Spiking Neural Networks on FPGAs

CSE462M Spring 2025
Washington University in St. Louis
Ethan Morton, John Walters, Edgar Sarceno

Abstract

Spiking Neural Networks (SNNs) offer a promising path for energy-efficient, event-driven computation inspired by biological systems. This project demonstrates a hardware-accelerated implementation of a Leaky Integrate-and-Fire (LIF) SNN on the PYNQ-Z2 FPGA platform. Our system combines Verilog-based fixed-point LIF neurons with an AXI4-Lite register interface, enabling precise control and low-latency data exchange with the Processing System (PS). A FastAPI server running on the PS exposes a REST interface for configuration and readout, while a lightweight Python client logs real-time neuron activity over HTTP. We implemented a three-neuron untrained network, configured with synthetic input pulses to validate core functionality and inter-neuron connectivity. Results show clean spike-leak behavior, sub-threshold membrane dynamics, and sparse coincidence detection, all consistent with theoretical predictions. Additionally, resource analysis revealed a significant reduction in LUT and FF usage by transitioning from Q16.16 to Q2.6 fixed-point arithmetic. This work validates the feasibility of low-resource SNN hardware and provides a scalable foundation for future neuromorphic experiments on FPGA platforms.

Contents

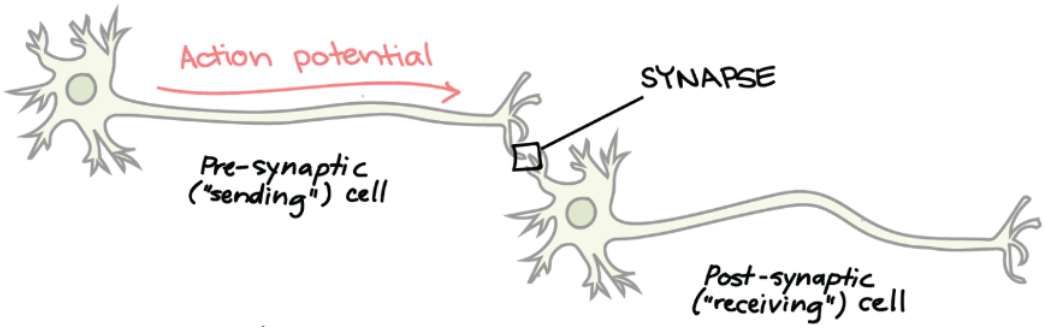


Figure 1: A simple image depicting the connection of two biological neurons. Obtained from [Goodman, 2023]

1 Introduction

In recent years, the fields of machine learning and "AI" have grown considerably. What started as simple Multi-Layer Perceptrons (MLPs) are now complicated network architectures that can capture hidden semantic meanings and perform operations with them. As the field has matured, many different methods for applying machine learning concepts have arisen. Among them is the use of biologically-centered neuron and network models which attempt to capture the magic of the human brain. These models are called Spiking Neural Networks (SNNs).

1.1 Spiking Neurons

SNNs are built off a slightly different fundamental building block than the neural networks you may hear about in the news. Other than recurrent neural networks which leverage self-connected neurons to maintain state during inference, most classical neural networks *do not* have state. Each "neuron" is just a placeholder for the numbers to flow through. The information resides in the weights obtained by intensive training. These weights are then used as a "black box" that you feed inputs into and then receive an output. Disregarding some mathematical subtelties about probability, each input maps to a specific range of outputs.

However, in SNNs and spiking neuron models, that is not the case. Each individual neuron *does* have a physical state. That state is external to the current computation, and is updated based on time. This makes SNNs more like a real-time simulations than a black-box computation. Each neuron in an SNN is intended to emulate or simulate a *real* neuron, just like the ones in the human brain. There are many models for biological neurons that can be used, but they all have a few basic traits in common.

The main feature is the neuron's *electrical potential*. This is the measure of how much charge has accumulated inside a neuron. In real human brains, it has been found that neurons communicate by building up charge from their inputs, and then once a specific charge threshold is reached, releasing that charge in one fell swoop to its outputs. That near-instantaneous release of charge is called a "spike," hence the name "spiking neurons." At it's base level, each neuron is essentially just an accumulator. It accumulates charge, until it hits a specific biological threshold, and then pushes the charge out in a burst. In the human brain, millions and billions of spikes will happen just by reading this paper.

Biological neurons are connected through two main structures: the *dendrites* and the

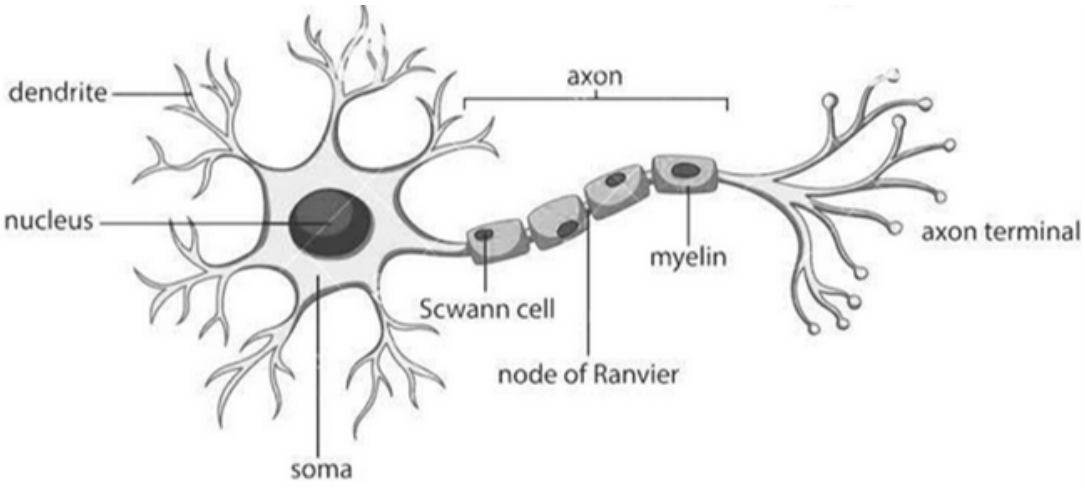


Figure 2: A graphic showing a more detailed physical neuron structure. Obtained from [Goodman, 2023]

axons. The *dendrites* are the mess of spiky prongs sticking out of the nucleus of the neuron, as seen in fig. 1. These dendrites are the input channel that accumulates from any connected neuron. The *axon* is the long, skinny part of each neuron. The axon is the output channel that transmits the stored charge to other neurons. In fig. 1, we see a simple model of how neurons connect to each other, axon to dendrite. A neuron may have more than one incoming connection, and more than one outgoing connection, though the specifics of how that works is much more complex. A slightly more detailed view of a neuron can be seen in fig. 2, which includes a more detailed biological description of the cell type.

