pyrtlnet: An End-to-End Python Workflow for Quantized Inference Hardware Development

Jeremy Lau
University of California, Santa Barbara
lauj@ucsb.edu

Timothy Sherwood
University of California, Santa Barbara
sherwood@cs.ucsb.edu

Abstract

As quantized neural network inference moves from cloud to edge, the demand for custom hardware accelerators tailored to specific models, precisions, and deployment constraints continues to grow. While frameworks for training and deploying quantized models in software are mature, the path from a trained model to working custom hardware remains steep and requires expertise spanning machine learning frameworks to hardware description languages, synthesis toolchains, and FPGA deployment flows. To ease innovation across these different layers of abstraction, we present pyrtlnet, an open-source Python-based framework that provides an end-to-end workflow from quantization-aware training in TensorFlow to custom inference hardware running on an FPGA, all without ever leaving Python.

The core idea behind pyrtlnet is to provide a sequence of matched workflows that progressively move from software to hardware: a TensorFlow quantization-aware training flow produces 8-bit integer weights and 32-bit integer biases; a LiteRT-based inference flow validates the quantized model in software; a NumPy reimplementation of that same inference exposes the quantization math in detail; a PyRTL hardware description reimplements the NumPy version as synthesizable digital logic; and finally a PYNQ-based deployment flow targets the design to an FPGA with AXI data transport. Because each workflow reimplements the same computation at a different level of abstraction, researchers can verify correctness at every step and build intuition for how software-level decisions (such as quantization bit-widths or layer structure) manifest in hardware cost and behavior. The generated hardware is easy to inspect, modify, and extend, so researchers can experiment with new layer types, quantization schemes, or dataflow organizations without navigating the complexity of industrial HDL toolchains. The result is a system that enables researchers to move from a trained model to synthesized hardware in an afternoon, opening the door to rapid co-exploration of model architecture and hardware design.

1 Introduction

Pushing the boundaries of efficient machine learning inference increasingly requires reasoning across layers of abstraction simultaneously. A researcher exploring extreme quantization, novel fault strategies, or new dataflow organizations needs to understand how choices made during training propagate through fixed-point arithmetic, into hardware structures, and ultimately onto silicon. This kind of cross-layer exploration demands a diverse set of expertise spanning ML frameworks, quantization math, hardware description languages, synthesis toolchains, and FPGA deployment flows. Yet the tools available at each layer are largely disconnected, making it difficult to trace a decision made in one domain through to its consequences in another, even for researchers with the background to do so.

Submitted to 7th Fast Machine Learning for Science Conference (FastML 2026). Do not distribute.

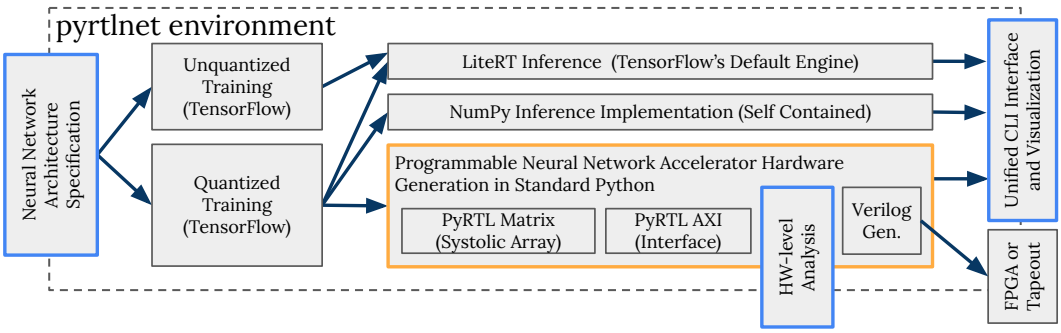


Figure 1: Overview of the pyrtlnet framework. A neural network architecture specification flows through matched workflows at increasing levels of hardware detail (from TensorFlow training through NumPy and PyRTL inference implementations to FPGA deployment) with a unified CLI providing consistent visualization at every stage.

