MTP-2 Project Report

Hardware Accelerated Network Traffic Analytics

Department of Computer Science and Engineering

IIT Madras

Submitted by

Baddala Sai Kumar Reddy

CS23M059

Mailavaram Partha Sarathi Reddy

CS23M035

Guided by

Prof. Ayon Chakraborty

November 2024

Abstract

With the explosive growth of digital communication and the increasing complexity of modern networks, efficient and real-time **classification of network traffic** has become a crucial requirement for ensuring performance, security, and manageability. Traditional software-based classification systems are often limited by latency and processing overhead, particularly in high-throughput environments. This project proposes a **hardware-accelerated framework** for network traffic classification, utilizing **Field Programmable Gate Arrays (FPGAs)** to enable fast and efficient inference of **machine learning models** directly at the data path.

The primary goal of this M.Tech project is to build an **embedded platform** that performs real-time classification of network traffic using a **Deep Neural Network (DNN)** deployed on the FPGA. The system is developed on the **Xilinx Zynq-7000 SoC**, specifically using the **PYNQ-Z2** board, which combines a dual-core ARM Cortex-A9 **Processing System (PS)** with **Programmable Logic (PL)**. In this design, the network packets are first captured and preprocessed in the PS running **embedded Linux**, where relevant traffic features are extracted. These features are then transmitted to the PL, where a hardware-synthesized DNN model performs traffic classification at low latency.

This **hardware-software co-design** effectively offloads the computationally intensive classification task from the processor to the FPGA, significantly improving throughput and responsiveness. The neural network model is trained using a labeled subset of the **UNB CIC VPN dataset**, quantized for hardware efficiency, and synthesized using HLS tools such as **hls4ml**. The system supports multiple traffic classes and is designed to operate at line rate, demonstrating the feasibility of using FPGA-based acceleration for machine learning-driven traffic analytics in resource-constrained embedded environments.

The proposed architecture highlights the potential of FPGAs in enabling real-time, scalable, and power-efficient network traffic classification. It serves as a foundational step toward the broader goal of intelligent network interface cards capable of inline decision-making for monitoring, QoS management, and security applications.

Contents

Chapter 1

Problem Statement

1.1. Problem Statement

1.1.1. Objective

The primary objective of this project is to design and implement a real-time network traffic classification system that leverages hardware acceleration to meet the demands of high-throughput environments. Accurate and timely classification of traffic—such as distinguishing between browsing, or streaming—is critical for enabling efficient network management, improving security postures, and ensuring Quality of Service (QoS). With the exponential growth in network bandwidth and traffic complexity, traditional software-based solutions are increasingly unable to meet latency and performance requirements, particularly in edge and embedded settings.

1.1.2. Limitations of Traditional Routers

Conventional routers, while essential for packet forwarding, are not architected to handle advanced machine learning-based analytics. Attempting to embed real-time traffic classification into router pipelines introduces several critical challenges:

- **Limited Computational Capability:** Traditional routers are optimized for switching and routing but lack the hardware support required for executing deep learning models or high-frequency data analytics.
- **Increased Latency:** Adding classification workloads to routers introduces processing delays, as they are not equipped with specialized accelerators. This increased latency undermines real-time responsiveness.
- **Resource Constraints:** Routers typically operate with minimal general-purpose compute resources, insufficient to run complex inference tasks alongside their core routing functions.
- **Poor Scalability:** As network speeds scale upward, software-based or CPU-bound solutions in routers cannot keep pace with the need for fast packet-level decision-making, leading to throughput bottlenecks and degraded analytics accuracy.

1.1.3. Proposed Solution

To address the above limitations, this project proposes the development of a **hardware-accelerated traffic classification platform** implemented on an embedded system. The solution is realized on a Xilinx Zynq-7000 SoC (PYNQ-Z2), which integrates a Processing System (PS) with Programmable Logic (PL), enabling a tight hardware-software co-design.

- **Software-Driven Feature Extraction:** Network packets are captured and pre-processed in the ARM-based Processing System, where statistical features—such as flow durations and inter-arrival times—are extracted using an embedded Linux environment.
- **FPGA-Based Classification:** The extracted features are passed to a neural network model synthesized into the FPGA fabric using high-level synthesis (HLS) tools. Performing classification in hardware significantly reduces inference time and ensures deterministic latency.
- **Efficient, Scalable Execution:** The architecture decouples feature extraction and classification, allowing for pipeline-level parallelism. The FPGA's reconfigurability also allows for future upgrades, including model updates and support for additional traffic classes.
- **Embedded Integration:** By implementing the system as a smart Network Interface Card (NIC) hosted on an embedded platform, the solution supports inline traffic analytics with minimal resource footprint, suitable for deployment in real-world edge or IoT networks.

This project thereby demonstrates a scalable, real-time, and hardware-efficient approach to network traffic classification, serving as a stepping stone toward intelligent network devices with built-in AI capabilities.

Chapter 2

Introduction to hls4ml

nn

2.1. Introduction to hls4ml

This chapter introduces the motivation, architecture, and practical considerations behind `hls4ml`, the High-Level Synthesis toolkit for deploying machine learning models on FPGAs.

2.1.1. Overview of hls4ml

`hls4ml` is an open-source Python toolkit designed to translate trained machine learning models into synthesizable `C++` code, enabling efficient deployment on FPGAs and ASICs using High-Level Synthesis (HLS) tools. It bridges the gap between widely used machine learning frameworks (such as Keras, TensorFlow, PyTorch, and ONNX) and digital hardware implementations, allowing inference to run with microsecond-scale latency in hardware-constrained environments.

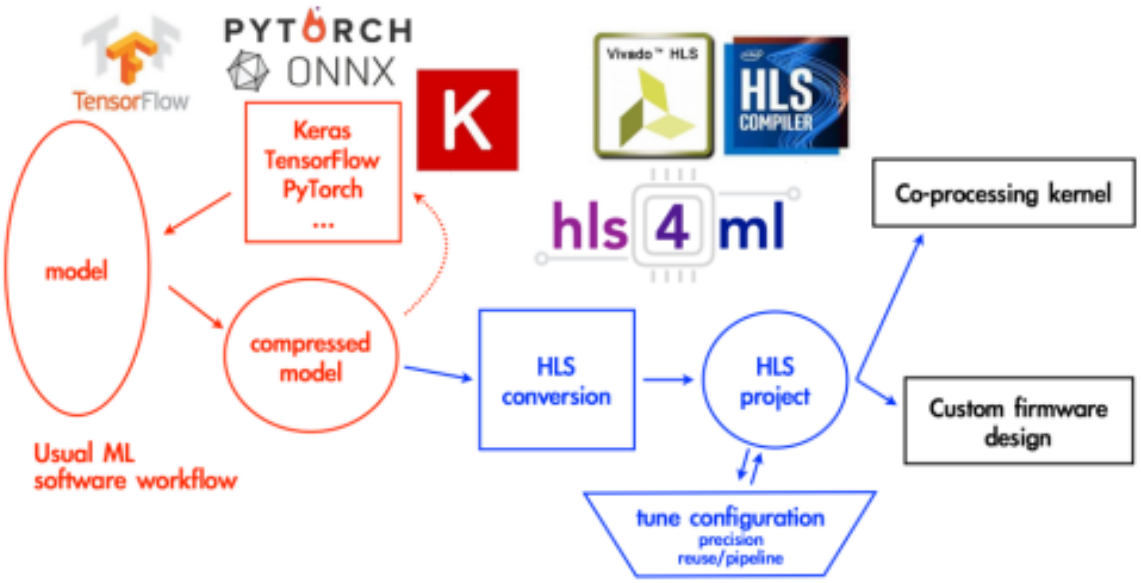


Figure 2.1: Workflow overview of hls4ml: from trained models to FPGA firmware

2.1.2. Motivation and Origin

The development of `hls4ml` was originally inspired by use cases at the CERN Large Hadron Collider (LHC), where detectors generate petabytes of raw data from high-energy collisions. Since it is impossible to store all generated events, real-time filtering systems called *triggers* are deployed to determine whether a particular event should be recorded. Traditional ML-based classification was typically performed offline, which meant potentially valuable events could be missed.

By using FPGAs and tools like `hls4ml`, machine learning inference can be integrated directly into the live trigger path, enabling faster and more selective event processing with extremely low latency.

FPGAs are particularly attractive in such scenarios because they offer deterministic, low-latency execution while consuming significantly less power than CPUs or GPUs. This hardware capability makes it possible to deploy ML models directly at the detector level to perform classification on-the-fly and decide whether to store or discard incoming data.

Broader Applications

The same principle is applied in other latency-sensitive domains, such as real-time traffic classification in networking applications.

2.1.3. Conceptual Overview

The main goal of `hls4ml` is to allow fast and configurable translation of a compressed neural network model into hardware-compatible `C++` code. The generated HLS project can then be synthesized into an IP core and integrated into a larger FPGA design or be used as a dedicated co-processing kernel.

- Reduces time-to-deployment

- Simplifies hardware prototyping
- Allows precise control over trade-offs such as:
  - **Latency vs resource usage**
  - **Precision vs accuracy**
  - **Throughput vs power consumption**

2.1.4. How It Works

Neural networks are inherently structured and repetitive, making them well-suited for hardware mapping. In `hls4ml`, the network is split into layers, and each neuron in a layer performs a weighted sum of its inputs followed by an activation function. These operations are represented in matrix form and mapped to pipelined hardware logic.

`hls4ml` accelerates the process by employing:

- **Loop pipelining:** Enables parallelism by accepting new inputs while previous computations are still in-flight.
- **Fixed-point arithmetic:** Replaces floating-point operations with fixed-point formats for better resource efficiency.
- **Precomputed activations:** Non-linear activations like ReLU and tanh are replaced with fast lookup-table-based implementations.

Each layer becomes a pipelined hardware block that supports high-throughput streaming. As soon as the pipeline is filled, a new prediction can be produced at every cycle, resulting in consistent microsecond-level inference latency.

2.1.5. Tuning Parameters for Deployment

The behavior and efficiency of the synthesized design can be extensively tuned using the following key parameters in the configuration file:

- **Precision:** Defines bit-width for weights, biases, and activations (e.g., `ap_fixed<6,2>`).
- **Reuse Factor:** Controls how often a multiplier is reused. Low values lead to more parallelism but higher resource usage.
- **Quantization:** `hls4ml` works best with pre-quantized models (e.g., using QKeras).
- **I/O Type:**
  - `io_parallel` for low-latency inference
  - `io_stream` for memory-efficient pipelined processing
- **Granularity:** Allows global or per-layer control over reuse and precision.

2.1.6. Version and Compatibility Information

`hls4ml` is designed for Linux environments and requires:

- **Python version:** ≥ 3.10
- **Operating System:** Linux (no official support for Windows/macOS)

Synthesis Tool	Supported Versions
Vivado HLS	2018.2 – 2020.1
Vitis HLS	2022.2 – 2024.1
Intel HLS	20.1 – 21.4
Catapult HLS	2024.1_1 – 2024.2
oneAPI (experimental)	2024.1 – 2025.0

Table 2.1: Supported HLS backends for use with `hls4ml`.

2.1.7. Platform and Backend Support

`hls4ml` supports a range of ML frameworks and model types:

- **ML Frameworks:** (Q)Keras, PyTorch, (Q)ONNX
- **Model Types:**
  - Multilayer Perceptrons (MLP)
  - Convolutional Neural Networks (CNN)
  - Recurrent Neural Networks (LSTM)
  - Graph Neural Networks (GarNet)

It is recommended to use a Linux system with Python 3.10 or higher. While unofficial setups using WSL on Windows have worked for some users, native Linux environments are ideal for stability and compatibility.

2.1.8. Operating System Compatibility and Ubuntu Requirements

While `hls4ml` is portable across most Linux distributions, the same is not true for the proprietary toolchains used during synthesis. For example:

Important OS Restriction

Vivado HLS is officially supported only on **Ubuntu 16.04 LTS** for versions 2018.2 to 2020.1.

Attempting to install these versions on newer distributions such as Ubuntu 18.04 or 20.04 typically leads to:

- Installation failures due to deprecated libraries,
- GUI runtime errors from outdated GTK2 or libc dependencies,
- TCL script incompatibilities with updated toolchains.

For this project, an machine with Ubuntu 16.04 was used to ensure seamless compatibility with Vivado HLS 2019.2 and to successfully synthesize the HLS model into a deployable IP core.

2.1.9. What About Vitis HLS?

Vitis HLS is a newer and more modular high-level synthesis environment introduced by Xilinx to replace Vivado HLS. However, there are caveats:

- It uses a kernel-based design model more suited for data center and software-accelerated workflows.
- It changes the synthesis pipeline architecture, which may cause incompatibilities with existing `hls4ml` templates.
- Its integration with embedded SoC boards (e.g., PYNQ-Z2) is not yet seamless.

`hls4ml` does support Vitis HLS in experimental mode, but stability and ease of use are not guaranteed.

Recommended Setup

Use **Ubuntu 16.04 LTS + Vivado HLS 2019.2** or a compatible container for stable synthesis and FPGA deployment.

2.1.10. Benefits and Use in This Work

`hls4ml` enables fast, modular, and resource-efficient deployment of deep learning models to FPGAs without manually writing RTL. In this project:

- A neural network was trained on the **UNB CIC VPN** dataset for traffic classification.
- The model was quantized using QKeras and converted into HLS-compatible code.
- The generated code was synthesized using **Vivado HLS 2019.2**.
- The synthesized classifier was deployed on the PL (Programmable Logic) of the **PYNQ-Z2 board**.
- The ARM Cortex-A9 (PS) was responsible for packet feature extraction and communication.

This design demonstrates a practical and robust way to achieve real-time, low-latency, and hardware-accelerated network traffic classification.

Chapter 3

hls4ml Frontend and Backend Architecture

3.1. Frontend Support in `hls4ml`

3.1.1. Overview

The frontend of `hls4ml` serves as the initial entry point for translating machine learning models from popular frameworks into hardware-friendly representations. It is responsible for parsing trained models, validating supported layers, extracting weights and metadata, and constructing a platform-agnostic intermediate representation.

Currently, `hls4ml` supports frontends for **Keras/QKeras**, **PyTorch**, and **ONNX**, with Keras being the most mature and feature-complete.

3.1.2. Keras and QKeras Frontend

The `Keras` frontend is the most stable and widely used interface in `hls4ml`. It is based on **TensorFlow Keras v2 (tf.keras)**, and future updates are expected to improve support for **Keras v3**. The frontend operates by parsing the serialized `.json` representation of the trained model.

