NTNU
Kunnskap for en bedre verden

Training a Spiking Neural Network for Object Detection with an Event Camera and a Neuromorphic Chip

Project Report

AIS4501 Specialisation Project in Mechatronics and Automation

Jon Urcelay San Roman

12th December 2025

Abstract

This project explores the integration of event-based vision with spiking neural networks (SNNs) for ultra-low-power object detection on neuromorphic hardware. The Prophesee EVK4 event camera and BrainChip Akida AKD1000 neuromorphic processor are selected as the core platform.

The work focuses on a complete, from-scratch reproduction of BrainChip's official event-based pupil center detection example. This includes training a spatio-temporal CNN (tennst backbone), INT8 post-training quantization with explicit calibration, CNN-to-SNN conversion, and attempted deployment on real Akida hardware. All steps were executed and documented in detail, addressing gaps in the official examples regarding code, hyperparameters, calibration datasets, and hardware compatibility constraints.

The project validates a functional event-based SNN pipeline and highlights critical practical limitations of current neuromorphic hardware for state-of-the-art spatio-temporal models, providing a solid foundation for future edge-deployable vision systems.

Contents

List of Figures

List of Tables

Acronyms

EC Event Camera.

DVS Dynamic Vision Sensor.

HDR High Dynamic Range.

NN Neural Network.

ANN Artificial Neural Network.

CNN Convolutional Neural Network.

LR Learning Rate.

RGB Red Green Blue.

YOLO You Only Look Once.

TENN Temporal Event-Based Neural Networks.

TENNST Temporal Neural Network Spatio-Temporal.

PTQ Post Training Quantization.

QAT Quantization Aware Training.

STE Straight-Through Estimator.

SNN Spiking Neural Network.

IF Integrate-and-Fire.

LIF Leaky Integrate-and-Fire.

BPTT Backpropagation Through Time.

CPU Central Processing Unit.

GPU Graphics Processing Unit.

NC Neuromorphic Chip.

AER Address-Event Representation.

TPU Tensor Processing Unit.

NPU Neural Processing Unit.

D-FOV Diagonal Field of View.

PHY Physical Layer.

PCIe Peripheral Component Interconnect Express.

LPDDR4 Low-Power Double Data Rate 4.

DMA Direct Memory Access.

IoT Internet of Things.

SoC System on a Chip.

NoC Network on a Chip.

FLOP Floating Point Operation.

HEVC High Efficient Video Coding.

INRC Intel Neuromorphic Research Community.

ASIC Application Specific Integrated Circuit.

FPGA Field-Programmable Gate Array.

STDP Spike-Timing-Dependent Plasticity.

VMM Vector Matrix Multiplication.

MAC Multiply-Accumulate operation.

NSoC Neuromorphic System-on-Chip.

NPO Neuromorphic Processing Node.

GDDR Graphics Double Data Rate.

CUDA Compute Unified Device Architecture.

IDE Integrated-Development Environment.

SDK Software Development Kit.

DDS Data Distribution Service.

HAL Hardware Abstraction Layer.

DWS Depthwise-Separable Convolutions.

1 Introduction

The main motivation for this project stems from recent advancements in event cameras, spiking neural networks (SNN), and neuromorphic chips, technologies that promise energy-efficient, real-time processing for dynamic environments, coupled with the lack of studies adapting these tools to maritime applications. Event cameras offer high temporal resolution (at μs scale), high dynamic range (120 dB), and low throughput, making them particularly suitable for challenging visual conditions at sea, such as intense glare, fog, rapid wave motion, and varying lighting. Similarly, SNNs and neuromorphic hardware provide biologically inspired computation with ultra-low power consumption (often in the micro-watt range), ideal for deployment on energy-constrained marine platforms where traditional frame-based vision systems struggle with power, bandwidth, and latency.

This project originally aimed to develop an event-based vision system for maritime object detection using a selected event camera and neuromorphic processor, with the intention of comparing performance against conventional RGB camera frame-based detection using established CNNs like YOLO or Faster R-CNN. However, the complexity of establishing a functional end-to-end SNN pipeline on real hardware led to a refocusing on replicating and extending a given official event-based detection example as a proof-of-concept. This shift enabled a deeper investigation of the practical challenges in SNN training and deployment, while maintaining successful integration and operation of the event camera throughout the project.

The report is structured as follows: Section 2 provides the theoretical background; Section 3 details the hardware, software, and workflow to be used; Section 4 presents the achieved results and limitations; and Section 5 and 6 conclude with reflections and future work. Ultimately, this project demonstrates a complete event-based SNN pipeline and highlights critical limitations of current neuromorphic hardware for state-of-the-art spatio-temporal models, laying a foundation for future edge-deployable vision systems.

2 Theoretical background

2.1 Event Cameras

Event cameras, also known as neuromorphic cameras, are bio-inspired sensors that differ from conventional frame-based cameras. Instead of capturing images at a fixed rate, they asynchronously measure per-pixel brightness changes, and output a stream of events that encode the time (t), pixel location (x, y) and sign of the brightness changes (p), as it can be seen in Figure 1.

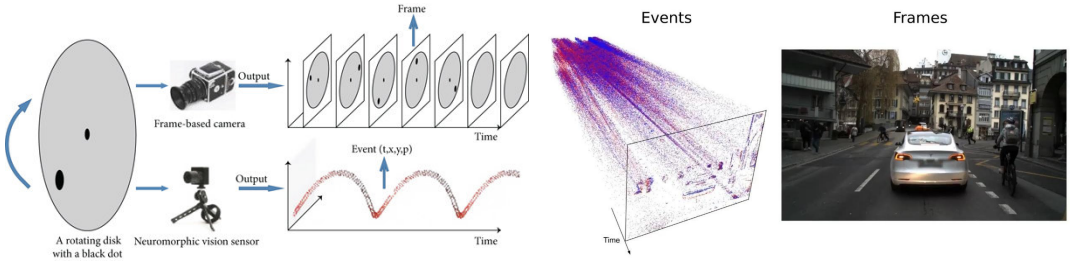

Figure 1: Output from frame-based camera and event camera.

For each pixel, an event is triggered if the logarithmic variation of its brightness (relative to the previous time step) is bigger than the threshold C, as described in Equation 1. This means that if the brightness change is smaller than C, there is no output for that specific pixel. See Figure 2 for a visual representation of a single pixel triggering events, and the circuit that makes this possible.

$$log(x, y, t + \Delta t) - log(x, y, t) = \pm C \tag{1}$$

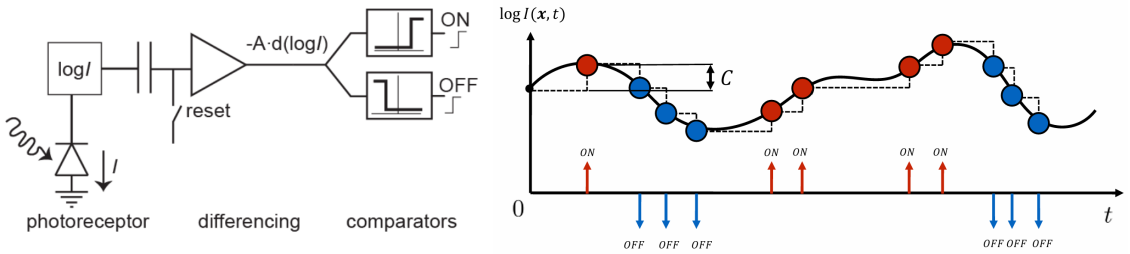

Figure 2: (Left) Single pixel circuit for event detection. (Right) Single pixel brightness intensity and event generation representation.

Event cameras offer some advantages compared to traditional cameras: high temporal resolution (in the order of µs), very high dynamic range (140dB vs 60dB), and low power consumption. Due to these advantages, event cameras have a large potential for robotics and computer vision in challenging scenarios for traditional cameras, such as high-speed environments, very bright/dark scenarios, and in low-power required applications.

However, in order to unlock their potential, novel methods are required to process the unconventional output of these sensors. This is a research area that has been active, at least, the last 15 years. To put into perspective, event cameras were first commercialized in 2008 by T. Delbruck (UZH&ETH) under the name of Dynamic Vision Sensor (DVS).

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models inspired by biological neural systems, consisting of interconnected nodes (neurons) organized in layers that process input data through weighted connections and non-linear activation functions. ANNs excel at learning complex patterns from data and form the foundation of modern deep learning, enabling tasks such as image classification, object detection, and regression.

These neural networks can be trained using supervised learning (Figure 3), which minimizes the loss function by using gradient descent on the weight values of the network. The loss function is an equation that relates, for the same input, the predicted outcome from the network and the expected prediction. This loss function decreases when both values are similar, and increases as the values differ. For this reason, a dataset is needed for supervised learning, which contains a series of inputs and their expected outcome (also called ground truth labels or targets). This is explained in detail in the videos referenced in [1–4].

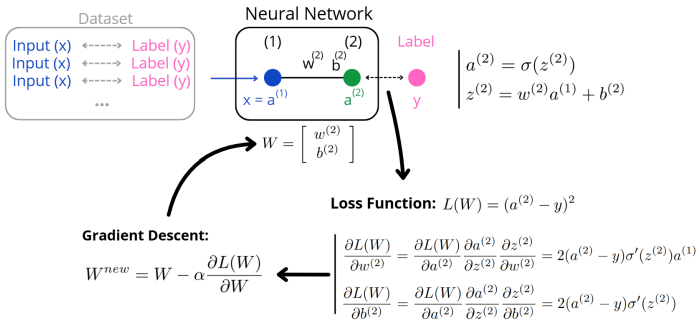

Figure 3: Supervised training visual graph of a single neuron NN.

Key hyperparameters control the efficiency and convergence of this training process:

- **Batch size:** the number of training samples processed simultaneously before updating the model's weights. Larger batches enable more stable gradient estimates and better GPU utilization, reducing per-epoch time by amortizing overhead across multiple samples.
- **Epochs:** the number of complete passes through the entire training dataset. When a single epoch is finished and backpropagation performed, validation loss can be computed in the updated network, stopping training if the loss is smaller than a given threshold or if the loss change is very small. This is called early stopping, and usually prevents overfitting.

- **Learning rate (LR):** the step size for weight updates during gradient descent. A high LR accelerates convergence but risks instability; too low slows down training.
- **AdamW optimizer with weight decay:** an adaptive optimizer (Adam) that decouples weight decay from the adaptive learning rate, adding L2 regularization to prevent overfitting. The decay term penalizes large weights, improving generalization without slowing convergence.
- **LR scheduler with linear warm-up and cosine decay:** starts with a gradual LR ramp-up to stabilize early training, followed by cosine annealing to smoothly reduce LR toward zero, enabling finer convergence in later epochs and typically improving final accuracy.

This is the basis of how supervised training works. More advanced methods improve the training process and the resulting performance of the network, but their explanation is not required to continue with this project. In addition, reinforcement learning (RL) offers a way of training neural networks without using a dataset of previously labeled values. This is neither required for the rest of this project, so it's content will be skipped.

To perform object detection with a traditional frame-based camera, like an RGB camera, Convolutional Neural Networks (CNNs) are widely used, such as YOLO [5] or Faster R-CNN [6]. These architectures extract hierarchical features from images via convolutional layers and enable near-real-time detection.

2.2.1 Temporal Event-Based Neural Networks

Unlike standard CNN networks that only operate on the spatial dimensions, Temporal Event-Based Neural Networks (TENN) [7] contain both temporal and spatial convolution layers. They may combine spatial and temporal features of the data at all levels from shallow to deep layers. In addition, TENNs efficiently learn both spatial and temporal correlations from data in contrast with state-space models that mainly treat time series data with no spatial components. Given the hierarchical and causal nature of TENNs, relationships between elements that are both distant in space and time may be constructed for efficient continuous data processing (such as video, raw speech, and medical data). Causal means using previous processed values of the time series to estimate current or future values.

The fundamental principle of TENNs is the factorization of spatio-temporal feature extraction into two sequential stages [8], illustrated in Figure 4. First, temporal convolutions (1D, causal, non-strided kernels) are applied independently at each spatial location to capture local motion and timing patterns across successive time steps. These are followed by spatial convolutions (standard 2D kernels) that aggregate structural information within each resulting temporal feature map. This separation allows efficient modeling of both dynamics and appearance while maintaining low computational complexity. TENNs typically employ depthwise separable convolutions: depthwise operations process each channel independently (reducing parameters), while pointwise (1×1) convolutions enable cross-channel mixing.

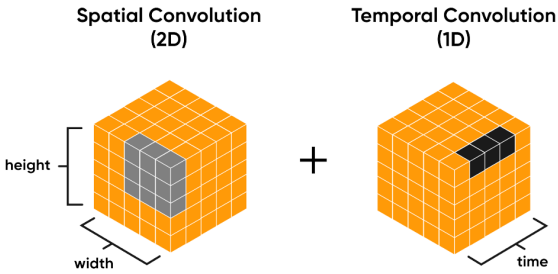

Figure 4: 2D Spatial Convolution and 1D Temporal Convolution for a TENN.

2.3 Quantization of a Neural Network

Quantization is a technique for deploying large neural networks efficiently, enabling faster inference and reduced memory footprint. It involves converting the model's weights and activations from higher precision (floating point bit formats) to lower precision (integer formats), which are described in section 2.3.1. Two primary quantization approaches exist, Post-Training Quantization (PTQ) and Quantization Aware Training (QAT), which are described in sections 2.3.2 and 2.3.3, respectively.

2.3.1 Bit Formats

In digital hardware, numerical values are represented using a fixed number of bits. The choice of format determines memory footprint, arithmetic complexity, energy consumption, and numerical precision of a neural network. Two main families are used for weights, biases, activations, and gradients:

Floating-point formats

These represent a number via a sign bit, an exponent, and a mantissa. Their layout and value computation are shown in Table 1, Figure 5, and Equation (2).

Name	Format	Bits	s	e	m	Signed	Approx. Range
Double Precision	Float64	64	1	11	52	Yes	$\pm 1.8 \times 10^{308}$
Single Precision	Float32	32	1	8	23	Yes	$\pm 3.4 \times 10^{38}$
Brain Floating Point	BFloat16	16	1	8	7	Yes	$\pm 3.4 \times 10^{38}$
Half Precision	Float16	16	1	5	10	Yes	± 65504

Table 1: Common floating-point formats used in deep learning

$$X = (-1)^S \times 2^{E-(2^e-1)} \times (1+M) \tag{2}$$

where S, E and M are:

$$S = b_0$$
$$E = b_1 2^{e-1} + b_2 2^{e-2} + ... + b_e 2^{e-e}$$
$$M = b_{e+1} 2^{-1} + b_{2+2} 2^{-2} + ... + b_{e+m} 2^{-m}$$

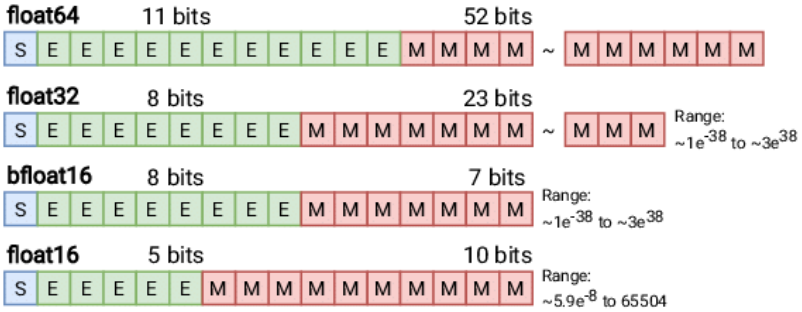

Figure 5: Floating-point formats representation.

Fixed-point / integer formats

These represent numbers as integers scaled by a (global or per-tensor) factor. Their layout and value computation are shown in Table 2 and Equation (3).

Name	Format	Bits (i)	Signed (u)	Approx. Range
Signed 8-bit Integer	INT8	8	Yes (1)	$[128, 127]$
Un-signed 8-bit Integer	UINT8	8	No (0)	$[0, 255]$
Signed 4-bit Integer	INT4	4	Yes (1)	$[8, 7]$
Un-signed 4-bit Integer	UINT4	4	No (0)	$[0, 15]$

Table 2: Common Fixed-point / Integer Formats

$$X = b_0 2^{i-1} + b_1 2^{i-2} + ... + b_{i-1} 2^0 - u 2^i \tag{3}$$

In a neural network, every weight, bias, and activation can be stored in one of these formats. Reducing precision from Float32 to INT8 typically reduces memory and compute energy by 4×; moving further to INT4 yields another 2× improvement. These gains come at the cost of potential accuracy degradation, which must be mitigated by suitable quantization techniques, as PTQ or QAT, discussed in the following sections.

