To the Graduate Council:
I am submitting herewith a dissertation written by Charles Peter Rizzo entitled "Exploration of Event-Based Camera Data with Spiking Neural Networks." I have examined the final paper copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank, Major Professor

We have read this dissertation
and recommend its acceptance:

James S. Plank

Catherine D. Schuman

Garrett S. Rose

Stephen A. Sarles

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the Graduate School

To the Graduate Council:
I am submitting herewith a dissertation written by Charles Peter Rizzo entitled "Exploration of Event-Based Camera Data with Spiking Neural Networks." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank, Major Professor

We have read this dissertation
and recommend its acceptance:

James S. Plank

Catherine D. Schuman

Garrett S. Rose

Stephen A. Sarles

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Exploration of Event-Based Camera Data with Spiking Neural Networks

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Charles Peter Rizzo

May 2024

© by Charles Peter Rizzo, 2024
All Rights Reserved.

To God for the discipline,

my wife for the support,

my mother for the confidence,

and my friends and family for reminding me to smell the roses.

I love you all.

In memory of my grandparents:

Pietro Angione (1932 - 2007) and Lena Angione (1932 - 2008)

Acknowledgements

I would like to thank the Department of Electrical Engineering and Computer Science Bodenheimer Fellowship (especially Dr. Robert Bodenheimer and Mr. Mike Crabtree) for providing me funding over the past several years to conduct my research and enjoy my time at UT. I would also like to thank the EECS staff and IT department for their willingness to put up with me and provide the necessary support, though often behind the scenes.

I would like to thank my advisor Dr. Jim Plank for the guidance and patience with me throughout these years. I would also like to thank my committee members and PIs Dr. Catherine Schuman for the unwavering support and care, Dr. Garrett Rose for the career advice and encouragement, and Dr. Andrew Sarles for his willingness to be on yet another TENNLab student's committee.

Additionally, I'd like to thank the students in the group both past and present for helping me get where I am today. I would not have made it here without their constructive suggestions, debugging assistance, and willingness to pitch in and donate time to various projects. Most of all, I appreciate the laughs and camaraderie. This was a marathon that I could not have finished without you all.

"Dans ses écrits, un sage Italien
Dit que le mieux est l'ennemi du bien."

– Voltaire, Dictionnaire Philosophique

Abstract

Neuromorphic computing is a novel, non-von Neumann computing architecture that employs power efficient spiking neural networks on specialized hardware. Taking inspiration from the human brain, spiking neural networks are temporal computation units that propagate information throughout the network via binary spikes. Compared to conventional artificial neural networks, these networks can be more sparse, smaller in size, and more efficient power-wise when realized on neuromorphic hardware. Event-based cameras are novel vision sensors that capture visual information through a temporal stream of events instead of as a conventional RGB frame. These cameras are low-power collections of pixels that asynchronously emit events over time with microsecond level precision. Because of this, these cameras are often dubbed "neuromorphic cameras". The events generated by a camera are analogous to the spikes that occur within a spiking neural network. This results in the temporal event stream being a very natural input modality for spiking neural networks. This body of work explores pairing spiking neural networks more tightly with event-based cameras and explores not only developing spiking neural networks as classification or control agents, but also explores leveraging spiking neural networks as denoisers and spatial feature extractors. This work aims to further cement pairing neuromorphic systems with neuromorphic cameras in order to foster an extremely low-power computer vision system powered by brain-inspired neural networks.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

As an emerging non-von Neumann computing architecture, neuromorphic computing features spiking neural networks (SNNs) that transmit information temporally through spike propagation. This more closely mimics the biology of the human brain – which still remains to be the most efficient "computer" that we have – than traditional neural networks do. Unlike traditional computation platforms, both the memory and computation components are embedded within the spiking neural network. Neuromorphic computing focuses on the development of SNNs that solve specific problems and that target specialized hardware.

The appeal of neuromorphic computing and its hardware is its low Size, Weight, and Power (SWaP) requirements. Neuromorphic hardware like IBM's TrueNorth [3], Intel's Loihi [28], SpiNNaker [41], and several others have become standardized in the research community. This hardware intentionally does not support the traditional deep neural networks that have stemmed from Yann Lecun's [66] foundational work, though it is possible to map these types of networks onto neuromorphic systems. Instead, the hardware natively supports power-efficient spiking neural networks. Spiking neural networks are temporal neural networks that propagate information through spikes over time. Additionally, spiking neural networks can be sparse collections of neurons and synapses as opposed to the dense, layered configuration

of artificial neural networks. Furthermore, the spike operation is non-differentiable which renders traditional learning with backpropagation impossible. The inability to leverage backpropagation has resulted in SNNs being difficult to train.

Recently, spiking neural networks have been leveraging surrogate gradient backpropagation functions such as Slayer [124] or E-Prop [10] for training. Instead of calculating a cost function derivative, a surrogate or estimated value is used instead for performing weight updates. There are also conversion based algorithms like Whetstone [122] that start with an artificial neural network and, either over the course of training or by way of a mapping algorithm, convert the network into a spiking version. Additionally, there are genetic algorithms like LEAP [26] and EONS [117] that apply genetic operators like crossover and merge to populations of networks over the course of training to optimize the parameters of the network. With the exception of EONS, the other listed training methods necessitate a rigidly defined, layered network architecture where data travels in a feed-forward manner throughout the network. However, it is also possible to produce spiking neural networks without training them. Though far less popular than employing training algorithms, spiking neural networks can be hand-tooled or crafted, as is done by Severa et al. in [121] and Plank et al. in [105, 104]. This complex approach requires knowledge of the function one wants to model with the network and how to map the function to a temporal process that is specific to a precisely defined neuronal model and synaptic model. The drawback of this approach is that the network is often not generalizable to other neuroprocessor architectures. However the benefit of a hand-tooled architecture is that, assuming deterministic network operation and input application, the network is guaranteed to perform the desired function 100% of the time. Typically, hand-tooled networks also scale well to various input sizes which can be helpful when targeting various types of hardware with different resource constraints.

As spiking neural networks have become more prominent, so also has a novel type of vision sensor called a Dynamic Vision Sensor (DVS) or event-based camera. Event-based cameras (EBCs) have been dubbed neuromorphic sensors because of

their sparse, event-driven output. Unlike conventional cameras, they do not capture dense frames of pixel values and they do not suffer from motion blur. Instead, they are composed of collections of asynchronous pixels that capture changes in light intensity and emit an event (of either positive or negative polarity) when the light luminosity change at a pixel location exceeds a certain threshold. Events are emitted asynchronously and as a constant stream of data. Their spikey, asynchronous, event-driven output pairs well with spiking neural networks' information communication modality which is the cornerstone of neuromorphic computing. They have a high temporal resolution (around 1 MHz) and dynamic range that enables them to capture motion or changes in light intensity at great distances. There are several commercially available cameras spanning different resolutions and technical capabilities. Examples of these are the Samsung DVS camera [126] and the Inivation DAVIS346 camera [133]. The major difficulty in working with event-based cameras is that due to their high temporal precision, they tend to generate enormous amounts of data in the form of precisely timed events.

The overall goal of this work is to produce a software library that facilitates event-based camera research with neuromorphic computing in the TENNLab Software Framework. The framework currently lacks the ability to process event-based camera data, and EONS lacks the ability to train spiking neural networks on visual observations spaces larger than roughly two hundred features. Additionally, much of the existing work with event-based cameras involves applying traditional frame-based processing to non frame-based data. As is often the case with conventional image-based processing, visual observations need to be downsampled to a more manageable input dimensionality. This is conventionally accomplished with a stack of convolutional and pooling layers. For the non-rigidly structured or layered spiking neural networks trained with EONS, it is impossible to leverage these layers, and so alternative methods must be used to downsample visual inputs. As such, I introduce the following novelties in this work:

- A tool in the TENNLab Software Framework that allows users to experiment with event-based camera data
- Detailed specifications for hand-tooled, dimensionality reducing spiking neural networks
- An application of the dimensionality reducing spiking neural networks to the classification of the DVSGesture dataset
- An application that exposes Atari 2600 games for SNN development within the framework that also simulates event camera data online during training
- An application of the dimensionality reducing spiking neural networks coupled with EONS for developing SNNs that can play Atari games based on visual observations
- Detailed specifications for hand-tooled spiking neural networks that denoise and cluster event camera data
- An application of the denoising and clustering spiking neural networks to the filtration of events in the DSEC driving dataset

Much of the existing work pairing event-based cameras with neuromorphic computing techniques relies on conventional image-based approaches and training methods in order to develop spiking neural networks. In this work, I provide a set of methods for processing event camera data in a sparse event batch format that preserves the inherent sparsity of the data stream. Furthermore, the presented hand-tooled spiking neural networks that preprocess event streams performing either downsampling or denoising and clustering are inherently scalable to any sized input space without need for training.

1.1 Outline

This rest of this dissertation is organized into multiple chapters that are structured in the following way. Chapter 2 provides some background and helps contextualize this work. Chapter 3 details novel hand-tooled neural networks and the functionalities that they provide. Chapter 4 discusses and details usage instructions for a library/tool developed to facilitate the use of the hand-tooled neural networks with event camera data. Chapter 5 details a classification experiment performed with a subset of the networks presented in Chapter 3 and discusses the implications of its results. Chapter 6 covers the many components necessary for training spiking neural networks to play Atari games where the observations come from event-based cameras. Chapter 7 details an event filtration experiment and demonstrates how some of the networks presented in Chapter 3 can be used to denoise and spatially cluster an EBC's event stream. Lastly, Chapter 8 lists ongoing and future work opportunities, and Chapter 9 lists the conclusions drawn from the work.

Chapter 2

Literature Review

2.1 Neuromorphic Processors

The benefits of neuromorphic computing are found in its hardware. One of the frontrunner neuroprocessors was SpinNNaker (Spiking Neural Network Architecture) [41] out of the University of Manchester. The SpiNNaker neuroprocessor leveraged general purpose ARM cores and a software neuron model in order to maximize flexibility at the cost of performance. Following closely behind SpiNNaker was IBM's TrueNorth [3] neuromorphic chip. Unlike SpiNNaker, TrueNorth implemented a digital neuron model in order to take advantage of hardware acceleration. The TrueNorth chip can support over 1 million neurons and 256 millions synapses spread across 4,096 cores. Similarly, Intel introduced their own digital chip with Loihi [28], and while it supports less neurons and synapses per chip than TrueNorth does, it does support online learning. Online learning allows networks to learn and be trained on chip instead of having to train and develop networks offline with other hardware so that they may be then loaded onto the neuromorphic chip (as is the case with TrueNorth). The DYNAPS [83] neuroprocessor supports the least amount of neurons and synapses per chip out of others, but it is an analog chip. For neuromorphic, an analog implementation is naturally rad-hard and implies even lesser power consumption

when compared to its digital counterparts. The BrainDrop [88] neuromorphic chip (out of Stanford University) is another analog chip that improves upon the neuron and synapse count of DYNAPS [42].

Each of the neuromorphic chips have their own software stacks in either C or Python and varying degrees of software simulator support. Even though all of the chips are designed to run spiking neural networks, they differ greatly in implementation (e.g. how spikes are routed through the hardware). None of the chips are commercially available, which makes it difficult to work directly with the existing hardware.

2.2 Neuromorphic Machine Learning

As a machine learning methodology, neuromorphic computing has grown more prominent due to its promise of lesser power consumption and smaller model footprints compared to larger, conventional models. While often not performing as well as those conventional models, it can perform comparably with a fraction of the resources at a fraction of the power consumption. It has been applied to all kinds of classification tasks, ranging from the classical Wine, Breast Cancer, and Iris datasets [4, 142, 118, 36, 115] accessible in scikit-learn [48], to newer datasets featuring ECG signals [27]. Spiking neural networks have even been applied to gesture recognition and have appeared in notable works like [31, 94, 128]. These works often use techniques like Spike Time Dependent Plasticity (STDP) [79], spiking backpropagation [124], and even reservoir computing models [57, 75] to train SNNs.

Neuromorphic techniques are also increasingly being featured in robotic applications [53]. Whether they're being leveraged for PID motor control and drone control [130] or robotic arm control [29] or related Mujoco [134] applications [34, 30, 144], spiking neural networks' real-time control ability is progressively being realized.

2.3 Event-based Camera Simulators and Datasets

Event-based cameras are expensive sensors. Because of this, there is work across several different software projects that aim to simulate the event output of event-based cameras. In general, the software projects take as input a series of frames as images or as a video and then produce an event stream as output. One of the first projects to do this was pioneered by Kaiser et al. in [61]. The authors developed a plugin for the Gazebo simulation software, which was used for autonomous vehicle simulation. The plugin was simple in that, to imitate events, the difference of a pixel value between frames was thresholded to determine if an event "occurs" at that pixel. Improvements to this simple model were seen in ESIM [106], software which more closely modelled the event-based sensors by using the log luminance difference between frames to determine whether an event occurs. Additionally, to address the roughly 1Mhz temporal resolution of the sensors, an event's timing between two frames was linearly interpolated based on the magnitude of the of a pixel's difference. Newer and more complex software models that strive to better emulate event-based cameras are seen in the V2E [55] software and ICNS [59] project.

Aside from projects that aim to reproduce event-based camera outputs from conventional frame-based inputs, there are projects that aim to tie event-based vision to autonomous vehicles (as in [61] with Gazebo). The CARLA [33] project features simulated event-based vision for agents that are developed to control a car, and the AirSim [123] project does the same but for drone control. In a similar sense, there are a few libraries that support general control application agent development using DVS inputs, one of which is Mujoco-ESIM [93], which extends the Mujoco application suite to provide simulated DVS inputs to agents using the ESIM event simulation software.

For conventional datasets, there are two main kinds of DVS datasets. The first kind of datasets are conventional frame-based datasets converted to DVS versions by moving a camera around the datasets' images on a monitor (like Cifar10-DVS [67], N-Mnist and N-Caltech101 [92], and N-Imagenet [65]). The other kind

of datasets are created natively using some sort of DVS camera, like the DVSGesture dataset [5] which features different users performing different gestures and the N-cars dataset [125] which features cars at different poses, speeds, and backgrounds. There are also American Sign Language (ASL) datasets such as ASL-DVS [13] and the SL-Animals-DVS [136] dataset. Similar to the ASL datasets are those involving pose estimation[20, 85, 108], facial expression recognition [12], and action recognition like EV-Act [44] and DailyAction-DVS [72]. Some of these datasets have been trained with neuromorphic techniques or spiking neural networks, but many have been leveraged with conventional training algorithms instead.

Perhaps one of the most prominent applications for event-based neuromorphic vision is in the autonomous vehicle space. This is largely because spiking neural networks can offer real-time decision making while processing immense amounts of sparse data, a necessary facet when applying dynamic vision sensors to autonomous vehicles. The DET dataset [24] is a dataset created with a DVS camera that focuses on lane extraction, and Viale et al. used it to develop SNN models on Loihi [137]. It features complex traffic scenes with various labeled lane types. The goal for models developed on this dataset is to be able to identify the lanes. MA-VIED [82] is a driving dataset that collects driving scenarios across a variety of sensors like IMU sensors and GPS antennas in addition to event cameras. Some other notable driving datasets include DDD17 [14], MVSEC [146], DSEC [46], and the 1Mpx Automotive Detection Dataset [100].

2.4 Event-based Camera Processing

When processing events from event cameras, one can process the events asynchronously as they arrive event-by-event or in batches or groups of events. The difficulty in processing the events asynchronously is how sparsely the events are generated due to event cameras' high temporal resolution. The events lack spatial context which renders traditional convolutional neural networks and feature

extractors ineffective. Two of the most prominent methods for event-by-event processing are filters and SNNs. Filters have been applied to problems like noise reduction [17, 62, 51], image reconstruction [86, 8, 139], and even Simultaneous Localization and Mapping (SLAM) problems [43, 63, 64, 22]. For SNNs, much of the work has involved starting with some sort of rigidly defined ANN, training it on labelled data, and then converting the model to an SNN that processes the data event-by-event [99, 90, 32].

For processing events in batches, there are several different types of representations. By far the most popular approach for processing batches of events is to create event frames or images so that traditional deep learning approaches can be leveraged [42, 147, 78, 89]. Other representations include time surfaces, which are maps where each pixel stores a time value to represent intensity of motion [135, 84, 145] and voxel grids, which are 3D histograms of events that preserve the temporal information of events by not collapsing a batch of events into a 2D frame or representation [107, 7, 138]. While there are other approaches that also seek to preserve spatio-temporal information or motion, a 2D representation of some time slice of events remains to be one of the most popular approaches for leveraging conventional computer vision algorithms.

Spiking neural networks can be leveraged on neuromorphic hardware, and because event cameras are "neuromorphic cameras", it is natural to pair SNNs with event cameras. Their shared biological influence and sparse event processing paradigm reinforce the pairing. The events generated by DVS camera pixels can be thought of as spikes being applied to input neurons in a spiking neural network. An example of a more biologically inspired system operating on event camera data leverages SNNs trained using STDP to perform tasks like object detection [2, 1]. Object detection or recognition of event camera data is a popular application of spiking neural networks. Other works train spiking neural networks with alternative, less biologically-inspired learning algorithms [71, 74, 72]. Another popular application space for SNNs and event cameras is optical flow, which is the apparent motion of objects in a scene

caused by relative motion between the camera and moving objects [91, 96, 52, 6]. The low latency of event cameras and neuromorphic hardware also encourages their applications to real-time robotic control systems as in [23, 129]. Though there are several domains where pairing spiking neural networks with event cameras makes sense and has shown promise, the research is still pretty sparse. This is likely partially due to ongoing challenges associated with natively training spiking neural networks [116].

2.5 TENNLab Software Framework

The TENNLab Neuromorphic Computing Framework [103] facilitates the development of spiking neural networks to accomplish a variety of tasks. The framework is composed of several layers that are shown in Figure 2.1. A user may choose or define an application which encompasses some sort of problem to be solved by an AI agent (e.g. classification, control theory, event detection, etc.). The framework includes a suite of applications used for benchmarking that feature observations derived from LIDAR [102], accelerometers, and more. The framework also supports several notable training algorithms such as Whetstone [122] which is a backpropagation algorithm that trains a traditional artificial neural network and slowly "sharpens" the activation functions into thresholding functions. Spike timining dependent plasticity [127], reservoir computing [109], and the ability to convert random forests to spiking neural networks [120] are also supported.

The most prominently used learning algorithm, however, is the Evolutionary Optimization of Neuromorphic Systems, or EONS [117]. EONS is a genetic algorithm that applies genetic operators like crossover and merge to populations of networks over a series of epochs. One of these training algorithms is selected to train an agent on a specific application, and one of several hardware architectures/devices is used to define the resultant network structure and parameters. The number of neuromorphic devices the framework supports is numerous and includes, but is not

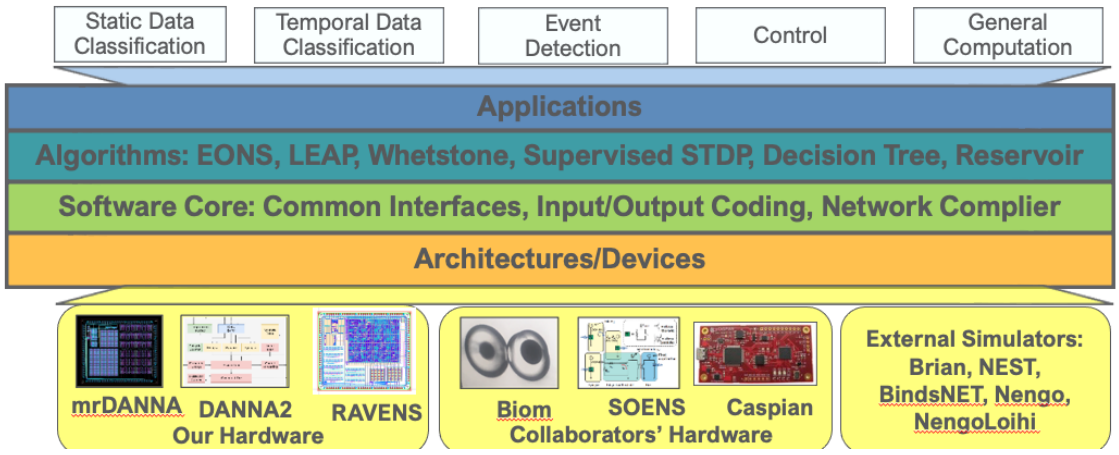


Figure 2.1: Figure illustrating the TENNLab Neuromorphic computing framework derived from [114]. Figure provided courtesy of Dr. Catherine Schuman.

limited to, an optoelectronic model in SOENS [18], popular simulators like Brian [50], NEST [49], and Nengo [9], and even some TENNLab proprietary device models like DANNA2 [80], a memristive version of DANNA, mrDANNA [19], and the most recent one, RAVENS [39]. The software core is the glue that combines each component and facilitates spiking neural network development for a wide variety of applications, algorithms, and neuromorphic devices.

2.6 RAVENS Neuroprocessor

The RAVENS neuroprocessor [40, 39] is the workhorse neuroprocessor of the TENNLab neuromorphic computing group. It has been designed for both digital and analog implementation, supporting deployment on ASICS, memristive hardware, FPGAs, and even microprocessors. It supports a standard Leaky Integrate and Fire (LIF) neuron model and has configurable STDP [127]. Our work leverages the simple LIF neuron model whose components are its threshold, leak value, and refractory period. For the LIF neuron model, when a neuron's charge surpasses its discrete threshold value, the neuron fires. Upon firing, the neuron enters a parameterized, dichotomized refractory period. At the end of its refractory period, the neuron's potential is simply reset to its standard resting potential. Each neuron in a RAVENS network has its own threshold value, refractory period values, and constant, linear leak value. If enabled, a neuron will leak away x units of charge at each timestep, and if disabled, the neuron will not leak away any charge at all. We have been able to run RAVENS on an FPGA at a speeds ranging between 1 and 100 MHz. Without loss of generality, in this work, we assume that a timestep or cycle of RAVENS is roughly 1 microsecond.

The synapse model for RAVENS is extremely straightforward and composed of only a discrete weight value and a discrete delay value. As an example, let's suppose we have two neurons with a synapse connecting them whose weight value is 3 and delay value is 4. If the presynaptic neuron fires at timestep 1, a spike whose charge

value is 3 will arrive at the postsynaptic neuron at timestep 5. The weight value for a synapse can be negative, meaning that the synapse is inhibitory to the postsynaptic neuron. Practically speaking, the smallest delay value a synapse may have is 1. This means that it will take only 1 cycle for a spike to arrive at a postsynaptic neuron.

Figure 2.2 illustrates the bio-inspired refractory period. After a neuron's threshold is exceeded and it fires, it slips into its absolute refractory period. During this period of time, the neuron will accumulate no charge, even if spikes are arriving at the neuron. After the absolute refractory period ends, the neuron enters its relative refractory period for some set number of cycles. At the beginning of this period, the neuron's potential is set to its parameterized refractory resting potential. During this period, the neuron can resume accumulating charge as normal, but the shift in resting potential for this period adjusts the neuronal threshold. This means that during the relative refractory period, a neuron can fire, but because the refractory resting potential is typically lower than the standard resting potential, it takes more activity and charge to cause it to fire than it normally would. At the end of the relative refractory period, if the neuron's potential is less than its standard resting potential, then the neuron's potential is reset to its standard resting potential, which is typically defined as being 0. For our work, we only leverage the absolute refractory period.

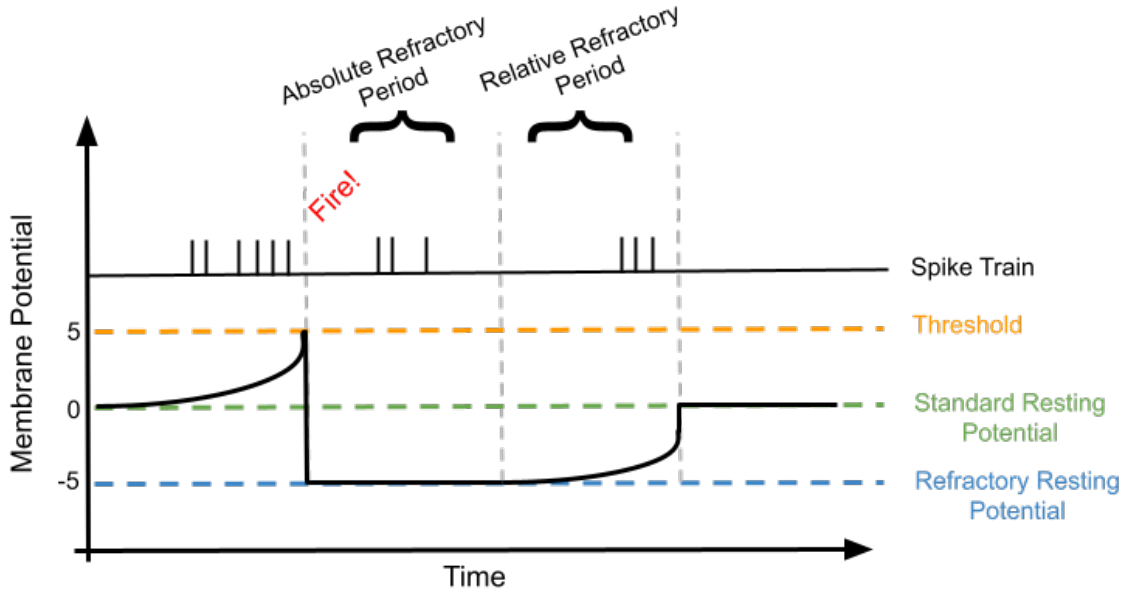


Figure 2.2: The RAVENS neuroprocessor implements a dichotomized neuronal refractory period. The figure shows an example of a neuron's membrane potential over time. The neuron's threshold is five, and after receiving six spikes, the neuron fires. The neuron's potential is set to its refractory resting potential of negative five and its absolute refractory period begins. During this period, the next three spikes that it collects are ignored. Once the relative refractory period starts, it resumes accumulating charge, and at the end of the relative refractory period, because the neuron's accumulated potential is less than its standard resting potential, the neuron's potential is set to its standard resting potential.

Chapter 3

Preprocessing Networks

In this chapter, we present four kinds of hand-tooled spiking neural networks that perform unique functions. The set presented in the first section focuses on regional event processing and downsampling input signals to produce more manageably sized observations, while the networks presented in the second section are focused on per-event filtration and classification. Most of the content in this chapter was presented in previous publications [110, 112].

3.1 Regional Event Processing and Dimensionality Reducing Spiking Neural Networks

© Owner/Author — Charles P. Rizzo 2023. The content in this section is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record (with the exception of one subsection) was published in [112], https://doi.org/10.1145/3584954.3584962.

