments and Results

4.1 Experimental Setup

The experiments were executed on a workstation powered by an Intel Core i7–12700 processor (25 hardware threads) and 8gb of RAM. The software stack comprised Ubuntu 20.04, Python 3.9, and a suite of libraries: `Flask` for the REST endpoint, `joblib` together with parallel pipelines, `pickle` and `base64` for model serialisation, `requests` for client–server exchange, and `scikit--learn` plus `PyTorch` for machine–learning routines, supplemented by standard preprocessing utilities.

The evaluation workflow begins with local training of model, which serves as the baseline model. The user then issues a `POST` request to the `Flask` server, specifying the desired primitive–block size n specifically for MLP. Upon receipt, the server, partitions the model into size–n primitives and creates multiple variants by applying different pruning ratios. Each variant is benchmarked for inference accuracy and for hardware cost—namely, the number of registers, multipliers, and adders derived from the mapping toolchain. A greedy selector rejects any design whose accuracy falls below the user-defined threshold and chooses the least-cost configuration among the surviving candidates. Finally, the selected primitive graph is serialised, encoded, and returned to the client for deployment, thereby delivering a resource-aware model that satisfies both performance and hardware constraints in a single round-trip.

4.2 Dataset & Model Setup

Our experiments are conducted using three widely adopted intrusion detection datasets: UNSW-NB15 [13], TON_IoT [14], and NSL-KDD [20]. These datasets provide a diverse range of network traffic patterns and attack categories, making them suitable for evaluating ML-based classification models in network environments. To capture complex relationships in the data, we employ two types of models: a Multilayer Perceptron (MLP) and a Decision Tree.

The MLP architecture is composed of four completely connected layers, each with a decreasing neuron count of 64, 32, 16, and 8, respectively. Each concealed layer implements the ReLU activation function to introduce non-linearity, while the final layer implements a softmax activation for multiclass classification. After that, we run this trained MLP model through our pruning and primitive mapping pipeline to optimize it for hardware.

We also train a Decision Tree classifier on the same datasets to determine how well our methods work for logic minimization and pipeline mapping. The tree structure is taken out, turned into Boolean expressions, and improved for staged P4 deployment. These two types of models show that our suggested framework can function with both arithmetic-heavy and logic-based inference paradigms.

4.3 Performance Evaluation

We begin by evaluating various performance metrics for both of the designed models. First, we analyze the ANN-based model, followed by the Tree-based model.

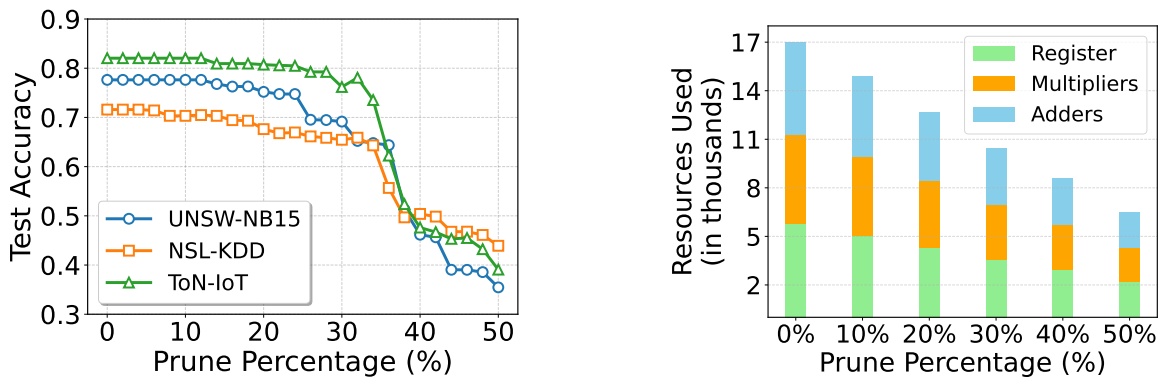
4.3.1 Results on ANN-based model

This subsection presents the performance evaluation of the optimized ANN-based model. We assess its behavior in terms of accuracy retention, inference latency, and hardware-aware complexity after applying pruning and primitive mapping techniques. The results highlight the trade-offs between model simplicity and inference quality.