2.3.2 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is a straightforward approach where a pre-trained model (typically Float32) is converted to a lower precision (usually INT8) without additional training

[9]. The entire process is performed after training of the original neural network has been completed. The algorithm of PTQ is represented in Figure 6, which consists of two main phases: calibration and mapping.

In the calibration phase, the process begins by computing the range (x_{min}, x_{max}) of the model parameters using a calibration dataset $\mathcal{D}_{\text{cal}}$. The scale factor s is then determined based on this range and the desired bit-width b. The zero-point z is calculated to align the quantized values appropriately. Then, in the mapping phase, each tensor T in the model parameters is quantized by scaling and rounding, followed by dequantization to obtain the quantized model parameters $\hat{\Theta}$.

PTQ is advantageous due to its simplicity and efficiency, making it suitable for scenarios where retraining is impractical or where computational resources are limited. However, PTQ may lead to performance degradation, especially in models that are sensitive to precision loss.

2.3.3 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) integrates the quantization process into the training loop, allowing the model to adapt its parameters to the lower precision representation. The primary objective of QAT is to minimize the loss function $L(\hat{\Theta}, B)$ while accounting for the quantization effects, thereby ensuring that the final quantized model maintains high performance. This method typically involves strategies such as fake quantization, where quantization operations are inserted into the computational graph, and Straight-Through Estimator (STE) for handling the non differentiable quantization steps during backpropagation.

As represented in the algorithm in Figure 6, QAT starts from a pre-trained FP32 model or a PTQ-initialized model with model parameters Θ, using a lower learning rate than in the regular network's training, and runs for a limited number of epochs.

during each training iteration a batch B is sampled from the training data D, and the current model parameters Θ are quantized to obtain $\hat{\Theta}$. This stage is called forward quantization, where the same quantizing and dequantizing operations as in PQT are used, converting values to low-precision integers and immediately back to floating-point for gradient computation.

The loss $L(\hat{\Theta}, B)$ is then computed using the quantized parameters and the batch data. Gradients g are calculated with respect to the loss using a STE, which approximates the gradients through the non-differentiable quantization function. The model parameters Θ are then updated using the learning rate η and the computed gradients. This process continues until convergence, resulting in quantization-aware trained parameters Θ^* that are optimized to perform well despite the precision reduction.

QAT typically results in better performance compared to PTQ, as the model can learn to compensate for the quantization-induced errors. However, it requires additional computational resources and time for training, making it more suitable for scenarios where maintaining high accuracy is critical and retraining is feasible.

Algorithm 1 PTQ

Require:
Pre-trained model parameters Θ
Bit-width b
Calibration dataset $\mathcal{D}_{\text{cal}}$
Ensure:
Quantized model parameters $\hat{\Theta}$
$(x_{\min}, x_{\max}) \leftarrow \text{Range}(\Theta, \mathcal{D}_{\text{cal}})$
$s \leftarrow (x_{\max} - x_{\min})/(2^b - 1)$
$z \leftarrow \text{round}(-x_{\min}/s)$
for each tensor $T \in \Theta$ **do**
    $q_T \leftarrow \text{round}(T/s) + z$
    $\hat{\Theta}[T] \leftarrow (q_T - z) \cdot s$ {Dequantization}
return $\hat{\Theta}$

Algorithm 2 QAT

Require:
Pre-trained model parameters Θ
Low learning rate η
Training dataset $\mathcal{D}$
Ensure:
Quant.-aware trained parameters Θ^*
while not converged **do**
    $B \leftarrow \text{SampleBatch}(\mathcal{D})$
    $\hat{\Theta} \leftarrow \text{Quantize}(\Theta)$
    $L \leftarrow \text{Loss}(\hat{\Theta}, B)$
    $g \leftarrow \nabla_{\Theta} L$ {Gradients via STE}
    $\Theta \leftarrow \Theta - \eta \cdot g$
return Θ^*

Figure 6: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

2.4 Spiking Neural Networks

SNNs are presented as bio-inspired neural networks that mimic the brain's architecture, using asynchronous and sparse spikes to communicate between nodes. In this way, SNNs are a natural fit to event cameras, as they are able to use the raw event stream as an input for training detection algorithms.

In addition, the basic operations of SNNs only include sums of weights, as spikes values are 0 or 1, they have the potential to use less power than conventional ANNs, which include multiplications of weights times the activation values from the neurons.

The behavior of a single neuron is represented in figure 7. As it can be observed, input spikes ($X(t)$) are passed to the neuron, which increases its internal state ($U(t)$) depending on the weight assigned to the input (W). When the internal state of the neuron crosses the threshold ($U(t) \geq \theta$), the neuron will cause an output spike ($S_{\text{out}}(t)$) and the internal state of the neuron will be reset.

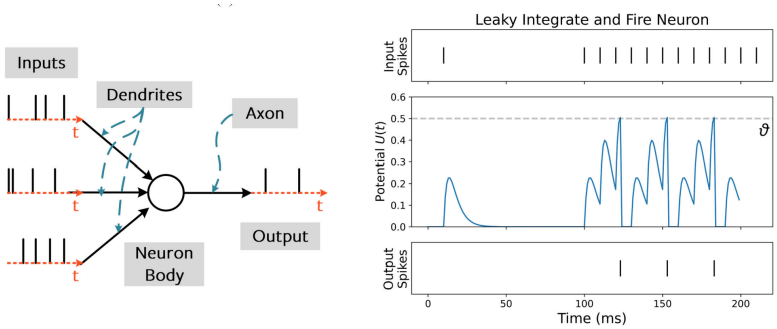

Figure 7: Spiking neuron representation.

Different neuron models can be used to simulate the neuron dynamics, with different reset methods and decaying functions, trading off biological realism against computational cost:

- **Hodgkin-Huxley model**

  This is the most biologically accurate description of neuronal membrane dynamics. It is governed by four coupled non-linear differential equations that model sodium and potassium channel conductances. Due to its extreme computational complexity, it is almost never used in large-scale SNN simulations or hardware implementations.

- **Integrate-and-Fire (IF) model**

  This is one of the simplest and most widely adopted models, which along with its variants (such as Leaky Integrate-and-Fire), is the dominant choice in current SNN research and neuromorphic hardware. The main reason for this is that the IF model is sufficiently expressive for most tasks while it remains computationally tractable and hardware-friendly.

  In the IF model, the membrane potential $U(t)$ behaves like a leaky capacitor, decaying exponentially between spikes and integrating incoming weighted spikes. When $U(t)$ reaches the threshold θ, an output spike is emitted and the potential is reset. The discrete-time formulation used in most training frameworks is:

$$U(t) = \underbrace{\beta U(t-\delta t)}_{\text{decay}} + \underbrace{WX(t)}_{\text{input}} - \underbrace{S_{\text{out}}(t-\delta t)\theta}_{\text{reset}} \tag{4}$$

$$S_{\text{out}}(t) = \begin{cases} 1 & \text{if } U(t) \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

  where $\beta \in [0, 1)$ is the decay factor, and the reset term implements hard reset-to-zero after spiking.

2.4.1 Training a SNN

Training spiking neural networks is an active research area in neuromorphic computing. Unlike ANN's with continuous activations, the non-differentiable nature of spike generation makes direct application of backpropagation impossible, forcing the community to develop alternative learning strategies. The main approaches can be classified as follows:

- **Shadow training (ANN-to-SNN conversion)**

  A conventional ANN is first trained with standard backpropagation. Then it's converted into an equivalent SNN by replacing ReLU activations with integrate-and-fire neurons and carefully mapping weights and thresholds. Explained in section 2.4.2.

- **Direct supervised training of SNNs**

  The network is trained end-to-end as an SNN from the beginning using spike-based signals.

  - **Backpropagation adapted to spikes**
    * **Temporal credit assignment**
      Methods that derive gradients from the precise timing of individual spikes, using the spike time as a continuous variable and applying the chain rule across exact spike events. These approaches are computationally expensive, memory-intensive (requiring storage of all spike times), and rarely used beyond small networks or theoretical studies.
    * **Backpropagation Through Time (BPTT) with surrogate gradients**.
      The non-differentiable spiking threshold is approximated by a smooth surrogate function (sigmoid, piecewise linear, etc.) during the backward pass, allowing standard autograd frameworks to train deep SNNs to state-of-the-art accuracy on many benchmarks. Explained in section 2.4.3.

- **Unsupervised local learning rules**

  Biologically inspired unsupervised or weakly supervised local learning rules in which weight updates depend only on information available locally at each synapse (pre/post spike times, membrane potential, etc.) without requiring error signals to be propagated backwards through the network. A popular approach is Spike-Timing-Dependent Plasticity (STDP), where the weight changes depend on the relative timing between pre- and post-synaptic spikes ("fire together, wire together" principle).

Section 2.4.2 describes the most relevant method for this project, which is shadow training (ANN-to-SNN conversion). In addition, section 2.4.3 describes surrogate-gradient-based backpropagation, which is not used in the project but is presented for understanding.

2.4.2 Shadow Training: ANN-to-SNN Conversion

Shadow training, also known as ANN-to-SNN conversion, is currently the most effective and widely adopted method for obtaining high-accuracy deep SNNs suitable for neuromorphic hardware deployment.

The approach, formalized in [10], consists of two independent phases. First, a conventional ANN (typically with ReLU activations) is trained to high accuracy using standard backpropagation and floating-point arithmetic (FP32). Second, this trained trained ANN is converted to a functionally equivalent SNN, which is done by replacing each ReLU with an IF neuron, which is done by mapping weights and biases so that the average firing rate of each spiking neuron matches the average ReLU activation of the original network.

That is exactly the key idea of rate-based conversion: to make each spiking IF neuron fire, on average, exactly as many times as its corresponding ReLU neuron would have output in the original ANN. For example, a ReLU that outputs 73.5 should become a spiking neuron that fires 73-74 times over the chosen simulation time.

In practice, this is achieved with almost no changes to the parameters:

- Weights W stay exactly the same as in the trained ANN. They are simply interpreted as the strength of incoming spikes instead of continuous inputs.
- Biases b are removed or absorbed into the threshold. Most conversion tools set $b = 0$ and compensate by slightly adjusting the threshold, because IF neurons usually have no bias term.
- Threshold θ of the IF neuron is set to the 99th percentile (or maximum) of the pre-activation values $z = W^\top a + b$ that a neuron actually receives during inference on a calibration set. In other words:

  $$\theta^{(l)} \approx \lambda \cdot P_{99}(z^{(l)})$$

  where $\lambda \leq 1$ is a small safety factor. This guarantees that the neuron fires on average $\approx P_{99}(z^{(l)})$ spikes over the simulation time T, which is very close to the original ReLU output $\text{ReLU}(z^{(l)})$.

Because the weights remain identical and the threshold is chosen to match real activation statistics, the average firing rate of each spiking neuron reproduces the average ReLU activation, layer by layer, with negligible error when enough time steps T are used. Additionally, robust normalization techniques (spike-normalization, data-based scaling) are usually applied after conversion to prevent vanishing or exploding firing rates in deep networks [10].

This is why the converted SNN achieves almost the same accuracy as the original ANN, even though it now runs with binary spikes and integer operations only.

For static images (C, H, W), which contain no temporal dimension, the conversion pipeline first encodes each pixel intensity into a spike train before feeding it to the SNN. Each pixel intensity $I \in [0, I_{\max}]$ is transformed into a Poisson spike train over T time steps with instantaneous firing rate:

$$\rho = \frac{I}{I_{\max}} \cdot f_{\max} \tag{6}$$

where $f_{\max} = 1/\Delta t$ is the maximum possible rate. The probability of emitting a spike in a given discrete time bin is therefore $p = \rho \cdot \Delta t = I/I_{\max}$. In practice, this yields a binomial distribution that converges to the desired mean rate as T increases.

2.4.3 Backpropagation Using Spikes

Although shadow training currently delivers the highest accuracy, direct supervised training of SNNs is an active and highly promising research direction. The main obstacle is the non-differentiable nature of the spiking mechanism: the Heaviside step function that generates a spike has zero gradient almost everywhere, in except of where the spike is generated, where the gradient is infinite. This leads to the so-called dead neuron problem (Figure 8). In addition, during backpropagation, no error signal can flow through a non-firing neuron, preventing

learning. This is also commonly referred as dead neuron problem, but it's not the same as the previous.

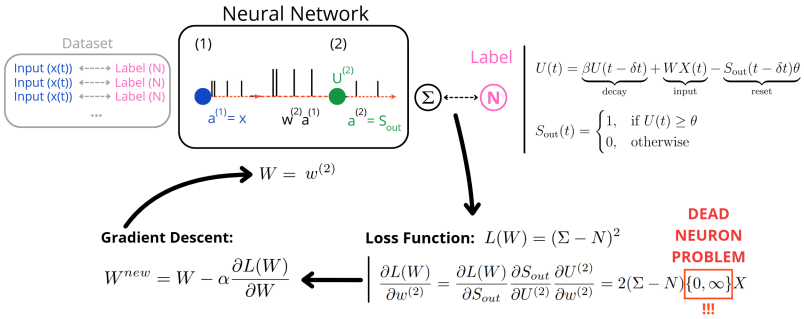

Figure 8: The dead neuron problem: the derivative of the spiking function is zero almost everywhere, blocking gradient flow.

The dominant solution is the surrogate gradient method [11]. During the forward pass, the network behaves exactly as a normal SNN. However, during the backward pass, the non-differentiable spike function is replaced by a smooth, surrogate function like a fast sigmoid, piecewise linear, or boxcar, whose derivative is non-zero in a neighborhood of the threshold (Figure 9). This approximation allows for the computation of gradients, which can be done with existing engines like PyTorch or TensorFlow, while preserving the exact spiking dynamics in the forward direction.

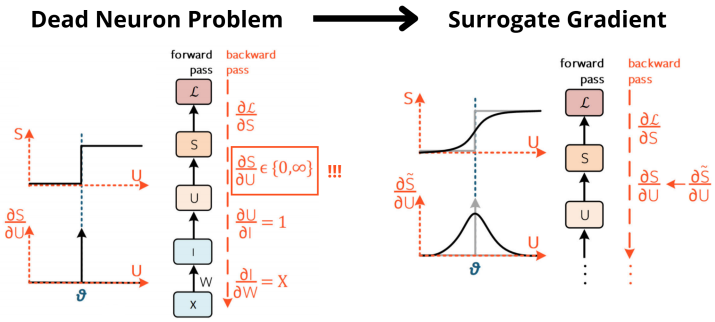

Figure 9: Surrogate gradient approach: the spike function is kept during the forward pass, but replaced by a differentiable surrogate only for backpropagation.

In the previous representations (Figures 8 and 9), the influence of the weight on the loss function has been assumed as a constant relative to time. However, since SNNs are recurrent over time, with the membrane potential depending on past inputs, the influence of a weight on the final loss decays exponentially with the temporal distance between the spike that activated the synapse and the spike that contributed to the loss (Figure 10). In other words, the influence of the weight on the loss function decays exponentially over time, governed by the leak factor $\beta^{(t-s)} \in [0, 1)$. This is why training is typically performed with Backpropagation Through Time (BPTT):

$$\frac{\partial L}{\partial w_{ij}}(t) \propto \beta^{t-s} \tag{7}$$

where s is the time of the presynaptic spike and t is the time of the postsynaptic spike that affects the loss, represented as well in Figure 10.

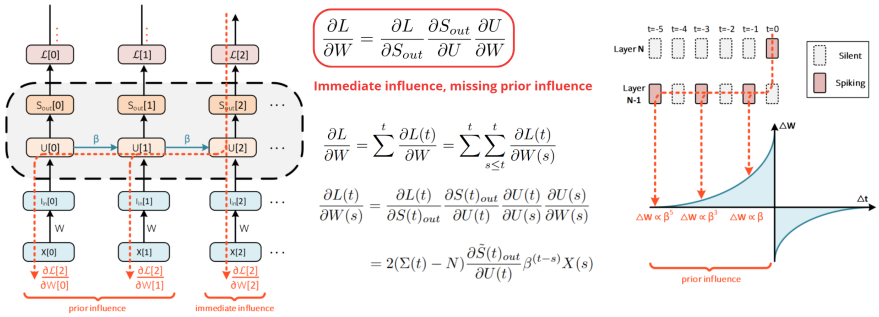

Figure 10: Temporal credit assignment in surrogate-gradient learning: influence of a weight decays exponentially with time (β^{t-s}).