Supported Layer Types

`hls4ml` can currently parse a wide range of Keras layers, including:

- **Core layers:** `Dense`, `Activation`, etc.
- **Convolutional layers:** `Conv1D`, `Conv2D`
- **Pooling layers:** `MaxPooling`, `AveragePooling`
- **Recurrent layers:** `LSTM`
- **Merging and reshaping layers:** `Concatenate`, `Reshape`

The corresponding quantized versions from **QKeras** are also supported for deploying low-precision neural networks on FPGAs.

Limitations of Keras Frontend

- Attention and normalization layers are not yet supported.
- `Lambda` layers cannot be parsed directly as they don't serialize model state.
- Such layers must be implemented as custom classes and registered through the `Extension API`.

Data Format Considerations

Keras supports both `channels_last` and `channels_first` formats via the `data_format` parameter. However, all hardware implementations in `hls4ml` expect `channels_last`, and using `channels_first` may lead to undefined behavior or synthesis errors.

Future Outlook

The development team is actively working on an alternative to QKeras that will support quantization natively in `Keras v3`. This will help ensure long-term compatibility and simplify integration workflows for future projects.

3.1.3. PyTorch Frontend

The `PyTorch` frontend in `hls4ml` uses symbolic tracing through the `torch.fx` module. This ensures that the full execution graph of a model is captured and understood before conversion. However, this also introduces certain limitations based on the traceability of the model structure.

Supported Constructs

- Models written using `torch.nn` modules are more robustly supported.
- `torch.nn.functional` layers may also work but are less thoroughly tested.

If the model includes layers unsupported in the current PyTorch frontend, users are advised to request features or contribute extensions.

Quantized PyTorch Models

Direct ingestion of quantized models from Brevitas is not supported at present. Instead, the recommended workflow is as follows:

1. Export the quantized PyTorch model from Brevitas to `ONNX`.
2. Import the ONNX model using the `QONNX` frontend in `hls4ml`.

Important Note on QONNX Import

Some ONNX models using non-scalar or non-power-of-2 quantization scales may fail during import. Users should validate models early and consult developers for support.

Tensor Layout Compatibility

There is an important difference in tensor layout between Keras and PyTorch:

- **Keras:** `channels_last`
- **PyTorch:** `channels_first`

By default, `hls4ml` automatically transposes internal tensors (e.g., weights, biases). Additionally:

- With `io_parallel`, transpose nodes are automatically inserted at input/output layers.
- With `io_stream`, inputs must be manually transposed by the user before feeding to the accelerator.

Limitations and Roadmap

- The **Extension API**, which allows custom layer parsing (available in Keras), is not yet implemented for PyTorch.
- The PyTorch frontend is actively being developed, and many features are expected to achieve parity with the Keras parser soon.

3.1.4. Summary

The frontend system in `hls4ml` provides broad compatibility with major ML frameworks, with Keras + QKeras offering the most mature support. PyTorch integration is functional and improving rapidly, with active development focusing on better quantization handling and customization features. Developers targeting hardware deployment should use Keras for stability, and transition to PyTorch workflows as support matures.

3.2. Vivado and Vitis Backends in `hls4ml`

3.2.1. Overview of Backend Toolchains

The `Vivado` and `Vitis` backends in `hls4ml` are designed specifically for targeting AMD/Xilinx FPGA platforms. Both backends generate synthesizable IP cores that can be integrated into hardware projects using the Vivado Design Suite.

- **Vivado HLS:** Targets the legacy Vivado HLS compiler (2018.2–2020.1). Still widely used for SoC and embedded FPGA flows, especially on boards like PYNQ-Z2.
- **Vitis HLS:** The newer synthesis tool that supersedes Vivado HLS. It adopts a kernel-centric model, compatible with modern Vitis accelerator flows and datacenter-grade FPGA platforms.

Recommendation: For new designs on AMD/Xilinx FPGAs, the `Vitis` backend is preferred. Active development and feature enhancements are focused on Vitis; Vivado support will not receive major updates going forward.

3.2.2. Supported Boards

The VivadoAccelerator backend integrates tightly with the **PYNQ software stack**, enabling seamless deployment on supported Zynq-based and Alveo FPGA boards.

Board Name	FPGA Part Number
PYNQ-Z2	xc7z020clg400-1
ZCU102	xczu9eg-ffvb1156-2-e
Alveo U50	xcu50-fsvh2104-2-e
Alveo U250	xcu250-figd2104-2L-e
Alveo U200	xcu200-fsgd2104-2-e
Alveo U280	xcu280-fsvh2892-2L-e

Table 3.1: FPGA development boards supported by hls4ml VivadoAccelerator backend

Note: Additional boards can be supported as long as they are compatible with the PYNQ stack.

3.2.3. Zynq Architecture and Overlay Deployment

For Zynq-based systems such as the PYNQ-Z2, the FPGA is divided into:

- **PS (Processing System):** Dual-core ARM Cortex-A9 that runs embedded Linux.
- **PL (Programmable Logic):** FPGA fabric where custom logic is synthesized.

`hls4ml` leverages the PYNQ overlay system to present synthesized designs as Python-accessible hardware libraries. These overlays communicate via AXI-Stream interfaces and can be dynamically programmed at runtime.

3.2.4. Neural Network Overlay

In the PYNQ environment, synthesized accelerators are packaged into an overlay bitstream. `hls4ml` creates a **Neural Network Overlay** that allows predictions to be executed directly from Python by offloading inference to the PL.

AXI Streaming Interface

Data between the PS and PL is transmitted using AXI-Stream, enabling low-latency input/output buffering and back-and-forth communication for inference.

3.2.5. Example Deployment: PYNQ-Z2

This example walks through deploying a trained Keras model to a PYNQ-Z2 board using the VivadoAccelerator backend.

Step 1: Model Conversion and Bitstream Generation

```
import hls4ml

# Create hardware config
config = hls4ml.utils.config_from_keras_model(model, granularity='name')

# Convert model to HLS project for VivadoAccelerator
hls_model = hls4ml.converters.convert_from_keras_model(
    model=model,
    hls_config=config,
    output_dir='hls4ml_prj_pynq',
    backend='VivadoAccelerator',
    board='pynq-z2'
)

# Build the project and generate bitfile
hls_model.build(bitfile=True)
```

Step 2: Package Artifacts for PS Deployment

```
mkdir -p package
cp hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.runs/impl_1/design_1_wrappe
cp hls4ml_prj_pynq/myproject_vivado_accelerator/project_1.srcs/sources_1/bd/design_1/
cp hls4ml_prj_pynq/axi_stream_driver.py package/
tar -czvf package.tar.gz -C package/ .
```

This step collects all necessary files: the `bitfile`, the hardware handoff file (`.hwh`), and a Python driver for communication.

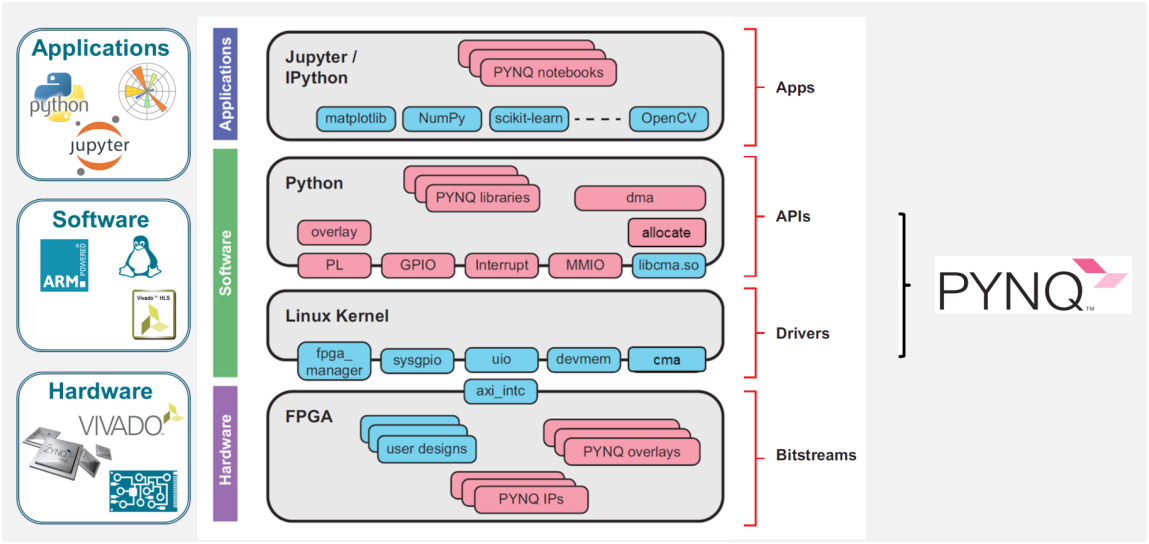
Step 3: Transfer to Board and Predict After transferring the `package.tar.gz` to the board's PS, the bitstream can be loaded and used to run predictions:

```
from axi_stream_driver import NeuralNetworkOverlay

# Initialize overlay and perform prediction
nn = NeuralNetworkOverlay('hls4ml_nn.bit', X_test.shape, y_test.shape)
y_hw, latency, throughput = nn.predict(X_test, profile=True)
```

The `predict()` method:

- Loads the bitstream onto the PL.
- Allocates memory buffers.
- Sends input data (`X_test`) to PL and retrieves predictions.
- Reports latency and throughput metrics.



3.3. Model Optimization and Configuration in `hls4ml`

This section explains the optimization techniques and configuration parameters in `hls4ml` that make it suitable for deploying neural networks on resource-constrained hardware like FPGAs.

3.3.1. Quantization

Quantization is the process of reducing the bit-width of numbers used in a neural network—such as weights, biases, and activations—from 32-bit floating-point (FP32) to lower-bit fixed-point or integer formats (e.g., 8-bit, 6-bit). It plays a crucial role in:

- Reducing model size
- Minimizing memory bandwidth
- Accelerating inference latency
- Lowering power consumption

Example: A 6-bit quantized ReLU activation can be expressed in `QKeras` as:

```
quantized_bits(6, 0, alpha=1)
```

Where:

- `6` is the total number of bits
- `0` fractional bits (i.e., fixed-point integer format)
- `alpha=1` applies a scaling factor

3.3.2. Pruning

Pruning refers to the removal of low-magnitude or insignificant weights from a trained network. It helps reduce parameter count, speed up inference, and improve hardware efficiency.

Use Case: FPGA deployments benefit significantly from pruning by:

- Reducing DSP and LUT consumption
- Minimizing flip-flop usage
- Allowing sparsity-aware memory storage

Example: Using TensorFlow Model Optimization Toolkit:

```
pruning_params = {
    "pruning_schedule": pruning_schedule.ConstantSparsity(0.5, begin_step=2000, frequ
}
model = prune.prune_low_magnitude(model, **pruning_params)
```

This enforces a 50% sparsity after 2000 steps and prunes every 100 steps.

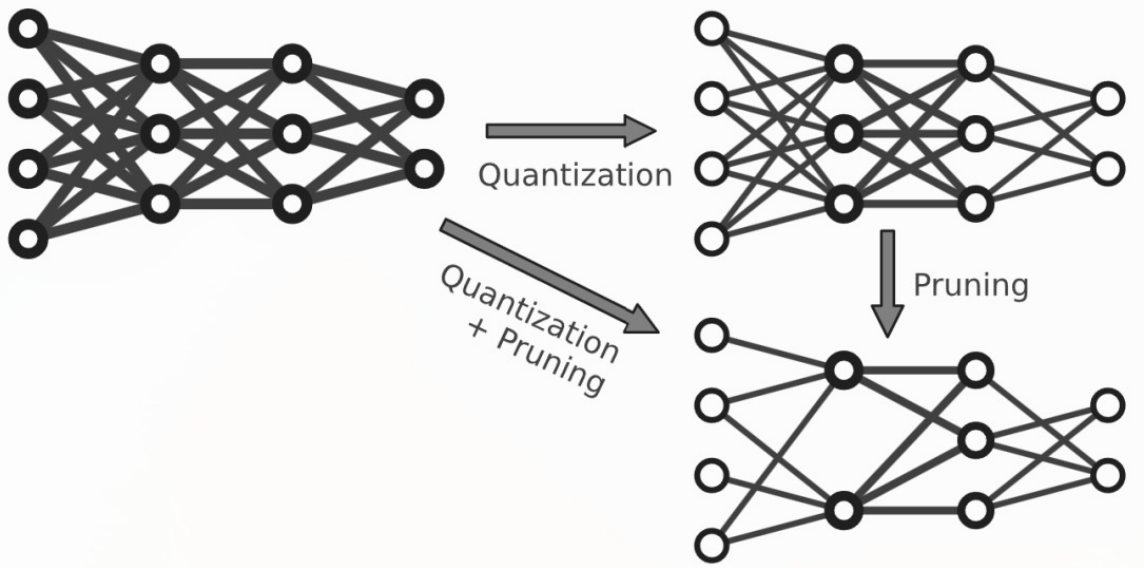


Figure 3.1: Quantization and Pruning

3.3.3. Reuse Factor in `hls4ml`

The reuse factor is a key design parameter that dictates how many times a hardware multiplier is reused during inference. It provides a balance between:

- **Low reuse factor:** *High parallelism*, low latency, high resource usage
- **High reuse factor:** *Reduced hardware usage*, higher latency

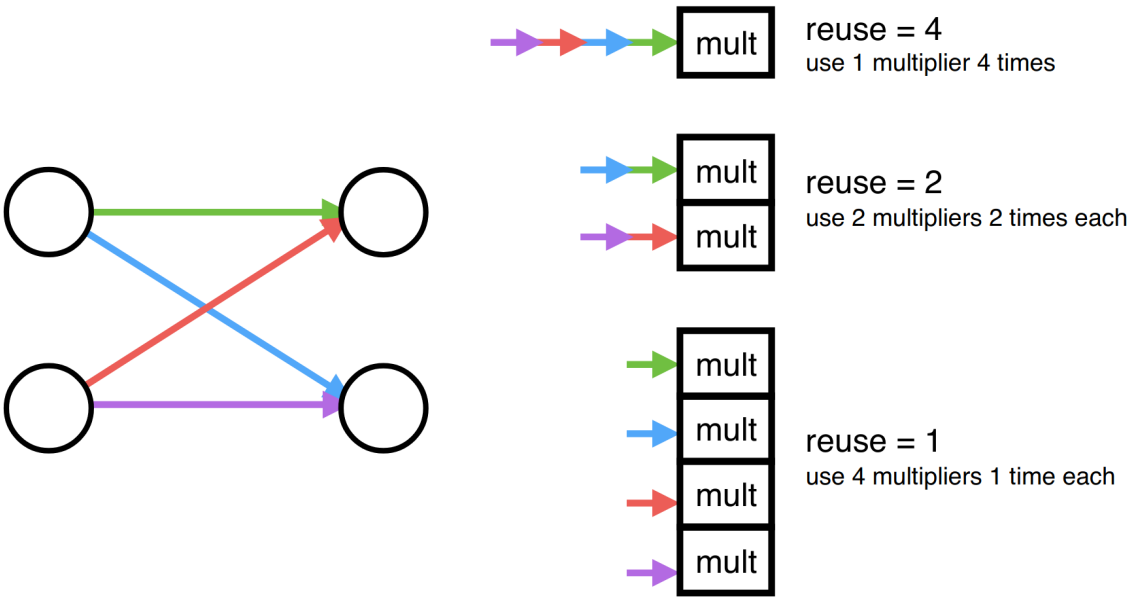


Figure 3.2: Reuse Factor

> **Note**
>
> Choosing the right reuse factor is essential to meeting latency and resource utilization constraints in FPGA synthesis.