4.3.1.1 Primitive Resource Utilization

For the model deployment, we considered each *neuron-primitive block* (N-PB) with six neurons and eight input features. This configuration of the primitive is selected based on our optimization, where we found that data communication overhead and resource wastage reduce with increasing configuration size. Also, we consider ten pipeline stages of the PISA architecture, where we introduced ten neuron primitives in each stage. We used Alveo U55c compute card to implement our framework. The native PISA design consumes 54.67% of the available LUTs. Introduction of 100 neuron primitives among 10 stages takes an additional 17% of the LUTs.

4.3.1.2 Pruning and Optimisation Trade-offs



(a) Classification Accuracy Based On Pruning Over Multiple Datasets

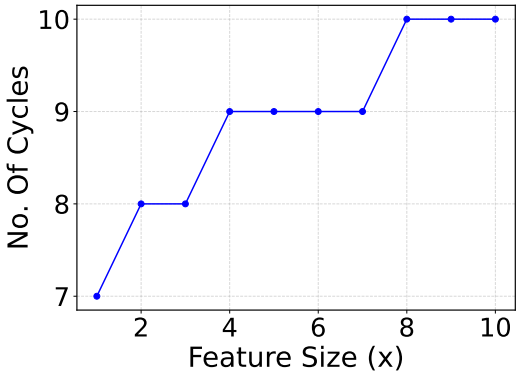
(b) Resources Used Based On Pruning On UNSW-NB15 Datasets

Figure 4.1: Relationship between pruning and model efficiency

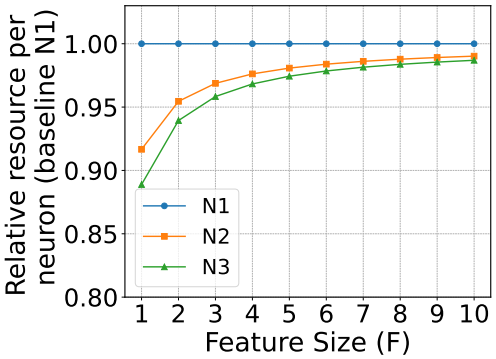
Pruning eliminates redundant neurons to meet stringent hardware constraints; however, excessive pruning can significantly degrade model accuracy. As shown in Fig. 4.1a, removing more than approximately 30% of weights leads to a sharp decline in test performance. Thus, we restrict pruning to a maximum of 30%, ensuring a balanced trade-off. Furthermore, complementary optimisations—such as floating-point to fixed-point conversion and activation function simplification—further reduce computational cost while introducing less than $\tilde{1}$% additional accuracy loss, making the overall approach highly suitable for embedded deployment scenarios.

4.3.1.3 Resources

It determines the number of registers, multipliers, and adders required to map this model in the hardware pipeline. As shown in Fig. 4.1b, the resource usage decreases as a function of the pruning fraction. This is because pruning removes neurons, which directly reduces the number of operations required for the model. Consequently, fewer resources, such as registers, multipliers, and adders, are needed to execute the pruned model, leading to lower resource consumption.



(a) Relationship between input feature size and inference latency for single neuron primitive



(b) Hardware resource requirements as a function of neuron primitive size (Neurons and Features).

Figure 4.2: Relationship between latency and resource with feature size and neuron

Relationship between input feature size and latency: The size of the input feature vector directly affects the latency of neuron computation. While multiplications for each feature-weight pair can be executed in parallel, the subsequent accumulation (addition) of these products introduces a sequential bottleneck. Specifically, the addition of $n + 1$ operands requires $\lceil \log_2(n + 1) \rceil$ steps when using a two-input ALU. Thus, as the number of input features increases, the latency of the primitive also increases logarithmically, as shown in Fig. 4.2a. This relationship underscores the importance of constraining feature width when designing latency-sensitive primitives.
Relationship between primitive size and hardware cost: Fig. 4.2b compares the normalized hardware resource consumption of multi-neuron primitives (e.g., N_2, N_3) to that of a single-neuron primitive (N_1), across varying feature sizes. Interestingly, while larger primitives encapsulate more neurons, they exhibit reduced overall

resource consumption per neuron. This is primarily due to the elimination of redundant registers and reuse of shared compute logic across neurons. However, designing larger primitives may increase mapping complexity and reduce placement flexibility. Therefore, selecting an optimal primitive size involves a trade-off between resource efficiency and structural modularity.