Thanks to surrogate gradients and BPTT, deep SNNs can be trained end-to-end on complex tasks and reach performance competitive with or superior to traditional ANNs while retaining the energy-efficiency benefits of event-driven computation [11].

2.5 Neuromorphic Chips

SNNs can be simulated on conventional von-Neumann processors (CPUs, GPUs), but their true energy-efficiency potential is realized on dedicated neuromorphic chips (NC) specifically designed to implement biological-like neuronal and synaptic dynamics in silicon (Figure 11).

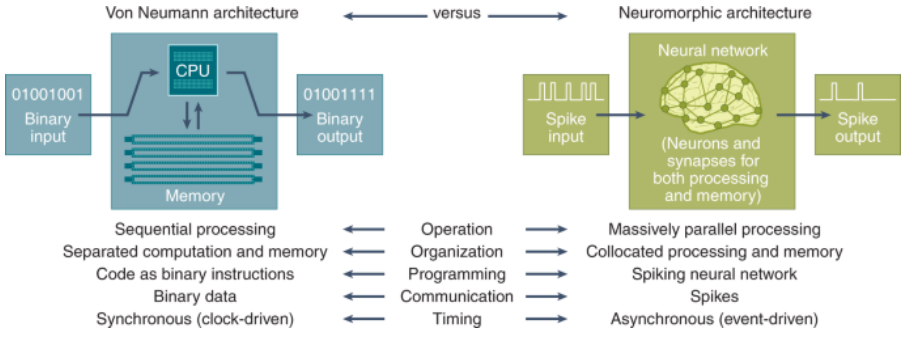

Figure 11: Von-Neumann architecture (left) vs Neuromorphic Architecture (right).

Neuromorphic chips abandon the traditional separation between processor and memory. Instead, each artificial neuron is implemented as a small mixed-signal or fully digital circuit (often

a leaky capacitor-resistor analogue of the biological membrane as represented in [12]) that directly emulates the IF behavior described in Section 2.4 (Figure 12). Thousands to millions of these neuron circuits, together with their synaptic connections, are distributed across the die and communicate asynchronously via binary spikes (address-event representation, AER). Because computation occurs only when spikes arrive, static power is virtually eliminated and dynamic power scales with event rate rather than clock frequency [13].

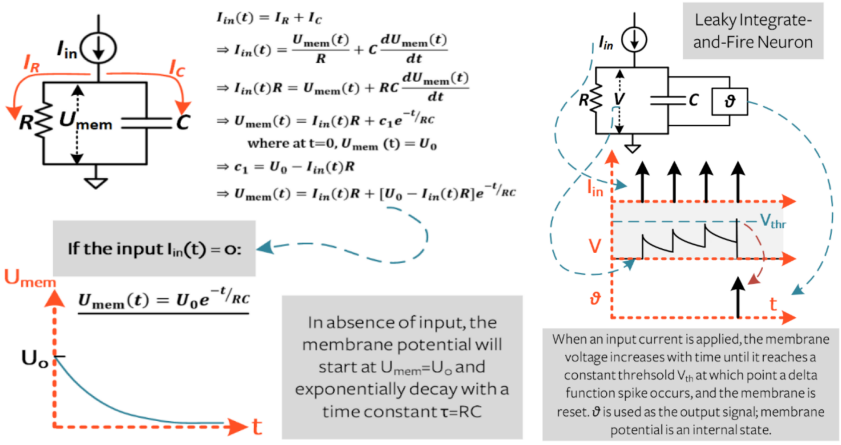

Figure 12: Physical realization of a single LIF neuron in neuromorphic hardware (RC circuit analogy).

At the architectural level, neuromorphic chips are typically organised as arrays of cores (Figure 13). Each core contains a crossbar array where input voltages are applied to rows and output currents from columns represent the weighted sum; the conductance of each memristor (or equivalent synaptic element) at the row–column intersection stores the synaptic weight. Multiple such cores are tiled across the die and interconnected by an event-driven routing fabric (usually an on-chip network-on-chip or address-event representation bus) that forwards spikes only to the cores that need them, eliminating global memory access and enabling massive parallelism with ultra-low power.

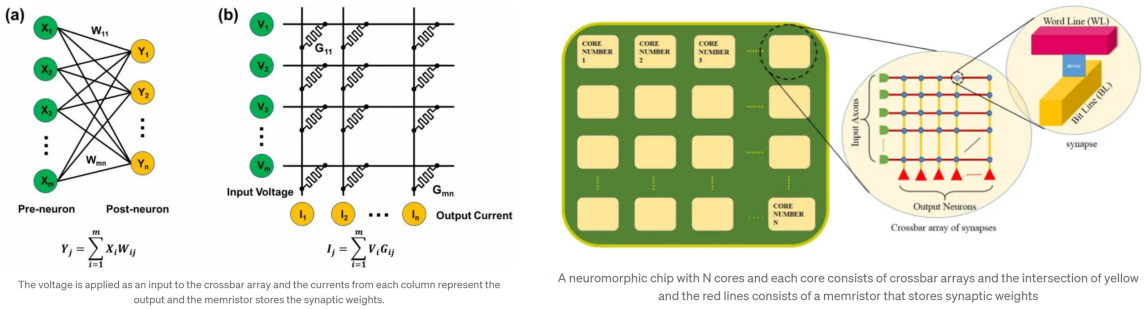

Figure 13: NC architecture: single core with crossbar array (left) and multi-core layout (right).

This event-driven, co-localized design yields improvements in energy efficiency while maintaining real-time performance on sparse, dynamic inputs such as those from event cameras.

However, it also introduces programming complexity and limited software ecosystems currently restrict widespread adoption to specialized applications. A detailed comparison of representative neuromorphic processors (IBM TrueNorth, Intel Loihi, BrainChip Akida, SpiNNaker, etc.) is provided in Appendix A.2.

3 Materials and methods

3.1 Hardware

3.1.1 Prophesee EVK4

The event sensor used throughout the project is the Prophesee EVK4 evaluation kit (Figure 14), based on the Sony IMX636 event-based vision sensor (co-developed by Sony and Prophesee). A self-made comparison of this event-based camera with other alternatives is shown in Appendix A.1.

The sensor provides 1280 × 720 pixel resolution with 4.86 µm × 4.86 µm pixel pitch and pixel-level latency below 100 µs, yielding an equivalent frame rate exceeding 10k fps. Dynamic range exceeds 120 dB, with nominal contrast sensitivity maintained down to a low-light cutoff of 0.08 lux and a virtual high-light limit of 100 000 lux. Power and data are supplied through a single USB 3.0 connection (5 V, no external supply required). Physical dimensions of the EVK4 module are 30 mm × 30 mm × 36 mm.

The camera is operated directly from the host PC, Raspberry Pi 5 (Section 3.1.3) or Jetson AGX Orin (Section 3.1.4), via the USB 3.0 cable using the official Prophesee Metavision SDK (Section 3.2.3).

Figure 14: Event Camera Prophesee EVK4.

3.1.2 Akida Neuromorphic System-on-Chip

The neuromorphic accelerator used in this project is the BrainChip Akida AKD1000, a first-generation production silicon containing 1.2 million neurons organized in 80 neural processing nodes (NPOs). The chip implements Akida IP version 1 and supports event-based, fully INT8-quantised (8-bit weights and activations) SNNs with on-chip learning capability.

The AKD1000 is deployed in three physical formats: M.2 2260 E Key Card, M.2 2260 B+M Key Card, and a PCIe Board (Figure 15), all interfaces exposing the same silicon chip AKD1000. Host-to-Akida communication occurs over the given interface, using the standard BrainChip Linux PCIe driver (Section 3.2.6) and MetaTF runtime (Section 3.2.5), which is included in the Akida SDK (Section 3.2.7). A self-made comparison of multiple neuromorphic processors is shown in Appendix A.2, including a comparison between the available Akida options.

Figure 15: BrainChip Akida AKD1000: M.2 Card E Key (left), M.2 Card B+M Key (centre), PCIe Board (right).

3.1.3 Akida M.2 Card with Raspberry Pi 5

As concluded in Appendix A.4, the low-power deployment platform consists of a Raspberry Pi 5 paired with a BrainChip Akida AKD1000 M.2 2260 B+M Key Card. The board's PCIe Gen 2 ×1 slot is connected to the Card's M.2 B+M Key interface via a PCIe to M.2 M Key adapter. Figure 16 shows the proposed setup from Akida (left image) and the setup used in this project with available components in the market (center and right) .

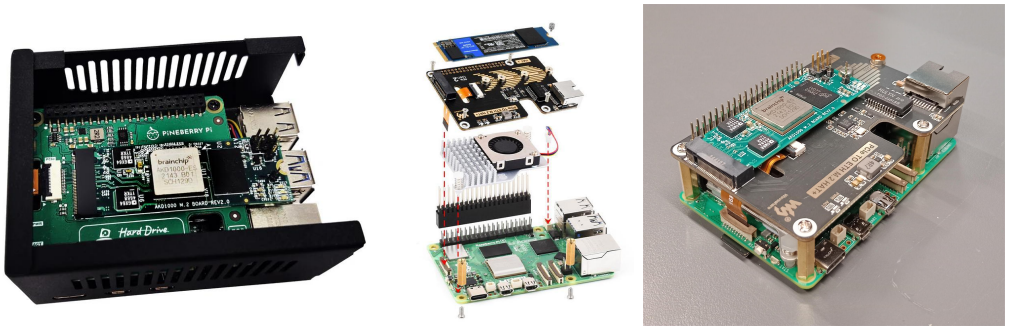

Figure 16: Raspberry Pi 5 with Akida M.2 Card setup: proposed by Akida (left), used in the project with active cooler, PCIe to M.2 M Key Adapter Hat and Akida M.2 B+M Key Card (center), used in this project fully assembled (right).

3.1.4 Akida PCIe Board with NVDIA Jetson AGX Orin

As concluded in Appendix A.4, the high-performance deployment platform uses an NVIDIA Jetson AGX Orin module paired with the BrainChip Akida AKD1000 PCIe Board. The module's full-size PCIe Gen 4 ×16-lane slot is used to connect to the Akida Board's PCIe Gen 2 x1-lane interface, as shown in Figure 17. The Akida PCIe card is powered directly from the PCIe slot; no auxiliary power connector is required.

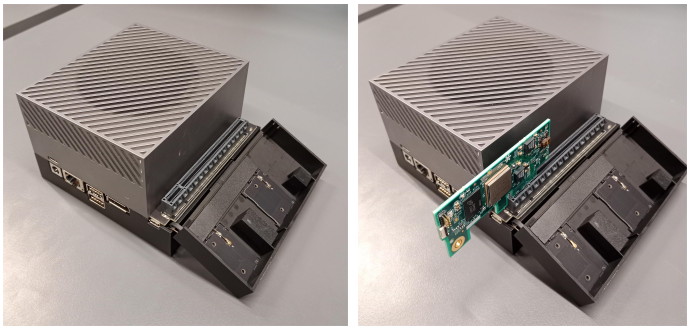

Figure 17: NVIDIA Jetson AGX Orin with Akida AKD1000 PCIe Board: Jetson AGX Orin's PCIe Gen 4 ×16-lane slot (left), and Jetson with installed Akida PCIe Board (right).

3.1.5 Alienware Area 51 with x2 GeForce GPU's

The development and training workstation used throughout the project is a Dell Alienware Area-51 running Ubuntu 24.04.3 LTS with CUDA 12.4. The system is equipped with an AMD Ryzen Threadripper 2950X 16-Core Processor (x86_64 architechture, 32 threads, base 3.5 GHz) and two NVIDIA GeForce RTX 2080 Ti GPUs (11 GB GDDR6 each, 4352 CUDA cores per card).

This workstation has served two critical roles. First, it has been used for flashing and reconfiguring the NVIDIA Jetson AGX Orin with different JetPack versions using NVIDIA SDK Manager (which requires an x86_64 Linux host). Second, it has been used for fast training, quantization-aware training, calibration, and CNN-to-SNN conversion of all models, leveraging multi-GPU acceleration and large batch sizes unavailable on the edge platforms. No Akida hardware has been connected to this machine.

Figure 18: Alienware Area-51 workstation: internal and external views.

3.2 Software

3.2.1 Ubuntu 24.04 on Raspberry Pi 5

The Raspberry Pi 5 runs the official 64-bit Ubuntu 24.04 Desktop image. Installation is performed using Raspberry Pi Imager on a laptop by writing the image directly to a 64 GB or 128 GB microSD card. The full desktop environment is retained to allow direct monitor/keyboard operation during development.

After complete project setup (Ubuntu 24.04 Desktop with system updates, Akida SDK, MetaTF, CPU-only TensorFlow 2.x and all Python dependencies, plus event-based datasets), total storage consumption is approximately 30 GB (5 GB OS, 2 GB software stack, 20 GB datasets). A 64 GB microSD card therefore represents the practical minimum, while 128 GB or larger cards are preferred for additional logging and dataset margin.

3.2.2 Re-flashing and SSD Boot Configuration of NVIDIA Jetson AGX Orin

The NVIDIA Jetson AGX Orin module is re-flashed and configured to boot from an external NVMe SSD instead of the internal eMMC. This is performed due to the limited capabilities of the original eMMC, which has a disk space of 64GB and a throughput of 1.85GB/s, replacing it with a NVMe M.2 SSD with 2TB of storage and a throughput of 8GB/s.

Flashing is performed from the x86_64 host workstation (Alienware Area-51, Section 3.1.5) using NVIDIA SDK Manager. SDK Manager cannot run natively on arm64 architecture, making a external x86_64 machine mandatory. The download and installation of SDK Manager in the Alienware Area-51 is done by following the steps documented in [14], which are also shown in the videos at [15] and [16]. These videos also explain how to physically install the SSD-card into the Jetson module, as well as to perform the re-flashing process with the SDK Manager, which is officially documented in [17].

JetPack 6.1 is the complete software development kit provided by NVIDIA for the Jetson platform. It bundles Ubuntu 22.04 (host and target filesystem), Linux kernel 5.15 with Jetson-specific drivers, CUDA 12.4, cuDNN 9, TensorRT 10, Vision Programming Interface (VPI), OpenCV 4.8, DeepStream 7.0, Isaac ROS 3.0, and additional NVIDIA-accelerated libraries into a single installation package, allowing the entire software stack to be flashed and configured in one operation via SDK Manager.

The complete JetPack 6.1 installation requires approximately 20 GB, the Akida SDK with CUDA-enabled TensorFlow adds 2 GB, and recorded event datasets occupy 20 GB, resulting in a total footprint of 42 GB. The 2 TB NVMe SSD (M.2 2280 M Key, PCIe Gen4) is physically installed via the Jetson carrier board's M.2 Key-M slot (Figure 19). After successful flashing and boot migration, the internal 64 GB eMMC is no longer used, but it's still available for storage.

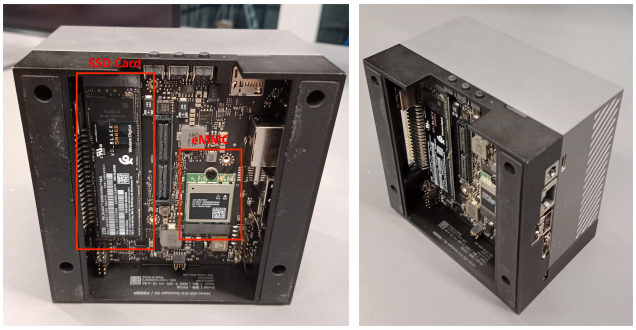

Figure 19: Jetson AGX Orin with installed 2 TB NVMe SSD M2 2260 Card.

3.2.3 Prophesee Metavision SDK

The Prophesee Metavision SDK [18] is the official software stack provided by Prophesee for all its event-based cameras, including the EVK4 used in this project (Section 3.1.1). It offers low-level access to the USB 3.0 event stream (via the C++ HAL), high-level algorithms for decoding, noise filtering, and event-to-frame accumulation, as well as Python bindings and visualization tools. It is used in this project because it is the only supported interface that guarantees full camera bandwidth and sub-millisecond latency.