The exact mechanics and theoretical variations of how that charge is released and how it is gathered is beyond the scope of this project, but suffice it to say, there are plenty, and they are as complicated as they are plentiful. A simple model to use as a basis for this project is called the Leaky Integrate-and-Fire (LIF) model. This model is just as simple as it sounds. This model has two main update rules: 1) the potential decays over time (this is the leaky" part) 2) if the potential crosses a specified threshold it fires ("spikes") The neuron is modeled intuitively with a simple differential equation, shown below in eq. (1).

$$\tau \frac{dv}{dt} = -v \tag{1}$$

Solving this, we arrive at a simple equation of exponential decay over time, shown in eq. (2). Due to the "spikey" nature of how the threshold works, there is not a clean mathematical way to put that in the equation, so it is done by just checking on each update.

$$v(t) = v_o e^{-\frac{t}{\tau}} \tag{2}$$

The setting of v_o is arbitrary, and more meaningfully represents the previous timestep. When neurons are simulated in a computer, there must be discrete timesteps taken, and as a result, the decay factor becomes a simple multiple of the previous potential. Our equation becomes a discrete time equation, and looks something more like eq. (3), where DF is the decay factor found by plugging a value for τ into the exponential of eq. (2).

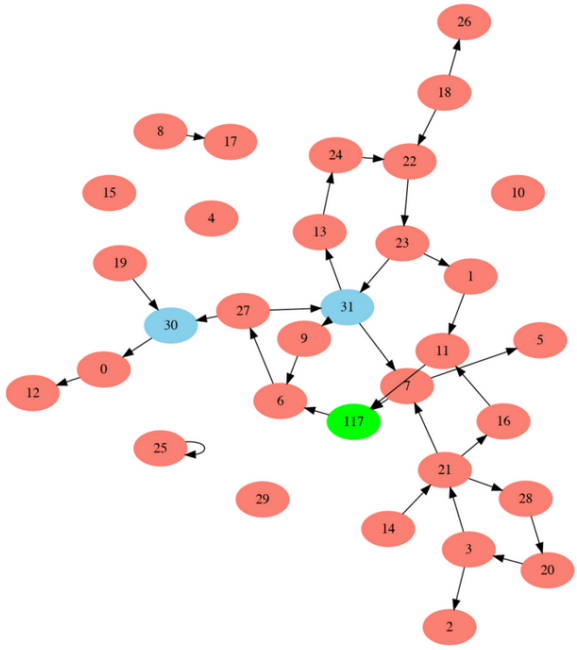


Figure 3: An example of a spiking neural network that was produced by using evolutionary algorithms through many iterations. Obtained from [Goodman, 2023]

$$v(n) = DF \cdot v(n-1) \tag{3}$$

1.2 Spiking Neural Networks

Just as classical perceptrons can be orchestrated into complex networks with many interconnections, so can these biologically-based spiking neurons. Due to their relative recency compared to classical neural networks, there are not many agreed upon architectures for use such as transformer models, diffusion models, etc. As a result, many networks are arbitrarily constructed, or evolutionary machine learning techniques are employed. In either case, the network often ends up looking somthing like fig. 3.

Compared to modern neural networks such as transformers, it seems appallingly simple, and it is difficult to tell where the information flows. This is another result of spiking neurons being more similar to a simulation than to a black-box computation. Instead of simply receiving an output that we then interpret, we must perform real-time analysis of the state of the network to determine what the network is understanding and processing.

There are many different methods for understanding how a network reacts to outside stimuli. A simple way is to simply look at the rate of spiking. Similar to how if the number produced for a neuron in a classical network is large then we call it "activated", if a spiking neuron is spiking very often, then it is "active." Though the process of training is outside the scope of the report, training is when specific neurons would be selected to act as "decision" neurons, meaning they are the output. This is easiest to see by looking at a tutorial posted in the snnTorch documentation [Eshraghian, 2021] that uses the standard MNIST handwritten digits and attempts to classify which digit is written.

There are as many input neurons as pixels in the image. Each input neuron is assigned a rate at which to spike based on the darkness of the pixel. In real scenarios, the inputs

will often be handled this way to act as a signal driver for the network, just as the small bones in human ears act as stimuli for the brain. The decision neurons are probed to determine which digit is detected by the network. +

2 Methods

Though there are many models and analysis methods available to try and understand and implement a simple SNN, we chose to keep it simple. Throughout this project, we used the LIF model for neurons, as it was the simplest to use and still produce interesting results. We also chose to encode "information" in the spiking rate of the neurons, in order to keep things easily interpretable.

2.1 Tooling

We were provided hardware and software for this project. The hardware was a PYNQ-Z2 FPGA development board made by Xilinx. This is an effective tool, with both a built-in ARM processor representing the Processing System (PS), and a small FPGA to implement custom Programmable Logic (PL) on. There is also built-in AXI compatibility with the FPGA, which can easily be called from languages such as Python. The software used was Xilinx Vivado, for custom HDL development. The reason for using Vivado rather than an HLS tool such as Vitis was the exploratory nature of our project. A large portion of this project consisted of research into how spiking neurons and SNNs worked, and as a result, we wanted to be able to have control over every aspect of the logic, rather than let it be abstracted away.

2.2 Overall Design

Our spiking neural network system is composed of three tightly integrated subsystems: the host PC, the Processing System (PS) on the PYNQ-Z2 board, and the programmable logic (PL) fabric. The host PC runs a lightweight client—implemented in MATLAB or Python—which issues HTTP requests for network control (e.g. start/stop, spike-rate adjustments) and visualizes membrane potentials in real time. The PS hosts a FastAPI server that exposes RESTful endpoints corresponding to control registers in the PL, and uses PYNQ's `Overlay` API along with a memory-mapped I/O interface (`MMIO`) to translate HTTP calls into AXI4-Lite transactions. The PL implements the two-neuron leaky integrate-and-fire network in Verilog, exporting all control inputs and neuron state through an AXI4-Lite slave peripheral.