Any network that has an input neuron for each pixel of an event-based camera will result in an a large spiking neural network that will be cumbersome to train on real-time tasks. While one can manually select smaller sub-regions of interest within the input space to operate on, as is done in [25], the question remains: what if most or

even all of the camera's observation space is important for an agent to make a decision? Taking inspiration from the way convolutional neural networks use a convolving kernel and pooling to reduce the dimensionality of the image being classified, this section aims to provide methods for downsampling a camera's event stream so that the spiking neural networks that make decisions based on the EBC inputs can do so with a more tractable input space. Furthermore, taking inspiration from spiking convolutional kernels [105], we reiterate that these spiking neural networks are hand-tooled. Before defining the networks, we demonstrate the convolution-like operation for processing the event-based camera data which we call the chipping methodology.

3.1.1 The Chipping Methodology

The output space of the DAVIS346 camera is very large and can be difficult for algorithms to operate on. To address this challenge, we break up the space into sub-regions, or chips. The chipping approach is defined by the following parameters: a **Chip_Size (r $\times$ c)** and a **Stride (s)**.

A chip is of size $r \times c$ pixels, and a new chip is created every s pixels. In that way, the stride value determines if there will be any overlap between the chips. This is functionally similar to the way a kernel convolves over an image in convolutional neural networks. To illustrate the chipping process, supposing we start at pixel location $i = 0, j = 0$, a chip is created that includes pixels $i = [0, r-1], j = [0, c-1]$. Using the stride value s, our next chip includes pixels $i = [0, r-1], j = [s, s+c-1]$. In this way, we sweep across the camera's output space (or the network's input space) in a row-major order fashion and create chips for each unique timestamp from the camera. Depending on the values of r, c, and s with respect to the camera's dimensions, it might be necessary to logically pad a chip with 0's. For example, if $r = c = s = 10$ and we use the DAVIS346 camera as an example, the 346 columns are not perfectly divisible by the chip size and stride. Therefore, the chips at the ends of each row will contain the six columns of pixels from the camera and 4 additional, padding columns

of 0 (or null) values. We'd like to stress that the chips are a logical organization of pixels that we use to help downsample the camera's event stream. We do not create chips from events, but use the chipping concept to filter the events.

Our primary downsampling operation is thresholding. We select a threshold t, and identify chips whose number of events meet or exceed the threshold. Camera events typically have polarity – positive when the pixel increases in intensity and negative when it decreases. We may choose to count only positive polarity events or to count all events without regard to polarity.

Figure 3.1 illustrates the chipping approach across an input space that is 20 rows by 30 columns. In this example, $r = c = 10$, $s = 5$ and $t = 7$. No padding is necessary here since both 20 and 30 are divisible by the stride value 5. The thicker grid lines denote the various chip boundaries, and three chips are identified as having events that meet or exceed the threshold. They are differentiated as either ignoring polarity or only counting events with positive polarity.

We denote the number of rows in the camera as cr and the number of columns as cc. For the DAVIS346 camera, $cr = 260$ and $cc = 346$. Generically for any size camera dimensions, the number of chips created per timestamp can be calculated by the following formula:

$$ceil\left(\frac{cr}{s}\right) * ceil\left(\frac{cc}{s}\right) \tag{3.1}$$

The benefit to the chipping approach is that regions of the input space identified as important (or having above a certain threshold of activity) can be passed on for further processing, while empty or sparsely noisy regions can be discarded. In this way, the network acts as a denoiser and reduces the amount of data that needs to be processed down the line (e.g., for classification). However, it is imperative that the network architecture be optimized in order to reduce the amount of time needed to process each chip.

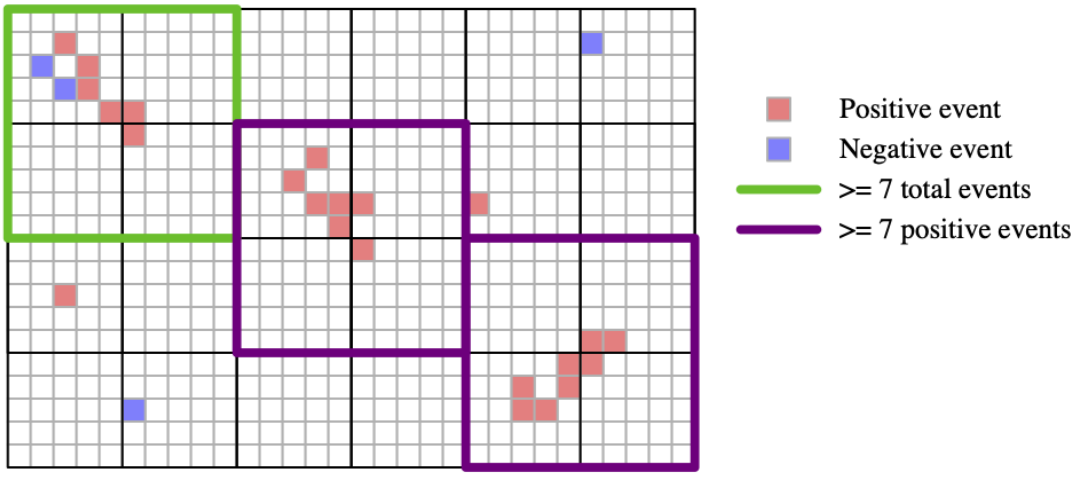


Figure 3.1: Figure showing how 10×10 chips are created when the stride is equal to 5. This figure was sourced from prior work [112].

3.1.2 Spiking Threshold Pooling (Event Counting) Neural Networks

We present three networks for counting events in chips. For these networks, we use the following parameters. The chip size and threshold are defined by r, c and t as before. We use an additional parameter f to denote a *timestamp coalescing* value. The first counting network is a **general counting network**. It has rc input neurons, one for each pixel in the chip. An input neuron receives a spike at timestep i if the camera has generated an event for the corresponding pixel at timestep i. The spike's value is 0 or 1, depending on the event's polarity. There is one output neuron in the network. If, over f timesteps, there are at least t events, then the output neuron spikes once. If there are less then t events, then the output neuron does not spike. There is a *network refractory period*, which is a number of timesteps before the network may be reused.

The purpose of the timestamp coalescing value is so that the network may work over discrete regions of time. Event-based cameras may operate at high temporal resolutions (around 1 μs)[138], meaning the event activity at any one unique timestamp is likely to be so sparse as to be of little use for an agent. Allowing the agent to coalesce several consecutive timestamps worth of input alleviates this issue. The trade-off with having a higher value of f is that it takes longer for the network to process the events.

The second counting network performs the same task as the general counting network, but limits f to be one. This limitation simplifies the network and has no network refractory period.

The third counting network is a **pixel counting network** that only counts one event for each pixel over the f timesteps. Like the general counting network, it has a network refractory period.

In the next subsection, we define the three counting networks precisely and provide examples. Figure 3.2 details how to read the networks presented in the next few figures.

General Counting Network Specification

The general counting network has rc input neurons labeled $I_{i,j}$, rc hidden neurons labeled $H_{i,j}$ and a single output neuron O. The input neurons have thresholds of -1, no leak and no refractory periods. The hidden neurons have thresholds of 0, no leak and no refractory periods. The output neuron has a threshold of t, no leak and an absolute refractory period of $2f - 1$ timesteps. There are three sets of synapses:

1. $I_{i,j}$ to $H_{i,j}$ with a weight of 1 and delay of 1.
2. $I_{i,j}$ to O with a weight of 1 and delay of 1.
3. $H_{i,j}$ to O with a weight of -1 and delay of f.

This network implements general counting as defined above, and has a network refractory period of $2f - 1$ timesteps. We provide an example below to illustrate the mechanics of the network.

Example 1: $r = c = 3$, $t = 4$, $f = 5$

We show the general counting network for these parameters in Figure 3.3. To declutter to drawing, we denote the I, H and O neurons by different colors, and do not show the subscripts.

To demonstrate, first suppose there are input spikes at timesteps 0, 1, 2, 3 and 4. It is immaterial which inputs actually spike, so we don't specify the i and j variables of $I_{i,j}$. We describe the network's activity as follows:

- Each of the inputs causes its corresponding $H_{i,j}$ to fire one timestep later, so at timesteps 1, 2, 3, 4 and 5, a neuron $H_{i,j}$ fires.

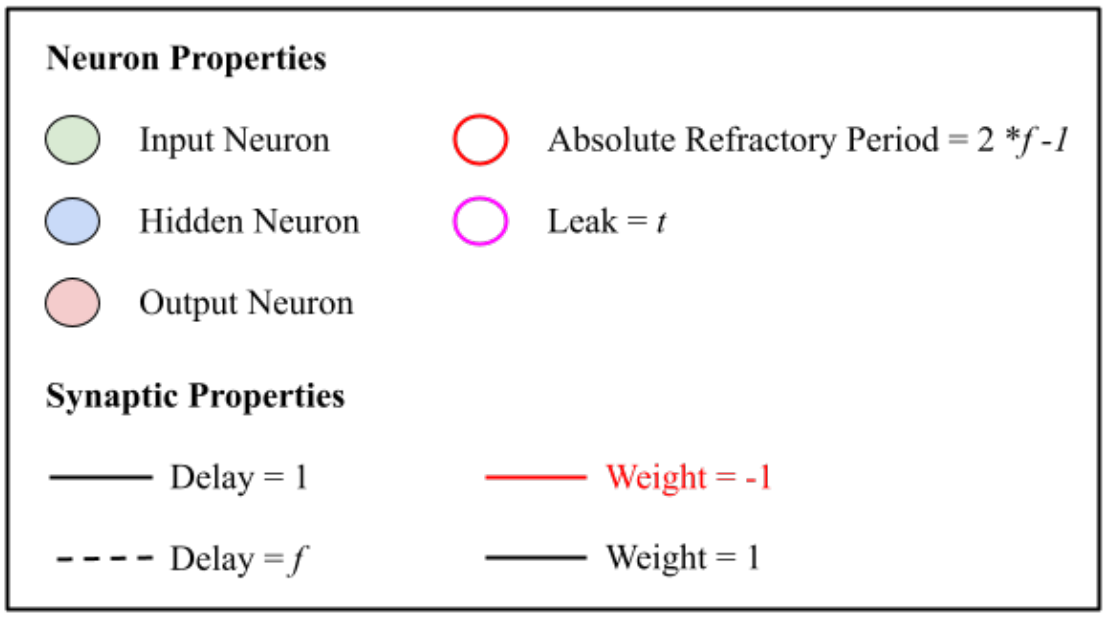


Figure 3.2: Legend detailing how to read networks in Figures 3.3 and 3.4. This figure was sourced from prior work [112].

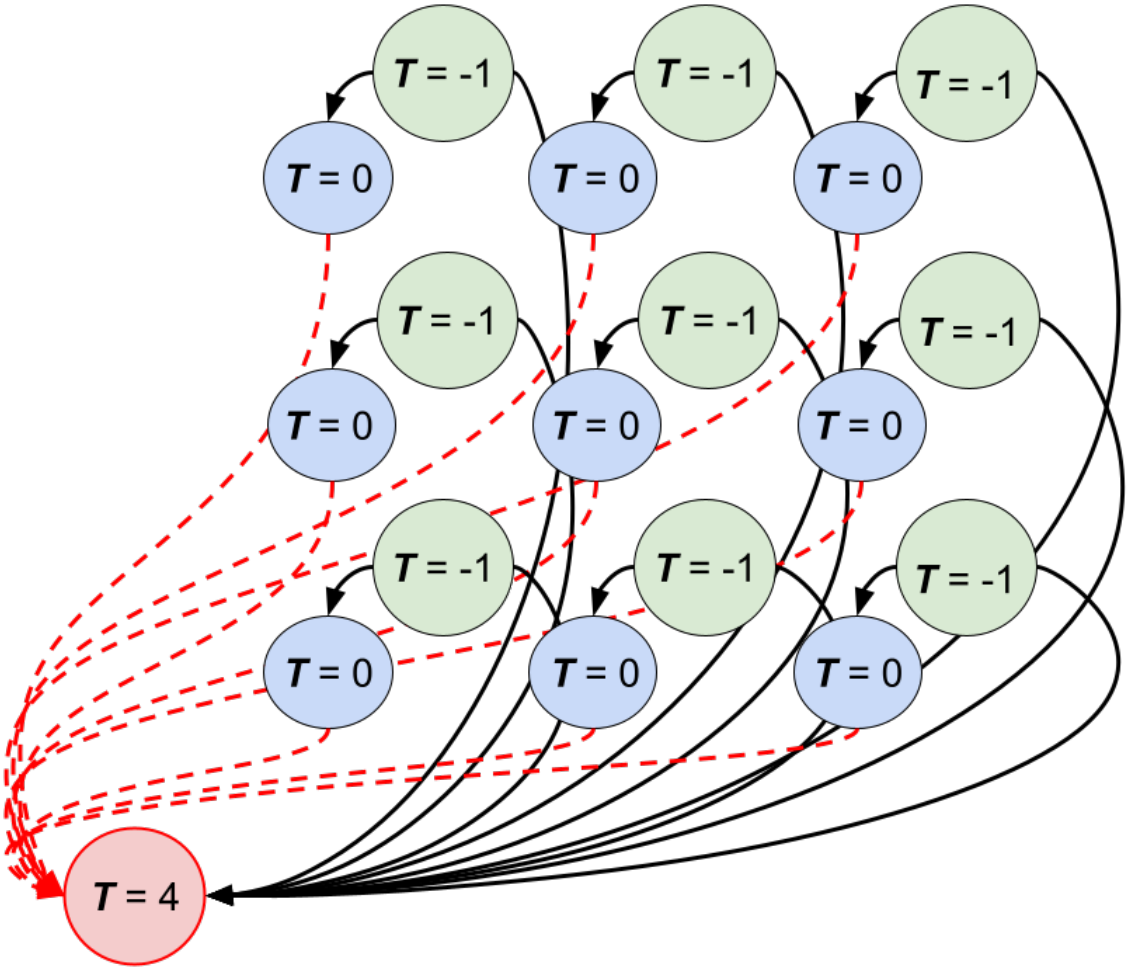


Figure 3.3: Network for Example 1 in Section 3.1.2. Features 3×3 input space where the threshold to fire is $t = 4$. This figure was sourced from prior work [112].

- The input neurons also cause O to receives spikes at timesteps 1, 2, 3, 4 and 5.
- At timestep 4, O's potential exceeds its threshold and it fires, denoting that four events have occurred on the chip. It enters its refractory period, which ends at timestep 13.
- The spikes from the $H_{i,j}$ arrive at O at times 6, 7, 8, 9 and 10. They are all ignored, because O is in its absolute refractory period.
- The network may be reused after timestep 13.

Now, suppose there are only input spikes at timesteps 0, 1 and 4. Spikes from $I_{i,j}$ arrive at O at times 1, 2 and 5. Since its potential is only three, O does not fire. Spikes from $H_{i,j}$ with values of -1 arrive at times 6, 7 and 10, resetting $O's$ potential to zero. The network may be reused after timestep 10.

To summarize, the network has $2rc + 1 = 19$ neurons and $3rc = 27$ synapses. It fires once during timesteps 1 to $f = 5$ if at least t input events have occurred, and has a network refractory period of $2f - 1 = 9$ timesteps.

Optimized Counting Network Specification

When $f = 1$, we have a special case of the network architecture that's smaller and more efficient than the network above. The input neurons are the same as those in the general counting network. However, there are no hidden neurons, and the output neuron doesn't have an absolute refractory period. Instead, the output neuron uses a leak value that is equal to its threshold t. There is only one set of synapses:

1. $I_{i,j}$ to O with a weight of 1 and delay of 1.

When $f = 1$, there is no network refractory period because any charge at the output neuron that does not cause a fire event is leaked away by the next timestep.

Example 2: $r = c = 3$, $t = 4$, $f = 1$

We show the improved network in Figure 3.4. To demonstrate, suppose five different input neurons each fire at timestep 0. At timestep 1, the output neuron O receives the five spikes from the five input neurons that fired. Since O's potential has exceeded its threshold, it fires at timestep 1.

Now suppose only three different input neurons fire at timestep 0. At timestep 1, O accumulates the three units of charge but, since its threshold is not exceeded, it does not fire. Instead, O leaks away t units of charge. At the start of timestep 2, O's potential has been reset to 0, and it is ready for reuse. Since there is no network refractory period, the network can process new inputs at every timestep.

To summarize, the network has $rc + 1 = 10$ neurons and $rc = 9$ synapses. It only fires at timestep 1 if at least t input neurons have received an event and fired at timestep 0, and it has no network refractory period: it can be reused immediately. When $f = 1$, the network's footprint is smaller, and it takes a minimal two cycles to process events.

Pixel Counting Network Specification

Alternatively, a network can count the number of unique pixels that fire in a chip or sub-region. For a sufficiently large enough chip size relative to the camera's dimensions and over a large enough collection of timestamps f, this could be indicative of motion at a specific chip location. This network might be more advantageous to use in datasets where motion capture is desirable instead of overall event activity. As with the general counting network, the pixel counting network is composed of rc input neurons, rc hidden neurons, and a single output neuron O. The hidden neurons and output neuron are parametrically the same as those in the general counting network. However in the pixel counting network, the input neurons $\mathrm{I}_{i,j}$ have absolute refractory periods equal to f. It has the same three sets of synapses that the general counting network does, and the network refractory period is $2f - 1$ as well.

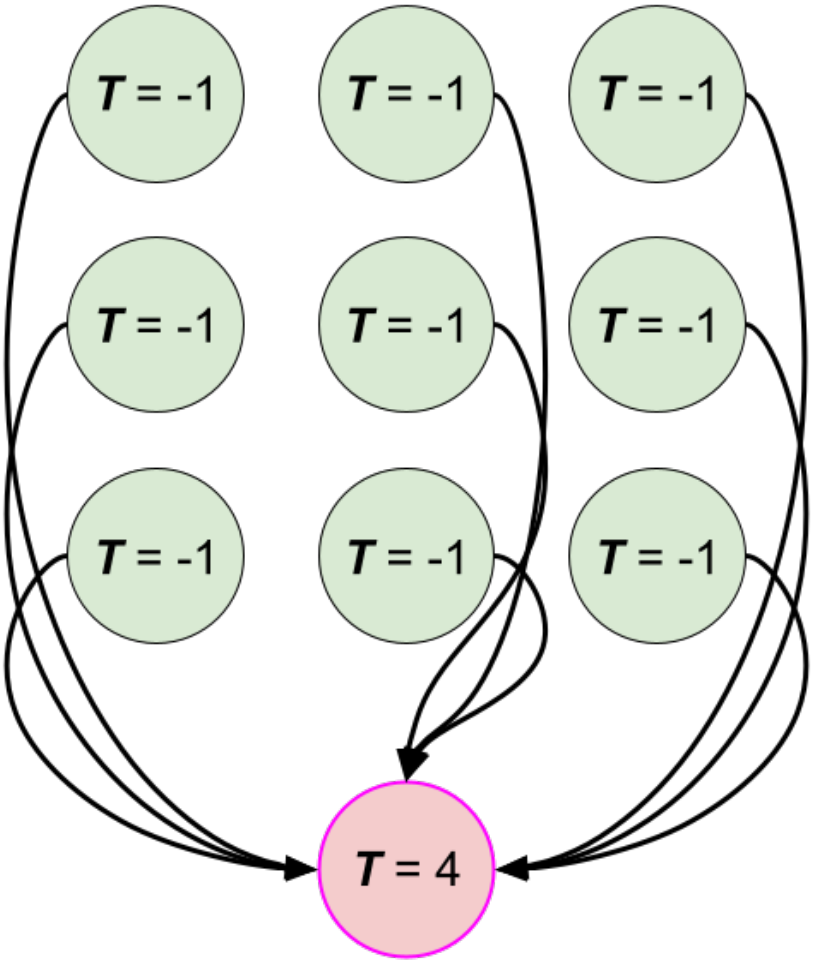


Figure 3.4: Optimized network architecture for Example 2 in Section 3.1.2. Features 3×3 input space where the threshold to fire is $t = 4$, and uses neuronal leak on the output neuron to clear any leftover charge. This figure was sourced from prior work [112].

The example in Section 3.1.2 works nearly identically for the pixel counting network. The only difference is when an input neuron $I_{i,j}$ fires, it immediately enters its absolute refractory period for f timesteps and will ignore any other input events during that period. So, for the output neuron O to fire between timesteps 1 and f, at least t different input neurons $I_{i,j}$ need to fire.

To summarize, the network has $2rc + 1$ neurons and $3rc$ synapses. It fires once during timesteps 1 to f if at least t input events have occurred across t different input neurons, and has a network refractory period of $2f - 1$ timesteps.

Practical Application of Event Counting Networks

This downsampling technique, also referred to as Spiking Threshold Pooling, is a way to downsample DVS camera data by applying the traditional pooling operation (by virtue of using one of the defined counting networks in Section 3.1.2) over the camera window but in a spiking manner. At a high level, a camera's output space (128×128 in the case of the DVSGesture dataset) is divided into chips or sub-regions that are valued at either 0 or 1 depending on the amount of events that occur within that region. We want to stress that we are using this technique as a means of organizing the events as they come from the camera. We are not building out event-frames in memory and striding over them performing convolution, as other works have done [141, 47, 5]. Our goal is to preserve the asynchronous, event-based characteristic of DVS camera data.

As an example, suppose we define an event counting network where the chip size is 10×10, the stride value is 10, and the threshold value is 5. For the 128×128 observation space of the DVS128 camera, when we apply the spiking threshold pooling operation, we reduce the observation space to

$$ceil\left(\frac{128}{10}\right) * ceil\left(\frac{128}{10}\right) = 13 * 13 = 169 \tag{3.2}$$

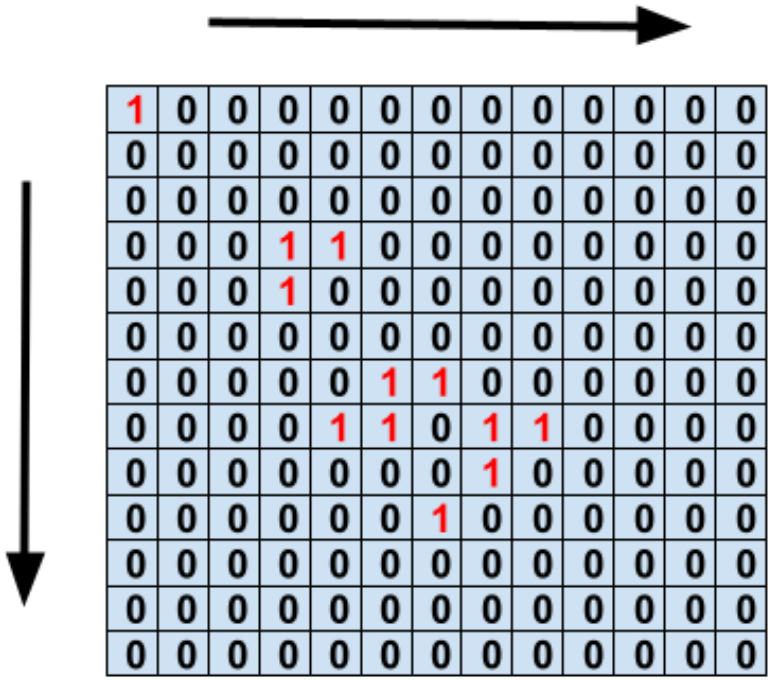
binary-valued observations. The chip value is 1 (or "of interest") if more than 5 events occur in that sub-region of the camera's observation space over some length of time. It is 0 otherwise. Effectively, the counting network is doing nothing more than counting or thresholding event activity at a certain region over some time interval. The benefits of this technique are that the counting network can be tiled and organized into a layer, producing a downsampled input signal that can be used as either inputs to successive downsampling passes or to a classification agent. An alternative to tiling the counting network is to time multiplex the input space and have only one counting network process the entire input space. Both of these approaches can be implemented neuromorphically which facilitates real-time classification by placing layers of downsampling networks behind a camera's output to reduce the input space for the final classifier network.

Additionally, all three types of counting networks may be restricted to counting only positive polarity events by changing the input neuron thresholds to one. Since negative polarity events have charge values of zero, they are ignored by the inputs.

3.1.3 Row/Column Collapse Network Specification

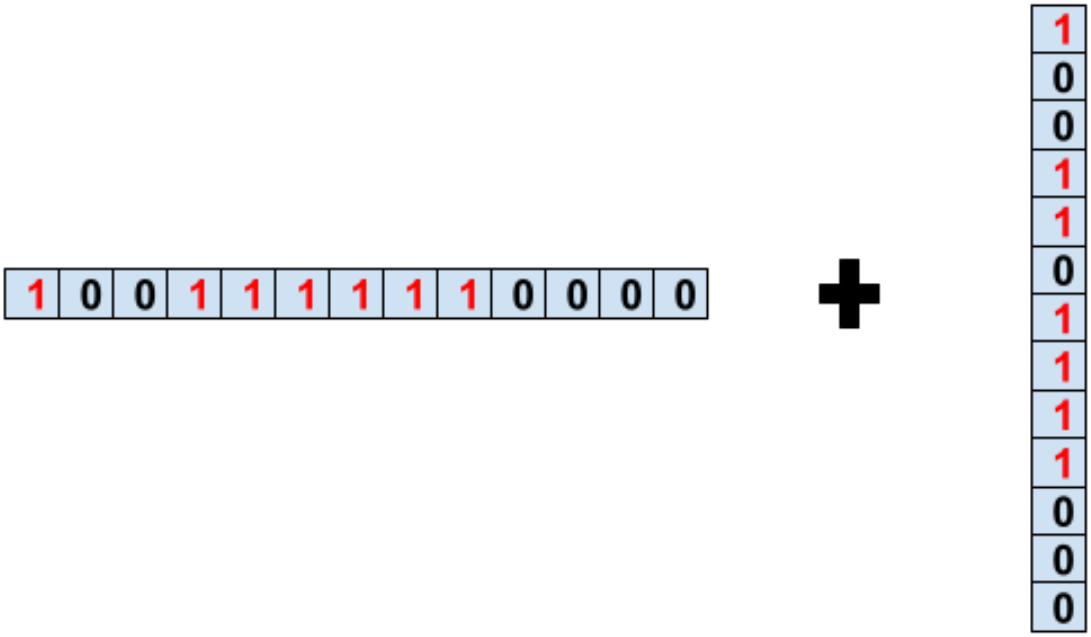
© Owner/Author — Charles Rizzo 2023. The content in this subsection is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record was published in [110], https://dl.acm.org/doi/10.1145/3589737.3605999.