The SDK has been installed on the Jetson AGX Orin (JetPack 6.1) using the official .deb packages and Python wheels provided by Prophesee. ROS 2 integration, including installation of ROS 2 Humble/Jazzy, metavision_driver, event_camera_py, and event_camera_renderer with all dependencies, is automated using the pandect-setup repository [19]. This setup script also installs the ros-event-camera stack [20] for publishing raw events and accumulated frames on ROS 2 topics.

This can also be done for the Raspberry Pi 5, in order to read event-data in real time.

All custom recording and dataset generation scripts rely exclusively on Metavision SDK. Reference implementations and benchmarks for event-camera software are taken from the community-maintained repository [21].

3.2.4 ROS 2 Components and C++ Nodes for Fast Event Decoding

Real-time processing of Prophesee EVK4 event streams demands extremely low processing latency. At high-contrast scenes the camera generates millions of events per second, delivered in approximately 1 ms packets. Any processing delay longer than the inter-packet interval causes buffer overflow in the Metavision driver and irreversible loss of events. To prevent this, three critical optimizations are implemented, which are implemented in the custom Event Camera package [22]:

- **ROS 2 components.** Instead of separate ROS 2 nodes communicating via DDS serialization/deserialization, all time-critical stages (event decoding, noise filtering, accumu-

lation) are loaded as components inside a single component containcer process. This enables zero-copy intra-process communication, which means passing messages between nodes using shared memory, resulting in reduced per-packet overhead.

- **C++ implementation.** Python-based nodes introduce overhead per packet. This is why the final pipeline is written entirely in C++.
- **Optimized event storage and indexing.** Events can be stored in a dynamic list, as an array with one event (t, x, y, p) on each element, whose length grows as more events are read. In addition, events can be stored in dictionaries that associate each pixel address with its latest timestamp and event count using hash-table lookups. However, dynamic lists and dictionaries are too slow because every event insertion triggers memory allocation or costly hash operations, and the data is scattered in memory, affecting CPU cache performance. Instead, two fixed-size 1280×720 arrays (`event_times_` and `event_counts_`) are permanently allocated in contiguous memory. Each incoming event updates only its corresponding pixel using direct array indexing (x + y × width), avoiding loops and memory allocation. In the visualization callback, a loop over the entire array is performed, but instead of looping over the entire length of the array (1280x720), only the indexes of the activated pixels are used, which are stored in `pixel_indices_`. In this way, pixels whose last event is recent are drawn with intensity proportional to the count, and the counter is reset.

3.2.5 BrainChip MetaTF

The Akida Development Environment MetaTF [23] is a complete machine learning framework that enables the creation, training, and testing of neural networks on the Akida Neuromorphic Processor Platform. It provides a high-level Python API for neural networks, which facilitates early evaluation, design, and fine tuning of neural network models.

MetaTF is comprised of four Python packages which leverage both the TensorFlow (through TF-Keras) and ONNX frameworks, and are installed from the PyPI repository via pip command. The four MetaTF packages contain:

- **Model Zoo (akida-models).** Directly load quantized models or Akida compatible models.
- **Quantization Tool (quantizeml).** Quantization of models using low-bitwidth weights and outputs.
- **Conversion Tool (cnn2snn).** Convert models to a binary format for model execution on an Akida platform.
- **Interface to the Akida Neuromorphic Processor (akida).** Includes a runtime, a Hardware Abstraction Layer (HAL) and a software backend. It allows the simulation of the Akida Neuromorphic Processor and use of the AKD1000 reference SoC.

3.2.6 Akida PCIe/M.2 Driver Installation and Configuration

Installation of the BrainChip Akida PCIe and M.2 drivers proved significantly more complex than the official documentation suggests, especially on the NVIDIA Jetson AGX Orin. For this installation, the tutorial referenced in [24] has been followed, with support from the original documentation referenced in [25,26].

TensorFlow Problems
First, additional complications arise from TensorFlow compatibility, as TensorFlow does not support the GPU from the Raspberry Pi (only supports CUDA) and no official TensorFlow wheels exist for the Jetson module's CUDA 12.6 + cuDNN 9 (JetPack 6.1). Therefore, TensorFlow is installed in CPU-only mode on both platforms.

Linux Headers Problems
Second, the standard procedure for the Akida driver installation (installing linux-headers-$(uname -r) followed by the driver install.sh script) works without issues on the Raspberry Pi 5. However, on the Jetson AGX Orin the Ubuntu package linux-headers-5.15.148-tegra does not exist in the standard repositories, causing the driver compilation to fail. This has been solved by manually downloading and expanding the kernel source, as explained in the official documentation referenced in [27]. This is described in the following steps:

- Skip "sudo apt install linux-headers-$(uname -r)"
- Go to the NVDIA Jetson Linux Archive [28] to Locate the Jetson Linux Version for the Jetson AGX Orin. In this case, Jetson Linux 36.4 is chosen.
- The previous step will open the page referenced in [29]. Scroll down to "DRIVERS" and click on "Driver Package (BSP)", which will download NVIDIA kernel source for Jetpack 6 (R36.4), as a zip folder "public_source.tbz2".
- Extract the downloaded "public_source.tbz2" to usr/src.
- Locate usr/src/Linux_for_Tegra/source/kernel_src.tbz2 and extract it to usr/src.
- Locate usr/src/kernel/kernel-jammy-src with all files inside.
- Follow the next commands on the terminal:
  - "export KDIR=/usr/src/kernel/kernel-jammy-src"
  - "cd /path/to/akida_dw_edma" (cloned from github)
  - "sudo KDIR=$KDIR ./install.sh"

After this was done, the issue was solved and verified by running the Global Akida workflow example from Akida's documentation [30].

ONNX Runtime Problems
Third and last, when following the official event-based eye-tracking example provided by BrainChip [31] (which shows the entire pipeline for obtaining a SNN for event-based detection),

onnxruntime dependency issues have been encountered in the Jetson AGX Orin. These issues occur when trying to export and import models in the quantization step. After been unable to solve them in the Jetson AGX Orin module, work has been proceeded at the x86_64 Alienware workstation, which haven't run into these onnxruntime problems. Later in the project, when trying to inference the SNN with Raspberry Pi 5 as a host, no onnxruntime problems have been encountered neither.

3.2.7 Python Environment and Third-Party Packages for Training Workflow

All neural network training, quantization, and conversion are performed in a Python 3.11 virtual environments. The software stack differs between platforms due to architecture and CUDA compatibility constraints. All the exloratory work is mainly performed in the Alienware workstation, which includes training the tennst model, as well as Akida-specific quantization and conversion steps. The Raspberry Pi is mainly used for inference of the Akida NSoC device, and the Jetson AGX Orin module can potentially be used for the same purpose if dependency issues are fixed.

- **Jetson AGX Orin.** Initially attempted as primary training platform. Although NVIDIA provides ARM64 PyTorch wheels for the Jetson AGX Orin, no compatible version have been found for JetPack 6.1 (CUDA 12.6). Therefore, PyTorch training has been abandoned on the Jetson, proceeding with the Alienware Area-51, which didn't give any problem with PyTorch compatibities.
- **Alienware Area-51 (x86_64).** Primary training platform. PyTorch 2.4.0 + CUDA 12.4 is used for training the Tennst. Full GPU acceleration on the dual RTX 2080 Ti GPUs enables large batch sizes and rapid experimentation. ONNX and onnxruntime-gpu are used for model export/import during intermediate validation steps. The BrainChip MetaTF suite (akida, akida-models, quantizeml, cnn2snn) is installed via pip from the private BrainChip PyPI index.
  - **Akida/MetaTF stack.** TensorFlow 2.19.1 (CPU-only), akida 2.17.0, akida_models 1.11.1, quantizeml 1.0.1, cnn2snn 2.17.0. These versions are the latest officially supported MetaTF releases at the time of writing and are installed from BrainChip's private PyPI index.
  - **Model interoperability.** ONNX 1.19.1, onnxruntime 1.19.2, onnxruntime_extensions 0.13.0. Used for intermediate model export/import and validation between PyTorch and TensorFlow workflows.
  - **Data handling and visualization.** numpy 2.1.3, opencv-python 4.12.0.88, matplotlib 3.10.7, h5py 3.15.1, tensorflow-datasets 4.9.9, tqdm 4.67.1, imagecorruptions-imaug 1.1.3, scikit-image 0.25.2.
  - **Standard scientific and utility packages.** scipy, pandas, protobuf 5.29.5, pillow 11.3.0, torchinfo 1.8.0, einops 0.8.1, and others listed in the full environment dump.

- **Raspberry Pi 5 and Jetson AGX Orin.** Akida NSoC device inference. TensorFlow (CPU-only) is installed, as mentioned in Section 3.2.6. The BrainChip MetaTF suite (akida, akida-models, quantizeml, cnn2snn) is installed via pip from the private BrainChip PyPI index.

3.3 Spatio Temporal Event Filter

A custom spatio-temporal filter is applied to the raw event stream from the Prophesee EVK4 to remove isolated noise events while preserving genuine activity. For each incoming event at timestamp t, position (x, y), and polarity p, the filter examines a local 3D neighborhood defined by ± 1 pixel in both x and y directions and a temporal window of 50 ms into the past ($\Delta t = 50\,000\,000$ ns).

The filter maintains two persistent arrays: `event_times_`‘ (last timestamp of activity at each pixel) and `event_counts_` (cumulative count of recent events at each pixel within Δt). For the current event, the neighborhood count is computed by summing `event_counts_` over the 3×3 spatial box (9 pixels). If this sum exceeds a threshold of 4 events, the event is considered correlated with recent activity and is retained; otherwise, it is discarded as noise. Retained events are stored in thread-safe buffers (`pixel_indices_` and `polarity_buffer_`) for subsequent accumulation and visualization. To prevent memory growth, an `active_indices_` list tracks recently updated pixels, and entries older than Δt are periodically cleared, resetting their timestamps and counts to zero. See Appendix B for full implementation details in the code.

3.4 Akida's Workflow to Obtain a SNN

3.4.1 General Overview

Brainchip Akida's documentation [23] introduces a user-guide, examples and API references that are useful to understand how Akida works and how a SNN is obtained in the Akida environment. Among the examples, the first one explains how the BrainChip Akida workflow [30] transforms a conventional floating-point CNN into an event-driven SNN that can be executed on the Akida NSoC. The process consists of three mandatory phases preceded by data-specific preprocessing:

1. **Training of a standard CNN (FP32)**. A regular Keras model is trained on the target task using standard supervised learning. The only Akida-specific constraints imposed at this stage are the layer types, which are restricted by the Akida device's IP version to be used. The layer types available for each version are specified in [32] for version 1 and in[33] for version 2. This correct match is required later in the process for mapping the model into the NSoC.
2. **Quantization to low-precision integers (8-bit or 4-bit)**. Quantization is a critical step in the workflow because the Akida neuromorphic processor is hardware-optimized for

low-precision integer operations (1-8 bits for weights/activations), not for high-precision floating-point (FP32) used in standard CNNs. The quantization is performed using the `quantizeml` toolbox on a calibrated dataset of representative samples (typically 500–2000 event frames or images). These calibration dataset can be self defined, taking samples from the training data, or it can also be skipped, as the toolbox will handle it automatically with random samples.

In the one hand, Post-training quantization (PTQ) will stop here and use the quantized model for the next phase. In the other hand, Quantization-Aware Training (QAT) will retrain and fine-tune the quantized model, using the training set with a low learning rate (typically 1e-5 to 1e-6) for a few epochs to recover accuracy lost during quantization.

3. **Conversion to Akida SNN**. The quantized model is transformed into an Akida-native binary representation (.fbz) using `cnn2snn.convert()`. During conversion:
   - Standard ReLU activations are replaced with spiking neurons (threshold-based integrate-and-fire).
   - Weights and activations are kept in 1-, 2-, 4-, or 8-bit integer format.
   - The resulting network becomes temporally-aware and event-driven: neurons emit binary spikes only when their membrane potential crosses the threshold.
4. **Mapping to Akida Device**. Finally, the converted model is loaded with `akida.Model()` and mapped to a physical (or virtual) device via `model.map(device)`, allocating layers to the available neural processing nodes.

   Akida contains 80 Neural Processing Nodes (NPNs), each composed of 8 Neural Processing Engines (NPEs) and local SRAM for weights and spikes. These nodes are interconnected by an event-driven mesh network. Mapping consists of the following operations performed automatically by the Akida runtime:
   - Each layer of the SNN is assigned to one or more NPNs according to its computational requirements and the available device resources.
   - Weights are distributed across the device's SRAM of the allocated nodes; only non-zero weights are stored, exploiting sparsity.
   - Inter-layer spike routing paths are configured so that output spikes from one node are delivered only to the nodes containing the downstream neurons that need them.
   - The mapper verifies compatibility with the device's IP version (v1 in the AKD1000, v2 in newer devices). Unsupported operations cause a mapping failure.

Once mapping succeeds, the model is fully resident on the device: no further host-side computation is required except for feeding accumulated event frames.

Although Akida implements true integrate-and-fire spiking neurons, the first-generation AKD1000 (IP version 1) operates in a frame-synchronous manner rather than being fully event-by-event asynchronous. This design choice ensures perfect compatibility with CNN-to-SNN conversion tools: the network still expects fixed-size input tensors, and inference is executed in discrete

time steps. Consequently, raw asynchronous event streams must be accumulated into short time windows (20-100 ms windows) before being fed to the network.

The device then executes the network in a truly event-driven manner, nodes remaining idle until they receive spikes, but the device processes the entire accumulated frame in a single burst. This hybrid approach preserves the ultra-low power advantage while allowing the use of standard deep-learning toolchains and converted CNN architectures.

3.4.2 Specific Workflow for Event-based Detection

The practical implementation follows the official BrainChip event-based eye-tracking example [31], which demonstrates a complete pipeline from raw Prophesee events to pupil center prediction using a converted SNN. This example is the only publicly available end-to-end event-based vision workflow for Akida. Therefore, it has been adopted as the primary reference. It's explained in the following steps:

1. **Training of the Spatio-Temporal CNN with Tennst backbone**.

   Network Input

   The Tennst network receives a 5-D input tensor of shape (B, C=2, T=50, H=96, W=128), where the two channels separate positive and negative polarity events, T=50 corresponds to temporal frames of accumulated events at 100 Hz (10 ms each frame), and the spatial resolution (H, W) is downsampled from the original 480×640 event resolution.

   Network Output

   The network outputs a 5-D tensor (B, C=3, T=50, H=3, W=4), where the three channels contain per-grid-cell confidence score, x-offset and y-offset, all of them being normalized values between 0 and 1. In this way, pupil center coordinates are reconstructed in the final 60×80 prediction space via weighted averaging over the coarse 3×4 grid, as shown in Figure 20.

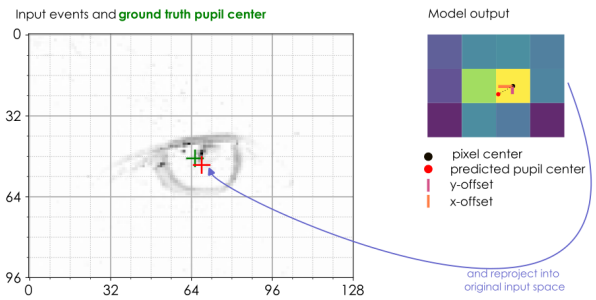

Figure 20: Model Output and Pixel Reconstruction for the Tensst in Akida's Eye-tracking Example.

Network Architecture

The proposed architecture is a stack of spatiotemporal convolutional blocks, each consisting of a temporal convolution followed by a spatial convolution (see Figure 21). These are designed to extract both fine-grained temporal features and local spatial structure from event-based input tensors.