We present pyrtlnet [1], an open-source Python-based framework that provides an end-to-end workflow from quantization-aware training in TensorFlow [3] to custom inference hardware running on an FPGA, all without ever leaving Python. The core idea is a sequence of matched implementations that progressively move from software to hardware: a TensorFlow quantization-aware training flow produces 8-bit integer weights and 32-bit integer biases; a LiteRT-based inference flow validates the quantized model; a NumPy re-implementation exposes the quantization math in detail; a PyRTL [6] hardware description re-implements the same computation as synthesizable digital logic; and a PYNQ [4] deployment flow targets the design to an FPGA with AXI data transport. Because each workflow re-implements the same computation at a different level of abstraction, researchers can verify correctness at every step and build intuition for how software-level decisions (quantization bit-widths, layer structure, fixed-point rounding, etc.) manifest in hardware cost and behavior. The generated hardware is defined in PyRTL, making it easy to inspect, modify, and extend, so researchers can experiment with new layer types, quantization schemes, or dataflow organizations without navigating the complexity of industrial HDL toolchains. The pyrtlnet framework is open source, well documented and tested, with automation provided by GitHub actions.

In this work we make the following contributions: we provide a complete, open-source, end-to-end workflow from quantization-aware training to FPGA deployment, unified in a single language (Python); we introduce a new output-stationary systolic array implementation in PyRTL along with a composable matrix library for expressing chained computations; we demonstrate matched inference outputs across all four implementation levels (LiteRT, NumPy, PyRTL simulation, and FPGA), validating correctness at every step; and we show that the full pipeline (from trained model to synthesized hardware) can be completed in an afternoon, opening the door to rapid co-exploration of model architecture and hardware design.

2 Framework Design

Figure 1 shows the overall structure of pyrtlnet. The framework is organized as a sequence of matched implementations, each re-implementing the same computation at a progressively lower level of abstraction. The entry point is a standard TensorFlow quantization-aware training flow [2] that produces a two-layer dense image recognition model on the MNIST dataset with 8-bit integer weights and 32-bit integer biases [8], training to 94% accuracy. From there, TensorFlow's LiteRT inference system serves as the reference implementation, producing intermediate tensor outputs at each layer that anchor the correctness of every subsequent re-implementation.

A key design decision is that all workflows share a unified command-line interface, shown in Figure 2. For each test image, the interface displays an ASCII-art rendering of the input (so the user can immediately see what the model is looking at), the intermediate tensor outputs from each layer (useful for tracing numerical differences between implementations), and a bar chart of the final classification logits. The model outputs a vector of 10 unnormalized probabilities, one per digit class; the bar chart reformats these so the prediction is visually obvious, with correct predictions shown in green and incorrect ones in red. Figure 2 (left) shows a correct prediction where the model identifies a

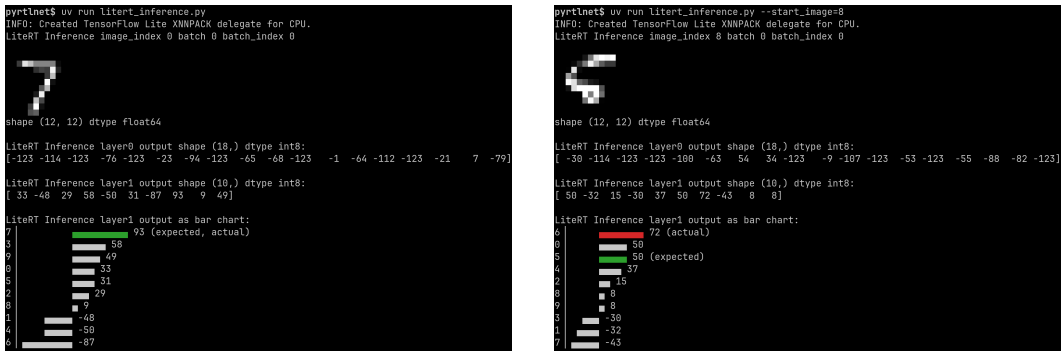
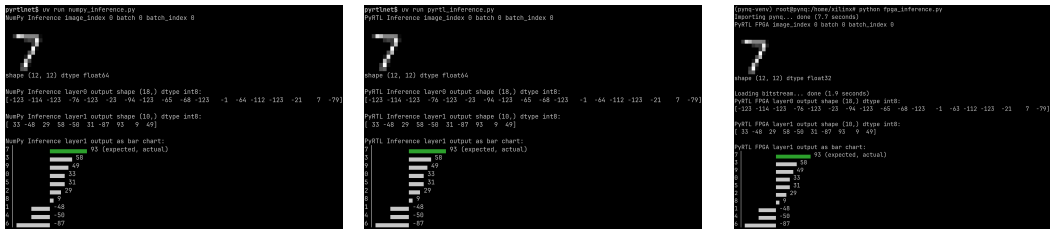