The row/column collapse technique flattens a two dimensional observation space by effectively performing an 'OR' operation across the rows and columns. This would be applied after a series of zero or more of spiking threshold pooling (or downsampling) layers and immediately precede the classifier. Continuing from the example shown in Equation 3.2, suppose after the sole layer of downsampling that we perform a row/column collapse. The 13×13 observation space is then transformed to a $13 + 13$ sized observation space. Figure 3.5 shows what this operation looks like intuitively.



(a) 13×13 reduced observation from Equation 3.2. 1's imply chips of interest; 0's are ignored.

Figure 3.5: This figure was sourced from prior work [110].



(b) Collapse down the rows by performing an 'OR' operation down each column. Do the same for the columns by performing an 'OR' operation along each row.

(c) Resultant observation is $13 + 13 = 26$ features where the first 13 correspond to the row collapsed values and the last 13 correspond to the column collapsed values.

Figure 3.5: (continued)

In the case of the DVSGesture dataset, different users perform the "right hand wave" gesture at different heights. The classifier might classify waving at the top of the observation space as something different from waving at the bottom of the observation space. With this technique, the rows and columns are flattened so that the signal for waving at the top of the observation space is the same as waving at the bottom of the observation space. Another benefit, like with the spiking threshold pooling mechanism, is that this technique can be performed by a spiking neural network which further facilitates a real-time, neuromorphic classification pipeline.

The spiking neural network for this technique, assuming an $r \times c$ observation size immediately preceding it ($I_{r,c}$), contains $r + c$ neurons and $2 * r * c$ synapses. The $r + c$ neurons are both input and output neurons, and they're defined as two sets:

1. $N_{0,c-1}$
2. $N_{c,r+c-1}$

Similarly, there are two sets of synapses defined as:

1. For $0 \leq i < r$ and $0 \leq j < c$, $I_{i,j}$ to $N_{j \in [0,c-1]}$
2. For $0 \leq i < r$ and $0 \leq j < c$, $I_{i,j}$ to $N_{c+i \in [0,r-1]}$

The neuronal thresholds are all 1 and the the synaptic weights and delays are also all 1. Neurons [0,$c-1$] each have r incoming synapses, and neurons [c, $r+c-1$] each have c incoming synapses. As the example in Figure 3.5 shows, if a column or row in the observation space has 1 or more chips of interest (or events if this is applied directly behind the camera's output), that column or row is represented by a neuron fire (a red 1 in Figure 3.5c) in the row/column collapse network. An example network is not shown because is simply a row of $r+c$ neurons that are connected as described to some preceding $r * c$ array of neurons.

3.2 Per-event Processing Networks

The work in this section is in preparation for publication. With the exception of one subsection, this work is available on ArXiv [113], https://arxiv.org/abs/2401.15212.

Instead of processing regions of an event camera's output and downsampling its output, we can instead produce networks that operate on a per-event basis. These networks do not change the dimensionality of the camera's output, and instead work at the event level to perform some sort of filtration or event denoising and event clustering. This is useful for a system in which the neuromorphic component acts as a preprocessing component before the data gets to a more heavyweight machine learning algorithm. Because these networks are more complex algorithmically than the spiking threshold pooling networks and the row/column collapse network, we also include spike tables to help explain the behavior of each proposed network at each timestep or hardware cycle.

3.2.1 Speed Filter Network

The speed filter function is a technique that filters out events based on their "speed". An input event x's speed is determined by the amount of events that occur within some radius epsilon (ϵ) of event x over the course of two timesteps. If some number of events occur within event x's radius that is greater than the speed threshold t, we say event x is moving fast. However, if the number of events is less than or equal to the speed threshold t, we filter out event x as moving too slow. Alternatively, we can identify events that are moving slower than the speed threshold t and filter out events moving faster than it. This filtering technique is functionally equivalent to the spiking threshold pooling (or counting) networks presented in section 3.1 but without the downsampling component. Even though we are technically thresholding neighboring event density around some event x, we are doing it over time and with

the understanding that events tend to represent regions or objects that are in motion. This is why we denote this technique as the "speed filter".

The "speed" filter is influenced by [87], in which Nagaraj et al. filter events from a DVS camera by constructing a spiking neuron layer whose dimensionality is the same as the camera. For a camera whose resolution is 640×480, this is over 307,000 spiking neurons. Additionally, they define the network's connectivity by "connecting" each neuron to nine pixels: the pixel it represents and the eight other pixels that are a Chebyshev distance of 1 away from it. This means that their speed filtration spiking neural network solution has over 2.7 million synapses. Instead, we present the same functionality as a modular network that can be parallelized and deployed to hardware systems that cannot dedicate such an immense quantity of resources to only event filtration.

For the network representation of the speed filter, there are three parameters that contribute to its architecture:

1. The speed threshold t_s
2. The radius epsilon ϵ_s (defined as the Chebyshev distance in this work)
3. Whether we filter out events that are moving faster or slower than the speed threshold

Slow Event Filter Network Specification

First, let us assume we filter out events that are moving slower than the speed threshold t_s. In defining ϵ_s as the Chebyshev distance, there are $(2 * \epsilon_s + 1)^2$ input neurons labeled $I_{i,j}$ and a single output neuron O. The input neurons $I_{i,j}$ have thresholds of 0, and the output neuron O has a threshold equal to the speed threshold t_s. It is worth noting that distance metrics other than the Chebyshev distance may be used (e.g. Euclidean, Mahalanobis, etc.), and only the number of input neurons is affected by this choice. The connectivity of the network is defined by one set of synapses:

1. $I_{i,j}$ to O with a weight of 1 and a delay of 1

Figure 3.6 illustrates what this architecture looks like for $\epsilon_s = 1$ and speed threshold $t_s = 10$.

Example 1

Let's suppose an event at pixel x,y occurs at time n and we want to determine whether or not to filter it out based on its speed. The current event's pixel is mapped to input neuron $I_{\epsilon_s,\epsilon_s}$, and the other adjacent pixels ranging from $x \in [x - \epsilon_s, x + \epsilon_s], y \in [y - \epsilon_s, y + \epsilon_s]$ simply map to the input neurons $I_{j\in[0,2*\epsilon],i\in[0,2*\epsilon_s]}$. The network cycles required to process the event at pixel x, y are detailed below:

1. **Cycle 0**: Apply events that occur at pixels $x \in [x-\epsilon_s, x+\epsilon_s], y \in [y-\epsilon_s, y+\epsilon_s]$ at time n - *1* to inputs $I_{j,i}$.

2. **Cycle 1**: Apply events that occur at pixels $x \in [x-\epsilon_s, x+\epsilon_s], y \in [y-\epsilon_s, y+\epsilon_s]$ at time n to inputs $I_{j,i}$. Current event x,y is applied to $I_{\epsilon_s,\epsilon_s}$ during this cycle.

3. **Cycle 2**: Output neuron O accumulates events as charge from events applied at cycle 0. If output neuron O's potential exceeds its threshold t_s, it fires.

4. **Cycle 3**: Output neuron O accumulates events as charge from events applied at cycle 1. If output neuron O's potential exceeds its threshold t_s, it fires.

The network requires a total of four cycles to process one event at pixel location x, y after which the network's state may be cleared so that it can process the next event. Table 3.1 shows an example spike table for Figure 3.6 where the event is not filtered out, and Table 3.2 shows an example spike table for Figure 3.6 where the event is filtered out.

Fast Event Filter Network Specification

If however we want to filter out events that are moving faster than the speed threshold t_s, we must slightly tweak the network architecture. In addition to the architecture

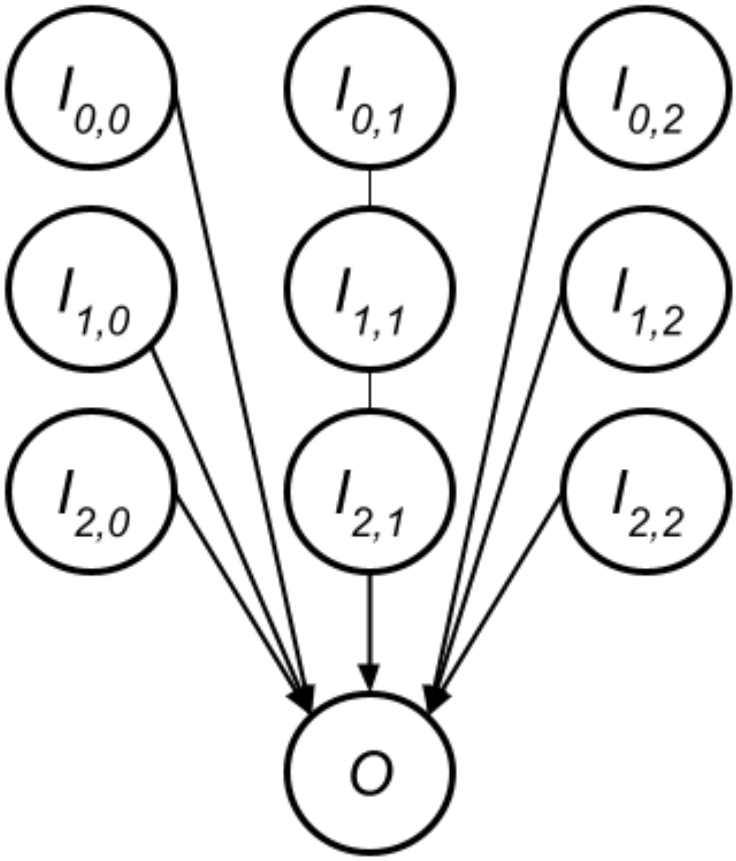


Figure 3.6: Speed filter network architecture with $\epsilon_s = 1$ that filters out events moving slower than speed_threshold $t_s = 10$. This figure was sourced from prior work [113].

Table 3.1: Example for speed filter shown in Figure 3.6. Current event of interest is applied to $I_{1,1}$ at timestep 1. Because more than the speed_threshold $t_s = 10$ events are applied between timesteps 0 and 1, output O fires at timestep 3. The event is not filtered out and passes through the network for downstream processing. This table was sourced from prior work [113]

Timestep	Apply spike to	*Fire*							
		$I_{0,1}$	$I_{1,0}$	$I_{1,1}$	$I_{1,2}$	$I_{2,0}$	$I_{2,1}$	$I_{2,2}$	O
0	$I_{0,1}$, $I_{1,0}$, $I_{1,2}$, $I_{2,1}$, $I_{2,2}$	-	-	-	-	-	-	-	-
1	$I_{1,0}$, $I_{1,1}$, $I_{1,2}$, $I_{2,0}$, $I_{2,1}$, $I_{2,2}$	*	*	-	*	-	*	*	-
2	-	-	*	*	*	*	*	*	-
3	-	-	-	-	-	-	-	-	*

Table 3.2: Example for speed filter shown in Figure 3.6. Current event of interest is applied to $I_{1,1}$ at timestep 1. Because less than or equal to the speed_threshold $t_s = 10$ events are applied between timesteps 0 and 1, output O does not fire. The event is filtered out. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*							
		$I_{0,1}$	$I_{1,0}$	$I_{1,1}$	$I_{1,2}$	$I_{2,0}$	$I_{2,1}$	$I_{2,2}$	O
0	$I_{0,1}$, $I_{1,0}$, $I_{1,2}$, $I_{2,1}$	-	-	-	-	-	-	-	-
1	$I_{1,0}$, $I_{1,1}$, $I_{1,2}$, $I_{2,0}$, $I_{2,1}$	*	*	-	*	-	*	-	-
2	-	-	*	*	*	*	*	-	-
3	-	-	-	-	-	-	-	-	-

already described, we add two neurons and three synapses. The first neuron is an additional bias input neuron we denote as I_b whose threshold $t = 0$. The second additional neuron is the new output neuron that we denote as O_n whose threshold $t = 2$. In our previous architecture, neuron O was our output neuron, but in this architecture, it is a hidden neuron. The connectivity is the same as described above but with the following additions:

1. I_b to I_b: This is a self-edge whose synaptic weight is 1 and delay is 1. This ensures that once neuron I_b fires, it continues to fire at each cycle until the end of the network simulation.

2. I_b to O_n with a weight of 1 and a delay of 1

3. O to O_n with an inhibitory synaptic weight of -1 and delay of 1

Figure 3.7 shows this modified architecture when $\epsilon = 1$ and the speed threshold $t = 10$. The dashed line in Figure 3.7 and latter figures represents an inhibitory synapse.

Example 2

Again, let's suppose an event at pixel x,y occurs at time n and we want to determine whether or not to filter it out based on its speed being too fast. The network cycles required to process the event at pixel x, y are detailed below:

1. **Cycle 0**: Apply events that occur at pixels $x \in [x-\epsilon_s, x+\epsilon_s], y \in [y-\epsilon_s, y+\epsilon_s]$ at time n - *1* to inputs $I_{j,i}$. Additionally, we apply a spike to the new bias neuron I_b

2. **Cycle 1**: Apply events that occur at pixels $x \in [x-\epsilon_s, x+\epsilon_s], y \in [y-\epsilon_s, y+\epsilon_s]$ at time n to inputs $I_{j,i}$. Current event x,y is applied to $I_{\epsilon_s,\epsilon_s}$ during this cycle. Bias input neuron I_b fires sending charge to itself and to output neuron O_n.

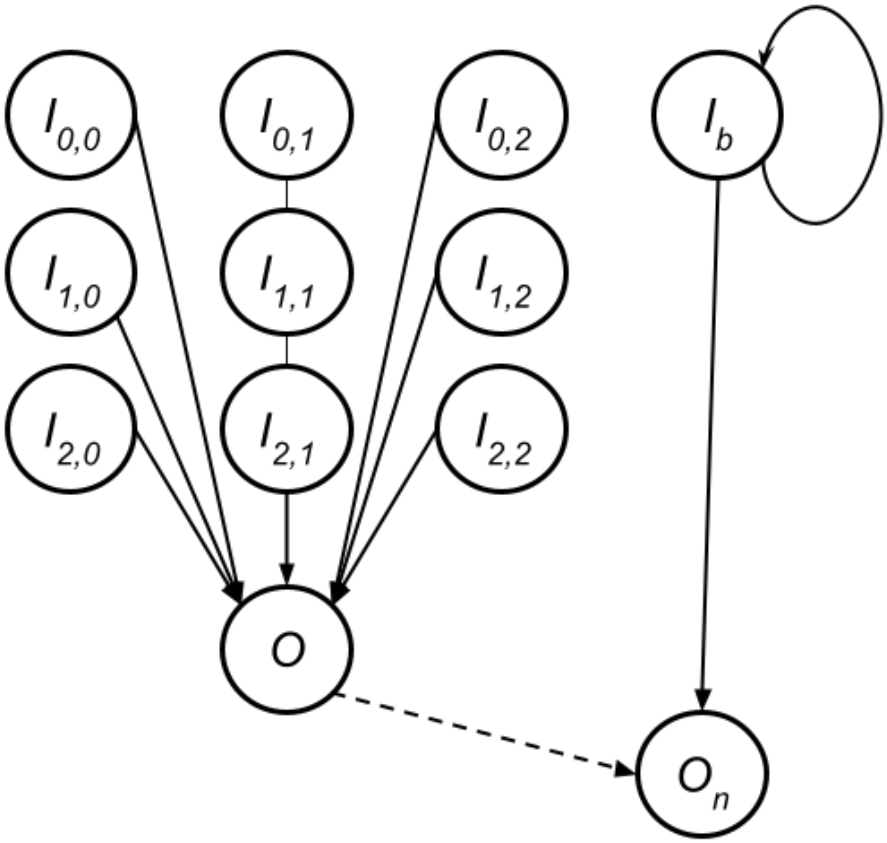


Figure 3.7: Speed filter network architecture with $\epsilon_s = 1$ that filters out events moving faster than speed_threshold $t_s = 7$. This figure was sourced from prior work [113].

3. **Cycle 2**: Hidden neuron O accumulates charge from events applied at cycle 0. If hidden neuron O's potential exceeds its threshold t_s, it fires with an inhibitory spike to output neuron O_n. Bias input neuron I_b fires sending charge to itself and to output neuron O_n.

4. **Cycle 3**: Hidden neuron O accumulates charge from events applied at cycle 1. If hidden neuron O's potential exceeds its threshold t_s, it fires with an inhibitory spike to output neuron O_n. Bias input neuron I_b fires sending charge to itself and to output neuron O_n.

5. **Cycle 4**: If no inhibitory spikes arrive at output neuron O_n from hidden neuron O, O_n fires meaning that we do not filter out the event. If instead one or more inhibitory units of charge arrive at neuron O_n from neuron O, the output neuron will not fire, and the event at x,y is filtered out.

Filtering out events that are moving faster than the threshold t_s requires that the network run for one additional cycle per event or five total cycles per event. Table 3.3 shows an example spike table for Figure 3.7 where the event is filtered out, and Table 3.4 shows an example spike table for Figure 3.7 where the event is filtered out. In summary, the following resources are required:

- To filter out events moving slower than speed threshold t_s:
  - **Neuron Count**: $(2 * \epsilon_s + 1)^2 + 1$ neurons
  - **Synapse Count**: $(2 * \epsilon_s + 1)^2$ synapses
  - **Runtime per Event**: 4 cycles
- To filter out events moving faster than speed threshold t_s:
  - **Neuron Count**: $(2 * \epsilon_s + 1)^2 + 3$ neurons
  - **Synapse Count**: $(2 * \epsilon_s + 1)^2 + 3$ synapses
  - **Runtime per Event**: 5 cycles

Table 3.3: Example for speed filter inverse functionality shown in Figure 3.7. Current event of interest is applied to $I_{1,1}$ at timestep 1. A spike is applied to the bias neuron I_b at timetstep 0, and it fires into itself and output O_n in each successive timestep. Because more than the speed_threshold $t = 7$ events are applied between timesteps 0 and 1, hidden neuron O fires at timestep 3. This fire directly inhibits output O_n from firing, and the event of interest is filtered out. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*							
		I_b	$I_{1,0}$	$I_{1,1}$	$I_{1,2}$	$I_{2,0}$	$I_{2,1}$	O	O_n
0	I_b, $I_{1,0}$, $I_{1,2}$, $I_{2,1}$	-	-	-	-	-	-	-	-
1	$I_{1,0}$, $I_{1,1}$, $I_{1,2}$, $I_{2,0}$, $I_{2,1}$	*	*	-	*	-	*	-	-
2	-	*	*	*	*	*	*	-	-
3	-	*	-	-	-	-	-	*	-
4	-	*	-	-	-	-	-	-	-

Table 3.4: Example for speed filter inverse functionality shown in Figure 3.7. Current event of interest is applied to $I_{1,1}$ at time step 1. A spike is applied to the bias neuron I_b at timetstep 0, and it fires into itself and output O_n in each successive timestep. Because less than or equal to the speed_threshold $t = 7$ events are applied between timesteps 0 and 1, hidden neuron O does not fire. Instead, by timestep 4, I_b has contributed enough charge to output O_n for it to fire, allowing the event to be passed through and not filtered out. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*							
		I_b	$I_{1,0}$	$I_{1,1}$	$I_{1,2}$	$I_{2,0}$	$I_{2,1}$	O	O_n
0	I_b, $I_{1,0}$, $I_{1,2}$, $I_{2,1}$	-	-	-	-	-	-	-	-
1	$I_{1,0}$, $I_{1,1}$, $I_{1,2}$	*	*	-	*	-	*	-	-
2	-	*	*	*	*	-	-	-	-
3	-	*	-	-	-	-	-	-	-
4	-	*	-	-	-	-	-	-	*

3.2.2 Density Based Clustering of Applications with Noise (DBSCAN) Network

Density-based spatial clustering of applications with noise (DBSCAN) [35] is a spatial clustering algorithm. It works by clustering all data points as either a core point, a border point, or a noise point. It clusters data based on the premise that spatially close data belongs to the same cluster. The clusters are not rigidly shaped and are composed of core points with border points forming the boundary of the cluster. For our use case, an event is a data point, and it is classified as either core, border, or noise based on the following:

- An event is a core event if in its radius epsilon (ϵ), at least threshold t (canonically referred to as min_points) events occur
- An event is a border event if in its radius epsilon (ϵ), between one and threshold t events occur where at least one of the events is a known core event
- An event is a noise event if neither of the prior conditions are met

Our network architecture has been validated against scikit-learn's [98] DBSCAN algorithm across a few different datasets including a few sequences from the DSEC dataset [46].

Network Specification

For the network representation of the DBSCAN algorithm, there are only two parameters that contribute to its architecture:

1. The threshold t_d or min_points value
2. The radius epsilon ϵ_d defined as the Chebyshev distance

Optionally, we can choose to recall and label noise events, but we argue that it makes more sense to filter them out as noise. Additionally, for our implementation of

the DBSCAN algorithm neuromorphically, the clusters are not labelled and identified. The events are only classified as either core or border events while noise events are filtered out.

We define two sets of input neurons $I_{i,j}$ and $I_{a,b}$. Each set of input neurons, like with the speed filter network, is composed of $(2 * \epsilon_d + 1)^2$ neurons, which results in a total of $(2 * \epsilon_d + 1)^2 * 2$ input neurons. The input neurons have thresholds $t = 0$. There are four hidden neurons and two output neurons. The output neuron O_c is the core output neuron and its threshold $t = 0$. The border output neuron is denoted as O_b and its threshold $t = 1$. There are also four hidden neurons denoted as $H_{i \in [0,3]}$ whose thresholds differ and will be explained in a later example that walks through the network activity. The connectivity is defined as follows:

1. $I_{i,j}$ to H_0 with a weight of 1 and delay of 1
2. $I_{i,j}$ to H_1 with a weight of 1 and delay of 1
3. $I_{a,b}$ to H_3 with a weight of 1 and delay of 1
4. H_0 to H_2, H_1 to O_c, H_2 to O_b, O_c to O_c, H_3 to O_b with a weight of 1 and delay of 1
5. H_1 to H_2, O_c to O_b with a weight of -1 and delay of 1

To better illustrate the connectivity, Figure 3.8 shows a network that performs the DBSCAN algorithm with an ϵ_d value of 1 and a threshold t or min_points value of 3.

Example 1

Like with our speed filter network examples, let's suppose an event occurs at pixel x,y and time n, and we want to determine whether this event is a core point event, a border point event, or a noise point event. Again, the current event's pixel is

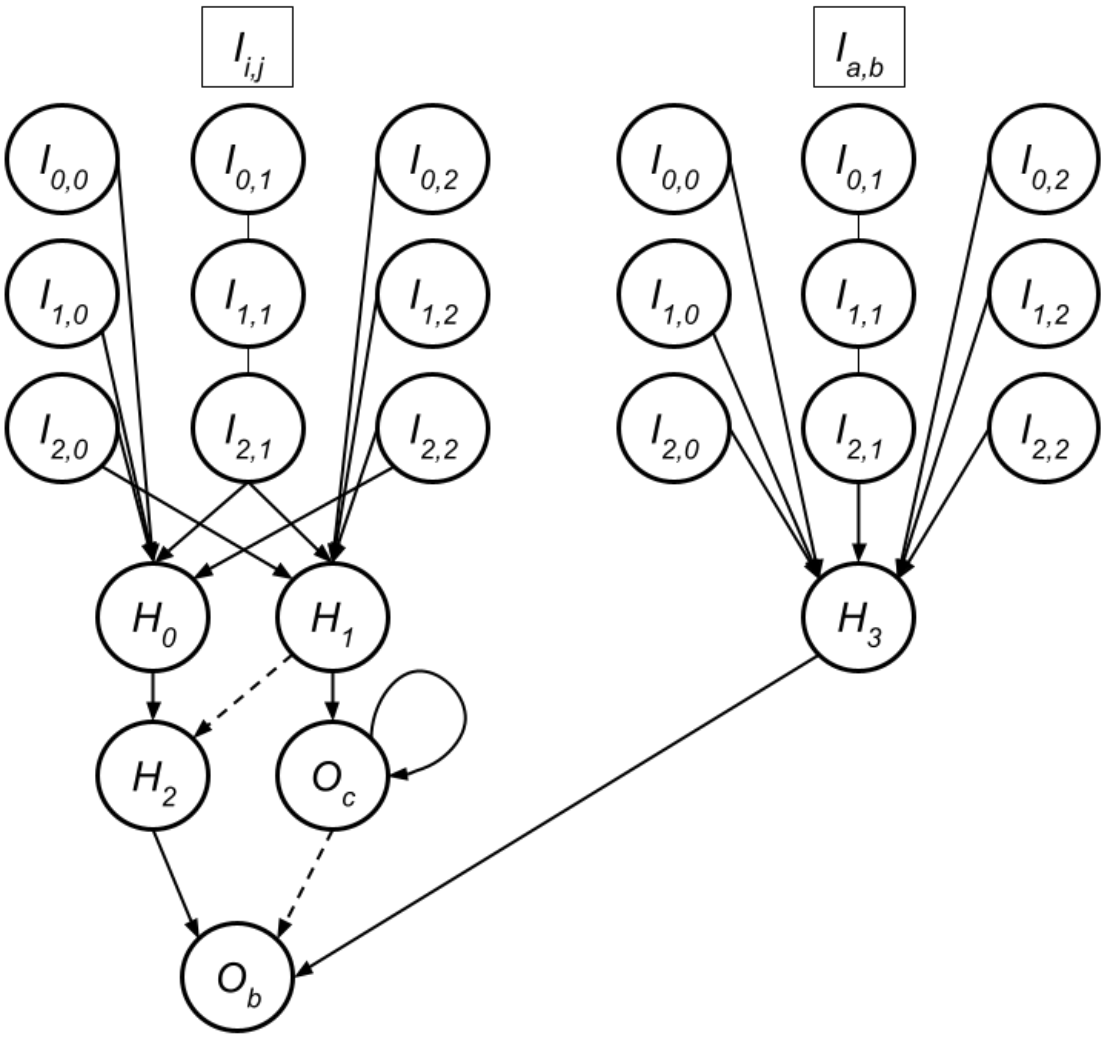


Figure 3.8: DBSCAN network architecture with $\epsilon_d = 1$ and threshold $t_d = 3$. This figure was sourced from prior work [113].

mapped to input neuron $I_{\epsilon_d \in j, \epsilon_d \in i}$, and the other adjacent pixels ranging from $x \in [x - \epsilon_d, x + \epsilon_d], y \in [y - \epsilon_d, y + \epsilon_d]$ simply map to the input neurons $I_{j \in [0, 2*\epsilon_d], i \in [0, 2*\epsilon_d]}$.