When the host client issues a command—such as calling `/enable` to toggle updates, `/n1_rate` or `/n2_rate` to set synthetic input divisors, or `/read` to sample membrane potentials—the PS performs the following sequence:

looks up the base address from

```
base_addr = int(overlay.ip_dict["axi_lite_slave_0"]["parameters"]["C_BASEADDR"], 0)
```

and instantiates

```
mmio = MMIO(base_addr, 0x1000)
```

to execute

```
mmio.write(offset, value)  % control writes
pot = mmio.read(0x0C)      % read neuron potential
```

All register accesses complete within a few FPGA clock cycles, yielding sub-microsecond latency suitable for real-time closed-loop experiments.

Inside the PL, the neuron update logic applies exponential decay and threshold comparison on every update tick. Control registers are mapped at offsets `0x00` (`update_enable`), `0x04` (`n1_stop_val`), and `0x08` (`n2_stop_val`); the current membrane potential (an 8-bit Q2.6 fixed-point value zero-extended to 32 bits) is read from offset `0x0C`. A compact AXI4-Lite FSM handles all handshakes, while a combinational multiplexer selects the correct register data for readback. This register-oriented design ensures deterministic, low-overhead communication between the PS and PL and simplifies integration with the host REST API.

2.3 PS-PL Communication using AXI4-Lite

The AXI4-Lite slave in our PL implements two dedicated finite-state machines (FSMs) to manage all bus transactions with minimal logic and latency. On the write side, the Write FSM sequences through the typical AXI4-Lite handshake phases: it asserts `AWREADY` only once `AWVALID` is seen, then waits for `WVALID` before capturing the write data and address; finally, it drives `BVALID` and the `OKAY` response once the data has been committed to the target register. This tightly controlled state progression ensures that control registers—such as `update_enable`, `n1_stop_val`, and `n2_stop_val`—are updated atomically on each valid transaction, preventing spurious or partial writes.

On the read side, a complementary Read FSM handles every incoming read request. When `ARVALID` is asserted by the PS, the FSM latches the read address and immediately deasserts `ARREADY` to throttle new requests until it can present valid data. Once the target register's value is available—whether it be the clock divider state or the current neuron potential—the FSM drives `RDATA`, asserts `RVALID`, and signals an `OKAY` response. After the PS deasserts `RREADY`, the FSM returns to its idle state, ready to service the next read cycle.

A small combinational data multiplexer sits alongside these FSMs to steer the correct 32-bit register out on the read data bus. Based on the latched address, it selects between the control/status registers (`update_enable`, `n1_stop_val`, `n2_stop_val`), the programmable clock-divider count, or the live Q2.6-encoded membrane potential. This approach avoids adding extra clock cycles or logic depth, since the multiplexer output simply feeds into the final read-data register of the Read FSM.

Finally, we incorporate a 32-bit programmable clock divider into the same AXI4-Lite framework. Its divisor register is written via the Write FSM, and the divider counter itself ticks every system clock. When the counter overflows, it pulses the neuron-update logic and resets; the counter value can be read by the PS to calculate precise polling intervals. Because the clock-divider register and counter state share the same bus and FSM mechanisms, we achieve sub-microsecond reconfiguration without extra clock domains or complex timing constraints.

2.4 AXI-SNN Communication

The AXI peripheral side of the PL is able to communicate with the SNN side of the PL through a few main data and control signals. There are two 32-bit data buses that go

between the AXI and SNN PL. One is the "addr" bus, and the other is the "data" bus. When the AXI peripheral places pushes a 32-bit address onto the addr bus, the SNN will immediately (combinationaly) place one of the three neuron potentials onto the bus, with the first 24 bits being zeros and the last 8 being the Q2.6 fixed point decimal number representing the neuron potential at that moment in time.

There is also an assortment of control signals to be used. The standard enable and reset control signals are included, as well as a separate "update_enable" signal, and two 32-bit input signals called "n1_stop_val" and "n2_stop_val." The update_enable signal controls whether or not the neuron update logic runs. If the signal is high, then the neuron potentials will update. If the signal is low, then they will not update. Having a separate enable signal for update logic versus the entire network enables better separation between the internal bus/latching, logic and the business logic of constructing and simulating the neurons.

The stop values correspond to how often the synthetic inputs to the two input neurons are fired. Each value is the number of update ticks (*not* clock ticks, update ticks) that pass by between firing the synthetic inputs. When the number of update ticks since the last firing matches the value on the corresponding stop_val input, that input will fire and reset it's internal counter to zero. Each input has it's own value, allowing easily customizable firing rates that can show differences in spiking rate.

2.5 PC-PS Communication

Efficient control and data exchange between the host computer (PC) and the Processing System (PS) on the PYNQ-Z2 board are crucial for configuring the spiking-neuron core and retrieving membrane-potential traces. Two complementary software paths are employed: a MATLAB-based visual interface for live monitoring and a Python-based FastAPI client–server stack for scripted acquisition. The following subsections outline the implementation of each path.

2.5.1 Live Plotting

Live plotting of the neurons was accomplished using Matlab, because there were issues with running an interactive plot using matplotlib through WSL2. The live plotting was accomplished by preparing three buffers to hold the neuron potentials for our three neurons and then polling the REST API periodically to add values to the buffers. The control signals are polled at startup and every time a button is pressed to update them, but nowhere else, so it is possible for the client state to become out of sync with the actual SNN state if there are multiple clients attempting to send control signal requests at the same time. A demonstration of the live system in action can be found at this link(if this link is broken and you need to see the video, email me at e.a.morton@wustl.edu and I can send you the video file upon request).

2.5.2 Static Analysis

While the live MATLAB client provides an oscilloscope-style glimpse of membrane potentials, our *static* pipeline captures long, loss-less traces for post-processing in Python. The workflow relies on two scripts:

1. `FastAPI.py` – executed on the PYNQ–Z2 processing system.

2. `Testing_API_2.py` – executed on the host PC and streams data to a `.csv` file.