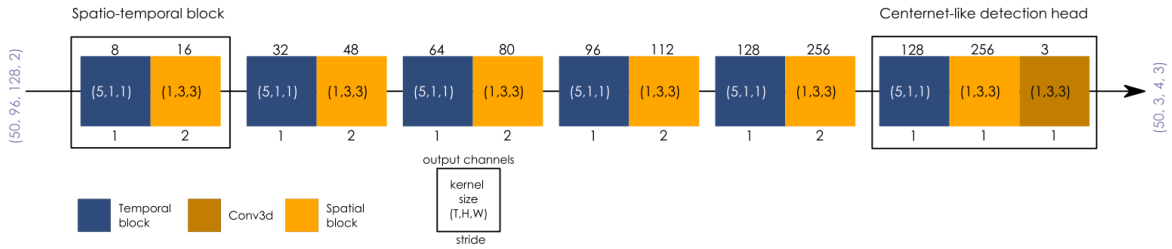

Figure 21: Proposed Network Architecture for the Tensst in Akida's Eye-tracking Example.

The key design feature for the network are explained below:

- **Causal Temporal Convolutions.** Output at time t depends only on input at time $\leq$ t.
- **Factorized 3D Convolution Scheme.** The spatiotemporal blocks perform temporal convolutions first, followed by spatial convolutions, as shown in Figure 22. Decomposing the 3D convolutions into temporal and spatial layers greatly reduces computation.
- **Depthwise-Separable Convolutions (DWS).** Both temporal 1D and spatial 2D convolution layers can optionally be configured as depthwise-separable to further reduce computation with minimal loss in accuracy.
- **No Residual Connections.** To conserve memory and simplify deployment on edge devices, residual connections are omitted. Since the model has a reduced number of layers, they are not critical to achieve SOTA performance.
- **Detection Head.** A lightweight head, inspired by [34], predicts a confidence score and local spatial offsets for the pupil position over the coarse spatial grid. The predicted position of the pupil can then be reconstructed.

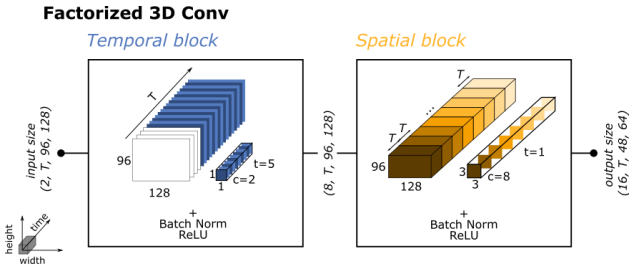

Figure 22: Factorized 3D Convolution Scheme Visual Representation.

Dataset Description

The original dataset [35] is formed by raw events in 480x640 resolution in the form of 4D-tuples (timestamp (μs), x, y, polarity (0 positive, 1 negative)), with ground truth labels in the form of (x-center, y-center, state (0-open, 1-closed)). These are labeled using accumulated event frames at 100Hz. The dataset comes divided in two splits, one for training and the other one for testing. Each of the splits is divided into different recording folders named as "X_Y", each representing a different person (X) being recorded multiple times (Y is the recording number for person X). Inside each folder, raw events are collected in a H5 formated file (named "X_Y.h5"), while the labels for those events come in a TXT file (named "labels.txt"). The dataset takes a total of 1GB of disk storage, with around 207M raw events distributed in 51 different recording folders. There are 42301 total labeled frames, with an average of 4893.6 events/frame. 39802 frames (94.09%) are labeled as open and 2499 (5.91%) as closed.

Data Pre-processing

In the pre-processing stage, these raw events are converted into tensors of shape (C=2, T=50, H=96, W=128), using causal event volume binning, which computes a decaying event volume relative to time, as shown in Figure 23.

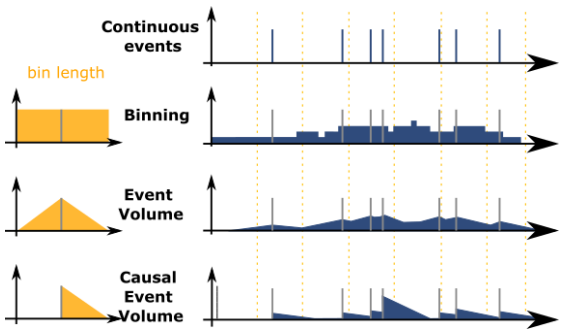

Figure 23: Event Binning Visual Representation.

In addition, for events inside the train split, closed eye labeled frames are ignored and data augmentation is applied. This data augmentation includes random scaling, rotation, translation, temporal scaling, and temporal flip (with polarity inversion).

Training details

The official example loads a pre-trained Tennst model, without showing further details about the training process or about data pre-processing. The only details mentioned are some hyperparameters used during training:

- Batch size of 32
- 50 event frames per segment (T)
- 200 epochs
- AdamW optimizer with base learning rate (LR) of 0.002 and weight decay of 0.005
- LR scheduler with linear warm up (for 2.5% of total epochs) and a cosine decay

A Github repository is references in the example [36], and even if they don't train a Tennst model, it shows some more details about the training process.

2. **Quantization to 8-bit integers**.

   The Tennst model is then quantized to INT8 using quantizeml with 1024 representative event samples as calibration data. In the example, the calibration data is imported, without showing how it has been constructed. In addition, no quantization-aware training (QAT) is performed in the official example.

   A critical side-effect of quantization is the collapse of the temporal dimension: the quantized model now expects 4-D inputs (B, C, H, W). However, no clear reason is provided for this. Each of the original T=50 time slices must therefore be processed sequentially as an independent frame. After feeding the network with a complete sample (50 temporal frames, which correspond to one sample) the internal FIFO buffers of the temporal convolutions are explicitly reset using quantizeml.tools.reset_buffers() to avoid carry-over effects. In addition, the event cooridnate reconstruction function from the networks predictions needs to be modified to use numpy functions instead of torch.

3. **Conversion to Akida format and hardware mapping**.

   The quantized model is converted with cnn2snn.convert(). The IP version of the converted model is version 2, as several Tennst layers are only supported in Akida IP version 2 [33].

   The resulting .fbz model can be executed correctly on a virtual IPv2 device (like SixNodesIPv2()). However, mapping to the physical AKD1000 NSoC fails, due to the IP version mismatch between the model (IP version 2) and the physical AKD1000 device connected to the Raspberry Pi (IP version 1). The alternative solution for this is to redesign and retrain the entire model using exclusively version 1 compatible layers [32] from the start, and skipping the Tennst use din the official example.

3.5 Proposed Workflow to Obtain a SNN

As mentioned in Section 3.2.7, the primary training platform is the Alienware Area-51 (x86_64 architecture and dual GPU GeForce RTX 2080 Ti). This is the workstation that is used for dataset pre-processing (Section 3.5.1), model definition (Section 3.5.2) and training, quantization (Section 3.5.3), and conversion to SNN and mapping (Section 3.5.4), which will be explained on this section.

In the other hand, the Raspberry Pi 5 with the Akida AKD1000 M.2 Card will be used for the last task, mapping the converted SNN into a device, explained in section 3.5.4.

3.5.1 Dataset Pre-Processing

As mentioned in section 3.4.2, the raw events and labels from the dataset need to be transformed into tensors with the input and output shape of the network. For this task, raw events and

labels related to closed eyes are excluded for training, but kept for testing, as the model will not predict the state of eye (open or closed). However, for simplifying the work, this project will not include the causal event binning, spatial transforms (rotations, scaling and translations), neither temporal augmentations (time scaling, and time flipping with polarity inversion), which are explained in Section 3.4.2. In this way, the data pre-processing step is simplified for this project.

3.5.2 Proposed Model Architectures

This section explains the architecture of the CNN models that are trained and compared in this project, which are the **Simplified Tensst** and the **Akida Example's Replicated Tensst**. For faster training, the next setting are used during the training of all proposed models:

- `torch.compile(model)` applies TorchInductor graph-level optimizations such as kernel fusion, memory planning, and CUDA graph capture. This typically achieves 30–60% speedup on GPUs by reducing Python overhead and kernel launch latency.
- Mixed-precision training with `torch.cuda.amp` performs most operations in FP16 while maintaining master weights in FP32. This saves memory usage and doubles arithmetic throughput on Tensor Cores without measurable accuracy degradation.
- DataLoader with `num_workers=12`, `persistent_workers=True`, and `prefetch_factor=2` – parallelizes event volume generation and augmentation across 12 CPU cores, keeps workers alive between epochs (avoiding repeated process spawning), and pre-fetches two mini-batches per worker to fully hide I/O latency behind GPU computation.
- Multi-GPU training using `torch.nn.DataParallel` distributes each batch across both RTX 2080 Ti GPUs, effectively doubling throughput and enabling larger batch sizes for faster convergence.

Simplified Tensst

This model is proposed with the objective of simplifying all the complexity from the original Tennst used in Akida's example for event-based detection. For this purpose, more intuitive inputs and output are used, which are described below.

- **Input shape:** (B, T=10, H=480, W=640) uint8
- **Output shape:** (B, 2) float32: x and y eye center coordinates.
- **Model architecture:** same as the one described in Section 3.4.2, but with the next adaptations:
  - Input layers adapter to simplifies input shape (B, T=10, H=480, W=640).

- Fully connected linear layers that progressively scale down from the output of the 3D Convolution to the input of the head.
- Output head, with the described output shape.

Akida Example's Replicated Tensst

The same architecture, input and output shapes of the original event-based eye-tracking example from Akida, described in Section 3.4.2.

3.5.3 Proposed Quantization

For this project, quantization to INT-8 and INT-4 are originally proposed, comparing default calibration and calibration providing samples from the train dataset.

As commented in Section 3.4.2, quantization results in the loss of temporal dimension (T=50), which will be considered with the previous calibration samples, as well as when inferring the quantized models obtained. It's also taken into account that after going through all the T=50 in one batch, buffer (FIFO) layers need to be reset.

Finally, due to its simplicity, lower computational requirements and limited scope of the project, only post-training quantization (PTQ) is performed; quantization-aware training (QAT) is not applied to the quantized models.

3.5.4 Proposed Conversion to SNN and Mapping to Akida Devices

The quantized models are converted to Akida SNN format using `cnn2snn.convert()`. At the outset of this stage, mapping to the physical AKD1000 device connected to the Raspberry Pi 5 was planned. However, during execution it was discovered that the available AKD1000 implements only Akida IP version 1, whereas the converted models require IP version 2 features present in the Tennst architecture (Section 4.4).

Consequently, none of the converted models could be mapped to the physical NSoC. Instead, all mapping and subsequent simulated inference are performed on a virtual IPv2 device (`akida.SixNodesIPv2()`) executed in software on the Alienware workstation. This virtual device fully supports IP version 2 layers and allows functional validation of the converted SNNs, but without the power and latency benefits of the real neuromorphic hardware.

4 Results and Discussion

4.1 Event reading from event camera

Event data acquisition from the Prophesee EVK4 was successfully implemented using the Metavision SDK on the Jetson AGX Orin platform. The custom ROS 2 pipeline enabled reliable real-time reading of raw events, with no packet loss observed during extended operation.

The spatio-temporal filter (Section 3.4) was applied to suppress isolated noise events. Visual inspection (Figure 24 and supplementary video [37]) confirmed effective noise reduction: background activity was significantly attenuated while genuine motion remained clearly visible. However, the filter introduced noticeable latency in the visualization stream, indicating that the current C++ implementation, although functionally correct, requires further optimization for truly real-time performance at high event rates.

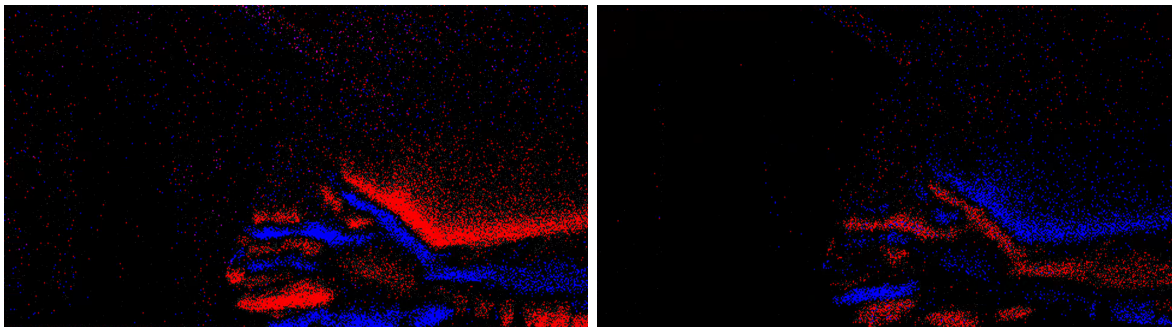

Figure 24: Event visualization without filter (left) and with spatio temporal filter (right).

Quantitative metrics such as noise reduction percentage or filtered event rate could not be reported due to the inability to reproduce the full experimental setup at the time of writing. Nonetheless, the qualitative results validate robust integration between the Metavision SDK and the ROS 2 pipeline, providing a solid foundation for dataset generation and model training.

4.2 Define and Train a CNN

Two CNN architectures, as proposed in section 3.5, are defined, trained from scratch, and evaluated on the event-based pupil center detection task to assess their suitability for quantization and conversion to SNN. Performance is quantified by the following metrics:

- Network size (million parameters).
- Preprocessed dataset characteristics: saved tensor shapes and total used storage (GB).
- Training configuration: batch size, number of epochs, optimizer, learning-rate scheduler, loss function, total training time, and convergence status.

Detailed results for the models **Simplified Tennst** and **Akida Example's Replicated Tennst**, are presented respectively in sections 4.2.1 and 4.2.2.

4.2.1 Simplified Tennst

This model contains approximately **6.7 million trainable parameters**. Multiple preprocessing strategies were explored to generate suitable input tensors for the model, balancing storage requirements, data fidelity, and training stability. Table 3 shows the evaluated dataset formats, their tensor shapes, and resulting storage sizes. The labels used in all the datasets use the same [N,4] structure, where N is the number of labels on a recording, and the 4 coordinates are: timestamp (μs), x and y coordinates of the center, and state of the eye (0 open and 1 closed).

#	Voxel		Label		Size (GB)
	Format	Shape	Format	Shape	
1	Float32	[N, T=10, H=480, W=640]	Float32	[N, 4]	472
2	Float32	[N, H=480, W=640]	Float32	[N, 4]	47.2
3	Uint8	[N, H=480, W=640]	Float32	[N, 4]	11.8
4	Float16	[N, T=10, H=480, W=640]	Float32	[N, 4]	236

Table 3: Preprocessed dataset variants for the Simplified Tennst model

The first dataset stored T=10 event frames per label (10 ms bins) in Float32 format, resulting in approximately 472 GB of storage. To reduce disk usage, the second attempt collapsed the temporal dimension to a single time bin per label (T=1), decreasing storage to 47.2 GB. The third attempt further reduced storage by a factor of 4 (to 11.8 GB) by changing the voxel datatype from Float32 to Uint8 while maintaining the collapsed temporal shape. See a visual representation of the stored event frames per label (T=1 and T=10) in Figure 25.

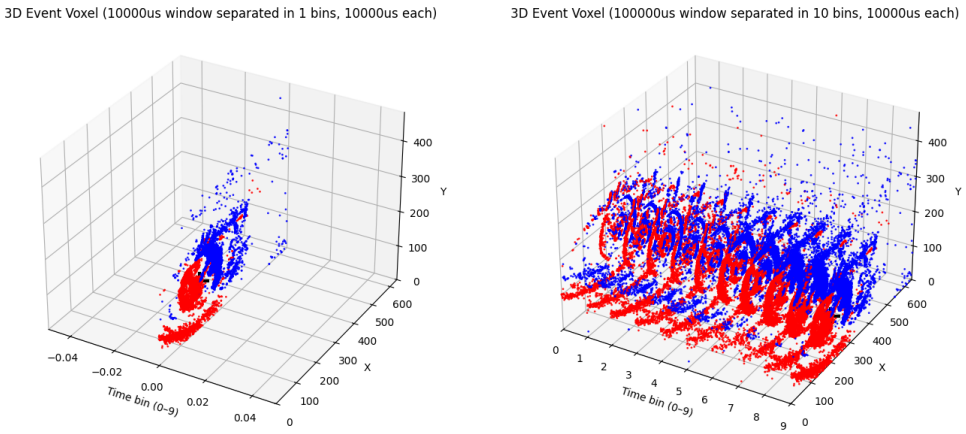

Figure 25: Visualization of stored event frames per label with T=1 (left) and T=10 (right).