Functionally, the inputs $I_{a,b}$ are equivalent to the input neurons $I_{i,j}$; however, instead of operating on the event of interest at pixel *x,y*, $I_{a,b}$ operates on each event that occurs in the radius ϵ_d from the event at pixel *x,y*. This is necessary because while inputs $I_{i,j}$ classify whether the event at *x,y* is a core, border, or noise point, the inputs $I_{a,b}$ must evaluate whether any of *x,y*'s neighboring events are core point events to validate a border point classification from inputs $I_{i,j}$ so as to accurately perform the DBSCAN algorithm neuromorphically. Additionally, hidden neuron H_3 has total leak enabled so that at the end of every cycle, if there is any remaining potential on the neuron, it is leaked away so that we may continue processing each neighboring event each cycle without delay.

The network activity is roughly detailed below:

1. **Cycle 0**: Apply events that occur at pixels $x \in [x - \epsilon_d, x + \epsilon_d], y \in [y - \epsilon_d, y + \epsilon_d]$ at time n to inputs $I_{j,i}$ where the event at pixel *x,y* maps to input $\mathrm{I}_{\epsilon_d \in j, \epsilon_d \in i}$. Additionally, apply the first neighboring event *x',y'* to $I_{a,b}$ where *x',y'* maps to $\mathrm{I}_{\epsilon_d \in b, \epsilon_d \in a}$; for each subsequent cycle, apply the next event in *x,y*'s neighboring list of events to the $I_{a,b}$ input neurons.

2. **Cycle 2**:

    - If hidden neuron H_1 fires, our event at *x,y* is a core event. It in turn fires into output neuron O_c which will continuously generate output spikes and send spikes with inhibitory charge to output neuron O_b to prevent a border point event classification.

    - If hidden neuron H_1 does not fire and only H_0 fires, we have a potential border point event classification that must be corroborated by the inputs $I_{a,b}$ and hidden neuron H_3 over the course of the next $t_d - 1$ cycles.

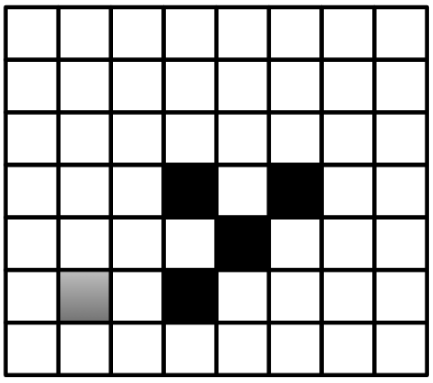
- If neither H_0 nor H_1 fire, the event at pixel x,y is classified as noise and filtered out.

3. **Cycle 3 to Cycle (min_points - 1)**: Input neurons $I_{a,b}$ continue processing events in list of neighboring events $x \in [x-\epsilon_d, x+\epsilon_d], y \in [y-\epsilon_d, y+\epsilon_d]$ (where each event is applied at $I_{\epsilon_d \in b, \epsilon_d \in a}$ in the $I_{a,b}$ input neurons). Supposing one of the events occurs at pixel location x',y', that event is applied at $I_{\epsilon_d \in b, \epsilon_d \in a}$ and events that occur at pixels $x \in [x'-\epsilon_d, x'+\epsilon_d], y \in [y'-\epsilon_d, y'+\epsilon_d]$ are mapped to and applied to $I_{b,a}$ similarly to how the event at pixel x,y and its neighboring events in radius ϵ_d are mapped to and applied to $I_{j,i}$.

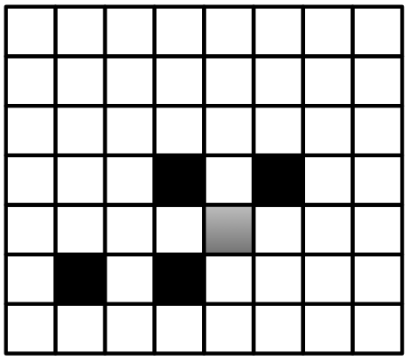
Figure 3.9 shows three event classification scenarios. Each event is classified as either a noise, border, or core event. These three scenarios are shown in Tables 3.5, 3.6, 3.7, 3.8, 3.9. Two tables are needed per example (with the exception of the noise classification), one for the $I_{i,j}$ inputs and one for the $I_{a,b}$ inputs.

This process of classifying events as either core point events, border point events, or noise point events requires threshold (or min_points) $t + 4$ cycles per event. Classifying an event as noise can be accomplished in 3 cycles, classifying an event as a core point can be done in 4 cycles, but classifying an event as a border point event can require up to threshold (or min_points) $t_d + 4$ cycles. This is due to the fact that classifying an event as a border point event requires the parallel classification of at least threshold (or min_points) $t_d - 1$ neighboring events. The intuition behind this is that if we believe an event at x,y is a border point event, there are threshold (or min_points) $t_d - 1$ or less events in its radius ϵ_d. We must then process in parallel up to a possible threshold (or min_points) $t_d - 1$ events at the $I_{a,b}$ neurons to determine if any of them are a core event point. In summary, the following resources are required:

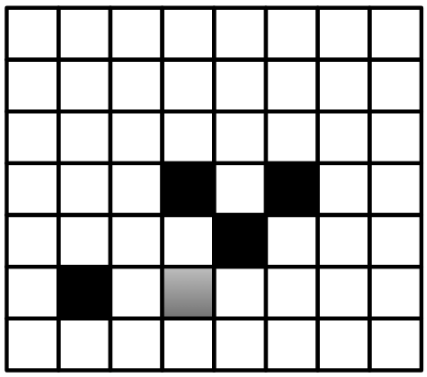
- **Neuron Count**: $(2 * \epsilon_d + 1)^2 * 2 + 6$ neurons
- **Synapse Count**: $(2 * \epsilon_d + 1)^2 * 3 + 7$ synapses
- **Runtime per Event**: threshold (or min_points) $t_d + 4$ cycles



(a) Example for classification of a noise event.



(b) Example for classification of a core event.



(c) Example for classification of a border event.

Figure 3.9: This figure was sourced from prior work [113].

Table 3.5: Example spike table for the $I_{i,j}$ input neurons for Figure 3.9a. There are no adjacent events, so we omit the table showing the $I_{a,b}$ neurons' activity as there is none. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*		
		$I_{1,1}$	H_0	H_1
0	$I_{1,1}$	-	-	-
1	-	*	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-
5	-	-	-	-
6	-	-	-	-

Table 3.6: Example spike table for the $I_{i,j}$ input neurons for Figure 3.9b. Since min_points = 3, and there are a total of 4 events within an epsilon ϵ_d of pixel x,y (including itself), both hidden neurons H_0 and H_1 fire. H_1 inhibits neuron H_2 from firing and causes core output neuron O_c to fire. O_c fires every cycle afterward because of its self-edge and continuously inhibits output neuron O_b from firing, as it still accumulates charge from the $I_{a,b}$ neurons. At the end of simulation, only output O_c would have fired, so the classification would be a core neuron classification. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*								
		$I_{0,0}$	$I_{0,2}$	$I_{1,1}$	$I_{2,0}$	$H0$	$H1$	$H2$	Oc	Ob
0	$I_{0,0}$, $I_{0,2}$, $I_{1,1}$, $I_{2,0}$	-	-	-	-	-	-	-	-	-
1	-	*	*	*	*	-	-	-	-	-
2	-	-	-	-	-	*	*	-	-	-
3	-	-	-	-	-	-	-	-	*	-
4	-	-	-	-	-	-	-	-	*	-
5	-	-	-	-	-	-	-	-	*	-
6	-	-	-	-	-	-	-	-	*	-

Table 3.7: Example spike table for the $I_{a,b}$ input neurons for Figure 3.9b. Since t_d or min_points = 3, there's a maximum of 3 events that will need to be processed by the $I_{a,b}$ neurons. We process the events adjacent to x,y in row-major order excluding event x,y. Because hidden neuron H_3 has total leak enabled, we can process all adjacent events in succession with no downtime. For all three adjacent events, there are only two events to apply at the $I_{a,b}$ neurons. This means that no adjacent event will cause neuron H_3 to fire. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*						
		$I_{0,0}$	$I_{0,2}$	$I_{1,1}$	$I_{2,0}$	$I_{2,2}$	$H3$	Ob
0	$I_{1,1}$, $I_{2,2}$	-	-	-	-	-	-	-
1	$I_{1,1}$, $I_{2,0}$	-	-	*	-	*	-	-
2	$I_{1,1}$, $I_{0,2}$	-	-	*	*	-	-	-
3	-	-	*	*	-	-	-	-
4	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-

Table 3.8: Example spike table for the $I_{i,j}$ input neurons for Figure 3.9c. We apply the two events to the $I_{i,j}$ neurons which causes hidden neuron H_0 to fire. Hidden neuron H_1 does not fire as its threshold has not been exceeded. H_2 then fires sending charge to O_b which fires because it has already received its supplementary required charge from the $I_{a,b}$ neurons at timestep 3 as is shown in Table 3.9. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*				
		$I_{0,2}$	$I_{1,1}$	H_0	H_2	O_b
0	$I_{1,1}$, $I_{0,2}$	-	-	-	-	-
1	-	*	*	-	-	-
2	-	-	-	*	-	-
3	-	-	-	-	*	-
4	-	-	-	-	-	*
5	-	-	-	-	-	-
6	-	-	-	-	-	-

Table 3.9: Spike table for the $I_{a,b}$ input neurons for Figure 3.9c. We process the sole neighboring event, and apply its three adjacent events to the $I_{a,b}$ neurons which causes hidden neuron H_3 to fire. Hidden neuron H_3 sends priming charge to output neuron O_b which doesn't fire until cycle 4, which is when it receives its supplemental charge from the $I_{i,j}$ neurons as is shown in Table 3.8. This table was sourced from prior work [113].

Timestep	Apply spike to	*Fire*					
		$I_{0,0}$	$I_{0,2}$	$I_{1,1}$	$I_{2,0}$	H_3	O_b
0	$I_{0,0}$,$I_{0,2}I_{1,1}$, $I_{2,0}$	-	-	-	-	-	-
1	-	*	*	*	*	-	-
2	-	-	-	-	-	*	-
3	-	-	-	-	-	-	-
4	-	-	-	-	-	-	*
5	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-

3.3 Summary

The network specifications presented in this chapter are organized by whether they're designed to process regions of events or singular events. For applications and hardware systems that are more resource constrained, it might only be possible to operate on an event-by-event basis. Additionally, for downstream AI agents, a downsampled version of the event camera's output may be necessary which requires regional event processing. If hardware resources permit, the spiking threshold pooling networks can be tiled several times into a larger network to process more of the input space in parallel. The per-event processing networks (dbscan and speed_filter) can also be duplicated and tiled to parallelize the processing of events. Furthermore, they can also be extended to process regions of the event camera output by reworking the connectivity with the mentality that we're placing an input neuron behind each pixel. The ability for these presented networks to be tiled, reconfigured to certain sized input spaces, and tailored to real hardware constraints results in customizable scalability that can be designed to suit the need of whatever hardware system they are to be leveraged in.

Chapter 4

Event-based Processing Tool and Library

The preprocessing networks' functionalities discussed in Chapter 3 are exposed in the TENNLab software framework in a library called "ebc_processing". The "ebc_processing_tool" application implements the functions in the "ebc_processing" library so that users may experiment with processing event-based camera data using the aforementioned networks. Listing 4.1 shows the list of commands made available by the "ebc_processing_tool" as terminal output.

```
UNIX> echo '?' | bin/ebc_processing_tool_ravens
EBC Processing Tool - the commands listed below are case-insensitive

FJ json       - Read a json from file or stdin defining input
                dimensions and layers
TJ [file]     - Write a json
PD            - Print dimensionality changes of input as it moves
                through the preprocessing stack
LCF [file]    - Load DVS camera file of events as CSV (use
                utils/aedat_to_csv.py to convert files if necessary)
```

```
VERB bool      - If true, print the output of every preprocessing
                 layer. Additionally, it will print the output of
                 each layer as a frame and not as a list of events
                 in the AER format. You can use this mode if you
                 want to have the output already in frame format,
                 though you will have to manually preprocess out the
                 intermediate layers if you don't want them. If
                 false, only print final layer's output.
NE bool        - If true, crosscheck CPU filtering implementation
                 against network's version per observation. If
                 false, only perform CPU method (which will run
                 faster).
PNETS          - List of network names to write each layer's network
                 component to. If no arguments are specified,
                 networks are dumped to stdout.
WOTF str       - If Write Observation To File is invoked with a
                 string, we're writing observations to that file.
SO int         - Set observation to process to this value
STEP           - Process one observation
SA             - Process all observations
?              - Print commands.
Q              - Quit.
```

Listing 4.1: EBC processing tool command list.

4.1 Commands

The `From_Json (FJ)` and `To_Json (TJ)` commands allow the user to parameterize the ebc_processing_tool by specifying dimensionality information about the data, whether or not to crop to a certain portion of the field of view, and what networks to use as "layers" to preprocess the data. The convention of using JSON to set parameters for an application is a precedent set by other applications and utilities in the TENNLab software framework. Listing 4.2 shows an example of a JSON file that

can be used to parameterize the ebc_processing_tool. A user may specify a JSON file as an argument to the `From_Json (FJ)` command, or he/she may opt instead to provide the JSON on the command line at runtime. The `To_Json (TJ)` command simply echoes the JSON used either to `stdout` or to a JSON file if the user opts to construct the parameterizing JSON while running the tool.

```
UNIX> cat cpp-apps/params/ebc_processing.json
{
    "input_rows": 480,
    "input_cols": 640,
    "segment_time_length" : 2000,
    "rc_collapse": false,
    "input_row_start": 200,
    "input_col_start": 450,
    "input_row_end":300,
    "input_col_end":600,

    "layers":[
        {
            "type": "speed_filter",
            "epsilon": 1,
            "threshold": 7,
            "inverse": true
        },
        {
            "type":"dbc",
            "epsilon": 5,
            "min_points": 20,
            "distance_type": "chebyshev",
            "recall_noise": false
        }
    ]
}
```

Listing 4.2: Example JSON file for the ebc_processing_tool.

In Listing 4.2, we define an input observation size that is 640×480 by specifying `input_rows` and `input_cols`. These are the only two parameters that are required to have values, and they should match the resolution or dimensionality of the data being operated on. Additionally, by specifying non-zero values for `input_row_start` and `input_col_start`, we are indicating that the region of the observation that we care about starts at row 200 and column 450 and terminates at row 300 and column 600 (as is indicated by `input_row_end` and `input_col_end`) respectively. This reduces the working observation space from 640×480 to 150×100.

The `Load_Camera_File (LCF)` command in Listing 4.1 requires that you specify a CSV file of events in the AER format. An example of this format is shown in Listing 4.3. Each line is a tuple parsed as: `x(column value),y(row value),timestamp, polarity`. The ebc_processing_tool reads in all of the events and creates observations whose time span is `segment_time_length` microseconds, which is exposed and set through the JSON provided to the ebc_processing_tool. The number of observations created are $length\_of\_file(\mu s)/segment\_time\_length$. If only `input_rows`, `input_cols`, and `segment_time_length` are specified, the event data will be organized into observations without any preprocessing techniques applied. If the value of `segment_time_length` is anything other than 1, then it is likely that, as the value increases, more events will be filtered out and ignored due to event truncation when constructing a dense event frame representation. For example, if an event occurs at some pixel location `x,y` at the beginning of an observation time slice and then another event occurs at the same location at the end of the observation time slice, the latter event will overwrite the previous event when the observation time slice is represented as a dense event frame. The implications and usefulness of this is discussed in latter chapters.

```
1 UNIX> cat test.csv
2 ...
3 117,7,2,1
4 118,7,2,1
```

```
119,7,2,1
120,7,2,0
121,7,2,0
122,7,2,0
123,7,2,0
124,7,2,1
125,7,2,1
126,7,2,1
...
```

Listing 4.3: Example csv file format for events.

The `Step` and `Step_All (SA)` commands communicate to the running process that an observation should be processed by passing the events in an observation through the preprocessing techniques specified in the `layers` argument in the JSON parameters. `Step` simply processes one observation and outputs the results, while `Step_All (SA)` steps through all of the available observations and outputs the results for all observations. It is also possible to select a specific observation number to process by using the `Set_Observation (SO)` command followed by a `Step` command. The output format of the `Step` and `Step_All (SA)` commands is also the AER format, matching that of the required input format. Additionally, the output for each observation is the final output of the last preprocessing technique. If instead the output of each technique in the in the preprocessing stack is desired, the `Verbose (Verb)` flag may be set to `true`. This will print the observation as it moves through each technique or layer of the preprocessing stack as a dense event frame. This is handy if a user wants to view the intermediate stages of observation processing or have the final output already formatted as a dense event frame for visualization's sake. If the `Write_Observation_To_File (WOTF)` command is specified along with a filename, the outputs will be written to the specified file instead of to `stdout`.

The `Print_Dimensions (PD)` command prints the transformation of the observations' dimensions as they move through the preprocessing stack. For the example shown in Listing 4.1, there would be no change in dimensionality because, as was

discussed in Chapter 3, the speed_filter and dbscan (dbc) networks do not alter the dimensionality of the observation space and operate on a per-event basis.

The `Print_Nets (PNETS)` command will print each preprocessing component of the `layers` JSON array as an individual RAVENS network to `stdout`. If there are `n` preprocessing techniques in the stack, `n` file names can be specified to write the networks to. The last command is the `Network_Evaluation (NE)` command that, when set to `true`, actually constructs in memory the networks for each of the preprocessing layer functionalities. Then, for each observation, the events are translated into spikes that are applied to the network which is then simulated for however many cycles are required. The output spikes are then decoded into classifications of the network. This is done for each network in succession such that the output of the first specified preprocessing technique is the input for the next preprocessing technique. The outputs are verified at each step in the preprocessing stack against the CPU implemented logic for each preprocessing functionality to ensure that the simulation of the actual networks on the inputs matches exactly what the CPU implemented logic yields. This acts as a debugging step and ensures that if a network is implemented on hardware, it will behave exactly as intended.

4.2 Example: DVSGesture

As an example, we leverage the ebc_processing_tool with one datafile from the DVSGesture dataset [5] (user 7 with LED lighting). The dimensions of the event files that compose the DVS dataset are 128×128. The files are also natively in the .aedat file format (as is produced by the DVS128 camera), but using the Python library AERManager we convert them to CSV files that look like the sample shown in Listing 4.3. Initially, we start with the JSON shown in Listing 4.4. We specify the `input_rows` and `input_cols` as both being 128, and we also specify that `segment_time_length` is 5000 microseconds, or 5 milliseconds.

```
1 UNIX> cat ebc_processing.json
```

```
{
    "input_rows": 128,
    "input_cols": 128,
    "segment_time_length" : 5000,
}
```

Listing 4.4: Initial JSON for processing DVSGesture dataset file.

When we run the ebc_processing_tool, load in the JSON file, and then load in the CSV datafile, we see the following output in Listing 4.5. We load in the JSON shown in Listing 4.4 using the `from_json` command and then print it out using the `to_json` command to verify it's correct. We then load in the datafile, which creates 15,159 observations that are each 5 ms long. This implies that the datafile as a whole is roughly 75.8 seconds long. Additionally, we can count all of the lines in the file, since each line contains one event tuple, to determine how many events compose the datafile. There are 2,897,920 events which results in roughly 38,231 events per second. Additionally, we specify the `step` command to print out all of the events that compose the first 5 ms observation (0-indexed).

```
UNIX> ./bin/ebc_processing_tool '>'
> FJ ebc_processing.json
> TJ
{"input_col_end":128,"input_col_start":0,"input_cols":128,
                "input_row_end":128,"input_row_start":0,
                "input_rows":128,"layers":[],
                "rc_collapse":false,"segment_time_length":5000}
> LCF ../data/user07_led.csv
Created 15159 observations
> step
93,24,0,-1
32,28,0,-1
61,54,0,-1
72,57,0,-1
71,64,0,1
```

```
46,71,0,-1
44,77,0,1
21,80,0,1
47,80,0,1
50,80,0,1
49,82,0,1
3,84,0,1
49,84,0,-1
...
```

Listing 4.5: Listing 1 for running the ebc_processing_tool.

Next, we want to apply two layers of the spiking threshold pooling network techniques to the data in order to downsample the data's dimensionality. The first spiking threshold pooling layer features a 10×10 sized kernel with row and column stride values of 10 and a threshold of 10. The second spiking threshold pooling layer features a 2×2 sized kernel with row and column stride values of 2 and a threshold of 2. Without needing to restart the ebc_processing_tool and reload in the datafile, we can simply modify the JSON by reusing the `from_json` command. We also print out the observation dimension change through each layer using the `pd` command. This is shown in Listing 4.6. The observation is reduced from 128×128 to 7×7. Furthermore, if we specify `verb` as true, we can see the observation as a dense event frame as it has moved through each layer of preprocessing. Additionally, if we reload the JSON but enable the `row_column_collapse (rc_collapse)`, the final preprocessed observation will be passed through the row_column collapse logic. The reason that we don't specify row_column collapse as a layer is because it can only be applied at the very end of the preprocessing stack. Instead, we expose it as a boolean option that can be applied to whatever the output of the last preprocessing technique is.

```
UNIX> ./bin/ebc_processing_tool '>'
> FJ
```

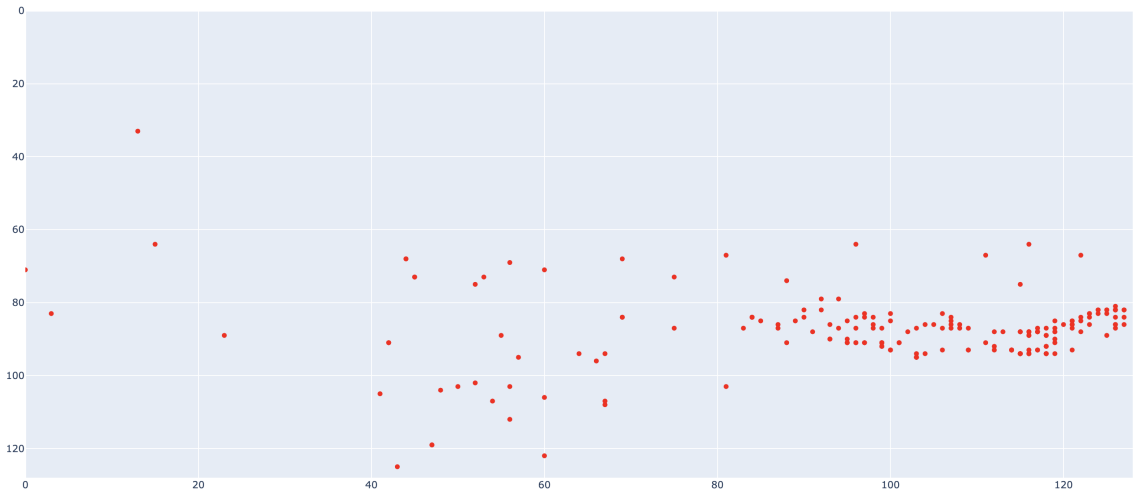
```
{"input_col_end":128,"input_col_start":0,"input_cols":128,
                "input_row_end":128,"input_row_start":0,
                "input_rows":128,"layers":[{"col_stride":10,
                "cols":10,"row_stride":10,"rows":10,
                "threshold":10,"type":"counting"},
                {"col_stride":2,"cols":2,
                "row_stride":2,"rows":2,"threshold":2,
                "type":"counting"}],"rc_collapse":false,
                "segment_time_length":5000}
> pd
128 x 128 ---> 13 x 13 ---> 7 x 7
> verb true
> step
Layer 0 Output 0000: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Layer 1 Output 0000: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
> sa
...
Layer 0 Output 8286: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
                0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Layer 1 Output 8286: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
```

```
> fj
{"input_col_end":128,"input_col_start":0,"input_cols":128,
                "input_row_end":128,"input_row_start":0,
                "input_rows":128,"layers":[{"col_stride":10,
                "cols":10,"row_stride":10,"rows":10,
                "threshold":10,"type":"counting"},
                {"col_stride":2,"cols":2,
                "row_stride":2,"rows":2,"threshold":2,
                "type":"counting"}],"rc_collapse":true,
                "segment_time_length":5000}
> pd
128 x 128 ---> 13 x 13 ---> 7 x 7 ---> 7 + 7
> step
Layer 0 Output 0000: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Layer 1 Output 0000: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Output 0000: 0 0 0 0 0 0 0 0 0 0 0 0 0 0
> so 8286
> step
Layer 0 Output 8286: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
                0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Layer 1 Output 8286: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

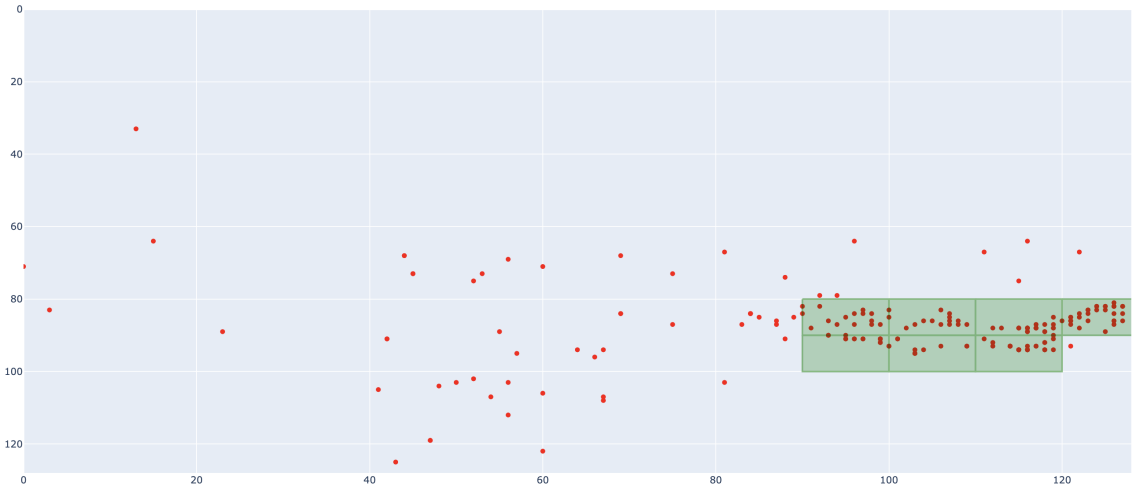
```
Output 8286: 0 0 0 0 0 1 0 0 0 0 0 1 0 0
```

Listing 4.6: Listing 2 for running the ebc_processing_tool.

Because looking at a dense array of 1s and 0s does not adequately display what is happening, we can plot the 0s and 1s as if they're pixels or regions using something like OpenCV [15] or Plotly [56]. Using Plotly, Figure 4.1 shows what the observation 8286 looks like as a scatter plot of events with no spiking threshold pooling downsampling, with one layer of spiking threshold pooling downsampling, and with both layers of spiking threshold pooling downsampling.

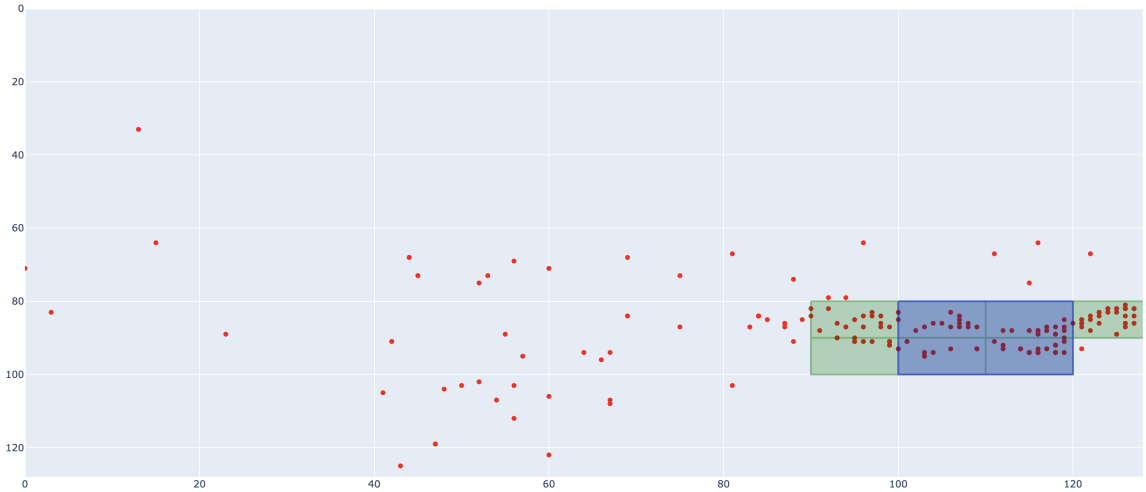


(a) Event scatter plot for the 5 ms of events that compose observation 8286. No spiking threshold pooling is applied here.



(b) Event scatter plot for the 5 ms of events that compose observation 8286. Only the first layer of spiking threshold pooling output is shown and is represented by the green boxes. The green boxes show lesser resolution than that of the individual event/pixel resolution.

Figure 4.1: Three different examples of the same observation with none, one, or two layers of spiking threshold pooling applied to the observation.



(c) Event scatter plot for the 5 ms of events that compose observation 8286. Both layers of spiking threshold pooling output are shown, and the second layer is represented by the blue boxes. The blue boxes show a lesser resolution than that of the green boxes (or the first layer of spiking threshold pooling output).

Figure 4.1: (continued)

Chapter 5

Classification Application

© Owner/Author — Charles Rizzo 2023. The content in this chapter is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record was published in [110], https://dl.acm.org/doi/10.1145/3589737.3605999.

In this chapter, we detail a classification experiment on the DVSGesture dataset highlighting leveraging two of the regional event processing and dimensionality reducing spiking neural networks presented in section 3.1.

5.1 DVSGesture Dataset and Preprocessing

Using a DVS128 camera by Inivation [69, 70], Amir et al. [5] created the DVSGesture dataset: a dataset featuring 11 hand and arm gestures performed by 29 different users under 3-5 lighting conditions. The 11 gestures include hand waving (both hands), arm rotations for both arms in both the clockwise and counterclockwise directions, forearm rolling, air drumming, air guitar, and an additional "Other" gesture invented by each user. Each gesture lasts for about 6 seconds and is separated by some length of time during which no gesture occurs. The format of the data is simply a stream of either positive or negative polarity events that occur at pixel *(x,y)* at a certain time *t*. Unlike other DVS datasets that are conversions from conventional, frame-based datasets and

lack the high temporal resolution of event-based cameras, the DVSGesture dataset (and others like it) have the high temporal resolution of dynamic vision sensors built in, as well as the natural noise that occurs when collecting data on the physical camera.

For prior work classifying the DVSGesture dataset, Amir et al. [5] have introduced an architecture featuring a 16-layer convolutional neural network capable of gesture classification accuracy upwards of 96%. Their architecture is composed of temporal cascade filters, convolutional layers, a Winner-take-all (WTA) decoding layer, and a sliding window filter that cleans up the instantaneous gesture classifications. What is most impressive about the architecture is that they managed to put the entirety of it on a TrueNorth [3] chip and introduce a hardware demonstration where gestures were captured by a DVS128 camera and then classified by the network on the chip. George et al. [47] and Xing et al. [141] have also performed classification on the DVSGesture dataset using relatively large spiking neural network architectures (parameter counts on the millions to tens of millions order of magnitude) with differing degrees of success. Our approach yields comparably performing networks with parameter counts in the thousands to tens of thousands range.

The DVSGesture dataset is presented as a collection of .aedat files generated by a DVS128 camera and corresponding label files that detail which of the eleven possible gestures is occurring between two timestamps, which are of microsecond temporal precision. Each .aedat file is the recorded event stream of a user in a specific lighting condition stored in the Address Event Representation (AER) [21] format, and initially for our work, we convert each file to a CSV representation where each event is represented by the following quadruple: *x, y, timestamp, polarity*. For our work, we shift the timestamps of the events so that the first event in a file occurs at timestamp 0 and all successive events occur at timestamps relative to the first event in the file. An example of the first five events and their conversions from the user 1 fluorescent lighting file is shown below.