4.3.2 Results on Tree-based model

We focus on decision trees within our tree-based learning approach due to their interpretability and distinct suitability for tabular mapping. After training the model, the decision logic is extracted and mapped into our proposed primitives (CBT, DTP, LCL). For training, we convert the multi-class problem into a binary classification task by labeling all benign traffic as class 0 and grouping all other traffic categories under class 1. Thus, class 1 represents the full spectrum of attack types or anomalous behaviors. The following evaluation compares the performance of our primitive-based hardware mapping against a standard non-primitive tree implementation across multiple datasets (D0–D5) of CICIDS-17.

4.3.2.1 Model Fidelity and Accuracy

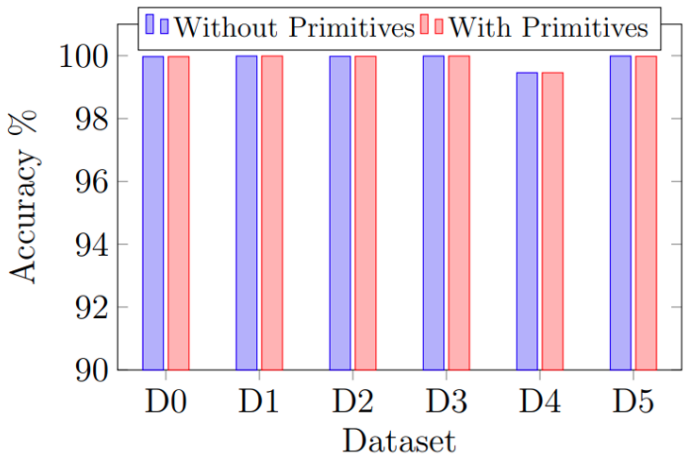


Figure 4.3: Accuracy comparison between the baseline model and the primitive-based mapping.

A primary concern when migrating logical models to hardware primitives is the

potential loss of fidelity. Figure 4.3 depicts the test accuracy across datasets D0 through D5. The results demonstrate that our mapping strategy is strictly lossless. The "With Primitives" implementation achieves identical accuracy (near 100%) to the baseline "Without Primitives" software model. This confirms that flattening the tree into Conditional Branch Tuples and utilizing the Default Transition Pointer does not compromise the decision boundaries or the classification capability of the system.

4.3.2.2 Computational Efficiency (Cycles)

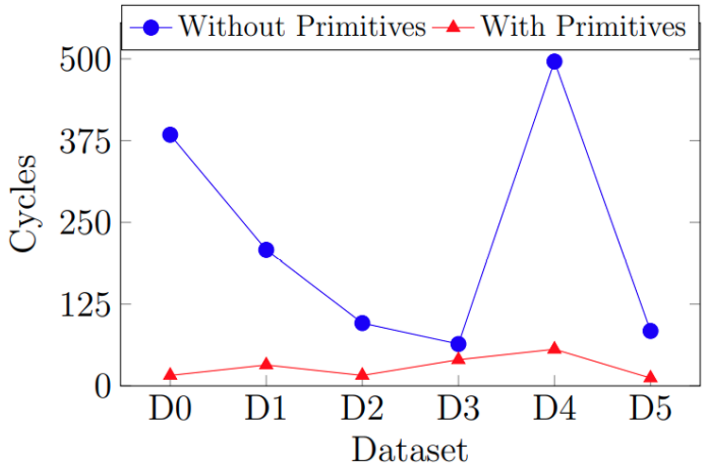


Figure 4.4: Processing cycles required for inference.