Snapshot endpoint The FastAPI service already exposes single-neuron read routes (`/neuron1`, `/neuron2`, `/neuron3`). For high-resolution logging a vectorised `/snapshot` route was added which reads all three membrane-potential registers and a global update counter in a single AXI transaction:

```
@app.get("/snapshot")
def snapshot():
    raw1 = mmio.read(N1_ADDR)
    raw2 = mmio.read(N2_ADDR)
    raw3 = mmio.read(N3_ADDR)
    clock = mmio.read(CLOCK_ACCUM_ADDR)
    return {"v1": raw1, "v2": raw2, "v3": raw3, "clock": clock}
```

Figure 4: Vector snapshot route returning simultaneous Q2.6 readings.

All values are returned in raw 8-bit Q2.6 format and converted off-board to preserve full precision.

Acquisition client `Testing_API_2.py` performs three roles:

1. **Board setup** – arms the neuron core through `/run` and programs the hardware clock divider via `/clock_target/{divider}`.
2. **Timed acquisition loop** – polls `/snapshot` at an interval

$$T_{\text{loop}} = \frac{\text{divider}}{f_{\text{fabric}}} \times 0.18,$$

   oversampling each neuron update by roughly 5.5× while keeping HTTP overhead negligible.
3. **Optional stimulation** – issues a `/stimulate` pulse every 5s to provoke deterministic spikes (disabled for the results presented in this report but retained for future work).

During each poll the script converts the raw register byte to volts using the helper in Listing 5. Results are appended to a timestamped `.csv` whose header is `timestamp_ms, stim, v1, v2, v3`. A `SIGINT` handler flushes and closes the file cleanly to avoid partial records.

```
def q26_to_float(raw32: int) -> float:
    raw8 = raw32 & 0xFF        # isolate low byte
    if raw8 & 0x80:            #  s i g n extend  if negative
        raw8 -= 0x100
    return raw8 / 64.0         # Q2.6  -> divide by 2**6
```

Figure 5: Local Q2.6 decoder mirroring `bin2q26()` on the board.

Data products The resulting comma-separated log (typically 3–5 MB for a one-minute run) forms the backbone of all static plots in Section 3. Because every hardware tick is archived, we can subsequently

- reconstruct spike rasters, inter-spike-interval (ISI) histograms, and voltage distributions;
- verify coincidence detection by aligning V_3 spikes with V_1 and V_2 events;
- benchmark REST latency by differencing successive `clock` fields.

By decoupling acquisition from visualisation, long experiments can run unattended on the FPGA while analysis proceeds later on any machine with Python support.

3 Results

The following analyses quantify both functional correctness and hardware efficiency of the FPGA-based spiking neural network. First, time-domain and event-based plots derived from the static-analysis pipeline illustrate membrane dynamics and spike statistics for three interconnected neurons. Subsequent figures examine the utilisation of core FPGA resources as network size scales, providing guidance for future expansions and optimisation strategies.

3.1 Graphing Neuron Potential

Fixed–point membrane potentials were logged for a 55s run at a hardware–paced update period of 150ms (cf. Section 2.5.2). The following figures present the recorded traces.

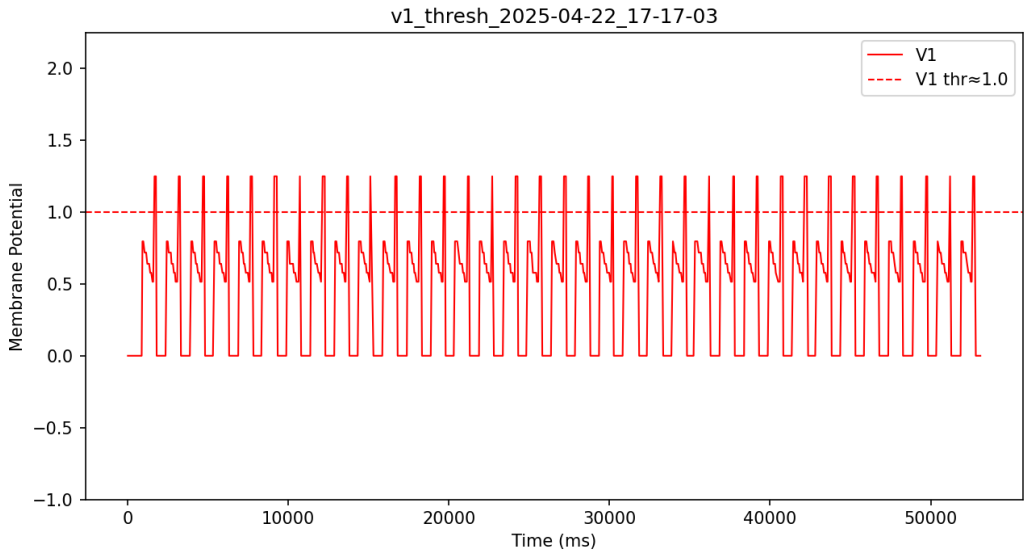


Figure 6: Neuron 1 membrane potential with a dashed threshold at 1V. Each vertical excursion above threshold constitutes a spike; the exponential decay that follows confirms correct implementation of the leaky term in the LIF model.