```
56,78,80046394,0  -->  56,78,0,0
63,78,80046412,0  -->  63,78,18,0
46,90,80046414,0  -->  46,90,20,0
69,87,80046427,0  -->  69,87,33,0
69,89,80046433,0  -->  69,89,39,0
...
```

For every user and lighting combination file, we create time series data samples. Using a similar notation as is presented in [141], we define *segment_time_length* as the length of time in microseconds that composes a segment. For example, if a dataset file is 114,170,024 microseconds long, and we segment the file with *segment_time_length = 5000*, the result is

$$floor(\frac{114170024}{5000}) = 22834$$

segments. The first segment in the file contains the events that occur within the first 5000 microseconds (or 5 milliseconds). The second segment in the file contains the events that occur between microseconds [5000,9999], and so on. Furthermore, we define an additional parameter called *segments_per_sample* that dictates how many segments compose one timeseries sample. If we assume *segments_per_sample = 150*, then a timeseries sample is composed of 150 5000-microsecond long segments. In other words, one timeseries sample is 0.75 real-time seconds. To determine the number of timeseries samples in our example file, we do the following:

$$floor\left(\frac{\frac{114170024}{5000}}{150}\right) = 152$$

Our example file has 152 timeseries samples where the first 5000-microsecond segment in the file has time 0, the next segment has time 1, and so on up until segment 149, which has time 149. Then, at segment 150, we start a new timeseries sample, so segment 150 has time 0, segment 151 has time 1, and so on. This temporal compression is similar to what George et al. [47] and others do in order to gain richer

spatial information through a more dense event collection by processing event time slices or batches as a "frame". The label for each timeseries sample is determined by whatever gesture window its last segment's last event's original timestamp occurs in. Because of the way the DVSGesture dataset is composed, there are lengths of time between each gesture that are unlabelled that we call transitional. The way we form our dataset makes it possible for a time series sample to start in this transitional time window (or even the previous gesture's time window) and end in a different gesture's time window. In the spirit of advocating for a more real-time, event stream approach of processing and classifying the DVSGesture dataset, we argue that this characteristic is realistic for classifying in real-time with a network right behind a camera.

Additionally, we alter the labelling scheme slightly. In the DVSGesture dataset, label 11 corresponds to the "Other" gesture that is different from user to user. It has been shown in many prior works that this label complicates classification and yields lower accuracies [47, 141]. Like these prior works, we choose to remove this label and its samples; instead, we assign label 11 to any time series samples that end in the transitional period of time between gestures. We believe this is a fair trade-off: instead of trying to classify a gesture that differs from user to user, we try to classify the periods of time between gestures as "transitional" and not one of the other ten standardized gestures. This change further supports the goal of classifying DVS camera data in real-time.

For our work, we use the the train/test user split recommended by Amir et al. [5]. This results in users 1-23 and all of their lighting conditions composing the training dataset and users 24-29 and all of their lighting conditions composing the testing dataset. While there are five lighting conditions possible, not every user is captured under every lighting condition. Table 5.1 shows which user and lighting condition combinations are not included in the dataset.

Table 5.1: Combinations of users and lighting conditions that are not present in the DVSGesture dataset. This table was sourced from prior work [110].

Lighting	User
Natural	7, 8, 10, 12, 18, 20, 23, 24, 25
Fluorescent	12, 16, 25
LED	11, 21
Fluorescent LED	–
Lab	3, 4, 11, 12, 14, 20, 24, 25, 27

5.2 Hyperparameter Grid Search and Evaluation

Our goal is to determine the effect of different neuromorphic spatial reduction techniques on a random forest classifier's performance on our version of the DVSGesture dataset. A random forest classifier is an ensemble of decision tree classifiers whose output is whatever output is selected most often among is constituent decision trees. Decision trees learn from data using simple if-then-else decision rules. Singular decision tree classifiers tend to overfit to training data, but averaging multiple decision trees that use random or semi-random decision rather than each learning from the data in a random forest tends to mitigate the overfitting. We perform a small grid search to examine how different downsampling parameters and layer combinations influence the classifier's performance on the testing dataset.

While both the *segment_time_length* and *segments_per_sample* values are parameterizable, changing them fundamentally changes event sparsity in the dataset. This work focuses more closely on the effect of downsampling parameters on a particular dataset and its event sparsity and not the impact of different levels of event sparsity. Table 5.2 shows the combinations evaluated.

To limit the size of our search, we only work with square chip sizes and stride values that are either equal to the chip size or half of it. We also only test up to two layers of downsampling and constrain the parameters for the second layers of downsampling. As an example, the first downsampling layer may count either events or unique pixels that fire in a region, but any secondary downsampling layer will only count the amount of chips from the first layer that "fire," and not the number of unique chips in its region that fire. Additionally, the threshold value for the second downsampling layer is always 1. However, the second downsampling layer features an additional chip size of 5×5 that is not used in any first downsampling layer.

There are some combinations that we define as illegal and leave out of the experiment. An example of an illegal parameter scheme would be where we have a first downsampling layer with chip size of 25×25 and a stride value of 25. This

Table 5.2: Parameters used for various configurations of neuromorphic spatial reduction techniques. * denotes that the options apply to the first spiking threshold pooling layer only. ** denotes that the options apply to the second spiking threshold pooling layer only. This table was sourced from prior work [110].

	Values
Chip Sizes	5×5**, 10×10, 15×15, 20×20, 25×25
Strides	*chip_size*, $\frac{chip\_size}{2}$
Thresholds*	10, 20, 30
Count Type*	Event, Pixel
Layers	1, 2
Row/Col Collapse	True, False
Transitions	True, False

would reduce our observation to $ceil\left(\frac{128}{25}\right) * ceil\left(\frac{128}{25}\right) = 6 \times 6$. If we use a second layer of downsampling where the chip size is 10×10, we do not have enough observations to fully occupy one chip. We declare this combination of parameters illegal. With this in mind, the result is 504 legal parameter combinations.

We have two sets of experiments. In the first, we include the transitional data samples, resulting in 11 classifications. In the second, we omit the transitional data samples. Even though our goal involves being able to process these transitional samples, we perform the experiment without the transitional samples to have a closer comparison with how classification is traditionally performed on the DVSGesture dataset in the literature. This yields a total of 1008 combinations. Finally, as a control, we test a combination where the chip size is 1×1 with a stride of 1 and a threshold of 1. This effectively performs no downsampling of the observation space at all and serves as a comparison data point to determine whether or not these neuromorphic spatial reduction techniques benefit classifier performance or not. This brings our total number of combinations up to 1010. We use scikit-learn [98] to create and train the random forest models on all combinations of the neuromorphic observation space reduction techniques.

5.3 Results

Figure 5.1 summarizes the results of the two experiments. It plots the F1-score for each random forest on the testing dataset. We chose F1-score as the evaluation metric because the sets of data have differing numbers of labels, and the transitional samples cause an imbalance in the label representation of the dataset. F1-Score is the mean of precision and recall, and for a multi-class classification experiment, it can be thought of as the average performance across all classes. This is useful in imbalanced datasets because a model's performance can be biased toward predicting whichever class represents the majority of the dataset. However, by using the average of performances across each individual class as the evaluation metric, the model is

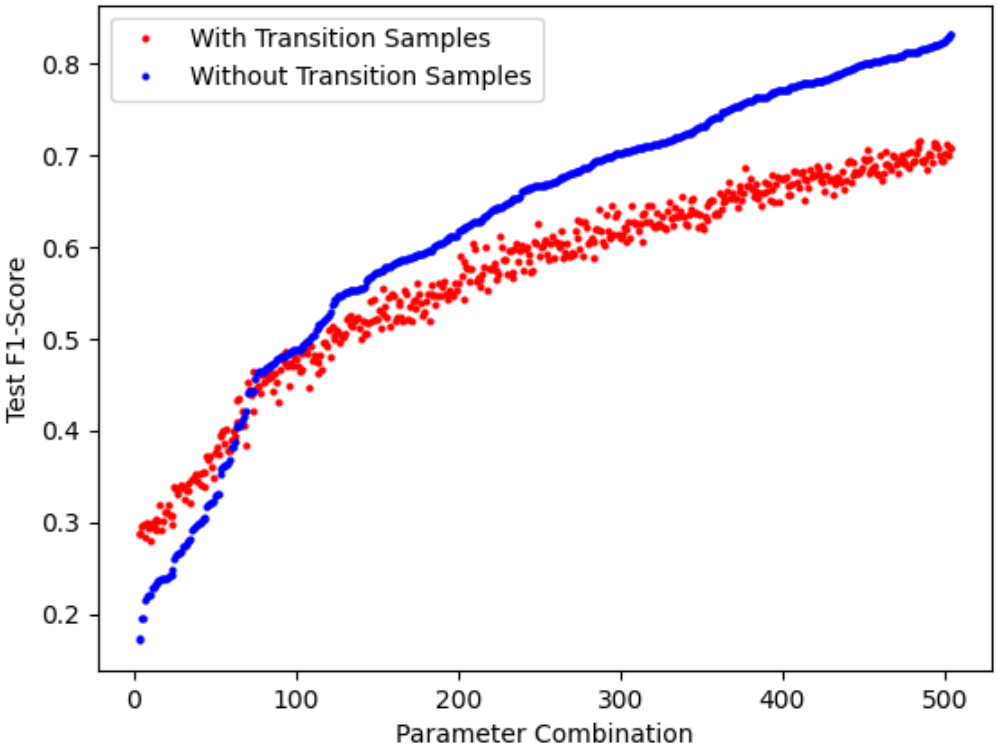


Figure 5.1: Overall test F1-score results for all parameter combinations. This figure was sourced from prior work [110].

pushed toward learning how to recall samples from all classes, even if it's to the detriment of overall performance.

Each parameter combination is numbered by how well it performs on the experiment without transition samples. The performance trend is similar for both experiments, and as expected, the overall performance on the experiment without transition samples is better than the experiment with transition samples. It is interesting that for the parameter combinations that perform poorly, they perform better on the dataset with transition labels. However, as the F1-Score rises above 0.5, the performance is markedly better without transition samples. We have no explanation for this phenomenon.

Figures 5.2 and 5.3 plot the performance of the classifiers with and without transitions, respectively, as a function of the number of features after downsampling and/or collapsing. In each figure, the best performance comes from reducing the input space of 128×128 to between 10 and 50 input features. This is an interesting result for two reasons. First, reducing the number of features, even though it loses resolution, can improve the performance of the classifier. Second, when the number of input features is reduced to the 10-50 range, the resulting size of the classifer, and therefore the SNN that implements it, is smaller, which is beneficial for SWaP.

Figure 5.4 shows the performance of classifying the data without transition samples, as a function of the number of neurons required when the random forest is converted to a spiking neural network [120]. This graph underscores the advantages of downsampling – the networks that perform best are in fact the smaller networks. This may be because of overfitting. We do not show the number of synapses because they scale linearly with the number of neurons (roughly 1.2 synapses per neuron). Moreover, we also omit the results of the data with transition samples. Although they require more neurons and synapses, their trends are the same.

Figure 5.5 shows the parameter settings of the 50 highest-scoring networks on the dataset without transition samples. Above each data point are shaded boxes indicating the parameter settings of the data point. For the boolean parameters

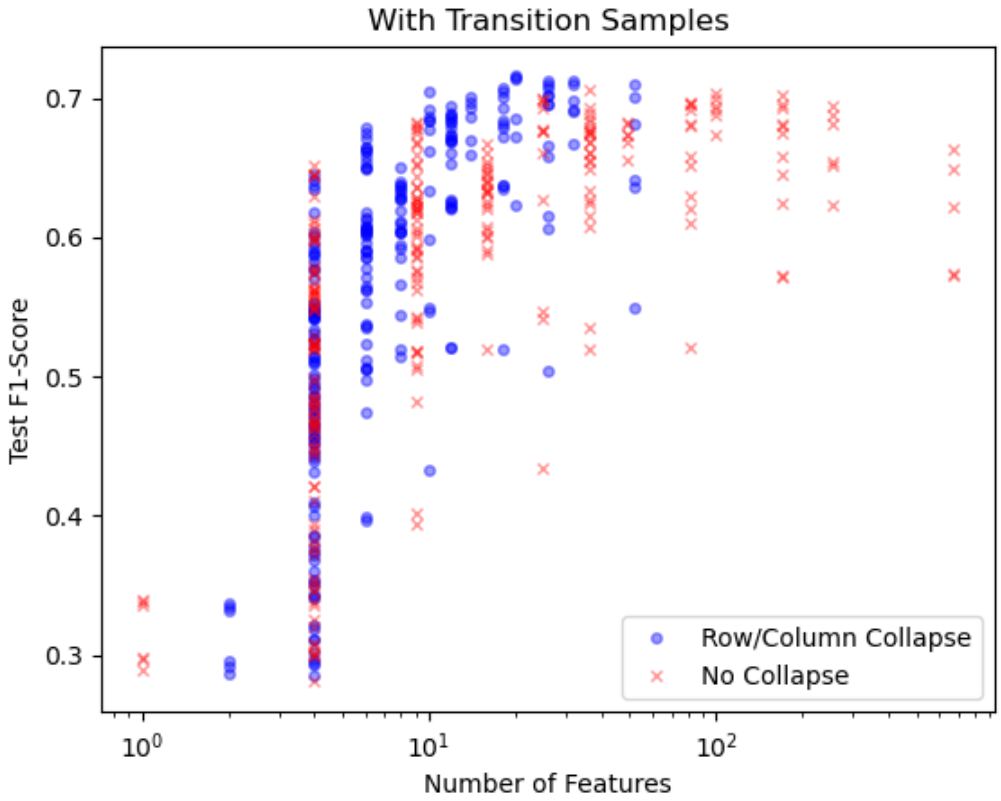


Figure 5.2: Figure showing the number of downsampled input features and their impact on overall testing F1-score for the datasets with transition samples. This figure was sourced from prior work [110].

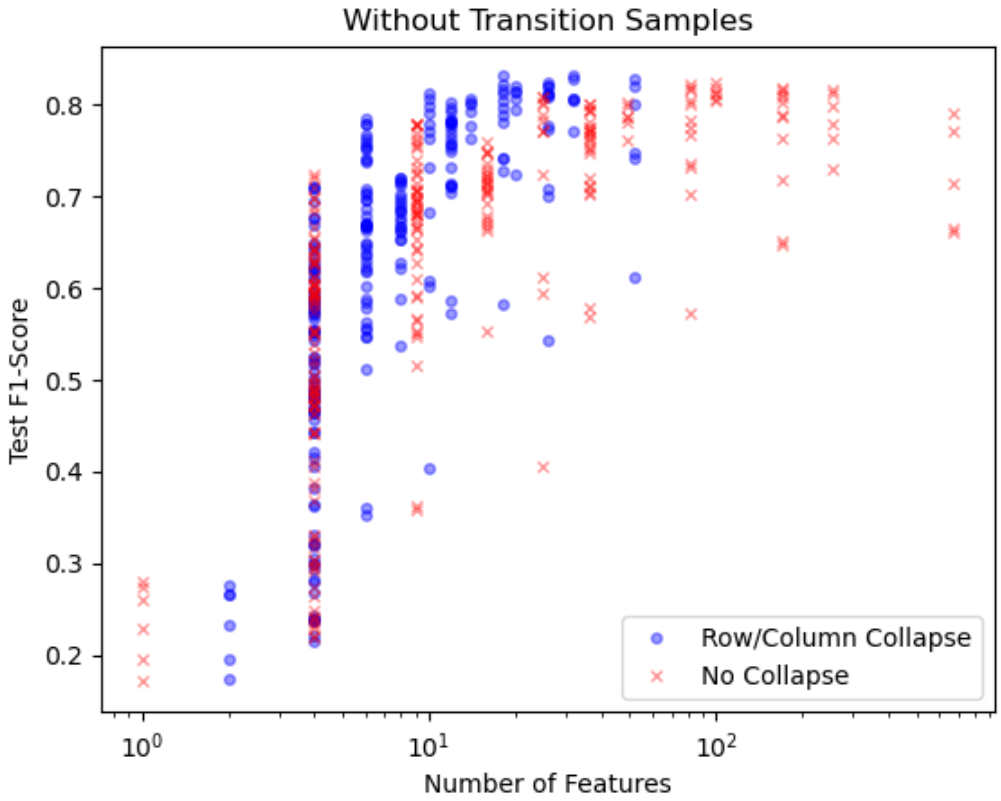


Figure 5.3: Figure showing the number of downsampled input features and their impact on overall testing F1-score for the datasets without transition samples. This figure was sourced from prior work [110].

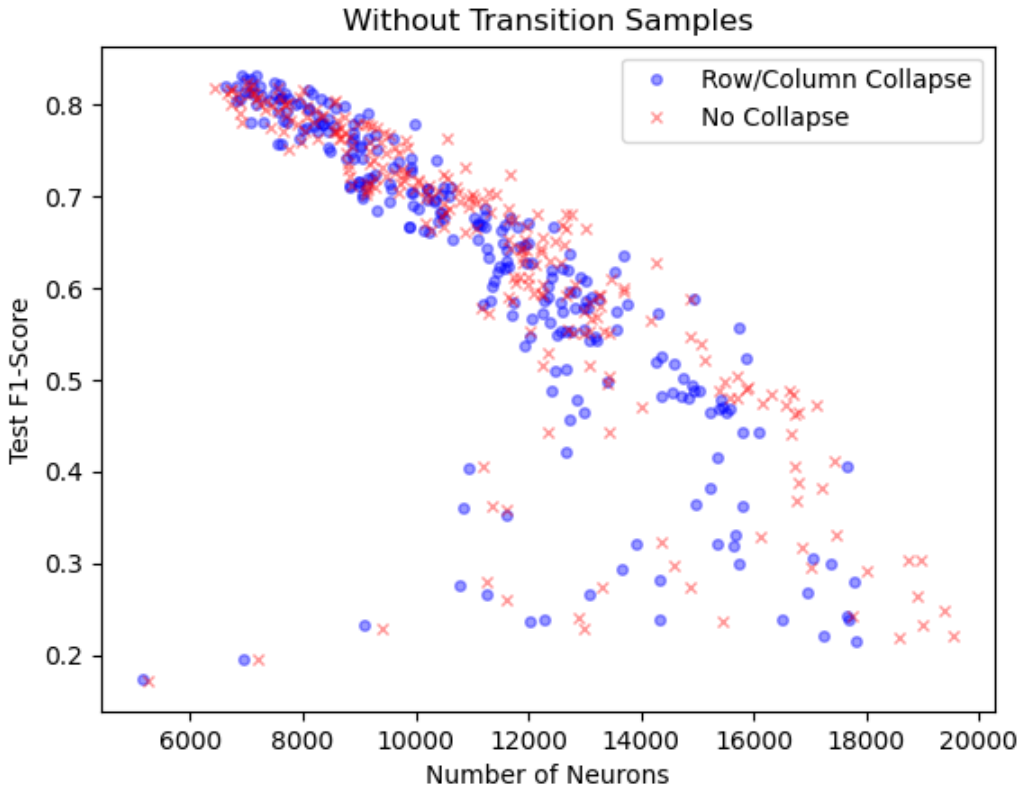


Figure 5.4: Figure showing the number of neurons required to convert each random forest classifier to a spiking neural network and its overall testing F1-score. This figure was sourced from prior work [110].

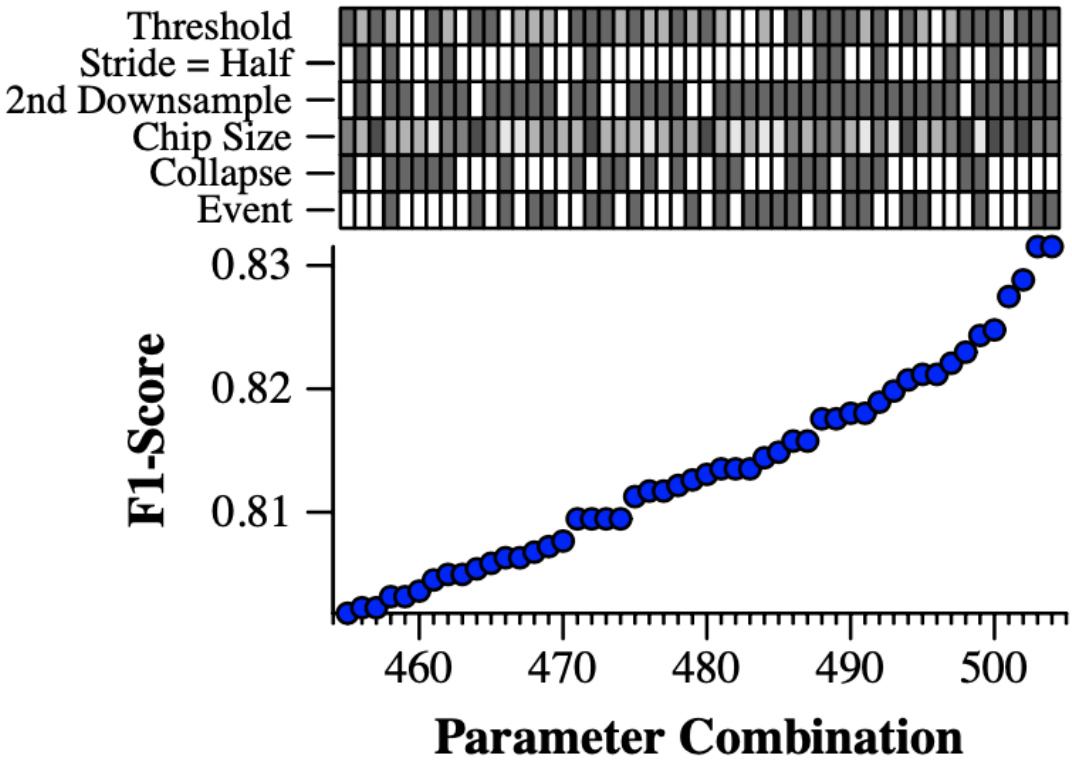


Figure 5.5: Figure showing parameter combinations of the 50 highest-scoring networks on the datasets without transition samples. This figure was sourced from prior work [110].

(Stride=Half, 2nd Downsample, Collapse and Event), a white box indicates *true* and a dark box indicates *false*. In the other two parameters (Threshold and Chip Size), lighter boxes indicate higher values.

There are three trends we can glean from this data. First, the majority of these networks did not use a second round of downsampling. Second, using a stride equal to half of the chip size generates better random forests. Third, row/column collapse was advantageous significantly more than it was not. We can use these observations to narrow the scope of further parameter searches, or to allow us to broaden the scope with respect to other parameters.

It is evident that the neuromorphic spatial reduction techniques can bear a large impact on the performance of the random forest classifier. The wrong combination of parameters can yield f1-scores on the testing dataset that are lower than 30%; however, the best combinations of parameters can yield scores as high as 85% when "transition" samples are ignored and 72% if they are not. The highest performing classifiers share the following similarities: the feature space is reduced to less than 50 down from the original 16,384, the row/column collapse technique is used, only one downsampling or spiking threshold pooling layer is used, and the chip stride is typically half the length of the square chip size.

5.4 Discussion

A point of confusion with this work is that our dataset is a timeseries dataset and is therefore not directly amenable to using a random forest classifier. We still fit a random forest model to the training data by "unrolling" each timeseries sample. If we suppose that we have downsampled to an observation space that is 5×5 and we know that *segments_per_sample* is 150, we can treat each feature at each timestep as a unique feature such that the classifier is working on a dense $5 * 5 * 150 = 3750$ feature observation.

At first glance, this seems to contradict the goal of being able to classify gestures in real time on neuromorphic hardware without additional preprocessing aside from the neuromorphic spatial reduction techniques. However, for the spiking neural network implementation, a dense observation need not be created. The camera's event outputs need only be routed to the proper input neurons at the proper time.

Additionally, the best performing networks have about 20,000 total parameters (neurons and synapses) when the transitional data is ignored and about 30,000 total parameters when the transitional data is recalled. While these totals are an order of magnitude smaller than the network sizes reported in [47, 141], these totals do not include the additional hardware requirements of the neuromorphic observation space reduction techniques that are employed. However, it is assumed that any spiking threshold pooling networks will be time multiplexed over the DVS camera's observation space. This involves using only one or a handful of counting networks in a spiking threshold pooling layer and striding them over the input space over time. This is similar in methodology to what has been presented by Severa et al. [121] previously. When using time multiplexing, only a few hundred additional neurons and synapses are necessary (depending on the chip size). We argue that the additional hardware requirements of the spiking threshold pooling layer(s) and row/column collapse techniques are negligible and do not total more than a couple thousand neurons and synapses.

As one of the first attempts evaluating neuromorphic observation space reduction techniques on DVS camera data, we acknowledge the following shortcoming with this body of work. Three parameter combinations from each experiment did not finish running due to memory errors. Each parameter combination had access to 96 Gb of memory, but if the feature space was too large (as it was for the 1×1 downsampling with a stride of 1 control experiment), the model could not be created and stored in memory alongside the data. This is likely due to *segments_per_sample* being too large. For the control experiment, the random forest classifier would have had to fit to data containing $128 * 128 * 150 = 2457600$ features per time series data sample.

While this is not ideal, it can still be stated that the neuromorphic spatial reduction techniques enable random forest classification of the DVSGesture dataset. Without it and with too large of an "unrolled" feature space for the classifier to use for training, random forest will not work on the dataset at all.

5.5 Conclusions

This work aimed to study the impact of neuromorphic spatial reduction techniques on the random forest classifier's ability to classify our version of the DVSGesture dataset. There are a few key takeaways from this work, but the most important one is that the neuromorphic spatial reduction techniques' parameters greatly impact the classifier's performance. In fact, without using the techniques, random forest cannot train on the data at all. Again, we specifically care about the random forest classifier because it can be represented as a spiking neural network on hardware and placed directly behind a camera for real-time classification. Furthermore, with this method of training, our resultant best performing networks are an order of magnitude smaller than other state of the art architectures shown in [3, 141, 47]. The tradeoff is that performance is roughly 10-15% worse; however, with a larger parameter space exploration, the we believe this performance gap can be closed substantially. Another takeaway is that the row/column collapse technique yields the best performance and should likely be used as a final neuromorphic observation space reduction technique before the classifier.

Chapter 6

Control Application

In this chapter, we present a library that simulates events from conventional frames to facilitate online learning on event-based data and a case study on one specific Atari game. This chapter aims to demonstrate the feasibility of using spiking neural networks coupled with event-based camera-style inputs to develop a control agent capable of solving complex but well-defined tasks. This chapter leverages the spiking threshold pooling network discussed in section 3.1 to help reduce the complexity of the input space for EONS-trained agents.

6.1 ALE_EBC Library

© Owner/Author — ACM 2022. The content in this section is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record was published in [111], https://doi.org/10.1145/3546790.3546817.

The Arcade Learning Environment (ALE)[11, 77] is a framework that facilitates the development of AI agents for Atari 2600 games. It is built on top of the Atari 2600 Stella emulator, and it currently supports over 50 classical Atari games. The Atari games are instances of control applications that require an agent or network to develop different skills in order to successfully play each game. For example,

with games such as Frogger and Freeway, skills such as object avoidance and path optimization are required to maximize the score, and with Enduro, the player is a car whose goal is to pass as many cars as possible while avoiding the boundaries. There are more complicated games for agents to play like Donkey Kong or Seaquest that have multiple different subtasks that must be performed in order to maximize the score while avoiding death.

We have implemented a wrapper for the Arcade Learning Environment that exposes the classical Atari 2600 games as applications for neuromorphic agents to train on in TENNLAb software framework [111]. We call it the ALE_EBC library.

The ALE_EBC library is written in C++11 for performance, and it interfaces with and implements the ALE_Interface class. The library is structured and intended to be used similarly to OpenAI gym [16] environments: create an instance of the ALE_EBC class, call `reset()` with a pseudo-random number generator seed, and then call `step()` in a while loop while the episode is not over (i.e., while the agent is surviving within the game). Both `reset()` and `step()` provide observations about the environment as their return values. The `step()` method also provides reward information about how the agent is performing, and it takes as an argument the action that the agent would like to perform. The simplicity of the ALE_EBC library allows it to be incorporated easily into other projects.

The only dependencies of the ALE_EBC library are ALE's compiled dynamic library and the Nlohmann::json library [73]. The ale_ebc constructor uses a JSON configuration file to initialize its values, and the nlohmann::json library simplifies this process. To use the ALE_EBC library after compilation, you need only include the ale_ebc.hpp header file in your project.

The library simply creates an episode instance of the user specified ROM file using ALE and then, for each time step, records the observation to be passed back to the agent and passes the agent's action to the episode. There are different types of observations that an agent can use as input, and they are detailed in the next section. The different observations and other ALE_EBC parameters are specified in the JSON

file of parameters. The ALE_EBC library is open-source code released under the BSD three-clause license, and is available on GitHub at https://github.com/charizzo/ale_ebc.git.

6.1.1 Observation Types and Event Simulation

RAM Observation

The classic Atari games only require 128 bytes of memory in order to be played. ALE exposes a function that can query the RAM state at each time step. The RAM observation is a vector of size 128 that represents the state of the current episode. Though this type of observation has nothing to do with the EBC contribution and novelty of this work, we include this observation type so that it may be used as a comparison point when evaluating different agents' performances across various observation types.

Grayscale or RGB Screen Observation

ALE also exposes functions that allow the user to query for the screen at any time step as an observation (represented as a one dimensional vector) in either grayscale or RGB values. Assuming a reported screen size of 210 $\times$ 160, there are 33,600 pixel values for the grayscale screen and 100,800 pixel values (33,600 * 3) for the RGB screen with each pixel value ranging from 0 to 255. Similar to the RAM observation, this observation type is exposed for the sake of comparison. The pixels are stored in row-major order where there are 160 columns and 210 rows for any normal frame. For the RGB observation, the color channels themselves are stored consecutively in the vector as R, G, B. As an example, to access the 50th pixel in the 4th row's blue value (assuming 0 indexing), the observation vector would be indexed as follows: $observation[(5 * 160) + (51 * 3) + 2]$.

Simple EBC Observation

The simple EBC observation simulation is straightforward: for each pixel in frame x, determine if there is an "event" at that pixel by taking the difference of the grayscale pixel value at frames x and $x-1$. The AirSim [123] event simulator code directly influenced all three of the event-based camera observation types. The simple observation type is most similar to the early EBC simulation work found in [61]. Listing 6.1 below shows the straightforward logic and implementation of the simple EBC observation generation code.