Although the collapsed formats significantly lowered disk storage, they introduced a trade-off: to provide the models with the required T=10 input during training, the 10 time bin voxels had to be reconstructed on-the-fly from the single stored bin voxels, by taking the last 10 bins. This reconstruction increases RAM usage during data loading while training. For this reason, the fourth and final dataset returned to storing the full T=10 bins explicitly in Float16, accepting 236 GB of storage to eliminate on-the-fly reconstruction, reduce RAM pressure, and simplify the training pipeline.

Training was performed using dataset variants #3 (11.8 GB) and #4 (236 GB). Hyperparameters were kept consistent: AdamW optimizer with learning rate 0.004 and weight decay 0.005, batch size of 8 (limited by RAM usage with training with the #3 dataset, as more RAM was available with the #4 dataset), mean squared error (MSE) loss on predicted coordinates, and OneCycle learning-rate scheduler with linear warm up (for 2.5% of total epochs) and a cosine decay.

Training was initially tested for both datasets, but training with dataset #3 required much longer, with ≈ **29 minutes / epoch**, comparing to the ≈ **12 minutes / epoch** with dataset #4. For this reason, training with dataset #3 was discarded, showing the results exclusively for dataset #4. In this was, training results are summarized in Table 4 and Figure 26.

Dataset	Epochs	Time/Epoch	Total Time	Train MSE	Val. MSE
#4	95	≈ 12 minutes	19 hours	≈ 1865 px	≈ 3078 px

Table 4: Training results for the Simplified Tennst model

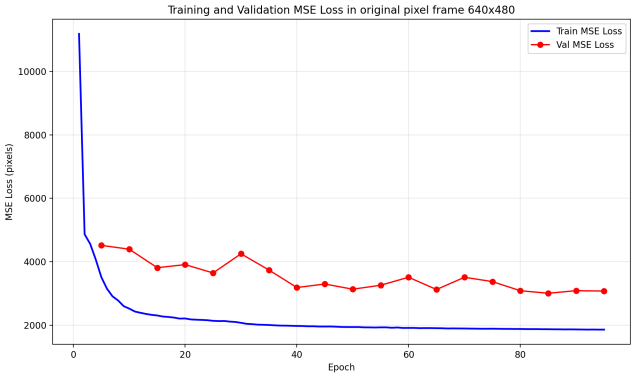

Figure 26: Training (blue) and validation (red) MSE loss for Simplified Tennst during training with dataset #4 (236GB).

Despite extended training and hyperparameter tuning, training resulted in very high converged values, with a **training MSE of ≈ 1865 pixels** and a **validation MSE of ≈ 3078 pixels**, indicating that the simplified architecture was insufficiently expressive for the regression-based pupil center prediction task on this challenging event dataset. This motivated replication of the official Akida example's Tennst architecture, which results are presented in the following section 4.2.2.

4.2.2 Akida Example's Replicated Tennst

This model contains approximately **1 million trainable parameters**. Table 5 shows the evaluated dataset formats, tensor shapes, and resulting storage size, which are done following the example and making the simplifications mentioned in section 3.5.1.

Voxel Format & Shape		Label Format & Shape		Size (GB)
Uint8	[N, C=2, T=50, H=96, W=128]	Float32	[N, C=3, T=50, H=3, W=4]	38.9

Table 5: Preprocessed dataset for the Akida Example's Replica Tennst model

Training was performed using the same hyperparameters: AdamW optimizer with learning rate 0.002 and weight decay 0.005, batch size of 128 (bigger batch size possible due to smaller model size), and OneCycle learning-rate scheduler with linear warm up (for 2.5% of total epochs) and a cosine decay. The used loss function for training is a combination of the cross-entropy (CE) loss in the confidence channel, and the mean squared error (MSE) loss on predicted reconstructed coordinates in original pixels, given by the formula $\text{loss} = \text{loss}_{CE} + 12\text{loss}_{MSE}$. Training results are summarized in Table 6 and Figure 27.

Epochs	Time/Epoch	Total Time	Train MSE	Val. MSE
70	2.66 minutes	3.11 hours	$\approx$ 1.5 px	$\approx$ 36 px

Table 6: Training results for the Akida Example's Replicated Tennst model

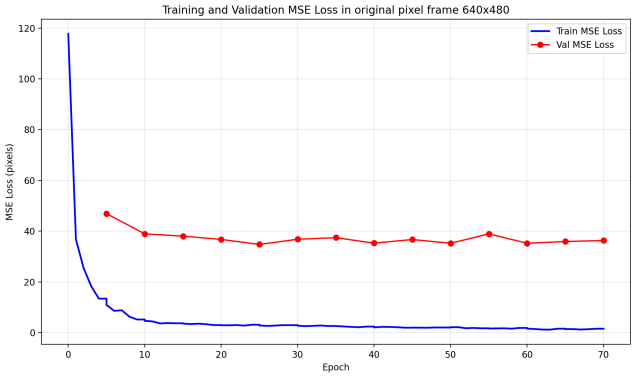

Figure 27: Training (blue) and validation (red) MSE loss for Akida Example's Replicated Tennst during training.

Results show that training converged at MSE loss on predicted reconstructed coordinates in original pixels of **$\approx$ 1.5 px in training data** and **$\approx$ 36 px in validation data**. The best model achieved an average MSE loss of **34.763 pixels** on the validation dataset. These is not an optimal result, as 34 pixels is not very accurate, but for this project is considered as a satisfactory result, puting focus on achieving a workflow to obtain the final SNN. See Figure 28

for a visual representation of the reconstructed eye-center pixel prediction on a single sample for the previous Simplified Tennst (sectione 4.2.1) and the current Akida Example's Replicated Tennst.

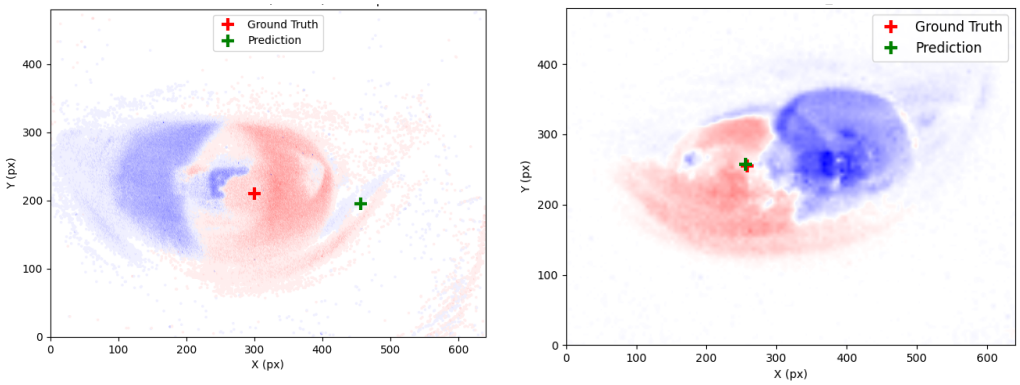

Figure 28: Eye-center prediction for Simplified Tennst (left) and Akida Example's Replicated Tennst (right), with sample MSE errors of 157.41 px and 2.53 px, respectively.

4.3 Quantization

As proposed in Section 3.5.3, quantization to both INT-8 and INT-4 was originally intended. However, INT-4 quantization was discarded after discovering that the `quantize()` function from the `quantizeml` package does not support 4-bit quantization for the replicated Akida Tennst model. Consequently, all efforts focused on INT-8 quantization of the original Akida Example's Replicated Tennst model, comparing the results obtained with default calibration against those achieved when explicitly providing calibration samples.

A critical side-effect observed during quantization is the collapse of the temporal dimension: the original floating-point model accepts 5D inputs of shape [B, C=2, T=50, H=96, W=128], but the quantized model produced by `quantizeml` expects 4D inputs [B, C, H, W]. The temporal axis is therefore removed without explicit explanation in the tool. This forces inference to be performed frame-by-frame (processing each of the 50 time slices independently), and the internal FIFO buffers of the temporal convolutions must be manually reset after each full sequence using `quantizeml.models.reset_buffers()` to prevent state carry-over between samples.

Table 7 shows the quantization results, comparing model size, accuracy, and latency. Accuracy is reported as mean squared error (MSE) in pixels on the original 480×640 frame, while latency is split into model inference time and total time (including pupil coordinate reconstruction) for a single input of shape [B=1, C=2, T=50, H=96, W=128]. These metrics have been computed in the test. These metrics have been recorded in the first 200 batches (with batch size 8) of the test dataset, which justifies the difference in accuracy with the 34.763 px provided in section 4.2.2, which uses the full dataset and a batch size of 128.

Calibration samples were provided using a batch size of 8, a maximum of 10 batches (4000 total samples, considering 50 temporal slices per sample), and 1 epoch.

Model	Size (MB)	MSE (px)	Latency (ms)	
			Model	Total
Akida Example Tensst (FP32)	12.3	32.295	436.59	436.62
Q-Int8 Default Calibration	1.1	40.064	3635.13	3637.93
Q-Int8 With Calibration Samples	1.1	44.467	3608.62	3611.28

Table 7: Quantization results for the Akida Example's Replicated Tennst model, recorded on the first 200 batches of size 8 of the test dataset.

The results confirm that explicit calibration with given sample does not necessarily improve the accuracy of the quantized model, does not affect the model size, but reduces the latency around 26.6 ms.

A striking result from quantization is the 8× bigger inference latency (from ≈436 ms to ≈3635 ms for a full sequence of 50 time slices). This is probably related to the quantized model being forced to process each time slice independently, as a result of collapsing of the temporal dimension from 5D [B, C, T=50, H, W] to 4D [B, C, H, W]. Additionally, the quantized model requires conversion of inputs from PyTorch tensors to NumPy arrays before each inference call, introducing an extra overhead.

Finally, it can be observed that the pixel reconstruction from the model's output takes a constant value of around 2.6 ms for the quantized models and 0.03 ms for the original model, which is not relevant comparing to the absolute latency values.

4.4 Convertion to SNN and Mapping to Akida Devices

The INT-8 quantized model with explicit calibration (Section 4.3) was successfully converted to an Akida-compatible SNN using `cnn2snn.convert()`. The resulting model retains the same architecture as the original Akida Example's Replicated Tennst but replaces ReLU activations with IF neurons and uses 8-bit integer weights and activations.

Mapping to the physical AKD1000 device connected to the Raspberry Pi 5 failed due to an IP version incompatibility. The AKD1000 physical device implements only Akida IP version 1 layers, which lacks support for the layer configuration of the tennst architecture, which uses Akida IP version 2 layers. As a result, the converted model could not be deployed on the real neuromorphic hardware.

Table 8 lists the layers supported by each Akida IP version. IP version 1 provides a limited set of basic convolutional and fully-connected layers suitable for early CNN-to-SNN conversion workflows. IP version 2 significantly extends the capabilities by introducing depthwise and transpose convolutions for more efficient separable designs, dedicated temporal buffering layers (`BufferTempConv` and `DepthwiseBufferTempConv`) to support causal spatio-temporal processing, and utility layers such as `Add` and `Concatenate` for residual and multi-branch architectures. These additions enable conversion of modern, high-performance models (including those with temporal dynamics like the tennst backbone) while maintaining compatibility with

the underlying hardware constraints. For detailed specifications of each layer, refer to the official documentation in [32] for IP version 1 and [33] for IP version 2.

IP Version 1 Layers	IP Version 2 Layers
InputData	InputData
InputConvolutional	InputConv2D
FullyConnected	Conv2D
Convolutional	Conv2DTranspose
SeparableConvolutional	Dense1D
	DepthwiseConv2D
	DepthwiseConv2DTranspose
	BufferTempConv
	DepthwiseBufferTempConv
	Add
	Concatenate
	Dequantizer

Table 8: Available layers for each Akida IP version

The converted SNN model consists of 19 layers with an input shape of [H=96, W=128, C=2] and an output shape of [H=3, W=4, C=3], processed in a single sequence. The final layer distribution is shown in Table 9, which produces a 3-channel output per grid cell (confidence, x-offset, y-offset) for pupil center regression, identical to the original Tennst model. As evident from the table, all layers in the converted model are exclusive to Akida IP version 2.

#	Layer Type	Output Shape
0	InputData	[96, 128, 2]
1	BufferTempConv	[96, 128, 8]
2	Conv2D	[48, 64, 16]
3	BufferTempConv	[48, 64, 32]
4	Conv2D	[24, 32, 48]
5	BufferTempConv	[24, 32, 64]
6	Conv2D	[12, 16, 80]
7	DepthwiseBufferTempConv	[12, 16, 80]
8	Conv2D	[12, 16, 96]
9	DepthwiseConv2D	[6, 8, 96]
10	Conv2D	[6, 8, 112]
11	DepthwiseBufferTempConv	[6, 8, 112]
12	Conv2D	[6, 8, 128]
13	DepthwiseConv2D	[3, 4, 128]
14	Conv2D	[3, 4, 256]
15	BufferTempConv	[3, 4, 256]
16	Conv2D	[3, 4, 256]
17	Conv2D	[3, 4, 3]
18	Dequantizer	[3, 4, 3]

Table 9: Layer distribution and output shapes of the converted Akida SNN model (IP version 2)

To enable functional validation and performance measurement, the model was instead mapped to a virtual IPv2 device (`akida.SixNodesIPv2()`) executed in software on the Alienware workstation. This virtual device fully supports IP version 2 layers and allows correct execution of the converted SNN, but without the power and real-time benefits of the physical NSoC.

Table 10 compares the three models across key metrics: model size, mean squared error (MSE) in pixels on the original 480×640 frame, and latency (model inference time and total time including pixel reconstruction) for a single input of shape [B=1, C=2, T=50, H=96, W=128]. These metrics have been recorded in the first 200 batches (with batch size 8) of the test dataset, which justifies the difference in accuracy with the 34.763 px provided in section 4.2.2, which uses the full dataset and a batch size of 128.

Model	Size (MB)	MSE (px)	Latency (ms)	
			Model	Total
Akida Example Tensst (FP32)	12.3	32.295	436.59	436.62
Q-Int8 With Calibration Samples	1.1	44.467	3608.62	3611.28
Converted Akida IPv2 (virtual device)	1.2	40.857	583.27	584.96

Table 10: Conversion results for the Akida Example's Replicated Tennst model

The converted model achieves a substantial reduction in inference latency, going down from 3608 ms in the quantized INT-8 model to 583 ms on the virtual IPv2 device, both been limited

to the frame-by-frame processing enforced by the quantized model structure. However, the achieved latency is still above the one from the original model, which is 436 ms. In the same way, when it comes to accuracy, MSE improves comparing with the quantized INT-8 model, but still falls behind the original model.

5 Conclusion

While the original goal of the project was to apply event-based SNNs to maritime object detection, the project revealed that achieving reliable SNN deployment on first-generation neuromorphic hardware (AKD1000) is itself a significant engineering challenge. The decision to focus on replicating and analyzing BrainChip's pupil-tracking example was driven by the need to establish a working baseline amid IP version incompatibilities and toolchain limitations. This narrower scope allowed thorough exploration of the full pipeline, from event acquisition to simulated spiking inference, and provided valuable insights into the current maturity of neuromorphic vision systems.

Two deployment platforms were initially proposed: a low-power setup using the Raspberry Pi 5 with an AKD1000 M.2 card, and a higher-performance setup using the NVIDIA Jetson AGX Orin with an AKD1000 PCIe board. The Jetson platform was ultimately discarded due to persistent dependency and compatibility issues in the SNN toolchain, despite successful event camera operation on this board. All final training, quantization, conversion, and simulated inference were therefore performed using the Raspberry Pi 5 for hardware interfacing and the Alienware workstation for compute-intensive tasks.

In the end, this project successfully demonstrated an end-to-end pipeline for event-based pupil center detection using spiking neural networks on neuromorphic hardware, reproducing and extending BrainChip's official Akida example.

Event data preprocessing was achieved without event transformations or binning, generating high-fidelity spatio-temporal tensors directly from raw event recordings. However, dataset preparation proved extremely storage-intensive, requiring up to 472 GB in early attempts and 236 GB in the final Float16 format with full temporal resolution (T=10). This highlighted the practical challenges of handling high-resolution event streams for deep learning workflows.

Training results revealed the difficulty of the regression task. The Simplified Tennst model (6.7M parameters) converged to very low accuracy, despite extended training and hyperparameter tuning, indicating that direct x-y coordinate regression is insufficiently expressive for this challenging event-based problem. In contrast, the replicated Akida Example Tennst converged reliably, achieving usable performance despite its smaller size and limited batch sizes imposed by RAM constraints, giving Out Of Memory (OOM) errors with larger batches. Training times were noticeably longer for the simplified architecture, underscoring the importance of well-designed spatio-temporal blocks.