In high-speed data planes, processing latency must be low and predictable. Figure 4.4 compares the computational cost defined in processing cycles. The baseline implementation (Blue) exhibits high variance and significant cost, peaking at 500 cycles for dataset D4. This behavior is typical of standard pointer-chasing tree algorithms, where depth varies per packet. In contrast, our primitive-based approach (Red) demonstrates an extremely low and flat profile, consistently requiring fewer than 25 cycles. This deterministic behavior is a direct result of the pipelined stage mapping, ensuring that inference completes within the strict time budget of line-rate packet processing.

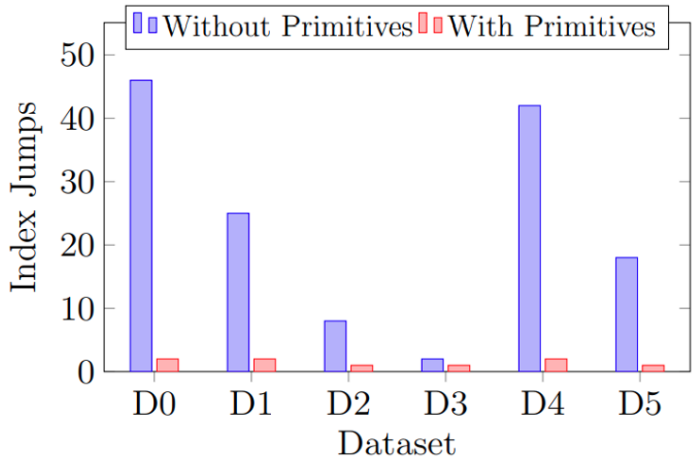


Figure 4.5: Comparison of memory index jumps during decision path traversal.

4.3.2.3 Structural Complexity (Index Jumps)

To evaluate the efficiency of memory access and state transitions, we analyze the "Index Jumps" required to reach a leaf node. As shown in Figure 4.5, the standard implementation without primitives requires a high number of jumps (e.g., over 40 for D0 and D4), necessitating frequent non-sequential memory accesses that can stall the pipeline. Conversely, the primitive-based model significantly reduces this overhead to negligible levels (typically 1–2 jumps). By utilizing the Conditional Branch Tuple (CBT) logic, multiple checks are performed in parallel within a single stage, and the Default Transition Pointer (DTP) streamlines the flow, resulting in a highly optimized memory access pattern suitable for TCAM/SRAM architectures.

Chapter 5

Conclusions and Future Scope

This thesis presents a consistent and efficient framework for putting machine learning inference directly into programmable data planes. It focuses on two basic model types: multilayer perceptrons (MLPs) and decision trees. It shows how each can be turned into lightweight, hardware-compatible primitives that follow the rules of modern switches, like register width, ALU count, and stage depth.

The framework for MLPs starts with organizing pruning to get rid of neurons and connections that don't have much of an effect, which makes the model much less complex. The pruned network is then broken down into a number of smaller arithmetic primitives, each of which is made only to run quickly on a match-action pipeline. To keep a balance between model fidelity and implementation practicalities, approximate computation techniques are only used when necessary.

In contrast, decision trees are naturally rule-based and are turned into sequences of conditional rules from the root to the leaf. In order to provide fewer and more compact rules, appropriate for TCAM or exact match tables on the data plane, each path is optimized using Boolean minimisation and literal reduction.

The results of our simulation reveal that both types of models can be used in programmable switches with the defined set of primitives. These primitives work well with diverse datasets and model sizes while keeping throughput and line-rate processing guarantees.

In the future, the framework can be improved to deal with more complex model

types, like tree ensembles or other MLP-based methods. In the future, improvements could potentially make it possible to change the configuration based on traffic patterns. Testing on real hardware platforms like NetFPGA or Tofino-2 would also help confirm latency and power performance.

In conclusion, this work shows that programmable switches can easily run reduced versions of MLP and decision tree models. The suggested primitive-based method introduces a way to embed ML in the dataplane for models to work at high speed with efficient use of resources.