```
for(i = 0; i < 210 * 160; i++)
{
    current_pxl = current_frame_buffer->at(i);
    previous_pxl = previous_frame_buffer->at(i);
    delta_luminance = current_pxl - previous_pxl;

    //Have not crossed threshold * max_pixel value,
    //move on; no event.
    if(abs(delta_luminance) < luminance_threshold * 255)
    {
        observation.push_back(0);
        continue;
    }

    //Determine if event is polarity increasing
    //(1) or decreasing (-1)
    polarity = (delta_luminance > 0) ? 1 : -1;
    observation.push_back(polarity);
}
```

Listing 6.1: Code used to determine whether or not an event should be simulated at each pixel.

Logarithmic EBC Observation

The logarithmic EBC observation is nearly identical to the simple EBC observation simulation. The only major difference is that the pixel values are passed through the natural logarithm prior to computing the difference. This necessitates a minor change to the thresholding check that determines whether or not there is an event. Calculating the polarity values for each each pixel based on change in log luminance intensity is justified in [60]; it more closely mimics the logarithmic response of the actual camera sensor. However, like in [61], because the Atari game environment is a controlled artificial scene, it isn't necessary to use log luminance logarithmic values for the pixels to determine if there are events or not. Regardless, this option is included for the sake of more closely adhering to how event-based hardware operates.

Temporally Interpolated Logarithmic EBC Observation

The temporally interpolated logarithmic EBC observation aims to even more closely mimic the behavior of event-based cameras. In addition to determining whether or not a pixel has captured an event based on the difference of log luminance intensity values, we more closely emulate the high temporal resolution of event-based cameras. The two aforementioned EBC observation methods have the same temporal resolution as the Atari episode simulation (about 60 FPS). However, event-based cameras can have temporal resolutions greater than 1 MHz.

Based on a pixel refractory period set by the user at runtime, the max number of events that can occur between two frames is calculated. The greater the difference between a pixel value's log luminance intensity between two frames, the greater the number of events generated by that pixel (up to that max_events amount). This means that the temporal resolution is increased by a factor of max_events, and the amount of events at each pixel are linearly interpolated between the previous frame's timestamp and the current frame's timestamp, starting at the previous frame's timestamp. Because we are effectively creating multiple events between two time steps of an

ALE episode, this observation becomes similar to a timeseries observation where the inner vector length is max_events and the outer vector length is the number of pixels.

Like with the logarithmic EBC observation, this level of precision is unnecessary for the artificial Atari environment. However, it is included for the sake of evaluating how well an agent performs when the amount of events are substantially increased such that the simulation more closely mimics the physical sensors. This method is what the AirSim [123] event simulator project implements. This observation option more closely mimics the ESIM [106] improvements made to the model used by Kaiser et al [61].

A quick comparison of the runtime for each of the observation types and amount of events or data over 300 timesteps is shown in Table 6.1. For each of the RAM and screen observation types, we consider every byte of RAM or every pixel value at each timestep to be an "event". The EBC_simple and EBC_log observations show how much static background data is filtered out when compared to either of the screen observation types. There is approximately 543× less data to be processed by the agent when using either of the EBC_simple and EBC_log observations as opposed to the screen_rgb observation. The memory footprint of the EBC_simple and EBC_log observation types is identical, and their disparity in runtime is likely due to passing each pixel value through the natural log function for the EBC_log observation. However for the EBC_log_ti observation, we increase the memory footprint by a factor of max_events, which is a function of the pixels' refractory periods. While it does take the longest to run, it still reduces the amount of data being processed by a factor of about 56. Naturally, the amount of events generated by any of the EBC observation types is dependent upon the threshold value. For this test, the thresholds were set to $255 * 0.05 = 12.75$ for the simple observation type and 0.05 for for the log observation types. These thresholds are pretty sensitive, or low, and increasing them would further reduce the amount of events.

Table 6.1: Comparison of wall clock computation time and number of events averaged over 10 uniquely seeded Freeway episodes at 300 timesteps per episode. This table was sourced from prior work [111].

Observation Type	Avg. Runtime (sec.)	Avg. Events
RAM	0.7693	38,400
Screen_Gray	0.8371	10,080,000
Screen_RGB	0.9725	30,240,000
EBC_Simple	1.6235	55,621
EBC_Log	3.0636	55,621
EBC_Log_TI	4.0338	532,540

6.2 Example with Freeway

© Owner/Author — ACM 2022. The content in this section is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record was published in [111], https://doi.org/10.1145/3546790.3546817.

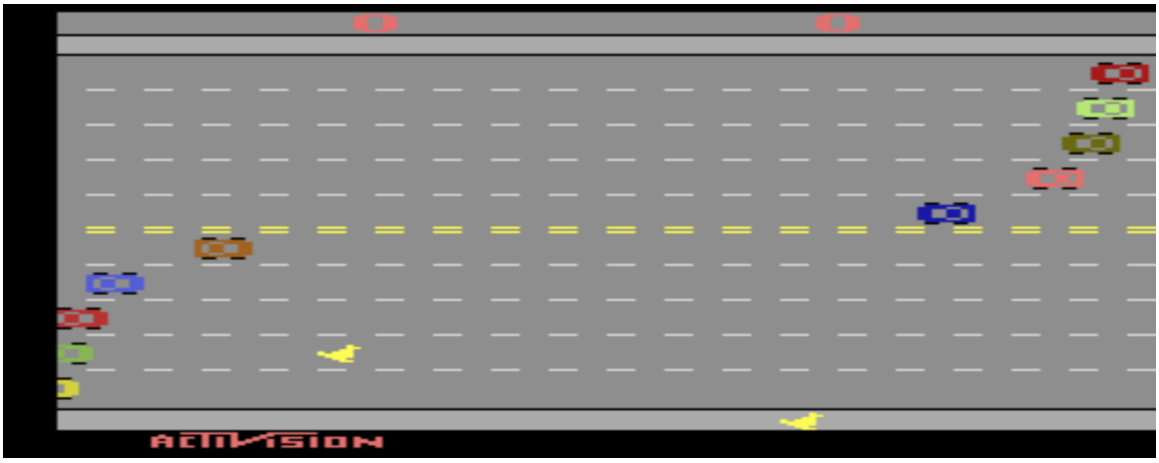
To illustrate how to use the library, we compile and run the example code in Listing 6.2. The freeway.bin ROM is specified in the parameter file JSON along with other necessary parameters that are explained in the GitHub repository. We use the "ebc_simple" input_type and record the events to a file (one event per line as timestamp, column, row, polarity (1 increasing/-1 decreasing)) in order to display them using the python program included in the repository that creates scatter plots of the events with Plotly [56]. The program allows users to "see what the agent sees" while training both as a sanity check and for demonstrations. The action is hard-coded to 2 on line 11 of Listing 6.2, which simulates pushing the Atari joystick forward to "move up". The other actions and encodings are specified in the ALE_EBC repository.