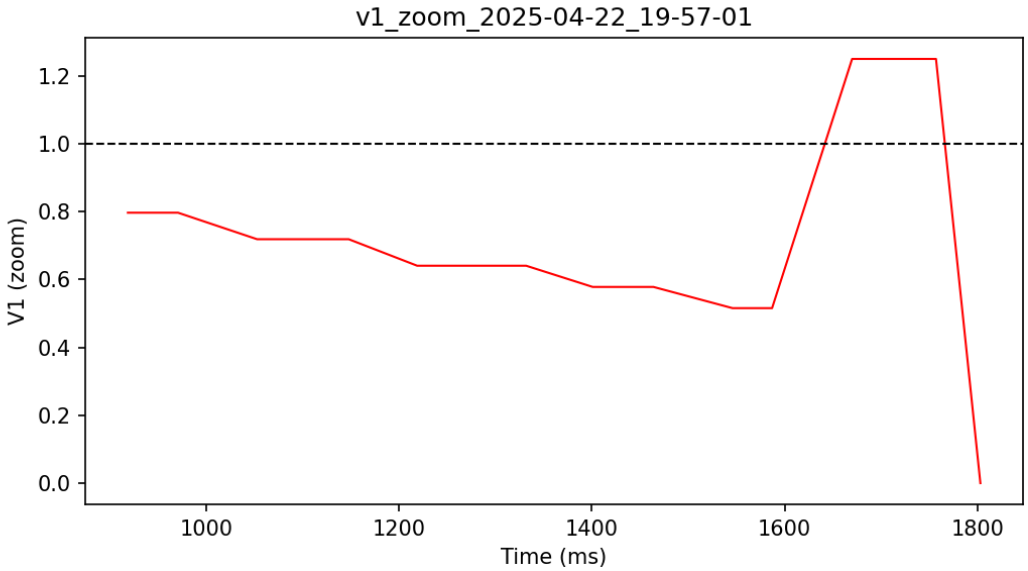


Figure 7: Scaled view of one inter-spike interval for Neuron 1. The discrete step pattern reflects the 8-bit Q2.6 arithmetic, while the monotonic descent illustrates leak-only dynamics in the absence of stimulation.

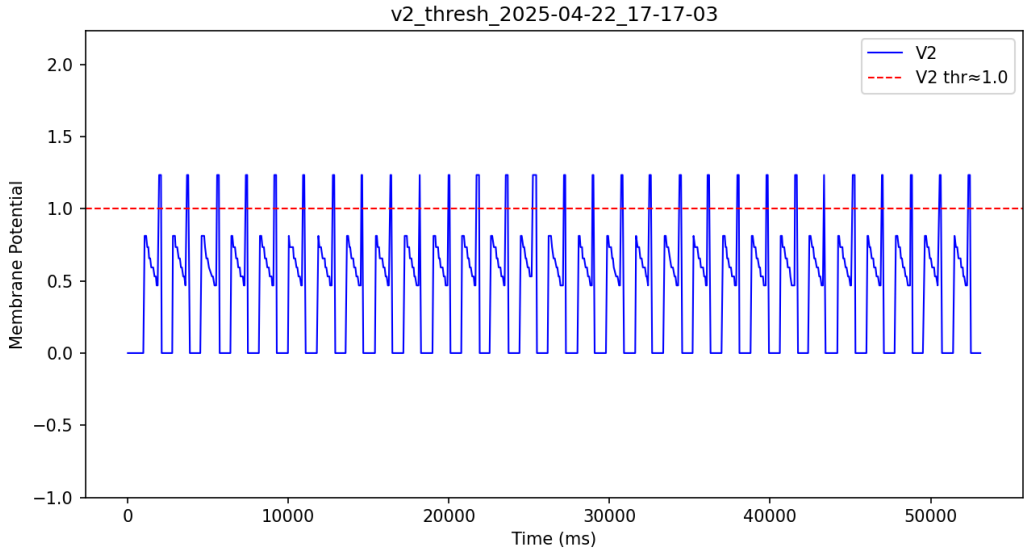


Figure 8: Neuron 2 exhibits identical threshold and decay parameters, but a lower external stimulus rate, leading to a longer inter-spike period (1.5s).

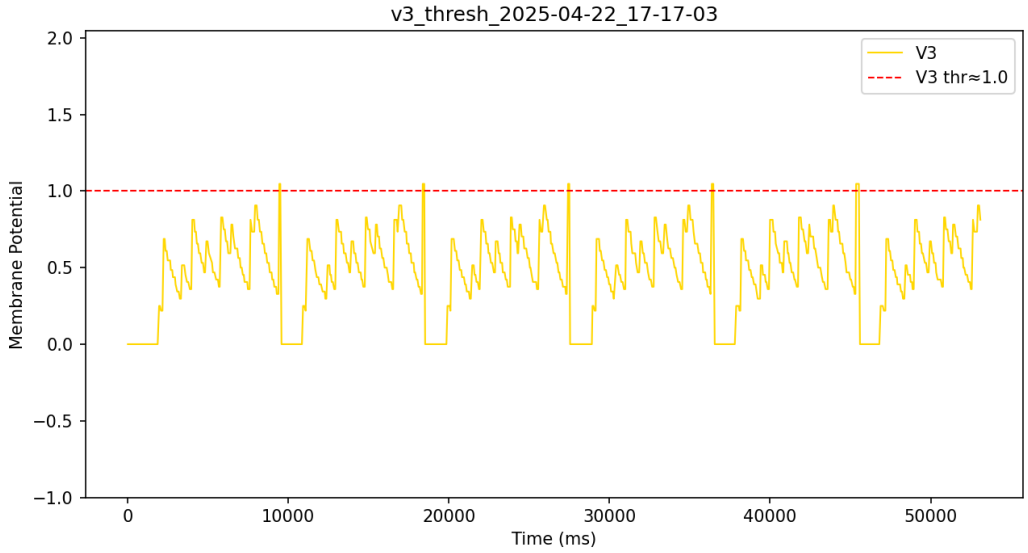


Figure 9: Neuron 3 integrates weighted spikes from Neuron 1 and Neuron 2 only. Irregular supra-threshold events corroborate the coincidence-detection objective of the design.