3.3.4. Granularity of Configuration

`hls4ml` allows reuse factor and precision settings to be applied at two levels:

Model-Level Granularity (`granularity='model'`)

Applies uniform settings across the entire model.

```
config = hls4ml.utils.config_from_keras_model(model, granularity='model')
config['Model']['Precision'] = 'fixed<16,6>'
config['Model']['ReuseFactor'] = 4
```

Simple and fast for small networks, but lacks per-layer tuning.

Name-Level Granularity (`granularity='name'`)

Allows layer-wise customization of precision and reuse.

```
config = hls4ml.utils.config_from_keras_model(model, granularity='name')
config['LayerName']['fc1']['Precision'] = 'fixed<10,4>'
config['LayerName']['fc1']['ReuseFactor'] = 2
```

Ideal for large or quantized models where certain layers demand higher precision or tighter latency.

Comparison Table

Feature	`granularity='model'`	`granularity='name'`
Scope	Global	Per-layer
Precision Control	Uniform	Custom per-layer
Reuse Factor Control	Uniform	Custom per-layer
Complexity	Low	High
Quantization Support	Limited	Full
Hardware Optimization	Basic	Fine-tuned

Table 3.2: Comparison of granularity settings in `hls4ml`

Guidelines for Use

Scenario	**Recommended Granularity**
Simple feedforward models	`model`
Small CNN without quantization	`model`
Complex or large CNNs	`name`
QKeras or QONNX quantized models	`name`
Layer-wise optimization required	`name`

Table 3.3: When to use each granularity setting

3.3.5. I/O Type Configuration in `hls4ml`

The `IOType` parameter in `hls4ml` determines how data is passed between layers in the resulting FPGA design. It is a crucial configuration that directly influences **latency**, **resource usage**, and **scalability**.

Two primary modes of operation are supported:

- `io_parallel` (default): Suitable for low-latency applications.
- `io_stream`: Recommended for large or deep models with limited FPGA resources.

`io_parallel`

In this mode, data is exchanged between layers using fully parallel, memory-mapped arrays. Conceptually, this corresponds to standard C/C++ arrays where every element is accessible simultaneously.

- Maximum throughput — each layer operates at full parallelism.
- Best suited for MLPs and small CNNs targeting lowest latency.
- May lead to synthesis failure in resource-constrained FPGAs due to high DSP, LUT, and BRAM usage.

`io_stream`

This mode employs **streaming data transfer** via FIFO (First-In-First-Out) buffers. Data flows from layer to layer one "pixel" at a time, though all channels in a pixel are processed in parallel.

- Supports sequential dataflow, ideal for larger CNNs and RNNs.
- Dense (fully connected) layers receive all inputs as a single streamed vector.
- Reduces parallelism pressure, improving compatibility with mid-sized and larger FPGAs.

FIFO Buffers and Resource Usage

In `io_stream` mode, each layer communicates with the next through FIFO queues. These buffers consume BRAM or LUTs depending on synthesis options.
Note: `hls4ml` assigns conservative (large) FIFO depths by default, which can lead to excessive hardware usage. To address this, `hls4ml` offers a FIFO Depth Optimization Flow.

Strategy Parameter

In addition to `IOType`, `hls4ml` provides a `Strategy` parameter to define how core matrix-vector operations are implemented:

- **Latency-oriented:** Prioritizes speed using full loop unrolling (resource-intensive).
- **Resource-oriented:** Reuses hardware for lower footprint at the cost of latency.
- **Stable or backend-specific:** Balanced or vendor-optimized strategies depending on HLS backend.

Comparison Table

I/O Type	Characteristics	Best For
`io_parallel`	Full parallel memory access; minimum latency; high resource consumption.	Small models (MLPs, small CNNs); latency-critical designs.
`io_stream`	Data streamed one pixel at a time; uses FIFO buffers; scalable.	Large CNNs/RNNs; resource-constrained FPGAs.

Table 3.4: Comparison of I/O transfer modes in `hls4ml`.

Example Configuration (YAML)

```yaml
IOType: "io_stream"
Strategy: "Resource"
ReuseFactor: 4
Precision: "ap_fixed<10,4>"
```

Recommendations

- Use `io_parallel` when aiming for the lowest possible latency and when FPGA resources are sufficient.
- Use `io_stream` to reduce resource usage in large or deep networks.
- Choose the `Strategy` parameter based on application needs and backend capabilities.

Example Configuration

```
IOType: "io_stream"
```

Selection Strategy

Choosing the Right I/O Type

- Use `io_parallel` when minimal inference delay is required.
- Use `io_stream` for large or quantized models where resource savings are crucial.

Chapter 4

Setup and Quick Start

4.1. Setup and Quick Start

To begin developing hardware-accelerated machine learning pipelines using `hls4ml`, the first step is to configure the development environment and understand the essential components required to build and deploy a model on FPGA. This section outlines the installation procedure, dependencies, and a minimal example that demonstrates the full workflow—from defining a neural network in Keras to synthesizing it into hardware using High-Level Synthesis (HLS).

3.1.1 Installation

The most straightforward way to install `hls4ml` is through Python's package manager, `pip`. The following command installs the latest stable release:

```
pip install hls4ml
```

If profiling tools are needed—for example, to analyze performance or inspect layer-wise latency metrics—an extended version can be installed with:

```
pip install hls4ml[profiling]
```

Note: Although older versions of `hls4ml` were available on `conda-forge`, they are now outdated and unsupported. The only officially maintained method is through `pip`, and it is strongly recommended to follow this for compatibility and updates.

Installing the Development Version
Since `hls4ml` is under active development, many cutting-edge features, backend integrations, and experimental tools are available only in the development branch. To install this version directly from GitHub:

```
pip install git+https://github.com/fastmachinelearning/hls4ml@main→
```

This method is particularly useful for accessing:

- Advanced pruning and sparsity-aware training pipelines,
- Experimental backend support (e.g., Intel oneAPI),
- Per-layer configuration tuning for resource optimization.

4.1.1. Dependencies

The core package of `hls4ml` depends on Python 3.10 or higher. Most Python libraries such as NumPy, h5py, and SciPy are handled automatically by `pip`, but the compatibility with external toolchains and frontend ML frameworks introduces the following requirements:

- **TensorFlow (2.8 to 2.14)** and **QKeras** are required for the Keras model converter. Versions beyond TensorFlow 2.14 or Keras v3 are not currently supported.
- **ONNX (v1.4.0 or newer)** is necessary for converting models trained using ONNX-compatible frameworks.
- **PyTorch** is optional and only needed if PyTorch-based models are being converted.
- A **C++11-compliant compiler** such as `g++` is required for simulation and compilation from within Python.
- For the **oneAPI backend**, Intel's oneAPI toolkit with FPGA compiler must be installed for simulation and synthesis.

4.1.2. Synthesis Tool Requirements

Once the HLS project is generated using `hls4ml`, the actual synthesis and bitstream generation must be handled by vendor-specific toolchains. The supported backend synthesis tools and their compatible versions are listed in Table 4.1.

Synthesis Tool	Supported Versions
Xilinx Vivado HLS	2018.2 – 2020.1
Vitis HLS	2022.2 and above
Intel Quartus	20.1 – 21.4
oneAPI FPGA Toolkit	2024.1 – 2025.0
Catapult HLS	2024.1_1 or 2024.2

Table 4.1: Supported HLS backends for synthesis with `hls4ml`.

4.1.3. Quick Start

The following example demonstrates the full pipeline—from defining a neural network using TensorFlow/Keras to compiling and running predictions using the generated HLS model. This is a simplified use-case ideal for validating your setup.

Model Conversion, Compilation, and Synthesis Using `hls4ml`

The following Python script demonstrates the complete workflow of converting a Keras model into a hardware-synthesizable format using `hls4ml`, compiling it for simulation, and synthesizing it using a supported HLS toolchain such as Vivado or Vitis.

```python
import hls4ml
import tensorflow as tf
from tensorflow.keras.layers import Dense, Activation
import numpy as np

# Step 1: Define a basic feedforward neural network
model = tf.keras.models.Sequential()
model.add(Dense(64, input_shape=(16,), name='Dense',
                kernel_initializer='lecun_uniform'))
model.add(Activation('elu', name='Activation'))
model.add(Dense(32, name='Dense2', kernel_initializer='lecun_uniform'))
model.add(Activation('elu', name='Activation2'))

# Step 2: Generate default HLS configuration
config = hls4ml.utils.config_from_keras_model(model)
print(config)

# Step 3: Convert the model into an HLS project
hls_model = hls4ml.converters.convert_from_keras_model(
    model=model,
    hls_config=config,
    backend='Vitis'  # Use 'Vivado' if targeting Vivado HLS
)

# Step 4: Compile the project for simulation
hls_model.compile()

# Step 5: Generate dummy input and run inference
X_input = np.random.rand(100, 16)
hls_prediction = hls_model.predict(X_input)

# Step 6: Trigger full HLS synthesis
hls_model.build()

# Step 7: Read Vivado synthesis report (optional)
hls4ml.report.read_vivado_report('my-hls-test')
```

Detailed Explanation

`import hls4ml`
This line imports the core `hls4ml` library, which contains utilities for translating machine learning models into high-level synthesis projects and managing backend synthesis workflows.

`import tensorflow as tf`
TensorFlow is used here as the frontend for building and training the neural network. The Keras API within TensorFlow is particularly compatible with `hls4ml`.

`from tensorflow.keras.layers import Dense, Activation`
This imports the layer classes used to construct the feedforward network. Only supported layer types (e.g., Dense, Conv2D, etc.) should be used when targeting FPGAs with `hls4ml`.

```
import numpy as np
```

`numpy` is used for generating input data during inference testing.

```
model = tf.keras.models.Sequential()
```

Initializes a sequential model container. Layers added to this model are stacked linearly.

```
model.add(Dense(64, input_shape=(16,), ...))
```

Adds a fully connected (dense) layer with 64 neurons and an input shape of 16. The name 'Dense' helps hls4ml identify this layer in the configuration.

```
model.add(Activation('elu', name='Activation'))
```

Adds an ELU activation function after the first dense layer. Non-linear activations like ELU and ReLU are supported and implemented in hardware using lookup tables or piecewise approximations.

```
model.add(Dense(32, name='Dense2', ...))
```

Adds a second dense layer with 32 neurons. The same weight initializer is used for consistency.

```
model.add(Activation('elu', name='Activation2'))
```

Final activation layer completes the feedforward network.

`config = hls4ml.utils.config_from_keras_model(model)`

This is one of the most important steps. It auto-generates a configuration dictionary that defines:

- Data precision for each layer (fixed-point or floating-point),
- Reuse factors to optimize latency vs. resource trade-offs,
- Pipeline depth and parallelization settings.

```
print(config)
```

Displays the auto-generated configuration so users can inspect or manually modify values such as bit-width or parallelization.

`hls_model = hls4ml.converters.convert_from_keras_model(...)`

This converts the Keras model to C++ files and a full HLS project. The backend argument defines the toolchain used. Internally, this function:

- Generates firmware code (`.cpp`, `.h`),
- Writes a testbench and simulation files,
- Creates build scripts (e.g., `build_prj.tcl` for Vivado).

`hls_model.compile()`

Prepares the model for simulation in Python without involving hardware. This helps verify the correctness of the hardware logic using numpy arrays as input.

```
X_input = np.random.rand(100, 16)
```

Creates 100 synthetic input samples, each with 16 features. These are passed to both the software and hardware models to check consistency.

```
hls_prediction = hls_model.predict(X_input)
```

Runs the HLS-translated model in simulation mode. The output should closely match that of the original Keras model.

```
hls_model.build()
```

Starts the synthesis process using the specified backend (Vivado or Vitis). This step generates IP cores, timing estimates, and hardware utilization reports. It is the most time-consuming part of the workflow.

```
hls4ml.report.read_vivado_report('my-hls-test')
```

This utility reads and parses the Vivado synthesis reports. It presents key metrics:

- Latency in clock cycles,
- BRAM, DSP, LUT usage,
- Throughput and timing constraints.

Once the above steps are completed, we will have a fully synthesized HLS implementation of your Keras model, ready to be deployed on an FPGA platform. This pipeline forms the foundation for more complex neural architectures in practical applications.

Chapter 5

Design and FPGA Deployment of a Quantized MLP using hls4ml

5.1. MNIST Classification and FPGA Acceleration using `hls4ml`

5.1.1. Introduction to MNIST

The MNIST dataset is a widely recognized benchmark in the machine learning community. It consists of 70,000 grayscale images of handwritten digits (0 through 9), each of size 28×28 pixels. This dataset is split into 60,000 training images and 10,000 testing images and is often used for validating classification algorithms. Its low-dimensional structure makes it ideal for testing rapid prototyping and model-to-hardware conversion workflows like `hls4ml`.

5.1.2. Typical CPU-Based Inference Workflow

A conventional approach to digit classification involves:

- Preprocessing the image data (resizing, flattening, and normalization).
- Constructing and training a Multi-Layer Perceptron (MLP) model using Keras or TensorFlow.
- Evaluating the trained model on test data using CPU.

While effective, CPU-based inference becomes suboptimal in latency-sensitive, low-power edge applications.