```
1  #include "ale_ebc.hpp"
2
3  using namespace ale_ebc;
4
5  int main(){
6      Ale_Ebc *test;
7      vector <int> observations;
8      bool done;
9      double reward, total_reward, action;
10
11     action = 2;
12
13     // Construct and initialize ALE_EBC object
14     // with params file
```

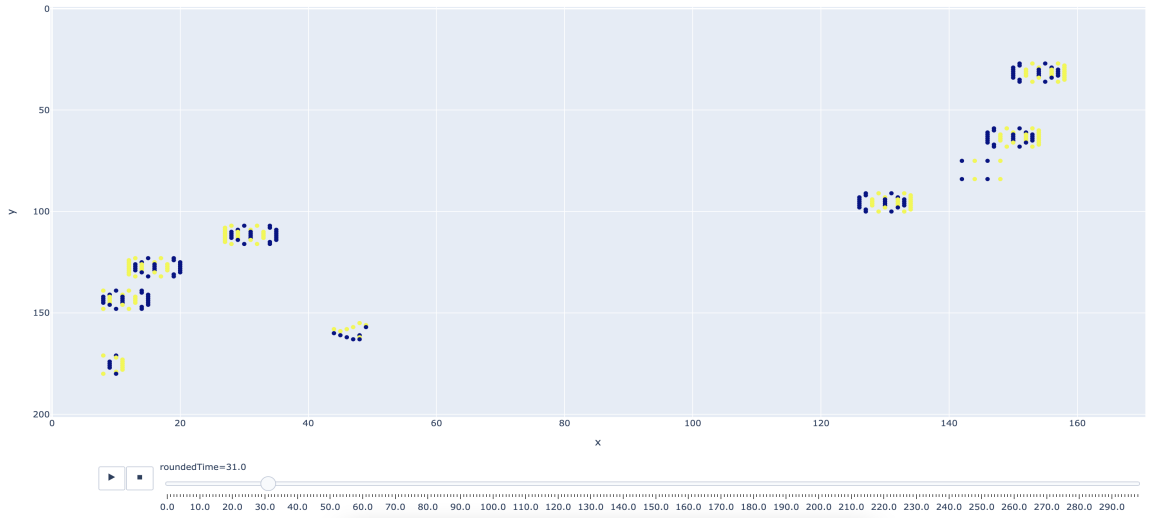
```
15      test = new Ale_Ebc();
16
17      // Let user seed the run by passing a
18      // seed to reset along with observations
19      // vector for initialization
20      test->reset(observations,2000);
21
22      done = false;
23      total_reward = 0;
24      while(!done){
25          done = test->step(action,observations,reward);
26          total_reward += reward;
27      }
28
29      delete test;
30      return 0;
31  }
```

Listing 6.2: Example code including and using ale_ebc library.

After running the program and displaying the events with the python program, we can see the scatter plot representation of the simulated events in Figure 6.1b, which is similar to current frame in Figure 6.1a. It highlights which of the cars are "in motion" from the previous frame to this current frame. It is interesting how, for this particular frame, the two lighter green cars are not captured by the event simulation, and the pinkish car in the top half is represented by 8 events that appear to be mapping the boundaries of the car's tires. This edge-like image of the car is not surprising given that the ebc_simple observation is used; this behavior of creating edge-like representations of moving objects when simple pixel difference thresholding is performed was pointed out in [106] as an issue with the model used by Kaiser et al. [61]. However, given that the other cars and the chicken are clearly visible, we argue that the ebc_simple observation type is likely sufficient as a proof-of-concept method for determining whether an agent can learn to play the game or not.



(a) Screenshot of Freeway episode.



(b) Event-based camera representation of events at the same timestep as the Freeway frame in Figure 6.1a.

Figure 6.1: Figure showing a conventional RGB frame of the Freeway game (a) and its event representation (b). This figure was sourced from prior work [111].

Examples of other Atari games can be found in the repository using the other observation types with links to videos of the conventional screen and the EBC simulated event plots.

6.3 Pong Case Study

© Owner/Author — Charles P. Rizzo 2023. The content in this section is the author's version of the work. It is posted here for personal use. Not for redistribution. The definitive Version of Record was published in [112], https://doi.org/10.1145/3584954.3584962.

Using the EBC_simple observation type, in order to motivate the use of counting networks as a neuromorphic downsampling utility for control applications, we walk through a case study where we train a spiking neural network as an agent to play the Atari 2600 game Pong. We use the EONS genetic algorithm to train the agent. EONS (Evolutionary Optimization of Neuromorphic Systems) [117] is a general-purpose genetic algorithm, implemented in the TENNLab neuromorphic software framework [103], which creates spiking neural networks without any specific topologies and applies genetic operators to them to hone both the structure and parameters of the networks. We leverage the wrapper application implemented for the Arcade Learning Environment (ALE) [11, 77] in the TENNLab software framework and the ALE_EBC [111] library to convert ALE screen observations to events using the EBC_simple observation type. Therefore, we have all of the pieces of the puzzle to explore training neuromorphic agents for Pong. As we describe the training, we can see the advantages of using the spiking threshold pooling networks for preprocessing.

ALE exposes two primary observation types at each timestep: the RAM that composes the game's current state and the screen which is a 210×160 vector of pixel values. Even though the size of the RAM at each step is only 128 byte values, it lacks spatio-temporal context and is therefore extremely difficult to train spiking neural networks on. The screen does have that context, but $210 \times 160 = 33,600$ inputs is a lot

of information for EONS to learn an agent on. By comparison, we have used EONS successfully to train spiking neural networks for multiple control applications with more complexity than pong [119, 102]. However, we typically limit the observations to small numbers of sensors, such as LIDARs, IR sensors, or accelerometers. With 33,600 inputs, the probability of EONS generating a reasonable agent is extremely low. ALE operates at about 60 frames per second (FPS) and commercial EBCs can operate at several thousands of FPS. While tools like V2E [55] do a better job representing what an EBC representation of a video or collection of frames looks like by artificially increasing the frame rate and having smoother frame-to-frame transitions, the ALE_EBC library can be used online during the training process. This means that as frames are generated by ALE, the ALE_EBC library can convert them to their event representations with little overhead so that EONS can train on observations that are presented as collection of events instead of as a dense array of RGB pixel values.

6.3.1 Observation Preprocessing

Even though we drastically reduce the amount of data processed at each timestep by simulating the frames as event streams, we still have an observation space of 33,600 pixels. To alleviate this issue, we use the event counting technique as a preprocessing step to downsample the input space. Before performing the downsampling step, we eliminate the first 30 rows and first 20 columns of each screen. This eliminates the scoring update banner at the top of the screen and the opponent's paddle from the input space. This is shown in Figure 6.2 and was done to remove sources of activity that were deemed unnecessary for the agent and to prevent the agent from trivially copying the movement of the opponent's paddle.

Using a chip size of 10×10, a stride value of 5, a threshold value $t = 8$, and a frame coalescing value of $f = 1$, we implement the event counting or spiking threshold pooling logic non-neuromorphically so that the agent can train on this

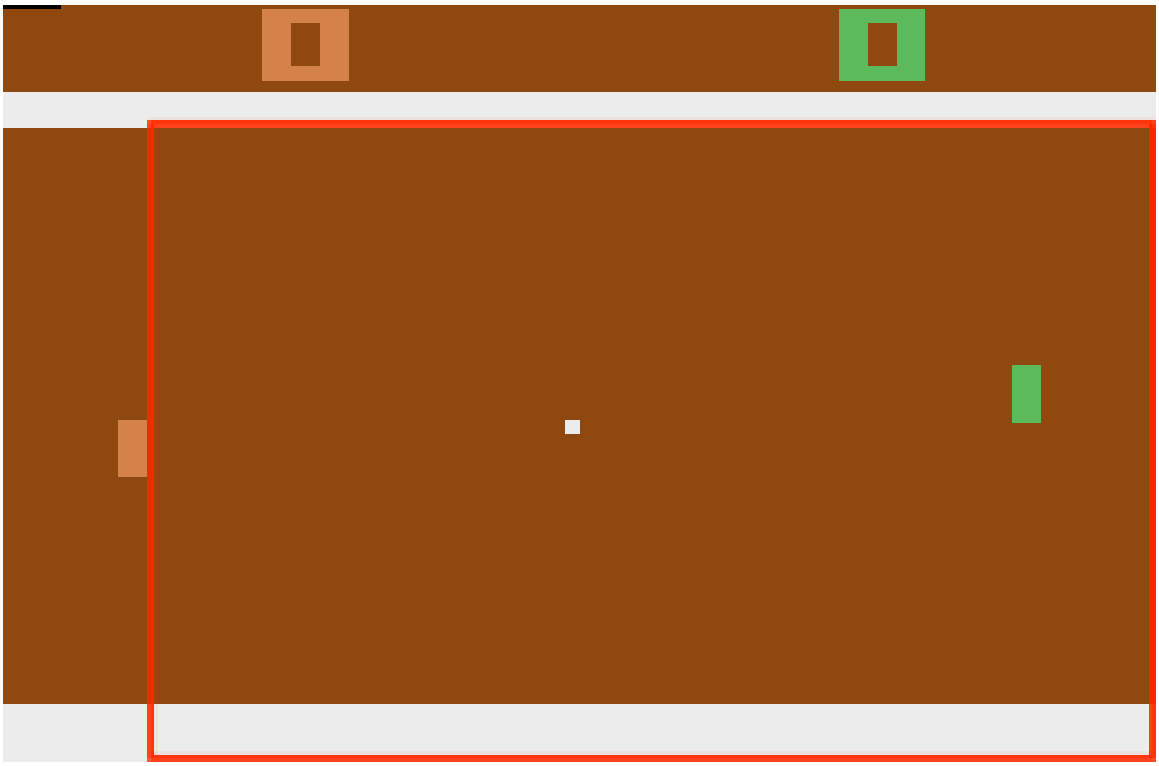


Figure 6.2: Image showing the entire Pong screen observation and the region of interest inside the red bounding box. This figure was sourced from prior work [112].

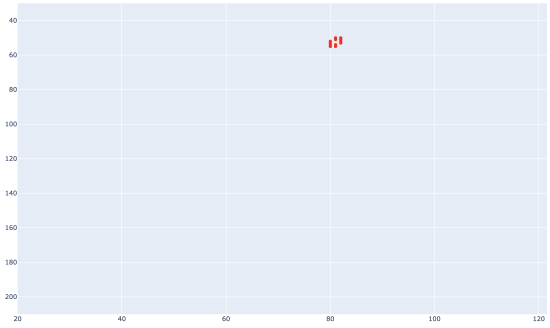
new, downsampled input space. Equation 6.1 shows the overall reduction in input space dimensionality.

$$ceil\left(\frac{160-20}{5}\right) * ceil\left(\frac{210-30}{5}\right) = 28 * 36 = 1008 \tag{6.1}$$

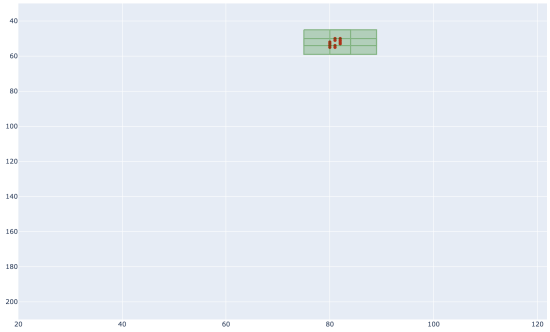
This first pass of downsampling reduces the input dimensionality from 140×180 to 28×36. This is $25\times$ less input data for the neural network to process during each timestep. Figures 6.3a and 6.3b show the original event representation of the 140×180 pixels and the downsampled version that is 28×36 green chips. In Figure 6.3b, the events are shown as red points, and the green boxes show the chip level view for the same timestep. Even though the ball appears to be centered in one green chip from the first level of downsampling, it is actually encompassed in four distinct green chips and is therefore now represented by four distinct chips/regions. While it appears to be a good representation of the input space, 1008 inputs is still a pretty large amount of inputs for EONS. Therefore, we further downsample the input space by applying the event counting technique to the newly created 28×36 input space. For the second pass of downsampling, we choose a chip size of 5×5, a stride of 5, a threshold value $t = 0$, and a frame coalescing value of $f = 1$. With these values, counting is equivalent to pooling. Equation 6.2 shows the further reduction of the input space.

$$ceil(\frac{ceil(\frac{160-20}{5})}{5}) * ceil(\frac{ceil(\frac{210-30}{5})}{5}) = 6 * 8 = 48 \tag{6.2}$$

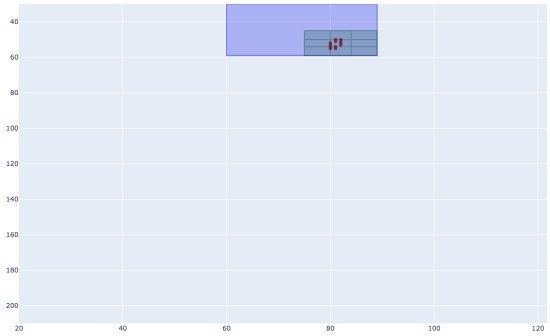
This second pass of downsampling reduces the input dimensionality from an initial 140×180 to 28×36 and finally to 6×8. This is an overall data reduction of $525\times$ per timestep. This size input space is much more tractable for EONS. Although the downsampling is performed non-neuromorphically for training, we compose the two levels of counting networks and the resulting network from EONS as a final network for testing. Figure 6.3c shows what the downsampling looks like by positioning the blue second-pass downsampling chip over the green first-pass downsampling chips.



(a) Shows the Pong ball represented as events simulated with the ALE_EBC library [111] using Plotly [56]



(b) Four green rectangles show the 4 overlapping chips that are flagged because the number of events composing the ball are greater than their thresholds. This is the first-pass downsampling view.



(c) The blue rectangle shows the second-pass downsampling view. In its region are 4 (green) chips that are flagged, which surpasses its threshold. Therefore, it in turn is flagged.

Figure 6.3: Figure showing the two-step downsampling process that results in a less resolute representation of the events. This figure was sourced from prior work [112].

The four first-pass downsampling green chips that represented the ball are now all encompassed in and represented by one blue chip when the downsampling is performed on the green chips' resolution of 28×36. While a 6×8 representation of a 140×180 frame is far less resolute, the motion and location of the pong ball are still adequately captured.

For each successive downsampling pass, each chip is binary: either it's a 0, which means not enough events have occurred in its encompassing region to cause the network to fire, or it's a 1, which implies the opposite. For Figures 6.3b and 6.3c, if a colored rectangle representing that chip is shown, it means that for that timestep, that chip has a value of 1. Even though event-based cameras differentiate between positive and negative polarity events, they can still be viewed as pseudo-binary in that each pixel either emits an event or it does not at each timestep. The counting network's binary valued output is functionally similar to how event-based cameras work, and they work well as a preprocessing layer between the camera's event stream output and the agent network.

6.3.2 Stochasicity in the ALE

In the ALE, episodic randomness is derived from a parameter called **repeat_action_probability**. When set to a number between 0 and 1, it is the probability that the agent's chosen action at time t is ignored, and instead, the action chosen at time *t-1* is repeated [77]. If set to 0 (its default value), every episode for most of the Atari games is completely deterministic. Machado et al. explore how an algorithm called "The Brute" learns to to exploit environmental determinism and how, for traditional machine learning algorithms and experiments in which score maximizing is the main goal, this property is undesirable. However, for our experiment, the determinism is desirable. It provides for a simpler experiment that allows us to test whether or not our neuromorphic downsampling approach is

effective at enabling EONS to develop a spiking neural network that can play a classical Atari game, even if it is just learning to exploit the determinism.

6.3.3 Experimental Evaluations

Because this work is focused on the plausibility of the workflow, we focused on a few tests; an in depth analysis of the training results is not included in the scope of this work. Our measure of success is the trained network's ability to reach a score greater than 0 when tested for 10,000 timesteps after training. Because the reward function for Pong is the difference of the opponent's score and the agent's score, at the start of training, the overall score for the episode was -13. This implies that the opponent scored on the agent 13 more times than the agent scored on the opponent. A positive score implies that the agent was able to score on the opponent more times than the opponent was able to score on the agent.

We expose two actions to the agent: move the paddle up and move the paddle down. This simplifies the action space down from the default 18 Atari controller actions. We chose not to include the NOOP action and modified the game difficulty such that the player's paddle size was reduced. We did this to encourage paddle movement in the agent without modifying the scoring function. For our first test, we trained with EONS for 1000 epochs using the full 140×180 EBC screen as input. Each training episode lasted only 2,000 timesteps. At the end of training, when we evaluated the network on 10,000 timesteps, the overall score was a -21, which means the opponent utterly defeated the neuromorphic agent. A score of 21 either for the agent or the opponent signifies that the game is over, and for our first experiment, the opponent reached that score in only 3,000 timesteps. When examining the agent's actions, it simply chose to move the paddle up at every timestep. We concluded that the observation space was too large for EONS to gain traction.

For our second experiment, we repeated the first experiment but this time, we used the singly-downsampled 28×36 input space. The results were the exact same

as the first experiment. Again, the observation space proved to be too complex for EONS.

For our last experiment, we again trained in the exact same manner as the previous experiments, but this time using the doubly-downsampled 6×8 observation space for input. The results were vastly different. During the 10,000 timestep testing simulation, the neuromorphic agent was able to rack up a score of 21 by timestep 8300. The agent had discovered a strategy to completely defeat the opponent. Upon visual inspection, the network learned that it needs to initially move the paddle down and then oscillate about that position at each timestep. The constant movement of the paddle reflects the ball at a steep enough angle that the opponent's paddle can not catch up to the ball in time, resulting in the agent gaining a point. The neuromorphic agent successfully learned how to exploit the game's determinism with the doubly-downsampled observation space. A video showing the agent's behavior for our last experiment can be found here: https://youtu.be/gLCBrd-TVJU.

6.3.4 Purely Neuromorphic Implementation

As noted above, this entire workflow can be done in a single spiking neural network. By tiling the first-pass event counting network over the input space, then tiling the second-pass event counting networks over the first-pass event counting networks' output space, and lastly connecting the agent network's inputs to the second-pass event counting networks' output space, we arrive at a purely neuromorphic implementation that performs both the downsampling and the decision making for Pong. Figure 6.4 shows the required neuron and synapse count for each layer and the total network size for Pong. In our example, we allow EONS to evolve the topology of the network by adding or removing synapses. This allows EONS-generated networks to have recurrence built into them.