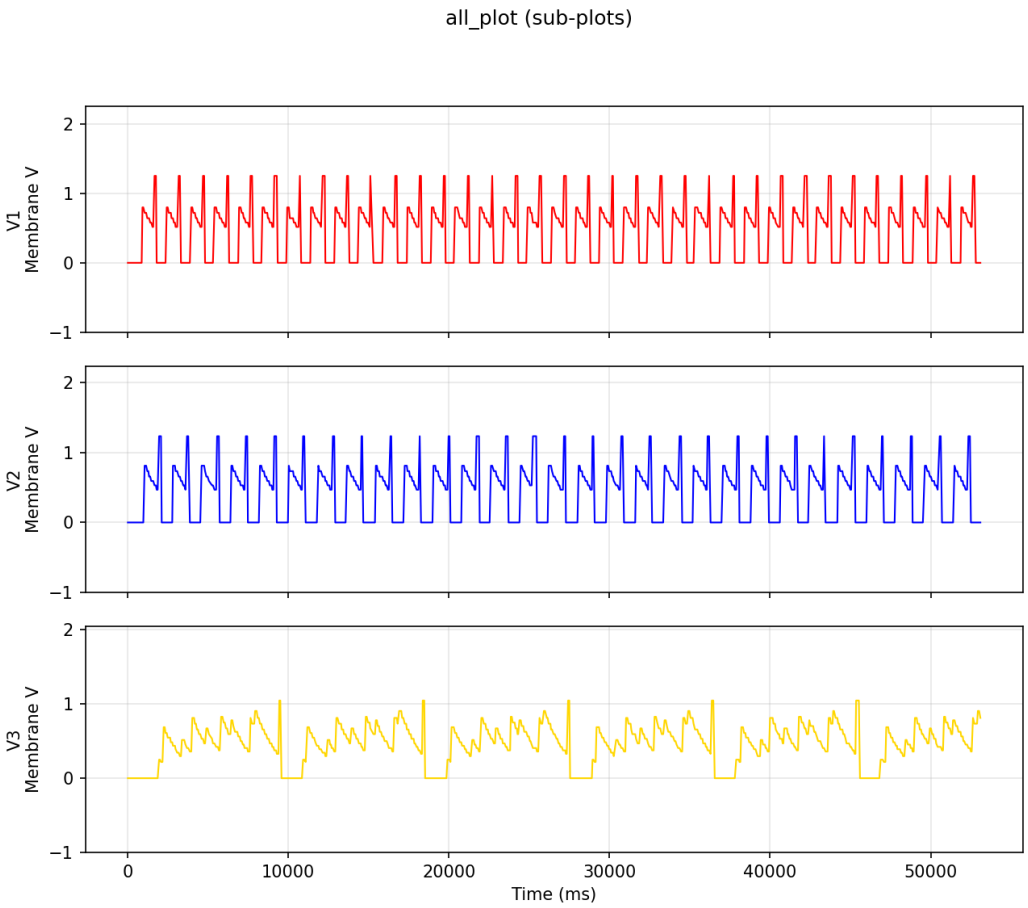


Figure 10: Superimposed traces of all three neurons. Divergent firing patterns arise exclusively from differing input schedules and synaptic weights; intrinsic decay and threshold constants are shared.

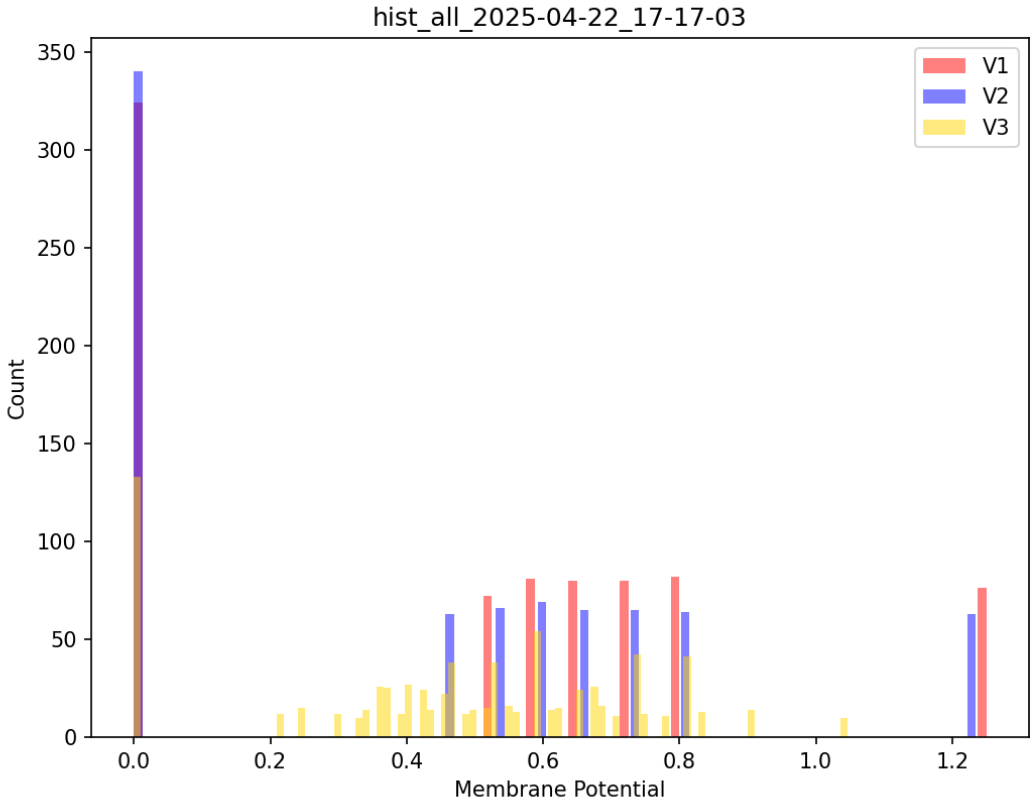


Figure 11: Histogram of every sampled voltage. The pronounced peak at 0V reflects the reset state; the broad cluster between 0.3V and 0.8V represents sub-threshold integration. Sparse tails above 1V correspond to true spikes.

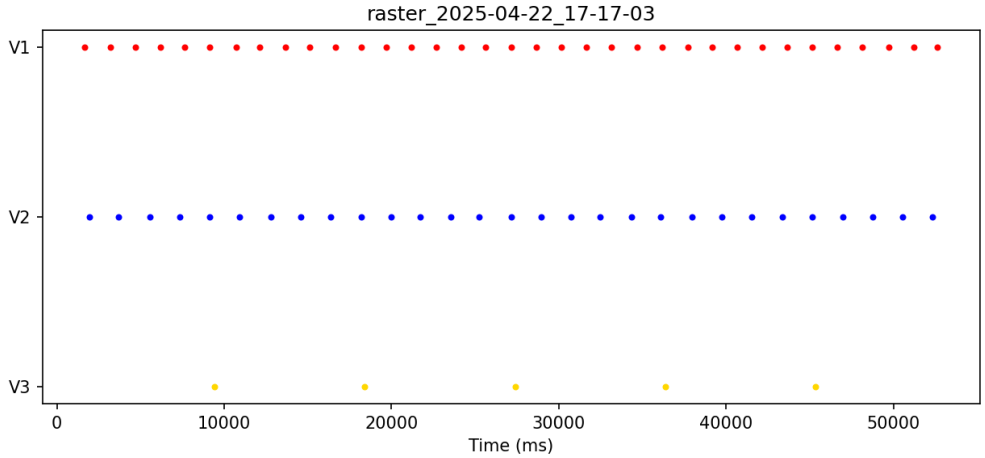


Figure 12: Spike-train raster condensed over the full acquisition window. Regular rows for V_1 and V_2 verify periodic stimulation; sporadic dots for V_3 occur only when upstream spikes coincide within the designed temporal window.