Quantization to INT-8 was successful, with explicit calibration samples dramatically not improving accuracy compared to default settings. However, latency was dramatically increased comparing the quantized models to the original. This is closely related to the collapse of the temporal dimension, which has been identified as the more significant side-effect of quantization, forcing frame-by-frame inference and requiring manual FIFO buffer resets. On the other hand, INT-4 quantization was not possible due to lack of support in the `quantizeml` toolbox for the target architecture.

Conversion to an Akida-compatible SNN succeeded, producing a functional spiking model with substantially lower latency than the quantized model on the virtual IPv2 device, even if being constrained to the temporal frame-by-frame processing. However, mapping to the physical AKD1000 failed due to IP version incompatibility, as the converted model relies on version 2 layers which are unavailable on the first-generation hardware. Although internally spiking and event-driven, the model still requires accumulated "event frames" as input, limiting the exploitation of full asynchrony.

Overall, the project validated the complete software pipeline from raw events to a deployable SNN in simulation, providing valuable insights into the practical limitations of current neuromorphic hardware for state-of-the-art event-based vision tasks. While real AKD1000 execution was not achieved, the results highlight clear paths for future improvements in both model design and hardware compatibility.

6 Future Work

Several directions remain open to extend and improve the current work. These extensions would transform the current proof-of-concept into a fully functional, real-time, ultra-low-power event-based vision system on edge hardware, while broadening its applicability to a wider range of practical applications.

Deployment on the physical AKD1000.

Redesign the spatio-temporal architecture using only Akida IP version 1 compatible layers, retrain on the full dataset, quantize, convert, and successfully map to the real NSoC on the Raspberry Pi 5. This would enable genuine ultra-low-power neuromorphic inference.

Power and performance benchmarking.

Once real hardware mapping is achieved, measure and compare power consumption, latency, and energy per inference against the FP32 baseline, quantized ANN, and virtual IPv2 execution to quantify the actual benefits of neuromorphic deployment.

Advanced data preprocessing and augmentation.

Implement causal event binning with exponential decay, spatial augmentations (rotations, scaling, translations), and temporal augmentations (random time scaling, time flipping with polarity inversion) to further improve model robustness and generalization.

Quantization enhancements.

Explore INT4 quantization (if supported in future MetaTF releases) and quantization-aware training (QAT) to recover potential accuracy while maximizing efficiency gains. Additionally, investigate the root cause of the temporal dimension collapse during quantization in the `quantizeml` toolbox, determining whether this limitation can be avoided or mitigated (through custom quantization configurations or alternative tools), and assessing the impact on latency and accuracy when preserving full spatio-temporal processing.

Detailed quantitative analysis of event processing.

Perform systematic measurements of the spatio-temporal filter's impact, including latency with and without filtering, total event count per recording, average events per packet, and processing latency per packet. Tune filter parameters ($\pm dx$, $\pm dy$, Δt, minimum event count) to optimize noise reduction versus responsiveness for real-time pupil tracking.

Full integration of the event camera with the Raspberry Pi 5 platform.

Install the Metavision SDK natively on the Pi to enable direct event acquisition without relying on the Jetson AGX Orin. In addition, integrate the filtered event stream directly with the mapped SNN for closed-loop, end-to-end real-time pupil center detection, including visualization and logging on the Pi. This would create a complete low-power edge system combining event sensing, filtering, and inference on a single board.

Custom dataset creation and replication of the workflow for other event-based detection tasks.

Record new event-based datasets for different vision tasks (like gesture recognition, object tracking, or obstacle avoidance) and apply the full pipeline: preprocessing, training, quantization, conversion, and deployment. This would demonstrate the generalizability of the approach on neuromorphic hardware.

Revisiting the original maritime application.

Future extensions could revisit the original maritime applications by building on the validated event-processing and SNN pipeline. Custom datasets recorded in real marine environments (high-glare water surfaces, fog, or rapid wave motion) could be collected using the Prophesee EVK4, combined with Akida IP version 1-compatible architectures to ensure successful mapping to the physical AKD1000. This would enable exploration of tasks such as obstacle detection, horizon tracking, or small-object recognition in challenging visual conditions, fully exploiting the low-power and high-dynamic-range advantages of event-based neuromorphic vision at sea.

References

[1] Grant Sanderson. *But what is a neural network? — Chapter 1, Deep learning*. YouTube video, 19:13 min, from Neural Networks series (explains core concepts of ANNs), Accessed: 2025-12-05. Oct. 2017. URL: https://youtu.be/aircAruvnKk?si=BeJRVyvP4YLk-8h4 (visited on 05/12/2025).

[2] Grant Sanderson. *Gradient descent, how neural networks learn — Chapter 2, Deep learning*. YouTube video, 20:13 min, from Neural Networks series (covers gradient descent and supervised training), Accessed: 2025-12-05. Oct. 2017. URL: https://youtu.be/IHZwWFHWa-w?si=b29N_C6UEGfE0TqZ (visited on 05/12/2025).

[3] Grant Sanderson. *What is backpropagation really doing? — Chapter 3, Deep learning*. YouTube video, 21:10 min, from Neural Networks series (explains backpropagation in supervised learning), Accessed: 2025-12-05. Oct. 2017. URL: https://youtu.be/Ilg3gGewQ5U?si=L5ZoBtdE8aCnxABx (visited on 05/12/2025).

[4] Grant Sanderson. *Backpropagation calculus — Chapter 4, Deep learning*. YouTube video, 21:32 min, from Neural Networks series (mathematical details of backpropagation for training), Accessed: 2025-12-05. Oct. 2017. URL: https://youtu.be/tIeHLnjs5U8?si=PvzlRMwqZJ-hxNte (visited on 05/12/2025).

[5] Joseph Redmon et al. 'You Only Look Once: Unified, Real-Time Object Detection'. In: *CoRR* abs/1506.02640 (2016). URL: https://arxiv.org/abs/1506.02640.

[6] Shaoqing Ren et al. 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks'. In: *CoRR* abs/1506.01497 (2015). URL: https://arxiv.org/abs/1506.01497.

[7] BrainChip Inc. *Temporal Event Neural Networks (TENNs) – Technical Whitepaper*. Whitepaper comparing TENNs to CNNs for event-based processing, Accessed: 2025-12-05. June 2023. URL: https://brainchip.com/wp-content/uploads/2023/06/TENNs_Whitepaper_Final.pdf (visited on 05/12/2025).

[8] BrainChip Inc. *Introduction to Spatiotemporal Models – Akida Examples Documentation*. Official explanation of spatio-temporal blocks for gesture recognition with TENNs, Accessed: 2025-12-05. 2025. URL: https://doc.brainchipinc.com/examples/spatiotemporal/plot_0_introduction_to_spatiotemporal_models.html (visited on 05/12/2025).

[9] Yongqi Li, Han Lu, Linjie Yang et al. *Quantization of Neural Networks: A Comprehensive Survey*. arXiv preprint arXiv:2411.06084 (survey on PTQ and QAT techniques), Accessed: 2025-12-05. Nov. 2024. URL: https://arxiv.org/pdf/2411.06084 (visited on 05/12/2025).

[10] Bodo Rueckauer et al. 'Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Neuromorphic Hardware'. In: *Frontiers in Neuroscience* 11 (2017). Seminal paper on rate-based ANN-to-SNN conversion with theoretical foundations and practical guidelines, p. 682. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.org/articles/10.3389/fnins.2017.00682/full (visited on 05/12/2025).

[11] Emre O. Neftci, Hesham Mostafa and Friedemann Zenke. 'Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-based Optimization to Spiking Neural Networks'. In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[12] Jon Urcelay. *RC Circuit for Leaky Integrate-and-Fire (LIF) Neuron*. YouTube video, 00:24 min, visual representation of the RC circuit analogy for LIF neurons, Accessed: 2025-12-05. Dec. 2025. URL: https://youtu.be/6e6ALpVKTAw (visited on 05/12/2025).

[13] Giacomo Indiveri et al. 'Neuromorphic Silicon Neuron Circuits'. In: *Frontiers in Neuroscience* 5 (2011), p. 73. DOI: 10.3389/fnins.2011.00073.

[14] NVIDIA. *Download and Run SDK Manager*. Accessed: 2025-04-05. 2024. URL: https://docs.nvidia.com/sdk-manager/download-run-sdkm/index.html (visited on 05/04/2025).

[15] JetsonHacks. *Jetson AGX Orin - NVMe SSD Boot*. YouTube video, Accessed: 2025-04-05. June 2023. URL: https://www.youtube.com/watch?v=DKI1k_aP0Qk (visited on 05/04/2025).

[16] NVIDIA Developer. *NVIDIA SDK Manager Tutorial: Installing Jetson Software Explained*. YouTube video, Accessed: 2025-04-05. 2024. URL: https://www.youtube.com/watch?v=Ucg5Zqm9ZMk (visited on 05/04/2025).

[17] NVIDIA. *Install Jetson Software with SDK Manager*. Accessed: 2025-04-05. 2024. URL: https://docs.nvidia.com/sdk-manager/install-with-sdkm-jetson/index.html (visited on 05/04/2025).

[18] Prophesee. *Metavision SDK Documentation*. Official Prophesee Metavision SDK documentation, Accessed: 2025-04-05. 2025. URL: https://docs.prophesee.ai/stable/index.html (visited on 05/04/2025).

[19] AIS-CPS-Lab. *pandect-setup: Automated installation of ROS 2 and Prophesee event-camera drivers*. Accessed: 2025-04-05. 2024. URL: https://github.com/AIS-CPS-Lab/pandect-setup.git (visited on 05/04/2025).

[20] Timo Stoffregen et al. *ros-event-camera: ROS driver and tools for event cameras*. 2024. URL: https://github.com/ros-event-camera (visited on 05/04/2025).

[21] UZH Robotics and Perception Group. *Event-based Vision Resources*. 2024. URL: https://github.com/uzh-rpg/event-based_vision_resources (visited on 05/04/2025).

[22] Jon Urcelay. *ros2_event_camera/EC: Real-time event processing in ROS 2 using components, C++ and fixed-size time-surface arrays*. Personal repository, Accessed: 2025-04-05. 2025. URL: https://github.com/jonurce/ros2_event_camera/tree/main/EC (visited on 05/04/2025).

[23] BrainChip Inc. *Akida Examples Documentation: MetaTF Development Environment*. Official documentation for MetaTF and Akida Runtime, Accessed: 2025-04-05. 2025. URL: https://doc.brainchipinc.com/ (visited on 05/04/2025).

[24] NeuroCortex AI. *Unleashing the Power of Spiking Neural Networks — Installing BrainChip Akida*. Oct. 2024. URL: https://medium.com/@neurocortexai/unleashing-the-power-of-spiking-neural-networks-installing-brainchip-akida-af9bb4480658 (visited on 05/04/2025).

[25] BrainChip Inc. *Akida PCIe Driver Installation Guide*. Official driver installation manual, Accessed: 2025-04-05. BrainChip Inc. June 2022. URL: https://brainchip.com/wp-content/uploads/2022/06/Akida-PCIe-Driver-Installation-Guide.pdf (visited on 05/04/2025).

[26] BrainChip Inc. *Akida PCIe Board User Guide*. 101g. Official user guide for the Akida AKD1000 PCIe development board, Accessed: 2025-04-05. BrainChip Inc. June 2022. URL: https://brainchip.com/wp-content/uploads/2022/06/Akida-PCIe_Board_UG_101g.pdf (visited on 05/04/2025).

[27] NVIDIA Corporation. *Kernel Customization – Jetson Linux Developer Guide*. Official NVIDIA documentation for manually downloading and expanding Jetson kernel sources, Accessed: 2025-04-05. 2024. URL: https://docs.nvidia.com/jetson/archives/r36.3/DeveloperGuide/SD/Kernel/KernelCustomization.html#to-manually-download-and-expand-the-kernel-sources (visited on 05/04/2025).

[28] NVIDIA Corporation. *Jetson Linux Archive*. Archive of all Jetson Linux releases and corresponding source packages, Accessed: 2025-04-05. 2025. URL: https://developer.nvidia.com/embedded/jetson-linux-archive (visited on 05/04/2025).

[29] NVIDIA Corporation. *Jetson Linux 36.4 (Driver Package BSP Sources)*. Public sources (public_sources.tbz2), 2025 – 04 – 05. 2025. URL: https://developer.nvidia.com/embedded/jetson-linux-r3640 (visited on 05/04/2025).

[30] BrainChip Inc. *Global Workflow – Akida Examples Documentation*. Official Akida end-to-end workflow example (training → quantization → conversion → inference), Accessed: 2025-04-05. 2025. URL: https://doc.brainchipinc.com/examples/general/plot_0_global_workflow.html#sphx-glr-examples-general-plot-0-global-workflow-py (visited on 05/04/2025).

[31] BrainChip Inc. *Event-based Eye-tracking with Akida – Akida Examples Documentation*. Official BrainChip example for training, quantizing, and converting an event-based SNN for pupil center detection using Prophesee event data, Accessed: 2025-04-05. 2025. URL: https://doc.brainchipinc.com/examples/spatiotemporal/plot_1_eye_tracking_cvpr.html (visited on 05/04/2025).

[32] BrainChip Inc. *Akida 1.x Layers – Akida API Reference*. Official documentation of Akida 1.x compatible layers and constraints, Accessed: 2025-04-05. 2025. URL: https://doc.brainchipinc.com/api_reference/akida_apis.html#akida-v1-layers (visited on 05/04/2025).

[33] BrainChip Inc. *Akida 2.x Layers – Akida API Reference*. Official documentation of Akida 2.x (second-generation) layers and supported operations, Accessed: 2025-04-05. 2025. URL: https://doc.brainchipinc.com/api_reference/akida_apis.html#akida-v2-layers (visited on 05/04/2025).

[34] Xingyi Zhou, Dequan Wang and Philipp Krähenbühl. *Objects as Points*. arXiv preprint arXiv:1904.07850 (CenterNet paper). Apr. 2019. URL: https://arxiv.org/abs/1904.07850 (visited on 05/04/2025).

[35] CVPR 2025 and Kaggle. *Event-based Eye Tracking – CVPR 2025 Challenge Dataset*. Official dataset download page (raw Prophesee recordings + 100 Hz pupil centre annotations), Accessed: 2025-04-05. 2025. URL: https://www.kaggle.com/competitions/event-based-eye-tracking-cvpr-2025/data (visited on 05/04/2025).

[36] EETChallenge Organizers. *3ET Challenge 2025: Event-based Eye Tracking Challenge – Official Repository and Baseline*. Official training and evaluation workflow for the CVPR 2025 Event-based Eye Tracking Challenge, Accessed: 2025-04-05. 2025. URL: https://github.com/EETChallenge/3et_challenge_2025 (visited on 05/04/2025).

[37] Jon Urcelay. *Event Reading: Filtered vs Not Filtered Events*. YouTube video, 00:19 min, visual representation of filter and not filtered events, read with EVK4 event camera. Dec. 2025. URL: https://youtu.be/fgZQ_I7QnAk (visited on 09/12/2025).

[38] Rachmad Vidya Wicaksana Putra, Pasindu Wickramasinghe and Muhammad Shafique. *Enabling Efficient Processing of Spiking Neural Networks with On-Chip Learning on Commodity Neuromorphic Processors for Edge AI Systems*. arXiv preprint arXiv:2504.00957, Accepted at IJCNN 2025. 2025. URL: https://arxiv.org/pdf/2504.00957.

[39] Jon Urcelay. *Akida SNN Event based Eye Center Detection*. Personal GitHub repository containing the full implementation of the Akida workflow for obatining a SNN, Accessed: 2025-12-11. 2025. URL: https://github.com/jonurce/Akida_SNN_Event_Cameras_Eye_Center_Detection (visited on 11/12/2025).

A Hardware Comparisons

On this section, the next comparisons are carried out: A.1, A.2, and A.3 will compare, respectively, different options for event cameras, neuromorphic chips, and host computers. In the comparison tables, positive aspects will be highlighted in green, while negative ones will be marked in red. The final choice, the bill of materials and budget are showed in A.4.