Figure 2: pyrtlnet's unified CLI interface for LiteRT inference. Left: a correct prediction where the model identifies a hand-drawn 7. Right: a misprediction where the model predicts 6 for a digit labeled 5; the ASCII-art display shows the prediction is still reasonable. The same interface is replicated across all inference workflows (NumPy, PyRTL, FPGA), enabling direct comparison.



(a) NumPy (b) PyRTL (c) FPGA

Figure 3: The same unified interface across three levels of implementation: (a) a pure-Python NumPy re-implementation of quantized inference, (b) a PyRTL hardware simulation, and (c) running on a Pynq Z2 FPGA. Identical inputs produce matching outputs at every level, letting researchers verify correctness as the design moves from software to silicon.

hand-drawn 7, while Figure 2 (right) shows a misprediction where the model predicts 6 for a digit labeled 5 (the ASCII-art display reveals this is a reasonable mistake). This same interface is replicated identically across the LiteRT, NumPy, PyRTL, and FPGA workflows (Figure 3), so researchers can run the same input through every level and directly compare outputs. This consistency makes it straightforward to trace a discrepancy (for example, an off-by-one difference in a ReLU output) back to the specific level of abstraction where it was introduced.

The first cross-layer step is a pure-Python NumPy [7] re-implementation of the LiteRT inference, with fixed-point arithmetic handled by the `fxpmath` library [5]. This implementation was built through careful study of TensorFlow's quantization specification [8] and makes every step of the quantized arithmetic (scale factors, zero-point adjustments, fixed-point accumulation, and requantization) explicit and inspectable in Python. The NumPy implementation produces nearly identical outputs to the LiteRT reference, with occasional off-by-one differences due to rounding behavior, providing a verified software baseline from which to build hardware.

PyRTL [6] is an open-source Python-based hardware description language designed around simplicity, usability, and extensibility rather than industrial optimization. Hardware designs are specified, simulated, traced, and tested entirely within Python, and because PyRTL uses elaboration-through-execution, designers can leverage the full power of Python (including introspection, object-oriented reuse, and programmatic generation) to construct and analyze hardware. A small, well-defined core intermediate representation supports custom transforms for debugging, analysis, and instrumentation, and completed designs can be exported to Verilog for synthesis or simulated directly in Python. This combination of accessibility and expressiveness has made PyRTL an effective platform for fast design iteration across domains including cryptography and machine learning [6], and it is this same combination that makes pyrtlnet possible: the entire hardware inference pipeline can be built, inspected, and modified without leaving the Python environment in which the model was trained.