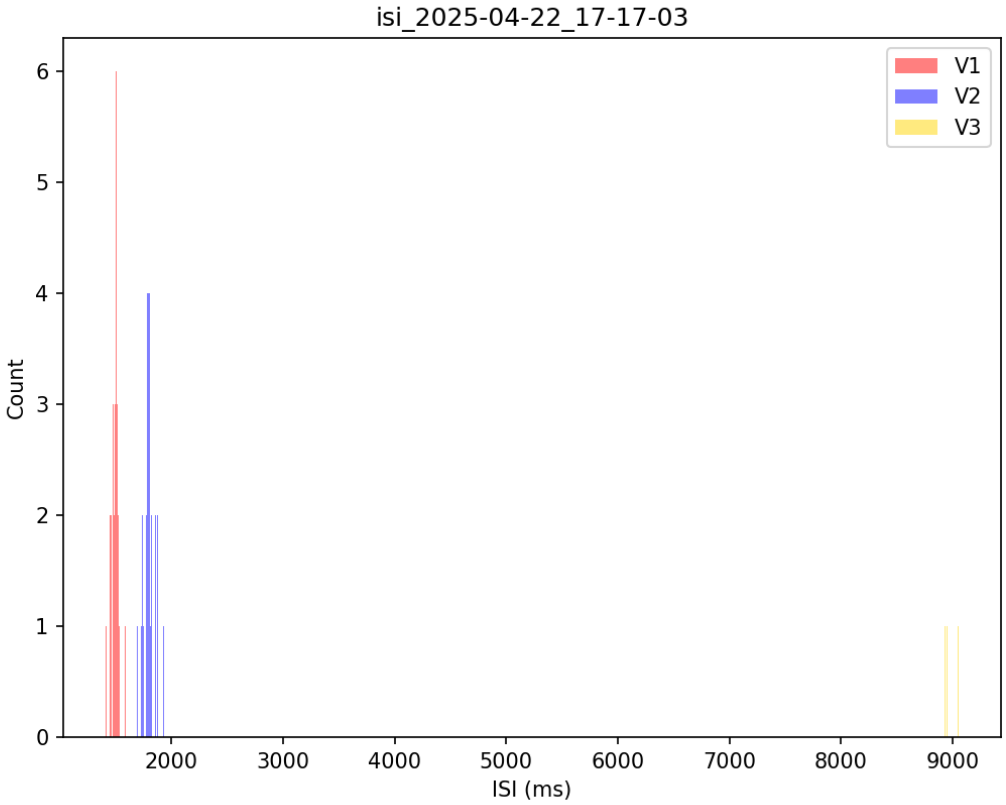


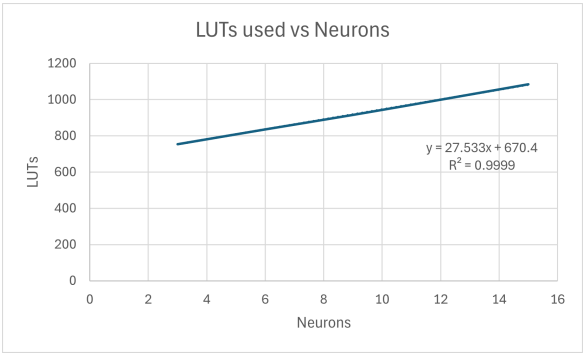
Figure 13: Inter-spike–interval (ISI) distributions. Narrow peaks at 1s (V_1) and 1.5s (V_2) affirm externally imposed periods, whereas V_3 exhibits a broad mode centred at 9s, consistent with the coincidence-detection requirement.

Conclusion of Static Analysis The collected traces validate the correct hardware realisation of the Leaky Integrate-and-Fire equations: (i) exponential leakage between spikes, (ii) deterministic firing at the programmed stimulus periods for input neurons, and (iii) sparse, coincidence-driven firing for the down-stream integrator. Histogram and ISI analyses further confirm that spiking activity is both event-driven and energy-efficient, with the hardware remaining in its low-power reset state for the majority of the run.

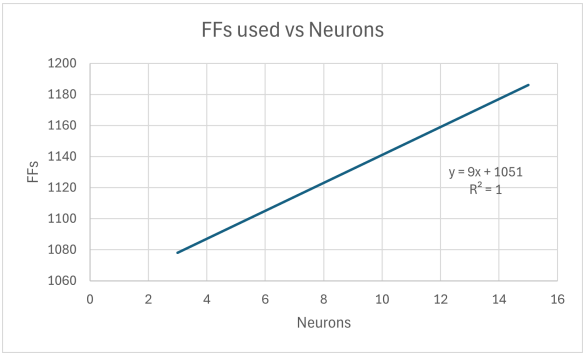
3.2 Resource Usage

An important facet of FPGA development is resource usage. How many LUTs, FFs, DSPs, etc, are used is critical to maintaining speed and efficiency throughout a computation. Early in our development cycle, we saw high resource usage with even just one neuron, up to 13% of LUTs used. This was a result of the neuron arithmetic being computed as 32-bit fixed point arithmetic, specifically, Q16.16. After further looking into SNNs and the necessity of high-precision arithmetic, we dropped the precision all the way down to Q2.6, which significantly reduced the resource usage. In figs. 14a and 14b we can see a visual representation of the resource usage as a function of the number of neurons.

There is approximately 28 LUTs used per neuron, with a base cost of roughly 670 LUTs, and approximately 9 FFs used per neuron, with a base cost of roughly 1051 FFs. These numbers represent roughly 1% of the total resources of even the small FPGA we were



(a) Graph of the LUT usage as a function of the number of neurons instantiated.



(b) Graph of the FF usage as a function of the number of neurons instantiated.