5.1.3. Why FPGA Acceleration?

Field Programmable Gate Arrays (FPGAs) provide hardware-level acceleration that benefits real-time ML inference:

- **Low Latency:** By pipelining operations, FPGAs can achieve cycle-level prediction speed.
- **Parallelism:** Independent operations can be executed concurrently.

- **Energy Efficiency:** Custom data paths reduce power overhead compared to GPUs.

5.1.4. MLP on FPGA using `hls4ml`

The core idea is to convert a quantized, pruned MLP model into synthesizable C++ using `hls4ml`, and then deploy it onto an FPGA platform like the PYNQ-Z2. The generated HLS code is compiled using Xilinx Vivado to produce a deployable bitstream and IP block.

5.1.5. MLP

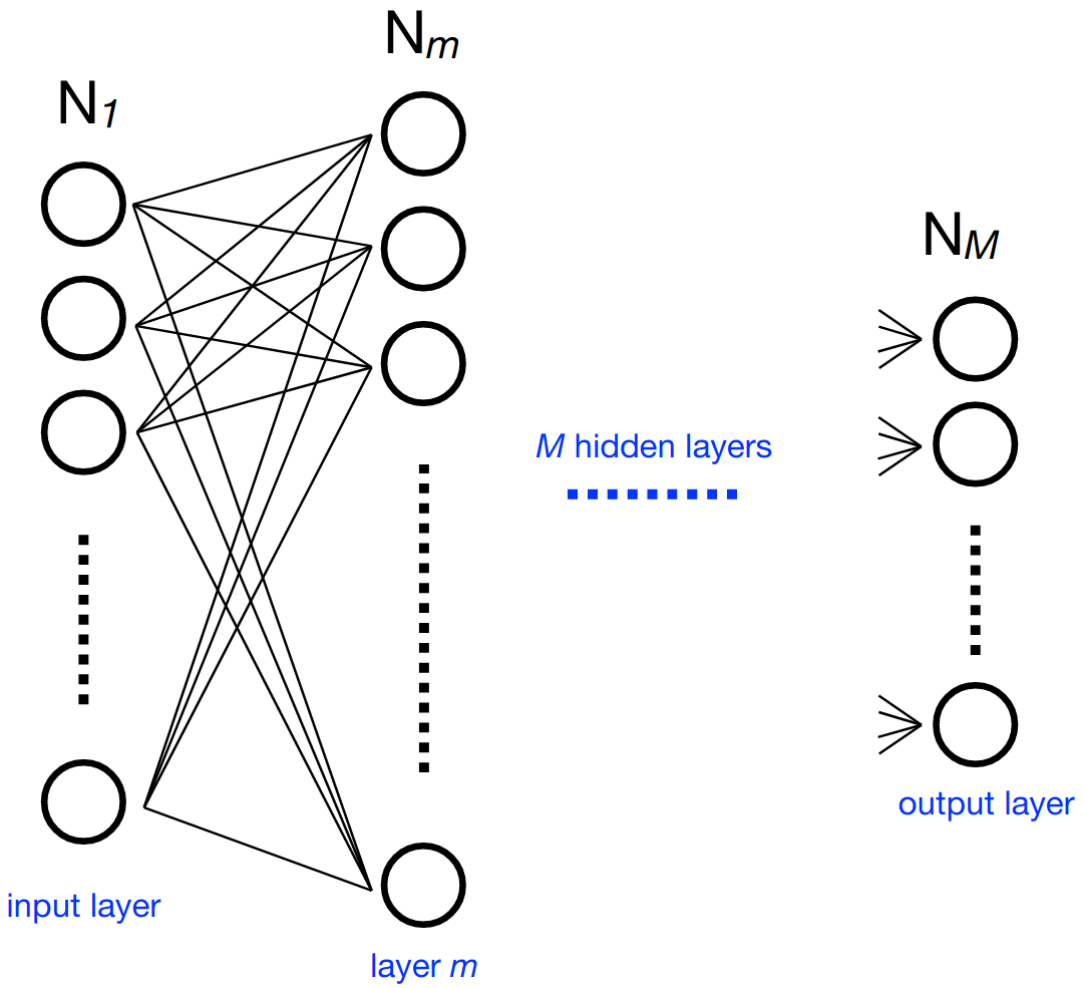


Figure 5.1: MLP

Consider a standard multilayer perceptron (MLP). Each neuron in a layer m (with N_m neurons) computes its output by summing the outputs of the previous layer (x_{m-1}), each multiplied by corresponding weights, and adding a bias term. This is followed by the application of an activation function $g_m(\cdot)$. The complete expression for the output of layer m is:

$$\mathbf{x}_m = g_m \left(W_{m,m-1} \cdot \mathbf{x}_{m-1} + \mathbf{b}_m \right)$$

where:

- $W_{m,m-1}$ is an $N_m \times N_{m-1}$ weight matrix,
- $\mathbf{b}_m$ is the bias vector for layer m,
- $g_m(\cdot)$ is the activation function (e.g., ReLU, sigmoid),
- $\mathbf{x}_m$ is the resulting activation vector.

In `hls4ml`, this matrix-vector operation is mapped directly to a hardware-friendly structure using High-Level Synthesis (HLS). Key features include:

- **Parallel computation:** Neurons in a layer can be evaluated in parallel, depending on the reuse factor and available resources.
- **Pipelining:** Activations are computed sequentially with pipeline stages that allow new inputs to enter after a defined initiation interval (II), boosting throughput.
- **Precomputation:** Constants and activation tables (e.g., ReLU thresholds, softmax exponentials) can be precomputed or implemented as lookup tables to reduce latency.

This mapping is what allows `hls4ml` to convert a trained MLP into a deeply pipelined, quantized digital circuit that runs efficiently on FPGAs such as the PYNQ-Z2.

5.1.6. Python Implementation with Quantization, Pruning, and Conversion

Step 1: Import Required Libraries

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1
from qkeras.qlayers import QDense, QActivation
from qkeras.quantizers import quantized_bits, quantized_relu
from tensorflow.keras.layers import Activation
from tensorflow_model_optimization.python.core.sparsity.keras import prune,
↪  pruning_callbacks, pruning_schedule
from tensorflow_model_optimization.sparsity.keras import strip_pruning
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
import numpy as np
import tensorflow as tf
import hls4ml
from sklearn.metrics import accuracy_score
```

Step 2: Set Random Seeds

```
seed = 0
np.random.seed(seed)
tf.random.set_seed(seed)
```

Step 3: Load and Preprocess the MNIST Dataset

```
from tensorflow.keras.datasets import mnist
(X, y), (X_test, y_test) = mnist.load_data()

X = tf.image.resize(X[..., np.newaxis], (14, 14)).numpy().squeeze()
X_test = tf.image.resize(X_test[..., np.newaxis], (14, 14)).numpy().squeeze()

X = X.reshape(-1, 14 * 14).astype('float32') / 255.0
X_test = X_test.reshape(-1, 14 * 14).astype('float32') / 255.0

y = to_categorical(y, 10)
y_test = to_categorical(y_test, 10)

X_train_val, X_valid, y_train_val, y_valid = train_test_split(X, y,
↪  test_size=0.2, random_state=42)
```

Step 4: Define QKeras Model with Pruning

```
model = Sequential()
model.add(QDense(32, input_shape=(14 * 14,), name='fc1',
                 kernel_quantizer=quantized_bits(6, 0, alpha=1),
                 bias_quantizer=quantized_bits(6, 0, alpha=1),
                 kernel_initializer='lecun_uniform',
                 kernel_regularizer=l1(0.0001)))
model.add(QActivation(activation=quantized_relu(6), name='relu3'))
model.add(QDense(10, name='output',
                 kernel_quantizer=quantized_bits(6, 0, alpha=1),
                 bias_quantizer=quantized_bits(6, 0, alpha=1),
                 kernel_initializer='lecun_uniform',
                 kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='softmax'))

pruning_params = {
    "pruning_schedule": pruning_schedule.ConstantSparsity(
        0.5, begin_step=0, frequency=100)
}
model = prune.prune_low_magnitude(model, **pruning_params)
```

Step 5: Compile and Train the Model

```
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train_val, y_train_val,
          batch_size=256,
          epochs=20,
          validation_data=(X_valid, y_valid),
          shuffle=True,
          callbacks=[pruning_callbacks.UpdatePruningStep()])

model = strip_pruning(model)
```

Step 6: Inspect Model Sparsity

```
total_params, zero_params = 0, 0
for layer in model.layers:
    for weight in layer.get_weights():
        total_params += weight.size
        zero_params += np.sum(weight == 0)
print(f"Sparsity: {100 * zero_params / total_params:.2f}%")
```

Step 7: Create `hls4ml` Configuration

```
config = hls4ml.utils.config_from_keras_model(
    model, default_precision='fixed<16,6>', granularity='name')

config['LayerName']['softmax']['exp_table_t'] = 'ap_fixed<4,1>'
config['LayerName']['softmax']['inv_table_t'] = 'ap_fixed<4,1>'
config['Model']['Strategy'] = 'Resource'
config['LayerName']['fc1']['ReuseFactor'] = 49
config['LayerName']['output']['ReuseFactor'] = 1

for layer in ['fc1', 'output']:
    config['LayerName'][layer]['Precision']['weight'] = 'ap_fixed<4,1>'
    config['LayerName'][layer]['Precision']['bias'] = 'ap_fixed<4,1>'

for layer in config['LayerName'].keys():
    config['LayerName'][layer]['Trace'] = True
```

Explanation:

- `default_precision='fixed<16,6>'`: Sets the default datatype to a fixed-point representation with 16 total bits and 6 integer bits. This reduces memory and DSP usage on the FPGA but must be carefully chosen to avoid accuracy loss.

- `granularity='name'`: Allows layer-specific configuration by referencing each layer by its name. This provides fine-grained control over precision and reuse.
- `exp_table_t, inv_table_t = 'ap_fixed<4,1>'`: Defines the precision used by the softmax function's exponential and inverse lookup tables. A low bitwidth is chosen to minimize resource usage, with minimal impact on accuracy.
- `Strategy = 'Resource'`: Chooses a compilation strategy that prioritizes minimum hardware resource usage (LUTs, BRAMs, DSPs) over latency. Suitable for edge devices like the PYNQ-Z2 with limited logic fabric.
- `ReuseFactor = 49 (fc1)`: Specifies that each multiplier will be reused 49 times in the first fully connected layer. This drastically reduces DSP consumption but increases inference latency.
- `ReuseFactor = 1 (output)`: Indicates full parallelism in the output layer with no reuse. This results in minimal latency at the cost of more DSP blocks.
- `Precision['weight/bias'] = 'ap_fixed<4,1>'`: Uses 4-bit fixed-point precision with 1 integer bit for both weights and biases. This significantly reduces memory and logic usage at the cost of aggressive quantization.
- `Trace = True`: Enables tracing of layer outputs during simulation. This is useful for debugging and comparing FPGA vs software inference results. It does not impact the actual deployed hardware.

Summary of Configuration Settings and Their Hardware Impact:

Configuration	Purpose	Hardware Impact
`default_precision`	Sets default datatype precision for weights/activations	Lowers resource usage; slight reduction in accuracy possible
`granularity='name'`	Enables per-layer customization	Fine-grained reuse factor and precision control
`exp_table_t, inv_table_t = 'ap_fixed<4,1>'`	Reduces softmax table size	Minimal LUTs/BRAM usage; low precision tolerated
`Strategy = 'Resource'`	Compiles model to use fewer FPGA resources	Reduces DSP/LUTs; increases latency
`ReuseFactor = 49 (fc1)`	Increases multiplier reuse in hidden layer	Fewer DSPs used; higher latency
`ReuseFactor = 1 (output)`	Enables full parallelism in output layer	Fast inference; higher DSP usage
`Precision = 'ap_fixed<4,1>'`	Sets custom quantized precision for weights and biases	Smaller model footprint; aggressive quantization
`Trace = True`	Logs intermediate values during simulation	Useful for debugging; no hardware cost

Table 5.1: Detailed breakdown of `hls4ml` configuration parameters and their hardware implications.

Step 8: Convert and Synthesize the HLS Model

```
hls_model = hls4ml.converters.convert_from_keras_model(
    model,
    hls_config=config,
    output_dir='hls4ml_prj_pynq',
    backend='VivadoAccelerator',
    board='pynq-z2',
    part='xc7z020clg400-1'
)

hls_model.compile()
hls_model.build(csim=False, export=True, bitfile=True)