The PyRTL hardware implementation re-implements the NumPy inference as synthesizable digital logic. At the core of this implementation is a new output-stationary systolic array that performs the quantized matrix multiplications. The systolic array is part of a composable PyRTL matrix library designed to express chained computations naturally (for example, the sequence $ax + b$ where the matrix product must complete before the bias addition can begin). This composability matters because quantized inference involves not just the multiply-accumulate but also requantization, clamping, and activation functions, and expressing these as a pipeline of composable matrix operations keeps the hardware description close to the mathematical specification. The resulting design can be simulated directly in Python with PyRTL's built-in simulators or exported to Verilog, and as Figure 3 shows, it produces outputs that match the software implementations exactly. While specifying hardware in Python raises natural questions about simulation performance, PyRTL provides multiple simulation backends that address this directly: `Simulation` is the default interpreted simulator for debugging, `FastSimulation` ahead-of-time compiles the circuit netlist into a Python function to eliminate per-cycle interpretation overhead, and `CompiledSimulation` generates and compiles C code into a shared library for simulation performance competitive with (and often outperforming) established tools such as Icarus Verilog and ModelSim. Switching between backends is a one-line change, so researchers can use the interpreted simulator for interactive debugging and move to compiled simulation for longer runs with a very simple change. The resulting design can also be exported to Verilog for use with tools like Verilator.

The final workflow takes the Verilog exported from the PyRTL design and synthesizes it for a Pynq Z2 FPGA with Vivado. This step extends the inference hardware with AXI transport (AXI-Stream for input image data flowing to the programmable logic, and AXI-Lite for output logits flowing back) so the complete system runs inference on real hardware with data managed from the ARM processing system. Figure 3(c) shows the same unified interface running on the FPGA, confirming that the full pipeline from training through synthesis preserves the inference behavior established at every prior level.

3 Results and Future Directions

The pyrtlnet framework successfully preserves inference accuracy across all four implementation levels: the quantized model achieves 94% accuracy on the MNIST test set, and this accuracy is maintained through the LiteRT reference, the NumPy re-implementation, the PyRTL hardware simulation, and the synthesized FPGA design. LiteRT runs inference on one image in 7 µs, NumPy in $3\,\mathrm{ms}$, and PyRTL in $67\,\mathrm{ms}$. The synthesized design targets a Pynq Z2 FPGA (Xilinx Zynq XC7Z020) and uses 3830 LUTs, 6 BRAM tiles, 120 DSPs, and runs at $25\,\mathrm{MHz}$, achieving an inference latency of $18\,\mu\mathrm{s}$ per image (449 cycles). The full pipeline from a trained TensorFlow model to a working FPGA demonstration can be completed in an afternoon by a researcher with no prior hardware design experience.

Because pyrtlnet's hardware is defined in PyRTL, the entire design is available to PyRTL's programmatic analysis and transformation infrastructure, and this opens several directions for future work. One natural extension is hardware instrumentation: PyRTL's intermediate representation supports custom transform passes that can inject cycle-accurate monitors, performance counters, or assertion checkers directly into the design, enabling researchers to observe the internal behavior of the inference hardware at a level of detail that is difficult to achieve with closed-source synthesis tools. Another is fault modeling for safety-critical applications. In domains such as high-energy physics and space-based computing, understanding how hardware faults (single-event upsets, stuck-at faults, or soft errors from radiation) propagate through an inference pipeline is essential. PyRTL's ability to programmatically modify circuit structure makes it possible to inject precise fault models at specific locations in the design and measure their effect on classification accuracy, providing a direct path from fault specification to measured resilience. Similarly, information flow analysis through the hardware (tracking how specific input bits influence outputs, or verifying that sensitive data does not leak through timing or structural side channels) becomes tractable when the entire design is represented as a transformable Python object. More broadly, we plan to extend pyrtlnet to support convolutional layers, deeper network architectures, and alternative quantization schemes (including sub-8-bit and mixed-precision), expanding the space of models and hardware trade-offs that researchers can explore end-to-end.

References

[1] pyrtlnet: A quantized neural network inference research framework. https://github.com/UCSBarchlab/pyrtlnet/.

[2] Quantization aware training in Keras. https://www.tensorflow.org/model_optimization/guide/quantization/training_example.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[4] Advanced Micro Devices, Inc. PYNQ. https://github.com/Xilinx/PYNQ.

[5] Franco Alcaraz. A Python library for fractional fixed-point (base 2) arithmetic and binary manipulation with NumPy compatibility. https://github.com/francof2a/fxpmath.

[6] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A Pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[8] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. http://arxiv.org/abs/1712.05877.