Figure 14: Graphs of resource usage as a function of the number of neurons instantiated.

provided. The number of neurons can easily scale into the hundreds or thousands with minimal optimization. When resources become an issue, the more complicated blocks, such as the DSP multiplication blocks created in the internal update logic for each neuron, can be shared across multiple neurons while decreasing the overall update speed of the network. Though this is a tradeoff, it is much better to be able to run larger networks at slower speeds than to only be able to run small networks at slower speeds, just like with classical neural networks. There are many more optimizations that could be made as the number of neurons scales up, including sharing of parameters by moving it to an input, and by allocating block RAM to hold values rather than allowing Vivado to try and implement thousands of single-bit FFs.

4 Conclusions

This report presented a complete, FPGA-resident prototype of a Leaky Integrate-and-Fire spiking neural network, together with a host–to–device software stack for configuration, data capture, and analysis. The implementation combined hand-written Verilog neurons, an AXI4-Lite control peripheral, and a Python/FastAPI interface, enabling sub-microsecond register access and sustained, loss-less logging of membrane potentials. Experimental traces confirmed the expected exponential leak, deterministic firing for externally driven neurons, and coincidence-triggered activity in a post-synaptic integrator. Transitioning from Q16.16 to Q2.6 fixed-point arithmetic reduced per-neuron logic utilisation to approximately 28 LUTs and 9 FFs, permitting hundreds of neurons to fit comfortably within the resource envelope of the PYNQ-Z2 fabric.

These results demonstrate that biologically inspired, event-driven computation can be realised on low-cost FPGAs with modest resource budgets and without sacrificing temporal fidelity. The separation of concerns—hardware neurons, register-mapped control, and REST-based monitoring—offers a flexible template for future neuromorphic studies.

Future Directions

- **Network Scaling.** Parameterising the neuron module and automating instantiation would allow network sizes in the 10^3–10^4 range; shared multipliers or time-multiplexed update logic could further lower the per-neuron cost.

- **On-chip Plasticity.** Incorporating spike-timing-dependent plasticity (STDP) or reward-modulated learning would convert the platform from inference-only to adaptive processing.
- **Power Characterisation.** Direct measurement of dynamic power during sparse versus dense firing would quantify the energy-efficiency gains commonly attributed to SNNs.
- **High-Level Synthesis.** Exploring HLS or Python-based HDL generators could accelerate design iterations while retaining the deterministic timing required for neuromorphic workloads.
- **Edge Integration.** Coupling the network with sensor front-ends—event cameras, microphone arrays, or bio-signal interfaces—would showcase the latency and power benefits in real-time edge applications.

Collectively, these extensions would transform the current proof-of-concept into a versatile research platform, capable of evaluating large-scale SNN architectures and on-device learning algorithms under realistic resource-constraint and latency conditions.

5 Deliverables

As stated in our formal proposal, the project's anticipated deliverables were

1. A deep description of how information is processed in spiking neural networks (SNNs) and the theoretical effectiveness of the strategies we implement.
2. A tool to synthesize Verilog code representing an SNN (hard-coded) or a Verilog library capable of dynamically instantiating arbitrary SNN topologies.
3. An analysis of resource usage per neuron (LUTs, FFs, DSPs, etc.) along with strategies for optimizing utilization as network size scales.

5.1 Completed Deliverables

- **SNN Description (Deliverable 1).** We provided a comprehensive background and mathematical treatment of Leaky Integrate-and-Fire neurons in Sections 1 and 2, including system-level diagrams and the REST/AXI-Lite control architecture.
- **Resource Usage Analysis (Deliverable 3).** In Section 3.2, we presented empirical graphs of LUT and FF usage versus neuron count (Figure 4), and discussed precision/format trade-offs (Q16.16 $\rightarrow$ Q2.6) to reduce resource footprints.

5.2 Uncompleted Deliverables and Reasons

- **Verilog Synthesis Tool (Deliverable 2).** We did not develop an automated HDL generator or dynamic Verilog library. Instead, we manually instantiated a three-neuron network in Verilog. This scope reduction arose from:
    1. Underestimating the complexity of the AXI4-Lite FSM integration, which consumed our available FPGA development cycles.

2. Extended debugging of the PS-PL RESTful/MMIO interface, which delayed higher-level tooling work.
3. Prioritizing a working proof-of-concept over broader tooling features to meet semester deadlines.

5.3 Process and Workflow Improvements

Reflecting on our experience, the following changes could streamline future projects:

- **Parallelize Hardware and Tooling.** Assign separate sub-teams for HDL integration versus code-generation tooling, with regular sync points to avoid bottlenecks.
- **Early Adoption of HLS or Code-Gen Frameworks.** Evaluate high-level synthesis (HLS) or Python-based Verilog generators at project kickoff to reduce manual HDL burden.
- **Continuous Integration for FPGA Builds.** Automate bitstream generation and AXI interface testbenches to catch integration regressions early.

6 Timeline

Figure 15: Gantt chart comparing planned versus actual timeline for the SNN-on-FPGA implementation.

As shown in Figure 15, we met our early milestones with minimal slip. These accomplishments provided confidence in both our custom PL logic and the RESTful control infrastructure.

However, the AXI/API Communication phase—during which we designed and integrated our custom AXI4-Lite slave—took significantly longer than anticipated. The added effort required to implement robust FSM-based handshaking and atomic register updates caused this task to overrun by nearly all of its allotted duration. As a direct result, subsequent tasks (SNN Design/Training and Full API Implementation) were either deferred or only partially completed.

Moreover, our original plan called for offline training of a larger, multi-neuron spiking model. Due to the extended AXI4-Lite development and overall time constraints, we did not perform any model training. We therefore scaled back to a three-neuron LIF network

with fixed thresholds and decay constants, and did not implement the broader neuron array originally envisioned.

7 Table of Responsibility

Member	Sections
Ethan	1, 2, 2.1, 2.4, 2.5.1, 3.2, 7
John	2.2, 2.3, 5, 6
Edgar	2.5, 2.5.2, 3, 3.1, 4

Table 1: Division of report sections among team members.

References

[Eshraghian, 2021] Eshraghian, J. K. (2021). snntorch documentation.

[Goodman, 2023] Goodman, D. (2023). Neuroscience for machine learners.