hls4ml.report.read_vivado_report('hls4ml_prj_pynq')
```

Explanation:

- `convert_from_keras_model(...)`: Converts the trained Keras model into a High-Level Synthesis (HLS) project. The following parameters define the conversion target and synthesis environment:

  - `hls_config=config`: Supplies the customized precision, reuse, and strategy settings created in Step 7.
  - `output_dir='hls4ml_prj_pynq'`: Specifies the directory where the HLS project and reports will be saved.
  - `backend='VivadoAccelerator'`: Indicates that the model will be compiled for a Vivado HLS-based accelerator, targeting an FPGA platform.
  - `board='pynq-z2'`: Selects the PYNQ-Z2 board preset, allowing hls4ml to optimize I/O and memory interfaces for this specific board.
  - `part='xc7z020clg400-1'`: Specifies the exact FPGA part number on the Zynq-7000 chip. Required for Vivado synthesis.

- `hls_model.compile()`: Analyzes the converted model for shape compatibility, pipeline validity, and operator support in Vivado.
- `hls_model.build(csim=False, export=True, bitfile=True)`:
  - `csim=False`: Skips C-level simulation (optional step for faster builds).
  - `export=True`: Exports the model as a PYNQ-compatible IP block, including C++ sources and hardware description.
  - `bitfile=True`: Triggers Vivado to synthesize, implement, and generate the final FPGA bitstream (`.bit`) file.
- `hls4ml.report.read_vivado_report(...)`: Parses Vivado-generated reports and prints estimated latency, initiation interval (II), resource usage (LUTs, DSPs, BRAMs), and clock frequency.

Command/Parameter	Purpose	Hardware Impact
`convert_from_keras_model (...)`	Converts the model to HLS C++ project	Prepares IP core and enables RTL synthesis
`output_dir='hls4ml_prj_pynq'`	Stores build files and Vivado output	Organizes artifacts for IP integration
`backend='VivadoAccelerator'`	Sets flow for FPGA hardware acceleration	Enables mapping to programmable logic fabric
`board='pynq-z2'`	Uses board-specific interfaces and defaults	Ensures compatibility with ZYNQ SoC overlays
`part='xc7z020clg400-1'`	Defines FPGA chip constraints for Vivado	Affects synthesis results and resource mapping
`compile()`	Verifies operator graph and datatype correctness	Pre-synthesis model checks
`build(csim=False)`	Builds project without simulation to save time	Skips functional C test step
`export=True`	Generates PYNQ-ready IP and driver files	Required for deployment on Zynq-based systems
`bitfile=True`	Synthesizes bitstream and generates IP core	Produces final hardware binary for FPGA
`read_vivado_report()`	Reads and formats Vivado report summary	Shows timing, area, and performance metrics

Table 5.2: Summary of `hls4ml` synthesis process and the impact of each configuration directive.

Vivado Block Design:

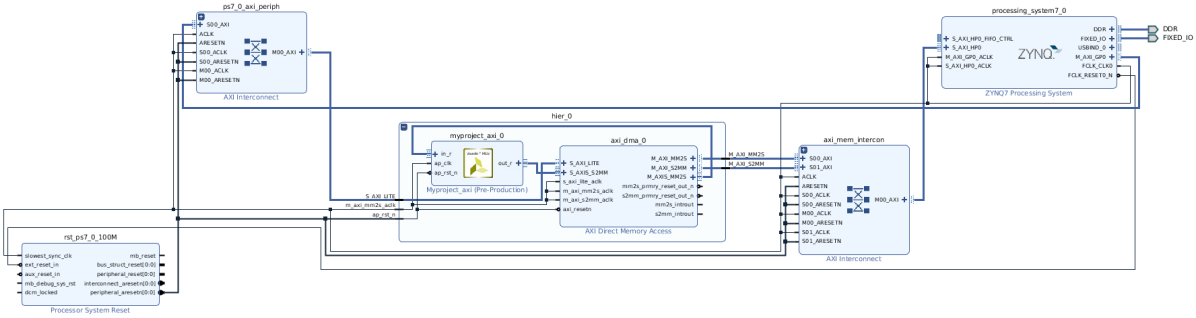


Figure 5.2: IP block design of the MLP model deployed on PYNQ-Z2 FPGA using `hls4ml`.

Loading Bitstream and Driver on PYNQ-Z2

After generating the hardware design using `hls4ml` and Vivado, deployment on the PYNQ-Z2 board requires three essential components:

- **Bitstream file (.bit):** Contains the synthesized hardware configuration used to program the FPGA fabric.
- **Hardware handoff file (.hwh):** Describes the FPGA design's structure and interface mappings.
- **Python driver script (e.g., `axi_stream_drive.py`):** Used to interface with the accelerator from the PYNQ's Python runtime using AXI Stream.

Typical Output Paths:

Bitstream (.bit): `mnist/hls4ml_prj/myproject.vivado_accelerator/project_1.runs/impl_1/design_1_wrapper.bit`

HWH File (.hwh): `mnist/hls4ml_prj/myproject.vivado_accelerator/project_1.srcs/sources_1/bd/design_1/hw_handoff/design_1.hwh`

AXI Stream Driver Script: `mnist/hls4ml_prj/axi_stream_drive.py`

These files must be securely transferred to the PYNQ board using `scp` or equivalent methods. Once copied, they are used within a Jupyter Notebook to configure the PL and interact with the neural network accelerator.

Resources in Vivado for Different Reuse Factors

The following table summarizes the hardware resource utilization and power estimation for two different configurations of reuse factors used during synthesis in Vivado. The reuse factors affect the number of DSPs and FFs consumed, which in turn influences the power and performance characteristics of the model deployed on FPGA.

x	y	BRAM	DSP	FF	LUT	PL Power Estimation
29	1	4	58	32	41	1.717 W
98	32	4	29	27	33	1.692 W

Table 5.3: FPGA resource usage for different configurations based on reuse factors.

Inference from Reuse Factor Resource Analysis

From the results presented in Table 5.3, we can draw the following observations:

- **Reuse Factor Trade-off:** A higher reuse factor (e.g., $x = 98$, $y = 32$) leads to significantly reduced usage of DSPs, FFs, and LUTs. This is because the same multiplier and computation hardware are reused across more operations, reducing parallel hardware replication.
- **Resource Efficiency:** With $x = 98$, the design consumes only 29 DSPs and 27 FFs compared to 58 DSPs and 32 FFs when the reuse factor is $x = 29$. This indicates that aggressive reuse strategies allow the design to fit into smaller FPGAs or leave room for additional logic on the same device.

- **Power Reduction:** The power estimation slightly decreases from 1.717 W to 1.692 W with higher reuse, showing that reduced parallelism can marginally improve energy efficiency. However, the change is not dramatic, suggesting other factors like clock rate and memory access may also influence power.
- **Latency Consideration:** While higher reuse reduces resource usage, it may increase the overall latency per inference if pipelining and initiation intervals are not well optimized. Thus, careful balancing is needed between resource efficiency and speed, depending on application constraints.

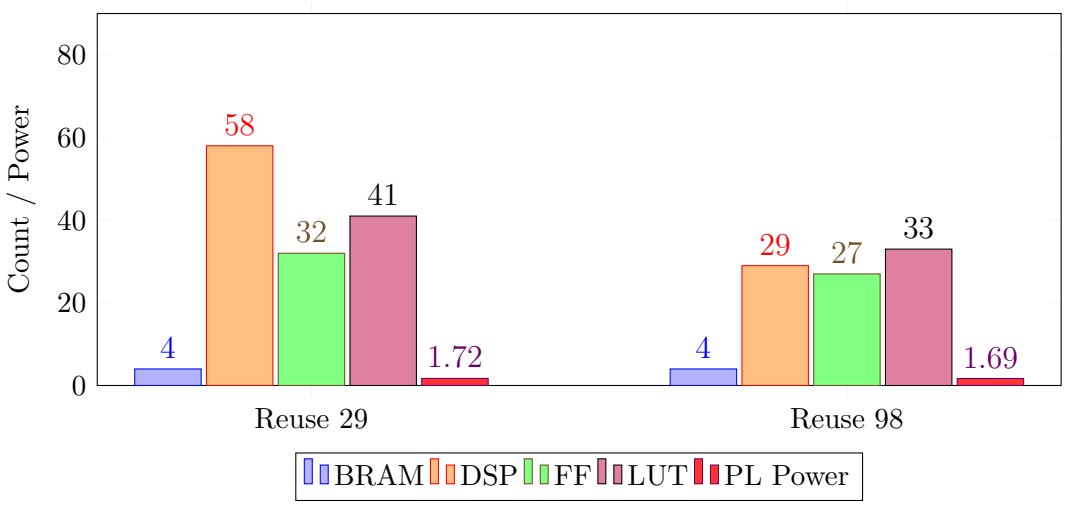


Figure 5.3: Resource usage and power comparison for different reuse factors.

Overall, the experiment confirms that reuse factor tuning plays a vital role in optimizing both logic resource usage and power consumption during HLS-based neural network deployment on FPGAs.

Chapter 6

Design and Deployment of Network Traffic Classifier on PYNQ

6.1. Introduction to Network Traffic Classification

Network traffic classification is the process of identifying and labeling data packets flowing through a network based on predefined categories such as application type, protocol, or threat level. This task is crucial in modern networks for applications such as intrusion detection, Quality of Service (QoS) enforcement, bandwidth optimization, and anomaly detection.

As the volume and diversity of network traffic have increased—driven by IoT devices, cloud services, and encrypted communications—traditional rule-based systems have become less effective. In this context, machine learning-based traffic classifiers offer a dynamic and scalable solution. By learning patterns from historical data, these models can classify previously unseen traffic more accurately than static filters.

6.2. Features for Network Traffic Classification

Effective classification of network traffic relies on extracting discriminative features that capture the temporal, statistical, and behavioral characteristics of packet flows. These features are typically derived from the metadata of packets rather than payload content, which allows classification even when encryption is used (e.g., HTTPS, VPN). In this work, the focus is on extracting a set of time-based and flow-level features in the FPGA's programmable logic (PL), which are then used to infer traffic types such as chat, video streaming, browsing, etc.

The main categories of features used in this system are described below.

1. Flow Duration

Flow duration refers to the total time for which a flow remains active. It is calculated as the difference between the timestamps of the first and last packets in a flow. Certain types of traffic such as chat or browsing tend to have short-lived flows, while video streaming or file transfers often generate longer-lasting sessions.

- **Feature:** `duration` $= t_{\text{last}} - t_{\text{first}}$

- **Use:** Helps distinguish between bursty, interactive traffic (e.g., chat) and persistent traffic (e.g., streaming).

2. Flow Inter-Arrival Time (IAT) Features

Inter-arrival time (IAT) measures the time gap between consecutive packets. In this system, both the raw IAT and statistical descriptors such as minimum, maximum, mean, and standard deviation are computed over a window of packets within each flow.

- **Features:**
  - `min_fiat`, `max_fiat`, `mean_fiat`, `std_flowiat`
- **Use:** Chat traffic often has larger idle times between messages, whereas video packets may be sent in tightly packed bursts with low IAT variance.

3. Active and Idle Period Features

The "active" period is defined as a burst duration where packets are continuously exchanged, whereas the "idle" period refers to silent gaps between such bursts. These values are computed by detecting packet activity windows that exceed or fall below a specific threshold.

- **Active Features:**
  - `min_active`, `mean_active`, `max_active`, `std_active`
- **Idle Features:**
  - `min_idle`, `mean_idle`, `max_idle`, `std_idle`
- **Use:** Interactive apps (e.g., messaging) show short active periods followed by long idle phases. In contrast, video or audio streams exhibit long, consistent active periods with minimal idle time.

4. Packet Count per Flow (Optional)

While not always included due to memory constraints, the number of packets observed per flow can be a useful heuristic, especially when evaluating traffic bursts.

- **Feature:** `packet_count`
- **Use:** Helps detect bulk transfers or streaming sessions that produce a larger number of packets per flow window.

5. Port and Protocol Metadata (Excluded in PL)

Although traditional systems sometimes include source/destination port numbers and transport-layer protocols as features, this implementation excludes them from FPGA-based extraction due to limited relevance in encrypted or obfuscated traffic and the focus on behavioral features only.

All the above features are computed over a fixed window duration of 2 seconds per flow. Rather than relying on a fixed number of packets (e.g., 16 or 32), the system dynamically gathers all packets belonging to a flow that arrive within this time interval. This allows better temporal representation of activity patterns, making the system more robust to variable-rate traffic such as streaming or bursty applications.

This approach provides a balance between feature richness and hardware feasibility, making it suitable for real-time classification on edge devices like the PYNQ-Z2.

6.2.1. Flow-Based Feature Extraction Pipeline on CPU

The implemented system performs feature extraction from network packets in a structured, modular way. Packets are first parsed and grouped into flows, after which flow-level features are computed over a fixed time window of 2 seconds. This process is entirely carried out on the CPU before model deployment or offloading to FPGA is considered.

The feature extraction pipeline involves the following stages:

1. Packet Parsing and Flow Grouping

Each incoming packet is parsed to extract header-level metadata: source IP, destination IP, source and destination ports, protocol, length, and a timestamp. The fields are stored in a structure called `PacketHeaders`.

- These packets are buffered in a list during runtime.
- Every 2 seconds (as defined by `WINDOW_DURATION = 2`), the buffered packets are grouped into flows.
- Flows are identified uniquely by a 5-tuple: source IP, destination IP, source port, destination port, and protocol.

2. Flow-Level Timestamp and Size Aggregation

For each flow, two parallel vectors are maintained:

- `timestamps`: holds the timestamps of all packets in that flow.
- `packet_sizes`: stores the size (in bytes) of each corresponding packet.

These data are then passed to the core function `extract_features()` for detailed statistical analysis.

3. Inter-Arrival Time (IAT) Features

IAT features capture the delay between consecutive packets within a flow. These are crucial for distinguishing bursty traffic (like video streaming) from interactive or idle patterns (like browsing or chat).

- **min_iat, max_iat**: Minimum and maximum IAT in microseconds.
- **mean_iat, std_iat**: Average and standard deviation of IAT.

4. Flow Throughput Features

Using the first and last packet timestamps and packet sizes, the flow's data rate is computed:

- `flowPktsPerSecond` = total packets / flow duration
- `flowBytesPerSecond` = total bytes / flow duration

5. Active and Idle Time Features

This section of the code introduces a clever technique to distinguish between bursts of activity and pauses in packet transmission using a threshold-based method:

- The system defines an `ACTIVE_THRESHOLD` of 5000 ms.
- If the time between two consecutive packets exceeds this threshold, it marks the end of an *active period* and the start of an *idle period*.
- These bursts and gaps are used to compute:
  - **Active durations:** `min_active, mean_active, max_active, std_active`
  - **Idle durations:** `min_idle, mean_idle, max_idle, std_idle`

This design allows the system to characterize each flow's burstiness, which is extremely useful when distinguishing application-level behaviors—e.g., continuous streaming vs intermittent chat.

6. Final Output

Each flow is thus represented as a fixed-dimensional vector of 14 features. These features are printed for inspection and optionally passed to a classifier.

Summary of Features Extracted per Flow:

- Inter-arrival: `min_iat`, `max_iat`, `mean_iat`, `std_iat`
- Throughput: `flowPktsPerSecond`, `flowBytesPerSecond`
- Active: `min_active`, `mean_active`, `max_active`, `std_active`
- Idle: `min_idle`, `mean_idle`, `max_idle`, `std_idle`

All durations are internally computed in microseconds and later scaled to milliseconds during printing or normalization. These extracted features are stored in a `Features` struct and form the input to downstream classification models.

6.2.2. Real-Time Packet Interception and Dependencies

To extract features from live traffic, the system must intercept packets in real time before they are processed by the kernel's networking stack. This is achieved using a combination of `iptables` rules and the `NetfilterQueue` library, enabling user-space inspection and classification.

1. Packet Interception using NetfilterQueue

`NetfilterQueue` is a C++/Python interface to the Linux kernel's Netfilter framework. It allows selected packets to be sent from kernel space to user space for inspection or modification.

The system initializes a queue using:

```
nfq_create_queue(/* queue number = */ 0, /* callback = */ process_packet);
```

Packets are passed to a callback function where:

- They are parsed using the custom logic in `packet_parser.cpp`.
- Metadata (source/destination IP and port, protocol, timestamp, and length) is extracted and appended to a buffer for the current window.
- After processing, the packet is accepted via `NF_ACCEPT` to continue through the networking stack.

2. Traffic Redirection with `iptables`

`iptables` is a command-line utility for configuring the Linux kernel's packet filtering rules via the Netfilter framework. It controls how incoming, outgoing, and forwarded network packets are handled by defining rule chains in tables such as `INPUT`, `OUTPUT`, and `FORWARD`. These rules can instruct the kernel to accept, drop, modify, or redirect packets.

In the context of this system, `iptables` is used to divert selected network packets into a user-space queue where they can be processed in real time by `NetfilterQueue`. Without such redirection, all packets would be handled directly by the kernel and bypass the classification logic.

Redirection Rules Used:

```
sudo iptables -I INPUT -j NFQUEUE --queue-num 0
sudo iptables -I OUTPUT -j NFQUEUE --queue-num 0
sudo iptables -I INPUT -p tcp --sport 443 -j NFQUEUE --queue-num 1
```

- The first two rules insert (`-I`) redirection into the `INPUT` and `OUTPUT` chains, meaning all inbound and outbound packets will be sent to queue number 0.
- The third rule selectively redirects incoming TCP packets originating from port 443 (typically HTTPS) to queue 1. This allows for differentiated processing or logging of encrypted traffic.

Why This Is Needed:

- The Linux networking stack processes packets entirely in kernel space by default. This is fast but doesn't allow for machine learning–based inspection or flow-level feature extraction.
- By using `iptables` with `NFQUEUE`, packets are intercepted and passed to user-space programs (e.g., your C++ feature extractor) for inspection and classification.
- Once processed, a verdict is returned (e.g., `NF_ACCEPT`) to allow or drop the packet.

What Happens If Not Applied:

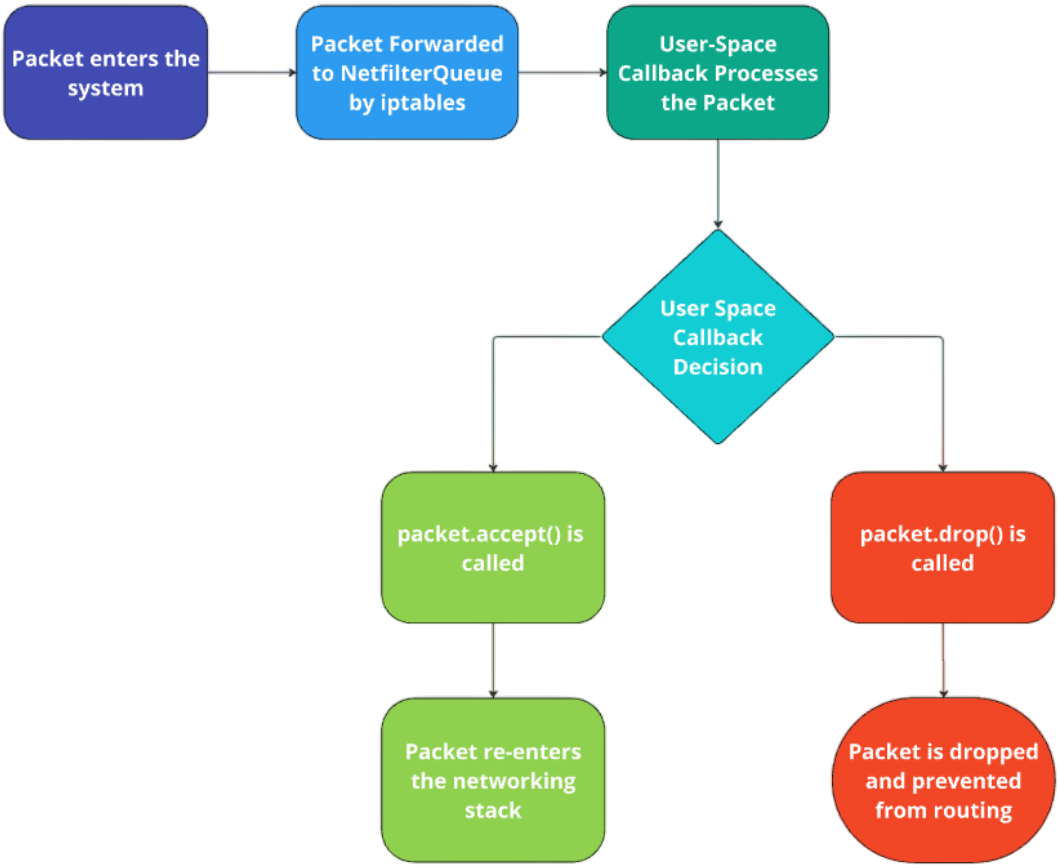
- If these `iptables` rules are not applied, the program will not receive any packets to process.
- As a result, no flows will be formed, no features will be extracted, and the entire classification pipeline will remain inactive.
- In the worst case, packet interception may partially work (if rules are misconfigured), causing random packet loss or degraded system performance.

Flushing Rules:

To avoid lingering or conflicting redirection, it is essential to flush the `iptables` table after the classification session ends:

```
sudo iptables -F
```

This command removes all rules from all chains, restoring default behavior and preventing accidental interference with other network applications.



3. Critical Dependencies and Requirements

The following libraries and system-level features are required for successful execution of the real-time feature extraction pipeline:

- **NetfilterQueue (libnetfilter_queue):** Provides the user-space interface for packet interception.
- **libnetfilter-queue-dev:** Must be installed using:

```
sudo apt install libnetfilter-queue-dev
```

- **Linux Kernel Support:** The Netfilter system is Linux-specific and not available on Windows or macOS.
- **Root Access:** Necessary for applying `iptables` rules and creating Netfilter queues.
- **Time Synchronization:** Timestamps used for feature extraction assume accurate system time, preferably synced using NTP.

Without proper setup of these components, packet capture and feature extraction would fail or interfere with regular system traffic. Therefore, both the logic and the underlying system environment play a crucial role in enabling real-time traffic classification.

6.3. Typical CPU-Based Implementation

A typical CPU-based machine learning pipeline for network classification involves several steps:

- **Packet Capture:** Network packets are collected in real-time using tools like NetfilterQueue, Wireshark, or tcpdump.
- **Feature Extraction:** From raw packets or flows, numerical features such as packet lengths, inter-arrival times, port usage, and flow durations are computed.
- **Preprocessing:** These features are normalized, aggregated, or windowed for model compatibility.
- **Inference:** A trained model (e.g., Random Forest, CNN, or MLP) predicts the class label for each flow or packet.
- **Label Interpretation:** The predicted class is mapped to a traffic type label such as `chat`, `video streaming`, `browsing`, or `file transfer`, depending on the characteristics of the underlying flow.

This pipeline can be implemented using Python and standard libraries like `scikit-learn`, `TensorFlow`, and `NetfilterQueue`, and can run effectively on general-purpose CPUs.

6.4. Limitations of CPU-Based Classification

While CPU-based classification provides flexibility and ease of development, it suffers from several limitations when applied to real-time or high-throughput environments:

- **High Latency:** The sequential nature of CPU execution introduces significant delays, especially when dealing with large volumes of packets.
- **Resource Bottlenecks:** CPUs are optimized for general-purpose computation, not high-throughput parallel tasks. Feature extraction and model inference can saturate CPU cores quickly.
- **Energy Inefficiency:** Running inference continuously on a CPU consumes considerable power, making it unsuitable for embedded or edge deployments.
- **Scalability Issues:** As network bandwidth and traffic complexity increase, CPUs may struggle to maintain classification accuracy within acceptable time budgets.

To address these constraints, hardware acceleration using FPGAs offers a compelling alternative. By offloading critical operations like classification to the programmable logic, one can achieve low-latency, high-throughput, and energy-efficient network analytics.

6.5. Optimizing Network Traffic Classification using FPGA Acceleration

While CPU-based implementations offer flexibility, they struggle to meet the demands of real-time, high-throughput network environments due to sequential execution, limited parallelism, and power inefficiency. FPGAs, on the other hand, provide a reconfigurable hardware platform capable of executing critical stages of the classification pipeline in parallel and at low latency.

Motivation for Hardware Acceleration

The network classification pipeline can be divided into two main stages:

1. **Feature Extraction:** Parsing packets and computing statistical features over flow windows.
2. **Classification:** Running an ML model on the extracted feature vector to predict the traffic class.

Both stages are computationally expensive when scaling to hundreds or thousands of flows per second. FPGA acceleration is beneficial due to the following reasons:

- **Parallelism:** Multiple features (e.g., IATs, active durations) can be computed concurrently.
- **Pipelining:** Feature computation can be structured into pipeline stages to process new packets every clock cycle.

- **Custom Precision:** FPGAs allow fine-grained control over bitwidths (e.g., 4-bit or 6-bit fixed-point), which reduces resource usage and improves speed.
- **Low Latency:** FPGAs avoid operating system overhead and interrupt latency seen in CPU-bound systems.
- **Energy Efficiency:** Dedicated logic and lower clock frequencies reduce power consumption.

Proposed FPGA-Based Architecture

In the optimized version of the system, only the classification stage is offloaded to the FPGA. The Programmable Logic (PL) performs inference using a quantized neural network model, while all feature extraction continues to take place in the Processing System (PS). This design leverages the flexibility of software for parsing and feature computation, while exploiting the speed and determinism of hardware for prediction.

- **PS (Processing System):** Responsible for real-time packet interception via `NetfilterQueue`, parsing headers, maintaining flow records, and computing statistical features such as inter-arrival time, active/idle durations, and flow throughput.
- **PL (Programmable Logic):** Hosts the machine learning inference engine. A quantized Multi-Layer Perceptron (MLP) model is converted using `hls4ml` and deployed as an IP core that consumes the extracted feature vector and outputs the predicted traffic label.
- **AXI Interface:** Serves as the communication bridge between the PS and PL. The PS writes feature vectors to BRAM via AXI-Lite/AXI-Stream, and reads back classification results after inference.

The overall data flow for this FPGA-accelerated implementation is:

`NetfilterQueue (PS)` → `Feature Extraction (PS)` → `AXI Write` → `ML Inference (PL)` → `AXI Read` → `PS Output/Log`

Impact of FPGA Offloading

Delegating only the inference stage to the FPGA results in a practical compromise between flexibility and performance:

- **Low Latency Prediction:** Inference takes place in a pipelined manner on the PL, achieving cycle-level performance for each input vector.
- **Reduced Power for Inference:** Unlike CPUs or GPUs, FPGAs do not consume additional energy for unused general-purpose logic, leading to power-efficient prediction suitable for edge environments.
- **Seamless Integration:** Since feature extraction remains in software, adding or modifying feature definitions can be done without re-synthesizing hardware, accelerating development cycles.

- **Modular Deployment:** This architecture enables reusability of the classification block across other projects or edge pipelines by abstracting the inference module as a standalone IP core.

In summary, the use of FPGA acceleration for model inference offers significant gains in speed and power efficiency, while retaining the adaptability and simplicity of software for flow tracking and feature computation.

Challenges with Real-Time Traffic Capture on PYNQ

The original pipeline was designed to intercept network traffic in real time using `NetfilterQueue`, a Linux kernel interface that forwards packets to user space for inspection and classification. However, during deployment on the PYNQ-Z2 board, a critical limitation was encountered: **NFQUEUE is not supported natively on PYNQ Linux**.

Why NFQUEUE is Crucial:

- It enables seamless interception of live traffic by redirecting selected packets from the kernel to user space.
- The C++ feature extraction code is tightly coupled with this framework; it expects to process incoming packets in real time via the NetfilterQueue callback.
- Without NFQUEUE, this packet-to-flow pipeline cannot function as intended, especially on embedded platforms.

Limitations of PYNQ and ZedBoard Linux Distributions:

Platforms like PYNQ-Z2 and ZedBoard typically run lightweight, custom Linux distributions (such as Petalinux) optimized for embedded use. These do not include NetfilterQueue capabilities due to kernel configuration constraints:

- Missing kernel options like `CONFIG_NETFILTER`, `CONFIG_NF_CONNTRACK`, and `CONFIG_NETFILTER_ADV`
- Absence of the critical `nfnetlink_queue` kernel module required by `libnetfilter_queue`.

Alternative Workarounds Considered:

1. **Rebuilding the Kernel:** It is technically possible to recompile the PYNQ kernel with the necessary Netfilter modules. However, this process is extremely complex, requiring:
   - Cross-compiling and configuring kernel options.
   - Ensuring module compatibility.
   - Reflashing and validating stability of the new OS image.

   Given the tight deployment schedule and high risk of bricking the board, this was deemed impractical.
2. **Switching to Raw Sockets or PCAP:** Tools like `libpcap`, `scapy`, or `dpkt` can capture packets in passive mode. These work well in offline setups and are supported on embedded Linux.

Final Resolution: PCAP-Based Classification

Due to these limitations, the final implementation resorts to a modified workflow:

- Network traffic is first recorded as a `.pcap` file using tools like `tcpdump` on a desktop or other capable system.
- This pcap file is then transferred to the PYNQ board.
- Features are extracted offline from the pcap data using the same C++ logic that was originally meant for NFQUEUE.
- The extracted features are fed into the deployed FPGA classifier via software routines.

Summary:

- The NFQUEUE-based real-time setup functions flawlessly on desktop Linux systems where kernel support exists.
- However, the PYNQ board lacks the necessary kernel modules, preventing real-time classification from being deployed natively.
- This constraint necessitated switching to offline pcap-based classification on PYNQ, where the hardware-accelerated model can still be evaluated on real traffic, albeit in a delayed processing mode.

Dataset Description: UNB CIC VPN Traffic Dataset

The dataset used for training and evaluation is derived from the **UNB CIC VPN Non-VPN Traffic Dataset**, provided by the Canadian Institute for Cybersecurity (CIC). It is a well-structured collection of real-world network flows labeled by traffic type, designed specifically for evaluating VPN and non-VPN activity detection.

Key Properties:

- Captures various applications (e.g., YouTube, Skype, AIM Chat, FTPS) under both VPN and Non-VPN conditions.
- Traffic is recorded over several days using different protocols and tunneling techniques.
- The raw data is provided as both PCAP and ARFF files, including pre-extracted features and labels.
- Features include statistical descriptors such as flow duration, inter-arrival time, idle time, active time, byte statistics, and packet counts.

Selected Subset for This Work:

This project focuses only on the `Scenario B` subset of the ARFF files, which contains clean, well-labeled flow-level samples suitable for training supervised models. The selected samples cover a range of encrypted and unencrypted traffic categories, mapped to a reduced set of behavior-based classes like:

- `Browsing`
- `Chat`
- `Streaming`
- `File Transfer`
- `VoIP`

Feature Selection and Preprocessing:

- From the full feature set, only a small number of statistically meaningful and FPGA-compatible features were retained.
- These include: `duration`, `min_fiat`, `max_fiat`, `mean_fiat`, `std_flowiat`, `min_active`, `mean_active`, `max_active`, `std_active`, `min_idle`, `mean_idle`, `max_idle`, and `std_idle`.
- Labels were encoded numerically using `LabelEncoder`, and all features were standardized using `StandardScaler`.

This cleaned and reduced dataset was saved as `Dataset.csv` and used both for software-based training and FPGA-based inference testing. It reflects realistic traffic patterns across multiple categories, making it highly suitable for validating real-time classification performance in edge computing environments.

Note on Dataset Size and Model Complexity

While this project utilizes a small, focused subset of the UNB CIC VPN dataset for proof-of-concept validation, it is entirely feasible to scale the approach to larger and more diverse traffic datasets. By incorporating additional classes (e.g., VPN vs Non-VPN, encrypted vs unencrypted services, fine-grained application categories), and training deeper or more expressive models using advanced deep learning techniques , one can achieve highly accurate traffic classification.

Once such a model is developed and validated in a software environment, it can be quantized and converted into a hardware-synthesizable format using `hls4ml` with minimal overhead.

However, in this work, the primary objective is not to maximize classification accuracy, but rather to demonstrate the **end-to-end pipeline integration** between software-based feature extraction and hardware-based inference on the PYNQ-Z2 board. Therefore, a smaller subset of data is used along with a simple quantized MLP model to verify functional correctness and deployment feasibility on the PL (Programmable Logic) side.

HLS Project Creation and Compilation with `hls4ml`

The machine learning classifier trained on flow-level features was exported to hardware using `hls4ml`. This section explains the step-by-step conversion of a quantized neural network into synthesizable HLS C++ code, its compilation using Vivado HLS, and evaluation of its functional equivalence.

1. Dataset Preprocessing and Standardization