A.1 Neuromorphic camera comparison

	Prophesee EVK4 (IMX636)	SynSense Speck	Inivation DAVIS346
Type	Event Camera	EC + Neuromorphic Processor	EC + Frames + IMU
Resolution	1280 x 720 = 921600	128 x 128 = 16384	346 x 260 = 89960
Temporal Resolution	$1\mu s$	Unspecified	$1\mu s$
Latency	$< 220\mu s$	Unspecified	$< 1000\mu s$
Dynamic Range	120dB	> 80dB	120 dB
D-FOV	47.7^{o}	77^{o}	Unspecified
Bandwidth	1.6 Gbps	Unspecified	Unspecified
Interface	USB 3.0 C with lock screws	USB 3.0 C	USB 3.0 micro B with lock screws
L x W x H	30 x 30 x 36mm	105 x 65 x 5 mm	40 x 60 x 25mm
Software	Metavision SDK	Tonic (datasets), Sinabs (SNN), Samna (interact with neuromorphic devices)	DV-Platform
Availability	Request a quote	Request a quote	Online Shop
Price	Unspecified	Unspecified	$5120
Shipping	Unspecified	Unspecified	Unspecified

Table 11: Neuromorphic Cameras Comparison

A.2 Neuromorphic chip comparison

As mentioned in [38], research processors refer to neuromorphic chips that are designed only for research and not commercially available, access to these processors being limited. Several examples in this category are SpiNNaker, NeuroGrid, IBM's TrueNorth, and Intel's Loihi.

Meanwhile, commodity processors refer to neuromorphic chips that are available commercially, such as BrainChip's Akida and SynSense's DYNAP-CNN.

In this way, research processors will be compared in Table 12, comparing Intel's NCs in a separate table (Table 13), and commercial NCs will be compared in Table 14.

	Neurogrid (Braindrop)	SpiNNaker	BrainScaleS	TrueNorth	Darwin	Dynap-SEL
Company / university	Standford	University of Dresden	Heidelberg University	IBM	Zhejiang University	Institute of Neuroinformatics (ETH Zurich) and SynSense
Release year	2014	2021	2022	2014	2022	2018
Neurons per chip	62K	0.152M	512	1M	2.35M	1024
Neurons per system	1M	7.6×10^{11} (5M chips)	512	1M	2.35M	4096 (x4 chips)
On-chip learning	No	Yes	Yes	No	Yes	Yes
Availability	End of life	Remote access		End of life	Contact Zhejiang University for collaboration	Contact SynSense
Price	N/A	Free		N/A	Unknown	
Shipping	N/A				Unknown	

Table 12: Research Neuromorphic Processors Comparison

	Oheo Gulch	Kapoho Point	Kapoho Bay	Nahuku	Pohoiki Springs
Chip	x1 Loihi 2	x8 Loihi 2	x2 Loihi 1	x8-32 Loihi 1	x768 Loihi 1
Neurons per chip	1.05M		0.131M		
Neurons	1M	8.4M	0.262M	1-4.2M	100M
Max processing rate	Unspecified				
On-chip learning	Yes				
Software	Lava				
OS	Linux				
Host interf.	PCIe 3.0 x8-lane (Arria 10 FPGA) 128Gbps	Ethernet	USB	Arria 10 FPGA Expansion Card	Many-Board System
Setup	Loihi and EC connected to host separately / Remote access from the Neuromorphic Research Cloud (vLab)				
Availability	Join INRC	Not released yet	Join INRC		
Price	Unspecified				
Shipping	Unspecified				

Table 13: Intel's Research Neuromorphic Processors Comparison

	Brainchip Akida PCIe Board	Brainchip Akida M.2 Card	SynSense Speck
Neurons	1.2M		0.32M
Max processing rate	Unspecified		
On-chip learning	Yes		No, off-chip training on Sinabs
Software	MetaTF		Samna
OS	Linux on ARM or x86-64 architectures - Ubuntu 20.04/22.04	Linux ARM-based - Ubuntu or Raspberry Pi OS	Linux - Ubuntu 18.04/20.04
Host interf.	PCIe 2.0 x1-lane	M.2 B+M or E Key with PCIe 2.0 x2-lane	USB 3.0 Micro-B
Interface bandwidth	4Gbps	8Gbps	4.8Gbps
Setup	Akida and EC connected to host (Single-Board Computer) separately	Akida and EC connected to host (Rasp. Pi, IoT SoC Dev. Boards or Single-Board Computer) separately	Speck connected to host (Single-Board Computer) and EC connected to Speck or to host
Availability	Brainchip Online Shop		Request a quote
Price	$289	$249	Unspecified
Shipping	7 days		Unspecified

Table 14: Commercial Neuromorphic Processors Comparison

A.3 Host Computer comparison

	Brainchip Akida Edge AI Box	Jetson AGX Orin	Jetson Orin Nano	Raspberry Pi 5
AI Performance	Unspecified	275 TOPS	67 TOPS	Unspecified
CPU	4x or 2x Cortex-A53 1.8 GHz (0.5ns)	12-core Arm Cortex-A78AE v8.2 64-bit CPU, 3MB L2 + 6MB L3	6-core Arm Cortex-A78AE v8.2 64-bit CPU, 1.5MB L2 + 4MB L3	4-core Arm Cortex-A76 64-bit CPU, 512KB L2 + 2MB L3 2.4GHz (0.41ns)
GPU	6 GFLOPS (high-precision) OpenGL ES 3.1/3.0, Vulkan, Open CL 1.2 FP, OpenVG 1.1	NVIDIA Ampere architecture with 2048 NVIDIA CUDA cores and 64 tensor cores	NVIDIA Ampere architecture with 1024 CUDA cores and 32 tensor cores	VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2
NPU	2x AKD1000 Chip Over PCIe Interface	-	-	-
Camera	-	x16-lane MIPI CSI-2 connector	2x MIPI CSI-2 22-pin camera connectors	2x 4-lane MIPI camera/display transceivers
PCIe Slots	-	x16-lane PCIe slot supporting PCIe 4.0 x8-lane	-	PCIe slot 2.0 x1-lane
M.2 B+M Key	-	M Key PCIe 4.0 x4-lane	M Key PCIe 3.0 x4-lane + M Key PCIe 3.0 x2-lane	-
M.2 E Key	-	PCIe 4.0 x1-lane	PCIe 3.0 x1-lane	-
OS	Linux			Raspberry Pi OS
USB Ports	USB 3.0 Type A + USB 2.0 Micro B	2x USB 3.2 Gen2 Type C + 2x USB 3.2 Gen2 Type B + 2x USB 3.2 Gen1 Type B + USB 2.0 Micro B	4x USB 3.2 Gen2 Type A + Type-C connector for UFP	2x USB 3.0 + 2x USB 2.0 + USB Type C for Power
Max. Encode	1x1080P60 / 2x1080P30 / 4x720P30 (H265 & H264)	2x 4K60 / 4x 4K30 / 8x 1080p60 / 16x 1080p30 (H265)	1080p30 supported by 1-2 CPU cores	-
Max. Decode	Same as encode	1x 8K30 / 3x 4K60 / 7x 4K30 / 11x 1080p60 / 22x 1080p30 (H265)	1x 4K60 / 2x 4K30 / 5x 1080p60 / 11x 1080p30 (H265)	4Kp60 HEVC
Memory	4GB LPDDR4	64GB 256-bit LPDDR5 204.8GB/s	8GB 128-bit LPDDR5 102GB/s	LPDDR4X-4267 SDRAM 16GB
Storage	32GB eMMC & Supports Micro SDXC card up to 1 TB	64GB eMMC 5.1	Supports SD card slot and external NVMe	Unspecified
Power	Unspecified	15-60W	7-25W	Unspecified
L x W x H	110 x 110 x 56mm	110 x 110 x 71.65mm	103 x 90.5 x 34.77mm	56 x 85 x 20mm
Availability	Brainchip Online Shop	RS-Online	RS-Online	Raspberry Pi Online
Price	$995	$2884	$360	$120
Shipping	10-12 weeks	1 week		

Table 15: Computer Comparison

A.4 Final choice and Budget

- Option 1. Heavy for detection from land: Prophesee EVK4 + Akida PCIe/Akida M.2 B+M Key Card + Jetson AGX Orin.
  - Best camera specifications.
  - High number of neurons.
  - On-chip training available.
  - Akida PCIe board is directly compatible with the Jetson.
  - Akida M.2 E Key 2260 is too long for Jetson's M.2 E Key slot.
  - Akida M.2 B+M Key 2260 is compatible with Jetson's M.2 M Key slot. A lenght adapter is needed to fit the 2260 card into the 2280 compartment.
  - Akida PCIe board and M.2 B+M Key card can be used separately and together in the Jetson.
  - All items are commercially available and fast delivery.
- Option 2. Light for possible on drone implementations: Prophesee EVK4 + Akida M.2 B+M Key Card + Raspberry Pi 5.
  - Best camera specifications.
  - High number of neurons.
  - On-chip training available.
  - Akida M.2 B+M Key Card is compatible with both Jetson and Raspberry Pi (PCIe to M.2 B+M Key adapter hat is needed).
  - All items are commercially available and fast delivery.

Object	On campus	Link	Price [NOK]
RGB Camera	Yes	RealSense D435	-
Event Camera	Yes	Prophesee EVK4	-
RC super fast boat	No	ATOMIC SR85	2805
Jetson AGX Orin	Yes	Jetson AGX Orin	-
Neuromorphic Processor	No	Akida PCIe Board	2875
2TB SSD Card	Yes	WD Black SN850 NVMe SSD	-
Raspberry Pi 5	No	Raspberry Pi 5 16GB	1223
Raspberry Pi Active Cooler	No	Raspberry Pi Active Cooler	51
Pi HAT for M.2 B+M Key	No	PCIe to M.2 M Key Adapter	218
Neuromorphic Processor	No	Akida M2 Card B+M Key	2460
MicroSD 64GB	Yes	SanDisk Ultra 64GB microSDXC	-
Total			9672 NOK

Table 16: Component List and Budget

B Project's Github Repository

The complete source code developed during the project is publicly available at: https://github.com/jonurce/Akida_SNN_Event_Cameras_Eye_Center_Detection.git [39].

The repository is structured as follows:

/EC

This folder contains the code for reading Events from EVK4, implementing ROS 2 components with C++. The main folder is **/EC/src/composition**

/RGB

This folder contains the code initially used to train a YOLO model for object detection, with the initial intention of the project to compare event-based SNN detection and frame-based CNN detection.

/akida_examples

This is the main folder of the project, where Akida's example workflow is replicated from scratch.

/akida_examples/0_global_workflow

- `0_global_akida_workflow.py`

  This scipt replicates Akida's Example Global Workflow, making sure that all dependencies are working.

/akida_examples/1_ec_example

- **quantized_models**

  This folder contains the quantized models, one of them quantized with default calibration and the other one with calibration providing samples. For the last one, this folder also stores the converted akida model.

- **training**

  This folder contains the code related to the first phase in Akida's SNN workflow, which is training a CNN. Datasets are also stored inside this folder.

  - **event-based-eye-tracking-cvpr-2025**
    Extracted original dataset with raw events and labels.
  - **preprocessed_akida**
    Preprocessed dataset for training the Akida Example's Replicated Tennst. Divided in train and test splits.

- **preprocessed_akida_test_small**
  Preprocessed dataset for testing the final converted Akida SNN model on the Raspberry Pi. Only some recordings of the test split are stored, to ensure low usage of storage on the Raspberry Pi, which has limited disk storage.
- **preprocessed_fast_open**
  Preprocessed dataset for training the Simplified Tennst. This dataset stores a voxel with T=10 temporal event frames for each label. Divided in train and test splits.
- **preprocessed_open**
  Preprocessed dataset for training the Simplified Tennst. This dataset stores a voxel with a single T=1 temporal event frame for each label. Divided in train and test splits.
- **runs**
  This folder is used to store the trained CNN models and their results, saving different versions of the Simplified Tennst and the Akida Example's Replicated Tennst.
- **plots**
  This folder saves multiple plots and figures (image format) generated during the project.
- **old_files**
  This folder contains old scripts that were used during the development of the project.
- `_0_check_cuda.py`
  This script checks if cuda is available in the host.
- `_0_discover_dataset.py`
  This script is used to analyze the original dataset with raw events **event-based-eye-tracking-cvpr-2025** and print multiple metrics.
- `_1_1_preprocess_bin_1.py`
  this script is used to generate the dataset **preprocessed_open** from the original dataset in **event-based-eye-tracking-cvpr-2025**.
- `_1_2_preprocess_bin_10.py`
  This script is used to generate the dataset **preprocessed_fast_open** from the original dataset in **event-based-eye-tracking-cvpr-2025**.
- `_1_4_preprocess_akida.py`
  This script is used to generate the dataset **preprocessed_akida** from the original dataset in **event-based-eye-tracking-cvpr-2025**.
- `_2_3_model_f16_last_10_gradual.py`
  This script defines the Simplified Tennst model.
- `_2_5_model_uint8_akida.py`
  This script defines the Akida Example's Replicated Tennst model.

- `_3_2_train_fast.py`

  This script trains the Simplified Tennst model `_2_3_model_f16_last_10_gradual.py` by using the dataset stored in **preprocessed_open**, saving the trained model in **runs**.

- `_3_3_train_fast_dataset_10.py`

  This script trains the Simplified Tennst model `_2_3_model_f16_last_10_gradual.py` by using the dataset stored in **preprocessed_fast_open**, saving the trained model in **runs**.

- `_3_4_train_fast_akida.py`

  This script trains the Akida Example's Replicated Tennst model `_2_5_model_uint8_akida.py` by using the dataset stored in **preprocessed_akida**, saving the trained model in **runs**.

- `_4_0_plot_training.py`

  This script is used to plot the training and validation losses during training, which are saved in a txt file under the respective model in **runs**.

- `_4_test_tennst.py`

  This script tests the Simplified Tennst model (saved in **runs**), in the test dataset and print the resulting metrics.

- `_4_test_tennst_akida.py`

  This script tests the Akida Example's Replicated Tennst model (saved in **runs**), in the test dataset and print the resulting metrics.

- `_4_1_test_tennst_plot.py`

  This script uses the Simplified Tennst model (saved in **runs**) to make predictions on a chosen sample in the test dataset, and plots a 2D and 3D representation of the prediction with the given input events.

- `_4_1_test_tennst_plot_akida.py`

  This script uses the Akida Example's Replicated Tennst model (saved in **runs**) to make predictions on a chosen sample in the test dataset, and plots a 2D and 3D representation of the prediction with the given input events.

- `_0_example_code.py`

  This script is Akida's eye-tracing example copy-pasted code, that check if certain functions work, making sure that all dependencies are working.

- `_5_0_quantize_default.py`

  This script quantizes the chosen model from **training/runs** by using default calibration samples, storing the resulting quantized model in **quantized_models**. In addition, it evaluates both the not quantized and quantized models, printing the resulting metrics and saving the results in the same **quantized_models** folder.

- `_5_1_quantize_calib.py`

This script quantizes the chosen model from **training/runs** by using customized given calibration samples, storing the resulting quantized model in **quantized_models**. In addition, it evaluates both the not quantized and quantized models, printing the resulting metrics and saving the results in the same **quantized_models** folder.

- `_6_qat.py`

  This script is a failed attempt to perform quantization-aware-training with an already quantized model from **quantized_models**.

- `_7_prep_akida_test_small.py`

  This script is used to generate the dataset **training/preprocessed_akida_test_small** from the original dataset in **training/event-based-eye-tracking-cvpr-2025**.

- `_8_check_cnn2snn.py`

  This script is used to analyze the version of cnn2snn.

- `_8_convert.py`

  This script is used to convert a chosen quantized model from **quantized_models** to Akida SNN, saving it to the respective folder in **quantized_models** and making sure that the converted model can be mapped into a chosen device.

- `_9_test_snn.py`

  This script evaluates the chosen Akida SNN model from **quantized_models**, mapped into a virtual device, in the test dataset inside **training/preprocessed_akida**, printing and saving the resulting metrics.

- `_10_deploy_pi_akida.py`

  This script evaluates the chosen Akida SNN model from **quantized_models**, mapped into the neuromorphic hardware device, in the test dataset **training/preprocessed_akida_test_small**, printing and saving the resulting metrics. This script is created to be used in the Raspberry Pi 5 with the connected Akida M2 Card, but is unsuccessful due to the incompatibility on the Akida IP version between the physical device and the trained model.