Bibliography

[1] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[3] Yaying Chen, Siamak Layeghy, Liam Daly Manocchio, and Marius Portmann. P4-nids: High-performance network monitoring and intrusion detection in p4. *arXiv preprint arXiv:2411.17987*, 2024.

[4] Archan Dhadhania, Jitendra Bhatia, Rachana Mehta, Sudeep Tanwar, Ravi Sharma, and Amit Verma. Unleashing the power of sdn and gnn for network anomaly detection: State-of-the-art, challenges, and future directions. *Security and Privacy*, 7(1):e337, 2024.

[5] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[6] Jose Gomez, Elie F Kfoury, Jorge Crichigno, and Gautam Srivastava. A survey on tcp enhancements using p4-programmable devices. *Computer networks*, 212: 109030, 2022.

[7] Sydney Mambwe Kasongo. A deep learning technique for intrusion detection system using a recurrent neural networks based framework. *Computer Communications*, 199:113–125, 2023.

[8] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. THC: Accelerating distributed deep learning using tensor homomorphic compression. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1191–1211, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL `https://www.usenix.org/conference/nsdi24/presentation/li-minghao`.

[9] Yibing Liu, Lijun Huo, Xiongtao Zhang, and Jun Wu. A multi-objective resource pre-allocation scheme using sdn for intelligent transportation system. *IEEE Transactions on Intelligent Transportation Systems*, 25(1):571–586, 2023.

[10] Mamatha Maddu and Yamarthi Narasimha Rao. Network intrusion detection and mitigation in sdn using deep learning models. *International Journal of Information Security*, 23(2):849–862, 2024.

[11] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[12] Sangeeta Mittal. Performance evaluation of openflow sdn controllers. In *International Conference on Intelligent Systems Design and Applications*, pages 913–923. Springer, 2017.

[13] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.

[14] Nour Moustafa, Mohiuddin Ahmed, and Sherif Ahmed. Data analytics-enabled intrusion detection: Evaluations of ton_iot linux datasets. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 727–735. IEEE, 2020.

[15] Xinglong Pei, Penghao Sun, Yuxiang Hu, Dan Li, Bo Chen, and Le Tian. Enabling efficient routing for traffic engineering in sdn with deep reinforcement learning. *Computer Networks*, 241:110220, 2024.

[16] Sankepally Sainath Reddy, Kosaraju Nishoak, JL Shreya, Yennam Vishwambhar Reddy, and U Venkanna. A p4-based adversarial attack mitigation on machine learning models in data plane devices. *Journal of Network and Systems Management*, 32(1):5, 2024.

[17] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, 2020. doi: 10.1109/JSAC.2020.3000405.

[18] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL `https://www.usenix.org/conference/nsdi21/presentation/sapio`.

[19] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frédéric Andres. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398:185–196, 2020. doi: 10.1016/j.neucom.2020.02.035.

[20] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE symposium on computational intelligence for security and defense applications*, pages 1–6. Ieee, 2009.

[21] Boyao Wang and Volodymyr Kindratenko. RL-Pruner: Structured pruning using reinforcement learning for cnn compression and acceleration. *arXiv preprint arXiv:2411.06463*, 2024. Available at `https://arxiv.org/abs/2411.06463`.

[22] Shie-Yuan Wang and Ying-Hua Wu. Supporting large random forests in the pipelines of a hardware switch to classify packets at 100-gbps line rate. *IEEE Access*, 11:112384–112397, 2023. doi: 10.1109/ACCESS.2023.3323297.

[23] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. pheavy: Predicting heavy flows in the programmable data plane. *IEEE Transactions on Network and Service Management*, 18(4):4353–4364, 2021.

[24] Guobing Zou, Tengfei Li, Ming Jiang, Shengxiang Hu, Chenhong Cao, Bofeng Zhang, Yanglan Gan, and Yixin Chen. Deeptsqp: Temporal-aware service qos prediction via deep neural network and feature integration. *Know.-Based Syst.*, 241(C), April 2022. ISSN 0950-7051. doi: 10.1016/j.knosys.2021.108062. URL `https://doi.org/10.1016/j.knosys.2021.108062`.