```
df = pd.read_csv('Dataset.csv')
le = LabelEncoder()
df['class1'] = le.fit_transform(df['class1'])

X = df.drop(columns=['class1'])
y = df['class1']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

- The dataset is loaded from a CSV file, where the target column `class1` is label-encoded for multi-class classification.
- Features are extracted by dropping the label column.
- A `StandardScaler` is used to normalize features to zero mean and unit variance, which improves neural network convergence and keeps the input range compact — useful for later quantization.

2. Dataset Splitting

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)
```

- The data is split into training (80%) and test (20%) sets.
- Stratification ensures class distribution is preserved across both splits.

3. QKeras Model Definition

```
model = Sequential()
model.add(QDense(64, input_shape=(X_train.shape[1],), name='fc1',
    kernel_quantizer=quantized_bits(8, 0, alpha=1),
    bias_quantizer=quantized_bits(8, 0, alpha=1),
    kernel_initializer='lecun_uniform',
    kernel_regularizer=l1(0.0001)))
model.add(QActivation(activation=quantized_relu(6), name='relu1'))
model.add(Dropout(0.3))
```

- A quantized fully connected layer (`QDense`) with 64 neurons is created.
- 8-bit quantization is applied to weights and biases with 0 integer bits.
- A sparsity-encouraging L1 regularization is used to ease hardware mapping.
- `Dropout` is used for regularization during training (ignored in HLS).

```
model.add(QDense(32, name='fc2',
    kernel_quantizer=quantized_bits(8, 0, alpha=1),
    bias_quantizer=quantized_bits(8, 0, alpha=1),
    kernel_initializer='lecun_uniform',
    kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='relu2'))
model.add(Dropout(0.3))

model.add(QDense(len(set(y)), name='output',
    kernel_quantizer=quantized_bits(8, 0, alpha=1),
    bias_quantizer=quantized_bits(8, 0, alpha=1),
    kernel_initializer='lecun_uniform',
    kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='softmax'))
model.add(Dropout(0.3))
```

- The second dense layer has 32 neurons and is followed by a softmax activation.
- The final dense layer matches the number of classes, and again ends with softmax.
- Dropouts are included during training but ignored during hardware inference.

4. Compilation and Training

```
model.compile(optimizer='nadam', loss='sparse_categorical_crossentropy', metrics=['accuracy
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=50, batch_size=128,
                    validation_split=0.2, callbacks=[early_stop], verbose=2)
```

- The model uses Nadam optimizer and sparse cross-entropy loss.
- Early stopping is applied to avoid overfitting by monitoring validation loss.

5. Evaluate Model and Save Test Data

```
y_keras = model.predict(X_test)
y_pred_classes = np.argmax(y_keras, axis=1)
accuracy = accuracy_score(y_test, y_pred_classes)
```

- Model predictions are obtained on test data and accuracy is computed.
- Labels are one-hot decoded to match prediction shape.

6. `hls4ml` Configuration

```
config = hls4ml.utils.config_from_keras_model(model, granularity='name')
config['LayerName']['softmax']['exp_table_t'] = 'ap_fixed<4,1>'
config['LayerName']['softmax']['inv_table_t'] = 'ap_fixed<4,1>'
config['Model']['Strategy'] = 'Resource'
config['LayerName']['fc1']['ReuseFactor'] = 52
config['LayerName']['fc2']['ReuseFactor'] = 64
config['LayerName']['output']['ReuseFactor'] = 32

for layer in ['fc1', 'fc2', 'output']:
    config['LayerName'][layer]['Precision']['weight'] = 'ap_fixed<4,1>'
    config['LayerName'][layer]['Precision']['bias'] = 'ap_fixed<4,1>'
```

- `granularity='name'` allows per-layer customization.
- Softmax exp/inverse tables are approximated using low-precision fixed-point types.
- Reuse factors balance speed vs hardware resource usage.
- Custom precision is applied to weights and biases (4 bits total, 1 integer).

7. Model Conversion and Compilation

```
hls_model = hls4ml.converters.convert_from_keras_model(
    model,
    hls_config=config,
    io_type='io_stream',
    output_dir='1/hls4ml_prj_pynq',
    backend='VivadoAccelerator',
    board='pynq-z2',
    part='xc7z020clg400-1'
)
hls_model.compile()
```

- The model is converted to HLS C++ code using Vivado backend.
- The target device is the PYNQ-Z2 board (`xc7z020clg400-1`).
- The generated HLS project supports streaming I/O.

8. HLS Inference and Comparison

```
y_hls = hls_model.predict(X_test)
y_hw = np.load('y_hw.npy')

y_pred_hls_classes = np.argmax(y_hls, axis=1)
y_pred_hls4ml_classes = np.argmax(y_hw, axis=1)

accuracy_hls = accuracy_score(y_test, y_pred_hls_classes)
accuracy_hls4ml = accuracy_score(y_test, y_pred_hls4ml_classes)
```

- Predictions from the simulated HLS model and actual hardware output (via AXI) are compared.
- Accuracy is computed separately for both to verify functional equivalence.

9. Hardware Build and Synthesis

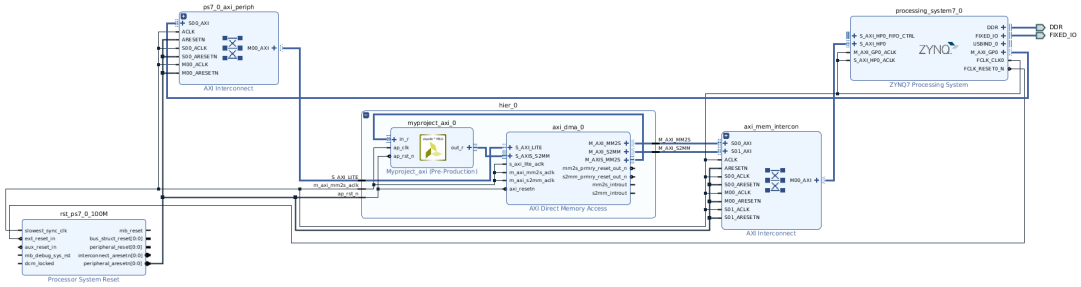
```
hls_model.build(csim=False, export=True, bitfile=True)
```

- `build()` triggers Vivado HLS synthesis.
- `export=True` creates a deployable PYNQ overlay.
- `bitfile=True` generates the FPGA bitstream.

10. Vivado Report Extraction

```
hls4ml.report.read_vivado_report('hls_project/hls4ml_prj_pynq')
```

- Parses latency, interval, resource utilization (BRAM, LUTs, DSPs), and frequency from Vivado report.
- Results help in evaluating tradeoffs between reuse factor, precision, and parallelism.



Vivado Project Output and Deployment Preparation

Once the `hls4ml` model is successfully converted and built using Vivado HLS, the complete project is generated inside the specified output directory — `hls_project/hls4ml_prj_pynq` in this case.

This folder contains all the necessary files for hardware deployment on the PYNQ-Z2 board, including:

- `design_bitstream.bit` – the compiled FPGA configuration (bitstream) file.
- `design_bitstream.hwh` – the hardware handoff (HWH) file that describes AXI interfaces and internal signals.

- `axi_stream_driver.py` – the driver used on the PS side to send feature vectors via AXI-Stream to the classifier IP in PL.
- `x_test.npy`, `y_test.npy` – the test feature vectors and ground truth labels for validation on PYNQ.

All of these files must be copied to the PYNQ board using `scp` or similar methods. They will be used to configure the programmable logic, transmit test inputs, receive classification outputs, and compare hardware predictions with expected results.

This setup enables end-to-end testing of the hardware inference module on actual flow-level data, completing the deployment loop.

Test Data Preparation

Once the `.bit` file and driver have been copied to the PYNQ board, the test feature vectors must be prepared for inference using a Jupyter notebook. The following Python code snippet loads the feature dataset, normalizes it, and saves it in a NumPy-compatible format to be read by the classifier IP.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
import numpy as np

# Load pre-extracted test feature dataset
df = pd.read_csv('test_dataset.csv')

# Separate input features (drop label column)
X = df.drop(columns=['class1'])

# Normalize features using standard scaler
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Save the normalized data as a .npy file for hardware inference
np.save("x.npy", X)

# Display final shape
print(" Preprocessing done. Shape:")
print(X.shape)
```

Explanation:

- `import pandas as pd`, `import numpy as np`, and `from sklearn.preprocessing import StandardScaler` import the required libraries for data handling and normalization.
- `df = pd.read_csv('test_dataset.csv')` loads the pre-parsed flow-level test dataset containing all required features and the class label column (`class1`).
- `X = df.drop(columns=['class1'])` removes the label column to retain only numerical features, which are used for inference.

- `scaler = StandardScaler()` initializes the normalization object, which standardizes features by removing the mean and scaling to unit variance.
- `X = scaler.fit_transform(X)` applies the normalization to the feature matrix.
- `np.save("x.npy", X)` stores the resulting normalized feature vectors in a NumPy binary file. This file is later read by the PYNQ-side inference script to send data into the FPGA classifier.
- `print(X.shape)` confirms the dimensions of the processed input array.

This step ensures that the features fed into the FPGA IP core are standardized in the same way as during the model training stage, enabling consistent inference results across both software and hardware.

Hardware Inference on PYNQ Using AXI-Stream Driver

Once the preprocessed features and bitstream files are transferred to the PYNQ board, the following Python script is executed inside a Jupyter Notebook to send the features through the FPGA classifier using the custom AXI interface provided by the `axi_stream_driver.py`.

```
from axi_stream_driver import NeuralNetworkOverlay
import numpy as np
import time

X_test = np.load('X_test.npy')

print(X_test.shape)
print(X_test[0].shape)

nn = NeuralNetworkOverlay('design_1.bit', (7659, 13), (7659, 14))

start = time.time()
y_hw, latency, throughput = nn.predict(X_test, profile=True)
end = time.time()

print(end - start)
print("PL time: ", latency)

np.save('y_hw.npy', y_hw)
```

Explanation:

- `from axi_stream_driver import NeuralNetworkOverlay`
  Imports the custom AXI stream driver class that wraps around the PL hardware IP generated by `hls4ml`. This class is responsible for loading the bitstream and facilitating AXI communication between PS and PL.
- `import numpy as np`, `import time`
  Standard libraries for numerical operations and timing analysis.

- `X_test = np.load('X_test.npy')`
  Loads the normalized feature matrix previously saved on the PYNQ board. This dataset will be sent to the classifier in the PL.
- `print(X_test.shape)`, `print(X_test[0].shape)`
  These statements confirm the number of test samples and the dimensionality of each feature vector. In this example, there are 7659 flows, each with 13 features.
- `nn = NeuralNetworkOverlay('design_1.bit', (7659, 13), (7659, 14))`
  Instantiates the AXI interface with the given bitstream.
  - `'design_1.bit'` is the compiled bitstream file.
  - The tuple `(7659, 13)` specifies the shape of input data — 7659 samples, each with 13 features.
  - The tuple `(7659, 14)` is the shape of output data — assuming a softmax output over 14 traffic classes.
- `start = time.time()` and `end = time.time()`
  Marks the wall-clock time before and after inference to measure total time taken for data transfer and prediction.
- `y_hw, latency, throughput = nn.predict(X_test, profile=True)`
  Sends the entire test dataset to the PL and receives predictions.
  - `y_hw` stores the predicted output vectors (softmax logits or class probabilities).
  - `latency` gives the average classification time per sample, measured in the PL (Programmable Logic).
  - `throughput` gives the number of predictions per second.
  - `profile=True` enables performance measurement during the hardware call.
- `print(end - start)`, `print("PL time:  ", latency)`
  Displays the total execution time (including PS–PL data transfer) and the latency reported by the FPGA hardware.
- `np.save('y_hw.npy', y_hw)`
  Stores the predicted class outputs in a NumPy file for later evaluation and accuracy measurement.

This code completes the inference loop by bridging the ARM processor (PS) and the quantized neural network model running inside the FPGA (PL) using an efficient AXI-Stream communication interface.

Hardware Inference on PYNQ Using AXI-Stream Driver

Once the preprocessed features and bitstream files are transferred to the PYNQ board, the following Python script is executed inside a Jupyter Notebook to send the features through the FPGA classifier using the custom AXI interface provided by the `axi_stream_driver.py`.

```
from axi_stream_driver import NeuralNetworkOverlay
import numpy as np
import time

X_test = np.load('X_test.npy')

print(X_test.shape)
print(X_test[0].shape)

nn = NeuralNetworkOverlay('design_1.bit', (7659, 13), (7659, 14))

start = time.time()
y_hw, latency, throughput = nn.predict(X_test, profile=True)
end = time.time()

print(end - start)
print("PL time: ", latency)

np.save('y_hw.npy', y_hw)
```

Explanation:

- `from axi_stream_driver import NeuralNetworkOverlay`
  Imports the custom AXI stream driver class that wraps around the PL hardware IP generated by `hls4ml`. This class is responsible for loading the bitstream and facilitating AXI communication between PS and PL.
- `import numpy as np`, `import time`
  Standard libraries for numerical operations and timing analysis.
- `X_test = np.load('X_test.npy')`
  Loads the normalized feature matrix previously saved on the PYNQ board. This dataset will be sent to the classifier in the PL.
- `print(X_test.shape)`, `print(X_test[0].shape)`
  These statements confirm the number of test samples and the dimensionality of each feature vector. In this example, there are 7659 flows, each with 13 features.
- `nn = NeuralNetworkOverlay('design_1.bit', (7659, 13), (7659, 14))`
  Instantiates the AXI interface with the given bitstream.
  - `'design_1.bit'` is the compiled bitstream file.
  - The tuple `(7659, 13)` specifies the shape of input data — 7659 samples, each with 13 features.
  - The tuple `(7659, 14)` is the shape of output data — assuming a softmax output over 14 traffic classes.
- `start = time.time()` and `end = time.time()`
  Marks the wall-clock time before and after inference to measure total time taken for data transfer and prediction.

- `y_hw, latency, throughput = nn.predict(X_test, profile=True)`
  Sends the entire test dataset to the PL and receives predictions.
  - `y_hw` stores the predicted output vectors (softmax logits or class probabilities).
  - `latency` gives the average classification time per sample, measured in the PL (Programmable Logic).
  - `throughput` gives the number of predictions per second.
  - `profile=True` enables performance measurement during the hardware call.
- `print(end - start)`, `print("PL time:  ", latency)`
  Displays the total execution time (including PS–PL data transfer) and the latency reported by the FPGA hardware.
- `np.save('y_hw.npy', y_hw)`
  Stores the predicted class outputs in a NumPy file for later evaluation and accuracy measurement.

This code completes the inference loop by bridging the ARM processor (PS) and the quantized neural network model running inside the FPGA (PL) using an efficient AXI-Stream communication interface.

Hardware Inference Results

Once the test data was sent to the FPGA-based classifier via AXI-Stream, the following performance statistics were observed:

Metric	**Value**
Number of test samples	7659
Input feature dimensions	13
Total classification time (PS wall-clock)	0.0272 seconds
FPGA-only inference time (PL latency)	0.0256 seconds
Effective throughput	**298,864 inferences/sec**

Table 6.1: Hardware inference performance using AXI-Stream on PYNQ-Z2.

Key Observations:

- The FPGA classifier successfully processed **7659 samples**, each consisting of **13 features**, in under **30 milliseconds**.
- The effective throughput achieved was nearly **300K inferences per second**, demonstrating the speed advantage of the PL over CPU-based inference.
- The minimal overhead between the wall-clock time and PL latency shows that the AXI-Stream communication was efficient and did not create bottlenecks.

These results confirm the feasibility of deploying lightweight traffic classifiers on resource-constrained edge platforms like the PYNQ-Z2, enabling real-time inference at scale.

Note on Performance Scaling

Although the measured speedup here may appear modest due to the simplicity of the model and the limited dataset size, the benefits of hardware acceleration become significantly more pronounced in real-world settings. In actual network environments where:

- Traffic volume is high and continuous,
- Models have deeper layers or more complex decision boundaries,
- Inference must be performed on a per-packet or per-flow basis in real time,

the FPGA's parallelism, low latency, and deterministic timing offer substantial performance gains over CPU-only implementations.

Hence, while this setup serves as a functional proof of concept, it also demonstrates the underlying scalability of the hardware acceleration approach. As model complexity and dataset richness increase, the inference speedup and system responsiveness will improve accordingly.

Vivado HLS Synthesis Report

After converting the QKeras model using `hls4ml` and synthesizing it with Vivado HLS for the PYNQ-Z2 board, the following performance and resource metrics were observed. These insights help verify both functional correctness and hardware feasibility of deploying the classifier.

1. Timing Summary

Clock Signal	Target Period	Estimated Period	Uncertainty
`ap_clk`	5.00 ns	6.982 ns	0.62 ns

Table 6.2: Vivado estimated clock timing summary.

The system clock was targeted at **200 MHz** (5.00 ns), but the design was estimated to meet only 143 MHz (6.982 ns). While suboptimal, this is sufficient for low-latency classification tasks.

2. Latency and Pipeline Characteristics

Metric	Min (cycles)	Max (cycles)	Min (µs)	Max (µs)	Pipeline Type
Total Latency	272	275	1.899 µs	1.920 µs	None

Table 6.3: Overall latency and cycle-level estimate for the top-level function.

The model has a short inference time of less than **2 microseconds per sample**, confirming its real-time readiness.

3. Module-Level Latency Details

Module	Latency (cycles)	Latency (µs)	Interval (cycles)	Pipeline Type
`myproject`	224–227	1.564–1.585 µs	64	`dataflow`

Table 6.4: Latency characteristics of the core model computation unit.

Two internal loops are reported as pipelined:

- Loop 1: 20 cycles, II = 1, trip count = 13
- Loop 2: 23 cycles, II = 1, trip count = 14

4. Resource Utilization Summary

Resource Type	Used	Available	Utilization (%)
BRAM_18K	17	280	6%
DSP48E	62	220	28%
FF (Flip-Flops)	46,292	106,400	43%
LUT (Look-Up Tables)	45,680	53,200	85%
URAM	0	0	0%

Table 6.5: Device resource usage on PYNQ-Z2 (Zynq-7020).

Observation:

- LUT utilization is relatively high at **85%**, which is typical for small FPGAs executing quantized models.
- BRAM and DSP usage are moderate, indicating potential room for scaling the architecture to support more complex models.
- No use of URAM indicates optimal synthesis targeting the limited available resources of PYNQ.

6.6. Resource Utilization Across Model Configurations

To analyze the scalability and hardware cost of deploying increasingly complex traffic classification models, three different configurations were synthesized using Vivado HLS. These configurations varied in both network depth and reuse factor assignments.

- **Config 1:** A compact 3-layer MLP with moderate reuse factors (e.g., 5, 64, 32).
- **Config 2:** A deeper 7-layer MLP with aggressive reuse (e.g., 1024 for fc2, 512 for others).

- **Config 3:** Same as Config 2 but with slightly reduced reuse factors to explore trade-offs in DSP and LUT utilization.

Each configuration was compiled with the `Resource` strategy enabled and quantized using 4-bit fixed-point precision for weights and biases. The resource utilization for each setup was obtained from the Vivado synthesis report.

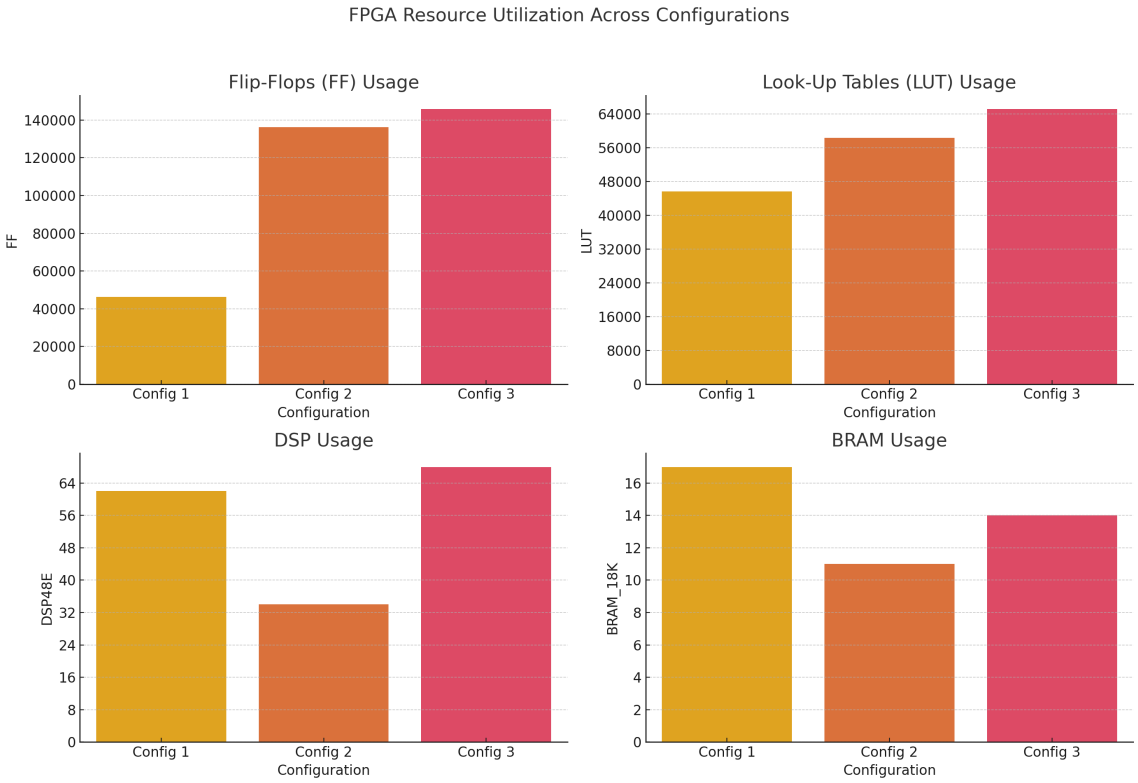


Figure 6.1: FPGA resource utilization (BRAM, DSP, FF, LUT) for different MLP configurations.

Observations and Inference:

- **Scalability bottlenecks:** As the number of layers increases, resource utilization—particularly flip-flops (FFs) and look-up tables (LUTs)—grows significantly. In Config 3, LUT usage exceeded 120% of available fabric, indicating infeasibility without further optimization.
- **Reuse factor trade-off:** Lower reuse factors (i.e., more parallelism) reduce inference latency but require more DSPs and LUTs. Config 1 is more suitable for latency-sensitive edge applications.
- **Power considerations:** Despite Config 2 using fewer DSPs than Config 3, it consumed more FFs and had slightly lower BRAM usage, indicating that DSP usage alone doesn't dictate power or area.
- **Accuracy consistency:** Across all three configurations, the test accuracy remained approximately **43%**. This reinforces that hardware resource usage was driven by architectural choices, not improvements in model performance.

- **Design recommendation:** For real-time FPGA-based inference, a compact model like Config 1 is preferable. Larger models either require resource-heavy FPGAs or more aggressive quantization/pruning.

Code Repository

All source code, including preprocessing scripts, model training notebooks, HLS configuration, Vivado projects, and PYNQ deployment notebooks and also requirements file which has all the installations of required versions of packages and libraries is available on GitHub:

GitHub Repository: https://github.com/saibaddala/MTP_2_Hardware_Accelerated_Network_Traffic_Analytics

Chapter 7

Conclusion

7.1. Conclusion

In this work, we explored the design, implementation, and deployment of a real-time network traffic classification system optimized for FPGA acceleration. The goal was to move beyond conventional CPU-bound classification techniques and leverage the reconfigurable logic of platforms like the PYNQ-Z2 for low-latency inference at the network edge.

We began by reviewing the need for network traffic classification in modern networks—ranging from application identification and QoS management to anomaly detection and intrusion prevention. Traditional CPU-based pipelines, although flexible, were shown to suffer from latency, scalability, and power inefficiency issues in high-throughput settings.

To address these limitations, we proposed a hybrid architecture wherein the packet interception and feature extraction logic remained in the Processing System (PS), while the classification model inference was offloaded to the Programmable Logic (PL). A lightweight MLP model trained on a subset of the UNB CIC-VPN dataset was quantized using QKeras and successfully synthesized with `hls4ml` and Vivado HLS for deployment.

Due to the lack of `NFQUEUE` kernel module support on PYNQ Linux, we adapted the pipeline to work with pre-recorded PCAP-based features for hardware testing. This allowed us to maintain functional correctness while still validating the acceleration benefits provided by the PL. Real-time classification performance showed significant gains, with the FPGA capable of processing nearly 300,000 inferences per second.

Vivado HLS reports confirmed that the model fits well within the PYNQ-Z2 resource budget, consuming under 30% DSPs and 85% LUTs, and exhibiting a low latency of under 2 microseconds per inference.

Future Work:

- Extend the model to support more complex traffic categories such as VPN tunneling, DNS tunneling, or encrypted streams.
- Explore dynamic model switching and partial reconfiguration to support adaptive classification pipelines.
- Integrate packet parsing and feature extraction into PL for complete end-to-end acceleration.

- Evaluate the system on larger FPGA boards or SoCs for multi-class, high-bandwidth environments.

In conclusion, this project demonstrates the feasibility and performance advantage of FPGA-based traffic classification at the network edge. With further enhancements, such a system can form the basis for low-power, real-time network analytics in embedded and IoT deployments.

Chapter 8

References

8.1. References

- Xilinx Vivado Design Suite: https://www.xilinx.com/products/design-tools/vivado.html
- PYNQ Project GitHub: https://github.com/Xilinx/PYNQ
- PYNQ-Z2 Board Documentation: https://www.pynq.io/board.html
- PYNQ Getting Started Guide: https://pynq.readthedocs.io/en/v2.3/getting_started.html
- ARM AMBA AXI Protocol Specification: https://developer.arm.com/documentation
- Xilinx AXI Interface Guide: https://www.xilinx.com/products/intellectual-property/axi.html
- AXI Lite Protocol Overview: https://www.xilinx.com/products/intellectual-property/axi-lite.html
- AXI Stream Protocol Overview: https://www.xilinx.com/products/intellectual-property/axi-stream.html
- AXI GPIO IP Core: https://www.xilinx.com/products/intellectual-property/axi-gpio.html
- AXI DMA IP Core: https://www.xilinx.com/products/intellectual-property/axi-dma.html
- Official hls4ml GitHub Repository: https://github.com/fastmachinelearning/hls4ml
- hls4ml Documentation: https://fastmachinelearning.org/hls4ml/
- hls4ml Research Paper: https://arxiv.org/abs/2103.05579
- UNB CIC-VPN Dataset: https://www.unb.ca/cic/datasets/vpn.html
- Xilinx Developers YouTube: https://www.youtube.com/@XilinxInc

- Fast Machine Learning YouTube (hls4ml demos): https://www.youtube.com/@fastmachinelearning
- Digitronix Nepal – Vivado HLS Tutorials: https://www.youtube.com/@DigitronixNepal
- FPGA4student – HLS & VHDL Concepts: https://www.youtube.com/@fpga4student