Figure 6.4 also shows the neuronal and synaptic requirements for any generic instance of an Atari game where we don't initially remove some rows and columns

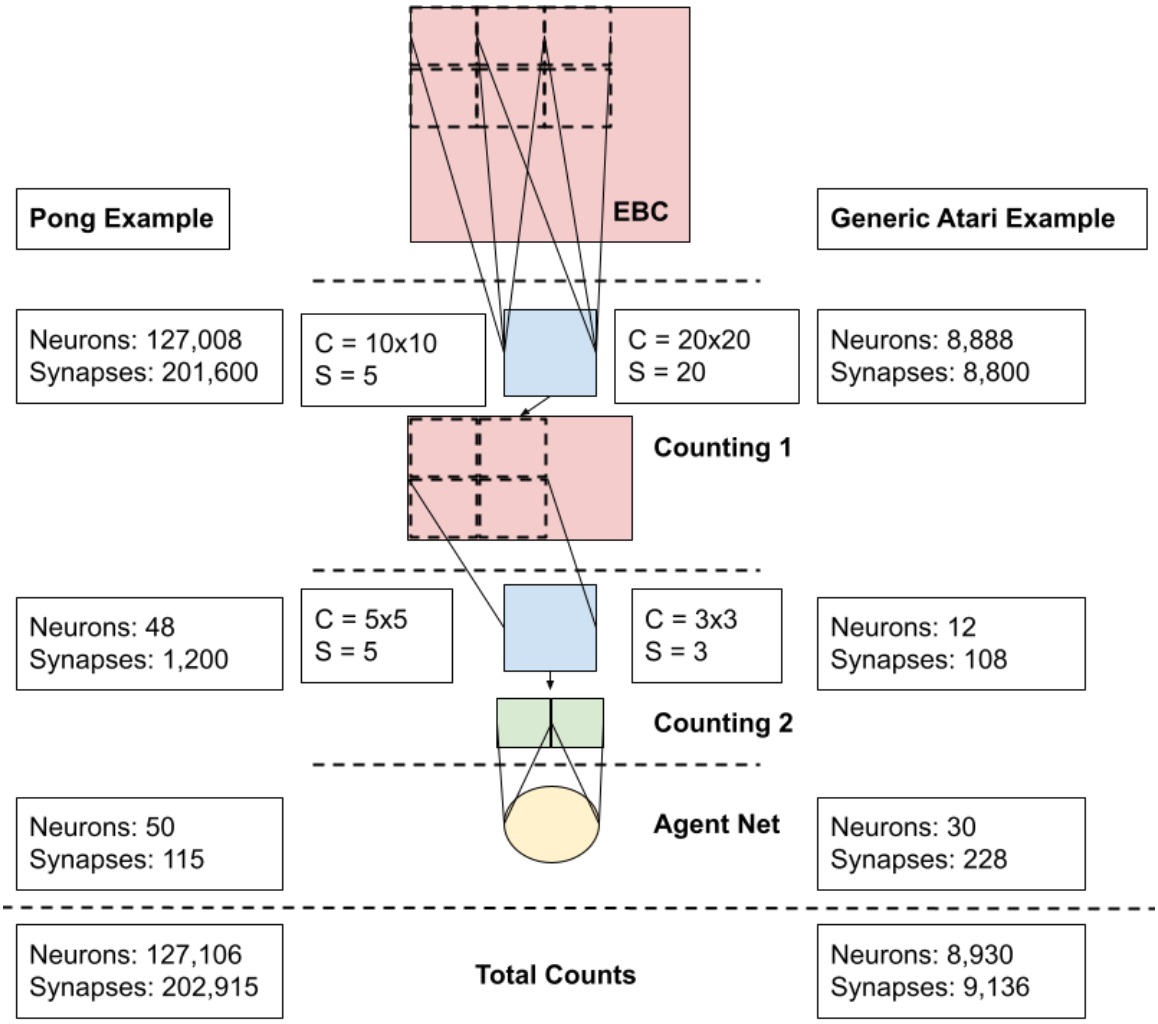


Figure 6.4: Figure showing neuromorphic workflow for both our specific Pong experiment (left) and a generic example using any other ALE-exposed Atari game (right). Different chip sizes and strides are used to highlight the impact on the final network footprint. The Pong example assumes an input dimensionality of 140×180 and the generic example assumes an input dimensionality of 160×210. This figure was sourced from prior work [112].

and reduce the size of the agent network's action space. It is worth noting that for the agent network, the numbers we use for the generic instance are estimates that assume a traditional, fully-connected, feed-forward architecture. We also assume different chip sizes and stride values for each downsampling pass to demonstrate how the required number of neurons and synapses are affected and can be reduced. Even though neuromorphic networks are typically smaller, we argue it is still compelling to have a purely neuromorphic workflow for solving more complicated control tasks where the inputs are based on event-based camera vision.

The numbers in Figure 6.4 are derived using formula 3.1 and using the counting network architecture shown in section 3.1.2. We know that when $f = 1$, as we assume it does for this case study, the amount of neurons in the counting network is $r * c + 1$ and the amount of synapses is $r * c$. Using formula 3.1 to calculate the amount of chips that need to be processed per timestep and assuming we tile the counting network over the input space that many times, we can calculate the amount of neurons and synapses that make up the first downsampling pass. Additionally, when chips overlap at the input layer as they do in our pong experiment, we need an extra neuron per input pixel and about four extra synapses per extra neuron. The neuron counts for successive downsampling passes only need consider adding an output neuron per second-pass downsampling chip because the input neurons will be the output neurons of the previous pass. The synapse count is simply the chip size multiplied by the amount of chips needed for the secondary pass. Lastly with the agent network, its size is dictated by whatever algorithm is used to develop it, whether all possible Atari actions are exposed, and by the amount of chips that make up the final downsampling pass, as its output neurons are directly connected to the input neurons of the agent network.

6.3.5 Conclusion

This work presents a method for divvying up the output space of an event-based camera and using neuromorphic solutions (spiking threshold pooling networks) to reduce the overall resolution of the field of view. Additionally, we performed a case study that motivates the usefulness of these event counting networks as neuromorphic downsampling utilities. The brief study shows that, for the EONS genetic algorithm, reducing the complexity of the input space results in a positive impact on the neuromorphic agent's performance when the observation space is simulated event-based camera footage of an Atari game screen.

We have also illustrated and highlighted that the workflow of nesting several event counting networks can be done purely neuromorphically with no need for additional preprocessing of the inputs on a CPU. For any event-based camera footage (real or simulated), layers of event counting networks can downsample the field of view by highlighting regions of importance. This aids the neuromorphic agent by reducing the amount of noise and the amount of data that needs to be processed all while retaining the spikey behavior of event-based cameras and maintaining important spatio-temporal locality information.

Chapter 7

Filtration Application

The work in this chapter is in preparation for publication.

In this chapter, we explore how the two per-event spiking neural networks presented in section 3.2 might be used in a real-time driving system to help reduce event throughput and noise while clustering the remaining events to further highlight objects of interest in the moving scene. This system is best illustrated in Figure 7.1.

7.1 Experiment

7.1.1 DSEC Dataset

To illustrate the capabilities of these networks as a real-time event denoising and clustering system, we apply them to ten sequences of the DSEC dataset [46]. DSEC is a stereo camera dataset in driving scenarios that contains data from both event cameras and global shutter color cameras. Due to event camera novelty, event camera datasets of driving scenarios are rare. DSEC is composed of 53 driving sequences that cover a variety of lighting conditions, traffic situations, and backgrounds. The event data is stored at VGA resolution, which is 640 $\times$ 480, in compressed h5 files [38] in the Address Event Representation (AER) format.

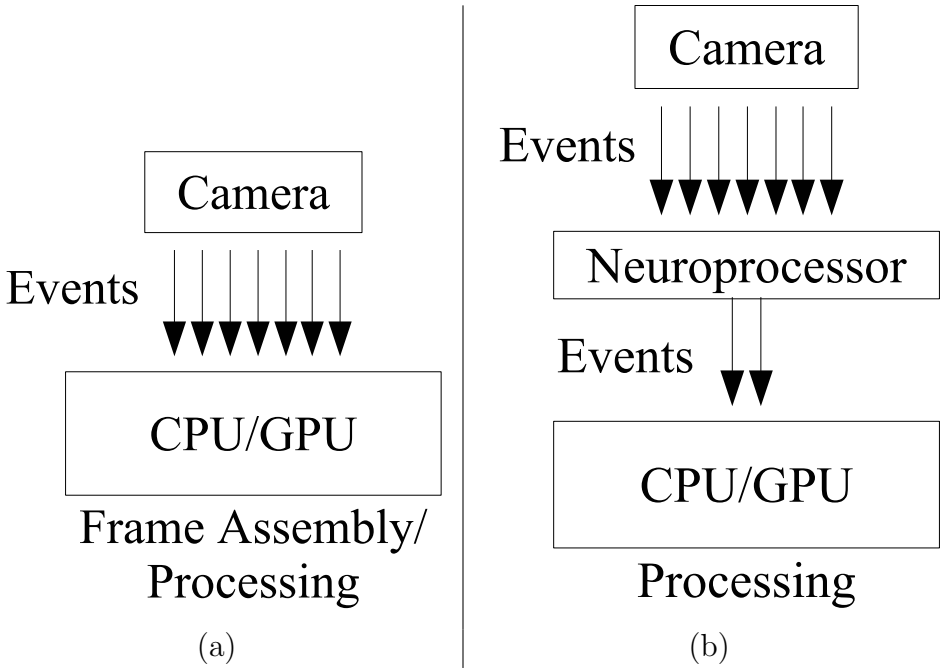


Figure 7.1: A traditional way to process an event stream from an EBC, and our proposed way to insert a low-power neuromorphic filter into the stream to lower event bandwidth and simplify processing.

The ten arbitrarily chosen sequences from the dataset that we examine are shown in Table 7.1. We also include their real-time video lengths and event file size in millions of events to highlight the quantity and density of data on which we are operating.

In each sequence, there are objects that are deemed important by the corollary DSEC-detection dataset [45]. These objects are pedestrians, riders, cars, buses, trucks, bicycles, motorcycles, and trains. In the DSEC-detection dataset, bounding boxes are drawn around these objects in the conventional camera frame capture at a frequency of 20Hz. The bounding boxes are drawn using the state-of-the-art Quasi-Dense Tracking algorithm [37, 95]. They were manually curated and contain IDs that are necessary to track a particular object over the course of a sequence. We leverage the object detection files that correlate to the ten sequences in Table 7.1 in order to track objects as they move through the event camera recordings of the respective scenes. An example of what this looks like as an event accumulation snapshot is shown in Figure 7.2. The green boxes are drawn around the objects according to the sequences' object detection files which identify multiple objects of interest and their boundaries at 50ms intervals.

Table 7.1: Table showing the ten sequences examined in this study, their event recording lengths, total number of events, and their average event densities per second.

Sequence Names	Event Recording Duration (s)	Total Events	Average Event Density (events / s)
interlaken_00_c	26.79	454,880,997	17,036,741
interlaken_00_g	66.69	605,959,064	9,086,205
zurich_01_a	33.99	315,269,622	9,275,364
zurich_01_d	39.69	501,254,909	12,629,249
zurich_02_a	11.69	177,098,240	15,149,550
zurich_04_b	13.4	129,548,333	9,667,786
zurich_04_f	42.9	361,419,518	8,424,697
zurich_09_b	18.3	166,829,066	9,116,342
zurich_13_b	15.7	186,428,145	11,874,404
thun_00_a	11.89	134,112,119	11,279,404

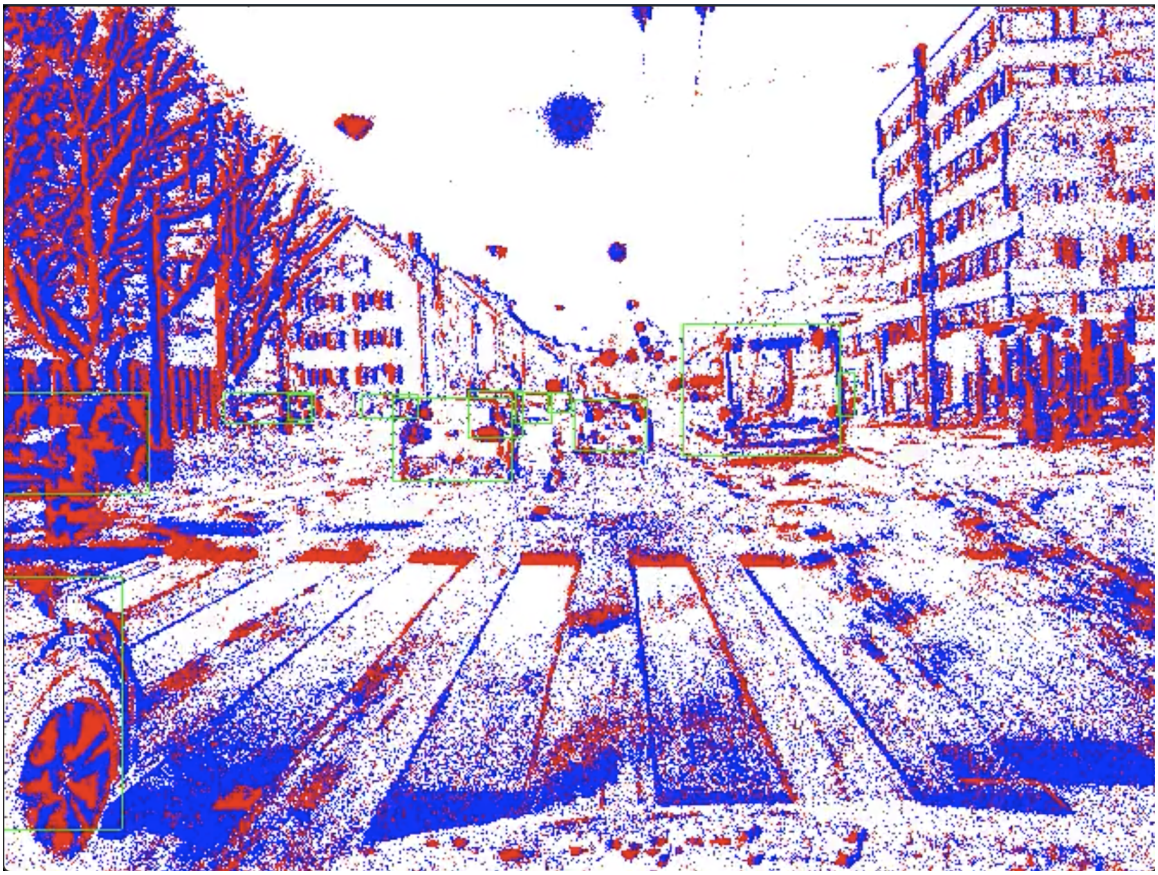


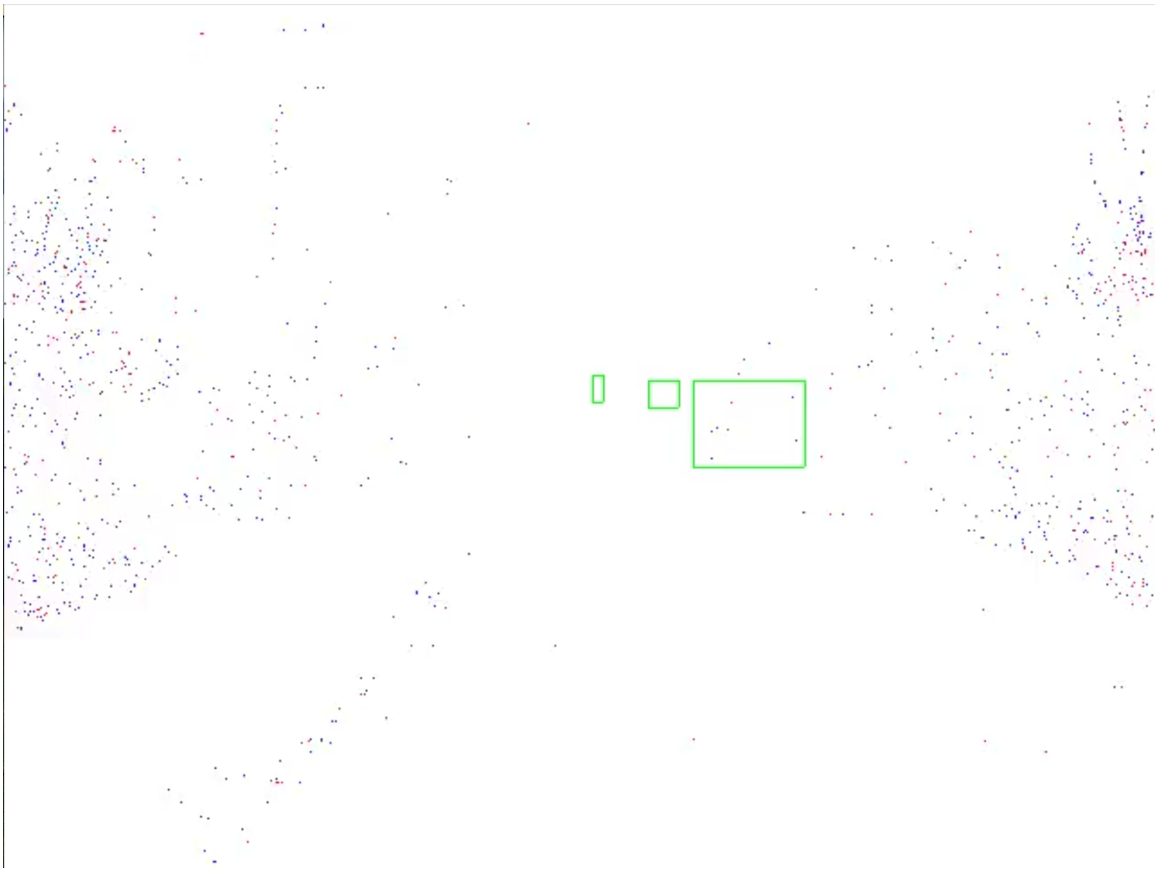
Figure 7.2: The first 50ms of events from the "zurich_01_d" sequence accumulated into an event frame.

7.1.2 Filtration with Segmentation

With the event camera generating events at a microsecond temporal resolution, we need to determine how often to flush events from the camera's event buffer to the neuroprocessor. Doing so every microsecond results in extremely sparse activity that fails to leverage the spatial feature capturing abilities of the DBSCAN filter network. Instead, we define "segment_time_length" (STL) as the amount of time during which we accumulate events from the event buffer, and then send the events to the neuroprocessor in a single batch. From the perspective of the neuroprocessor, all of these events are received in a single timestep, and following processing, a single batch of output events are sent from the neuroprocessor downstream. We discovered, while performing this activity on RAVENS, that we achieve an automatic form of filtration. When events from the same pixel are sent to RAVENS in a single batch, RAVENS discards the earlier events, and only processes the last event. This form of time aggregation is similar to building a video frame in the standard processing of events, as in Figure 7.1(a). However, unlike conventional computer vision algorithms, we do not actually construct an in-memory event frame representation; instead, we can simply operate on the sparse list of events and not waste memory by constructing a frame and imputing with zeroes or null values.

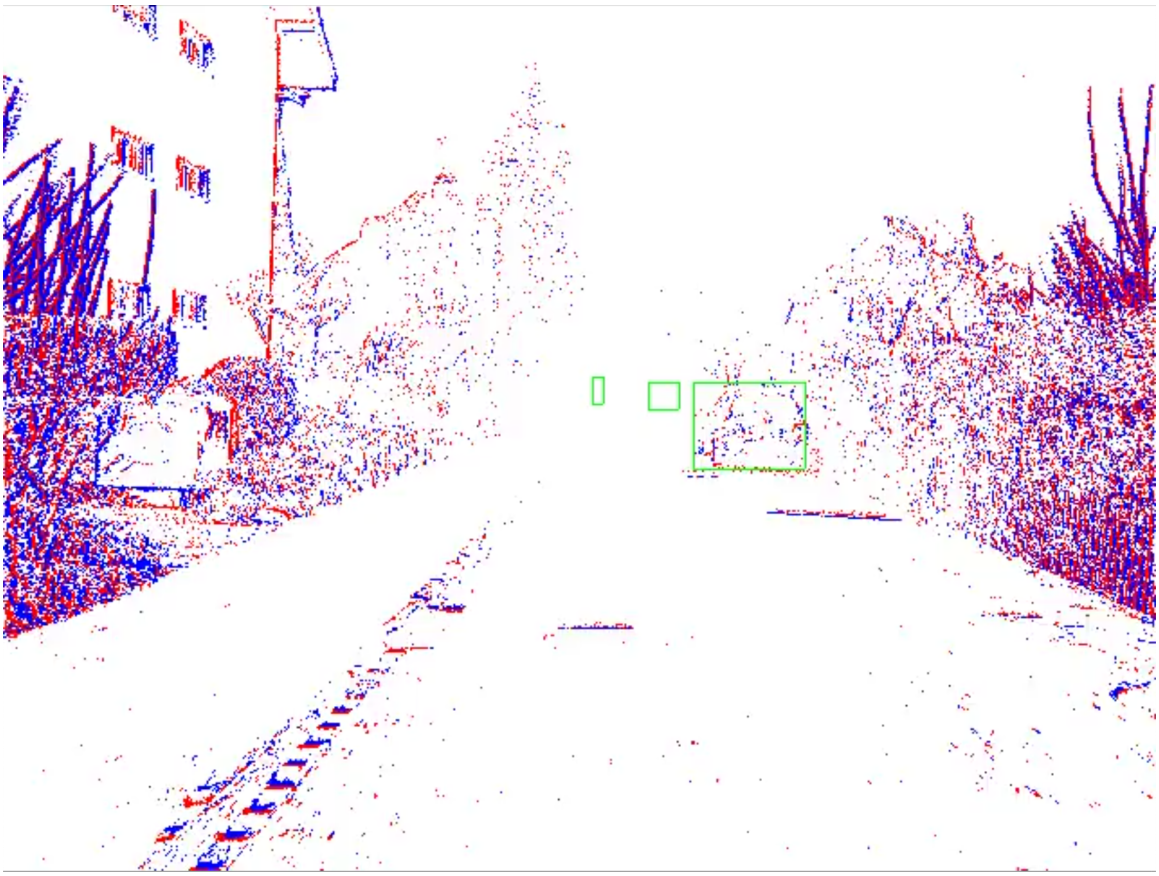
We test our neuromorphic system's ability to operate on event accumulations as short as 0.1 milliseconds. This is 500 times faster than the rate at which object labels are provided in the DSEC-detection dataset, and success on intervals this small would highlight the high-speed, real-time capability of the neuromorphic system. Being able to operate at a high frequency to denoise and cluster events is important for compatibility with more heavyweight object classification or detection algorithms downstream.

Figure 7.3 shows how differently the first 0.1ms, 5ms, and 50ms of accumulated events looks for the "thun_00_a" sequence. As the value of STL increases, spatial



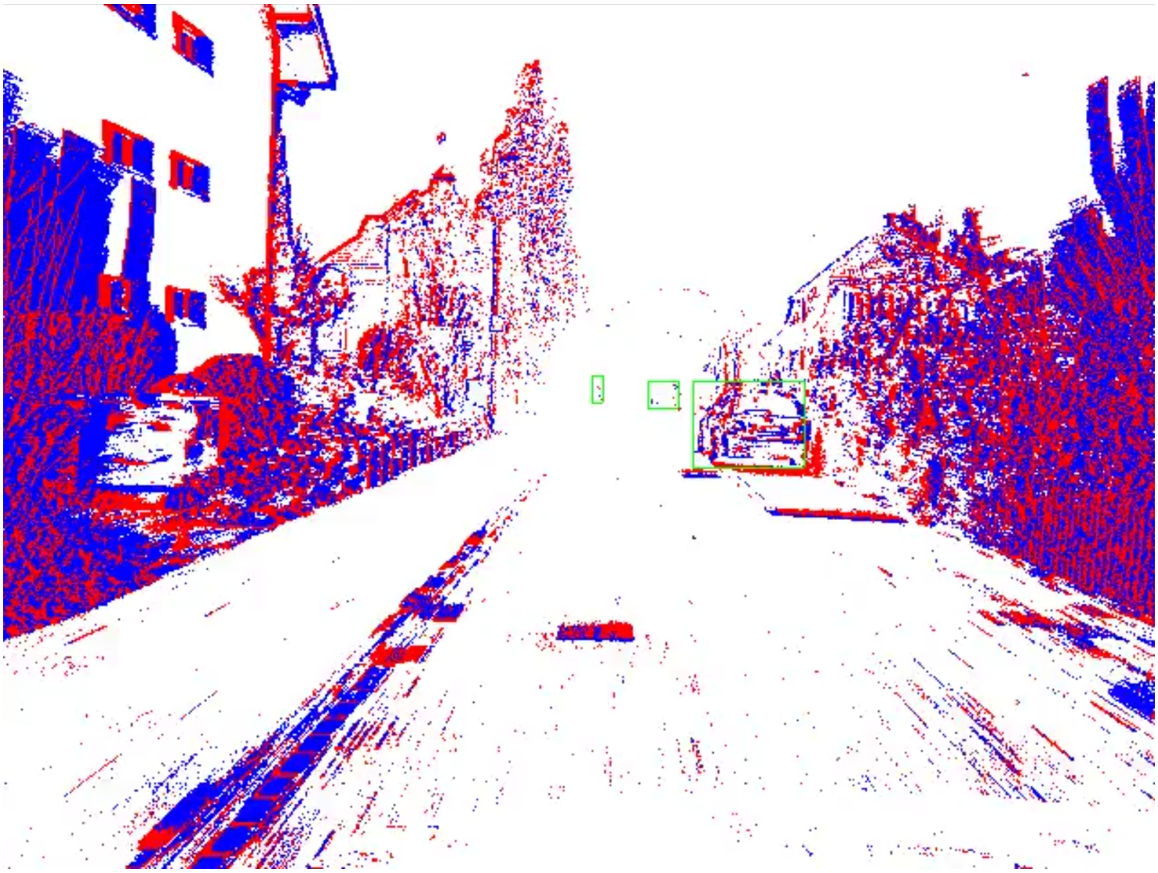
(a) The first 0.1ms worth of events accumulated into a frame for the "thun_00_a" sequence.

Figure 7.3: Figure showing various "segment_time_length" event time slices from the "thun_00_a" sequence.



(b) The first 5ms worth of events accumulated into a frame for the "thun_00_a" sequence.

Figure 7.3: (continued)



(c) The first 50ms worth of events accumulated into a frame for the "thun_00_a" sequence.

Figure 7.3: (continued)

context increases, and the event accumulation frame starts to look more and more like a binary image.

Figure 7.4 shows how many events are filtered by batching them to the neuroprocessor and increasing the STL. While the lower values perform less filtration, they also allow us to evaluate the effectiveness of the neuroprocessor working at very high speeds. Higher values perform more filtration and therefore may be more desirable in practical installations, but there are also situations where less aggressive batching filtration is desired and processing the events at higher frequencies is necessary. These include high-speed applications like autonomous driving, spacecraft maneuvering, etc. In our later experiments, we evaluate all STL values shown in Figure 7.4.

7.1.3 Parameter Exploration

Table 7.2 shows the parameters explored for both the speed filter network and the DBSCAN network implementations. For this work, we only test up to a two-tiered serial network approach where we have either a speed filter network followed by a DBSCAN filter network, or a DBSCAN filter network followed by a speed filter network. We perform all of the combinations shown in Table 7.2. There are a few invalid combinations in the table for the DBSCAN filter network where the threshold $t_d > (2 * \epsilon_d + 1)^2$, but we ignore those combinations and are left with 3,304 valid combinations per sequence.

Our goal is to leverage the speed filter and DBSCAN filter techniques to discard noise events and events related to objects that we are uninterested in (e.g., buildings, trees, etc.) while maximally recalling events that compose the objects labelled in the DSEC-detection dataset files for the ten sequences. To accomplish this, we perform a modest hyperparameter exploration of the two constituent networks in order to identify ideal network parameters per sequence.

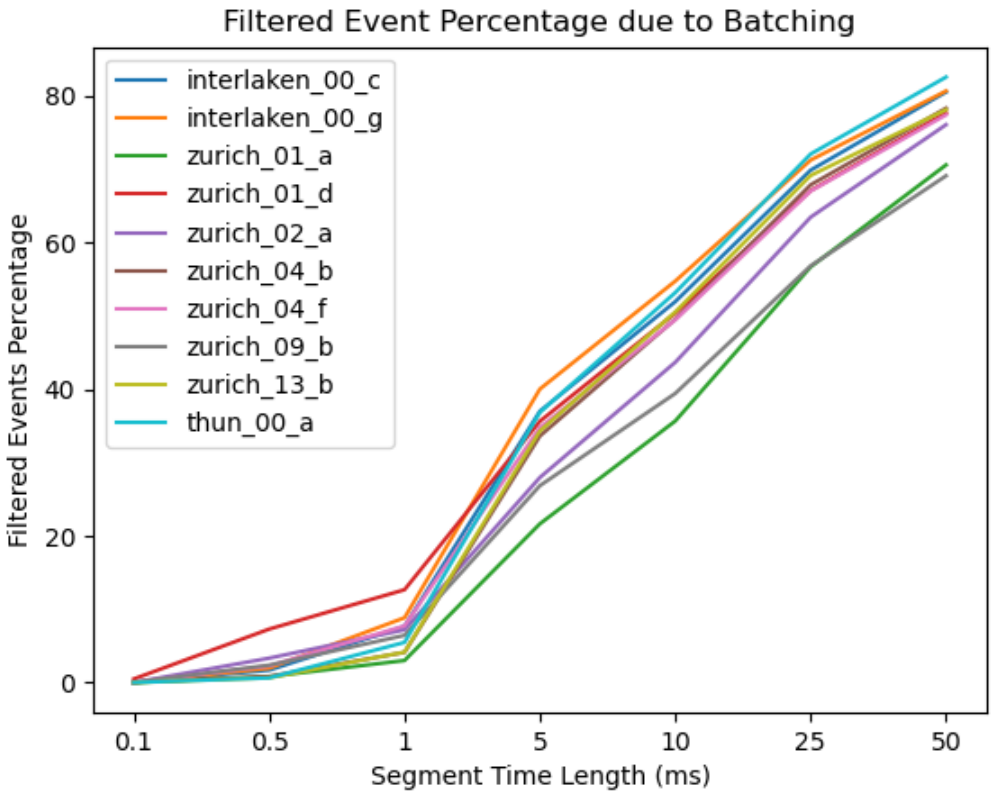


Figure 7.4: Figure showing events that are filtered by batching events at different STLs per sequence.

Table 7.2: Parameters explored for both the speed filter and DBSCAN filter techniques.

Parameter	Values
STL (ms)	0.1, 0.5, 1, 5, 10, 25, 50
ϵ_s	1
Threshold t_s	2, 4, 8, 16, 18
Speed Filtration Type	Fast, Slow
ϵ_d	2, 4, 6, 8
Threshold t_d	2, 4, 8, 16, 32, 64
Structure	Speed, DBSCAN, Speed/DBSCAN, DBSCAN/Speed

7.1.4 Ensemble Exploration

Each sequence contains objects moving at different speeds, so it is likely that no singular parameter combination will recall all of the events composing every object in a dynamic scene. In addition to the filtration networks' hyperparameter exploration, we performed a more in depth study on two sequences ("thun_00_a" and "zurich_09_b"). We denote this additional study as the ensemble experiment in which, for STL equal to 25ms, we examine how an ensemble of 2 of the possible 472 parameter combinations performs. There are $\binom{472}{2} = 111,156$ ways to create an ensemble, and across 2 datasets, we examine over 200,000 ensembles. The idea is to see if there's an ensemble of two network parameter configurations that complement each other so that object recall rate (true positive rate) and noise filtration (true negative rate) are higher than any singular network parameter combination on the respective sequence when STL is 25ms. To perform this analysis, we performed set operations on the filtered results of different parameter combinations with an unfiltered base case.

7.2 Results

Our primary evaluation metric is balanced accuracy, which is an average of the true positive rate and true negative rate. Simply put, in our experiments, true positive rate is the percentage of objects' events correctly recalled, and true negative rate is the percentage of events filtered out that are either noise or do not compose the objects we're interested in. While a higher balanced accuracy value indicates better performance, increasing recall and increasing filtration are conflicting goals. Presenting some sort of weighted balanced accuracy might make for more attractive results, but we instead opt to report the standardized balanced accuracy metric. Furthermore, all of the results in Figures 7.5, 7.6, 7.7, and 7.8 show the networks' performance after filtration due to batching occurs, while Figure 7.4 shows the filtration benefit that batching the events and sending them to the neuroprocessor

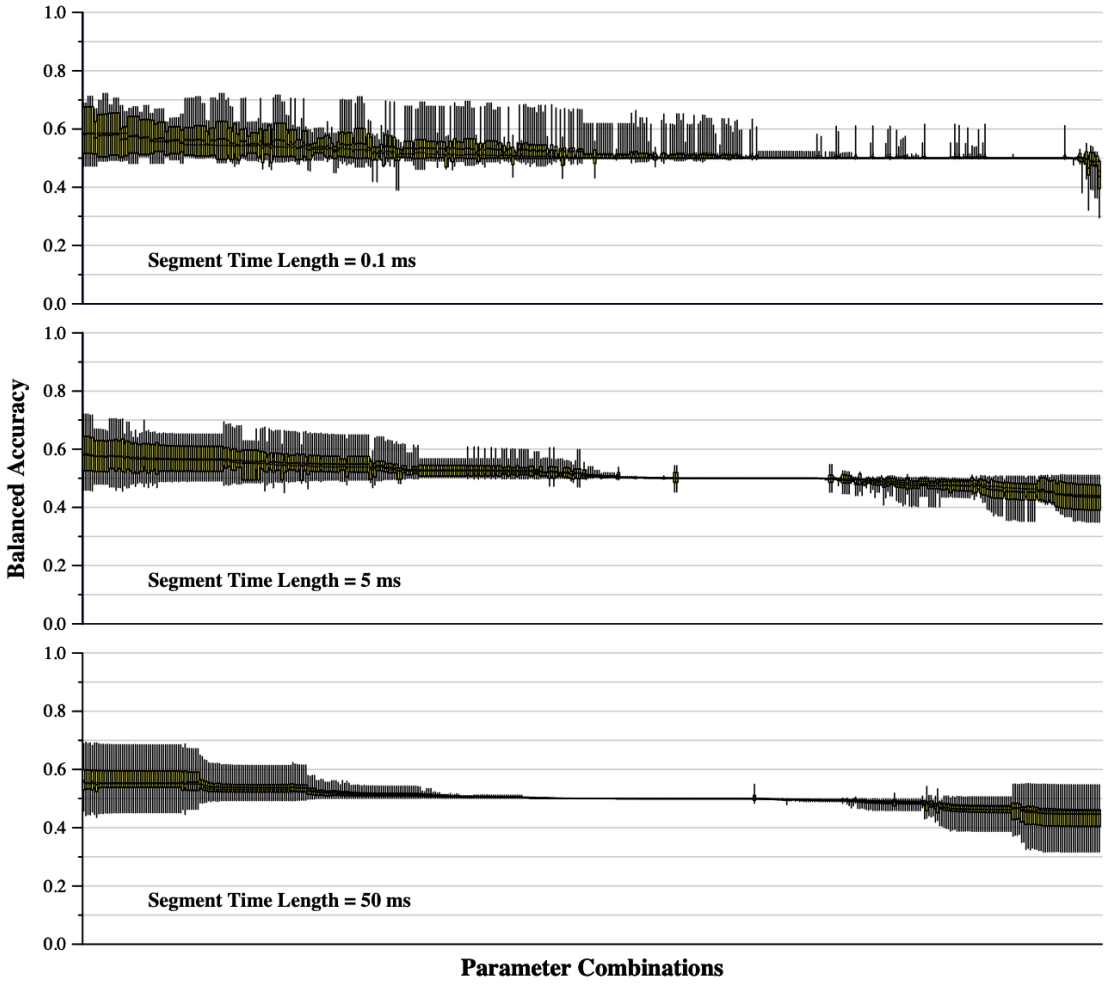


Figure 7.5: Each tukey plot represents a parameter combination of speed filter and/or DBSCAN filter. Each tukey plot is that parameter combination's balanced accuracy score across all ten tested sequences. They are sorted in descending order by median value.

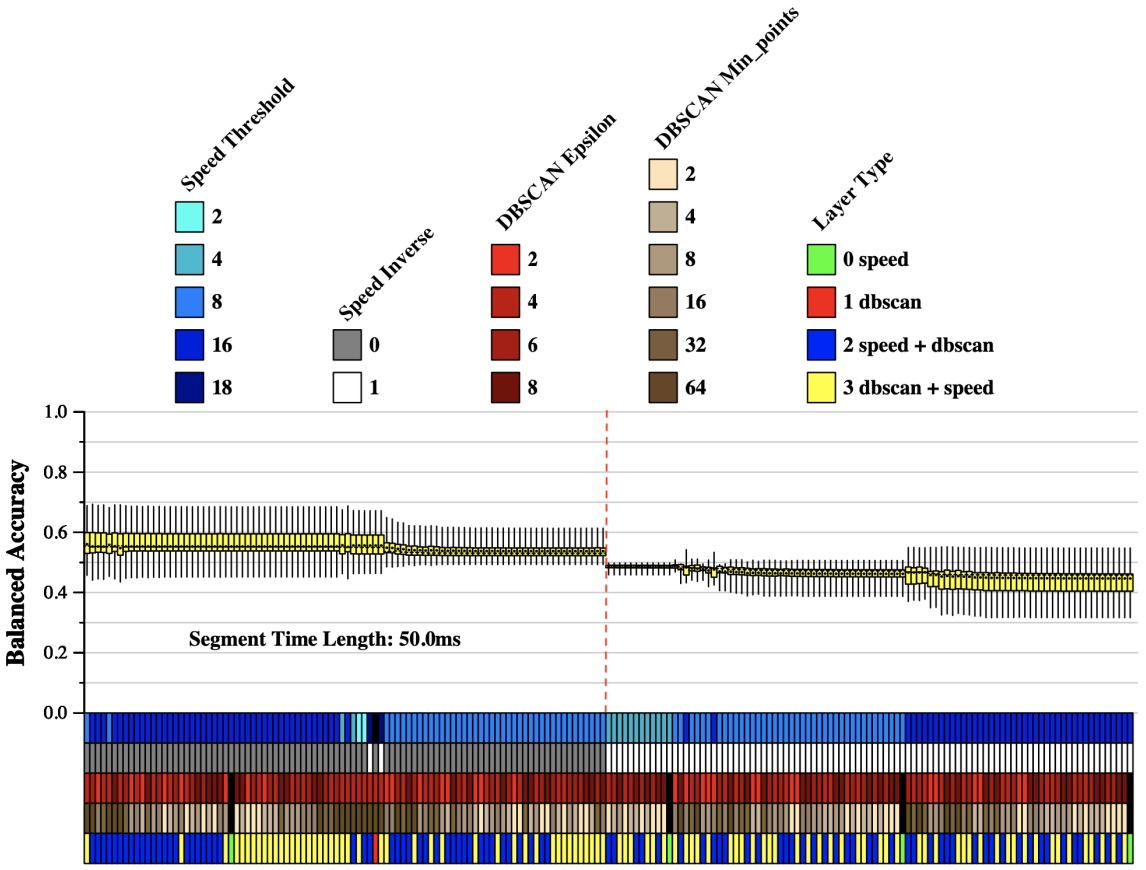


Figure 7.6: Each tukey plot represents a parameter combination of speed filter and/or DBSCAN filter. Each tukey plot is that parameter combination's balanced accuracy score across all ten tested sequences. They are sorted in descending order by median value. The vertical, red dashed line shows the split between the top 20% of performers and the bottom 20% of performers.

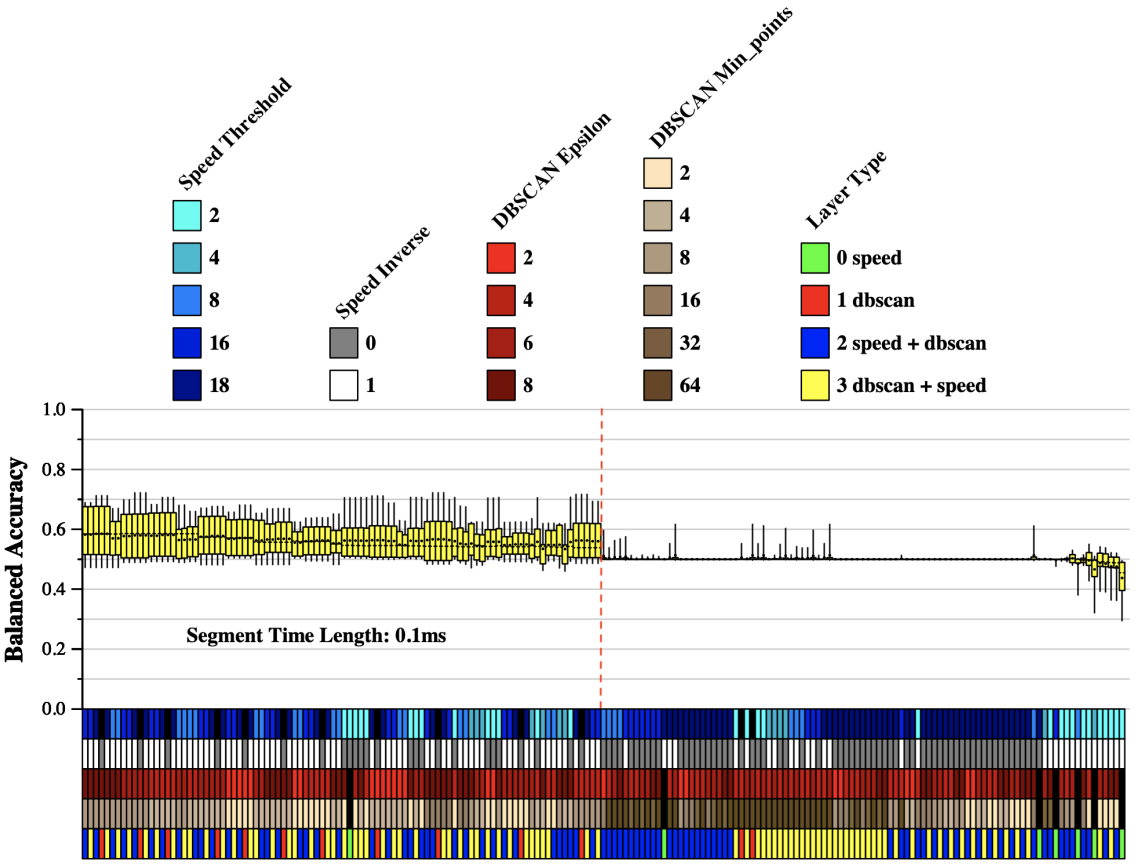
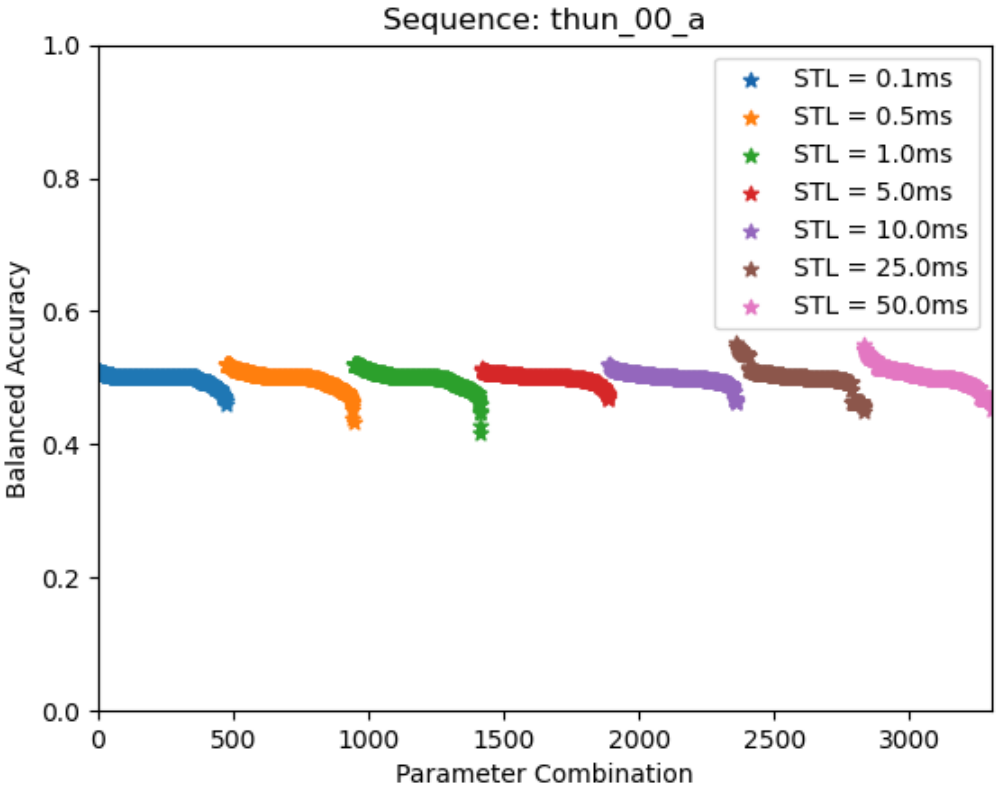
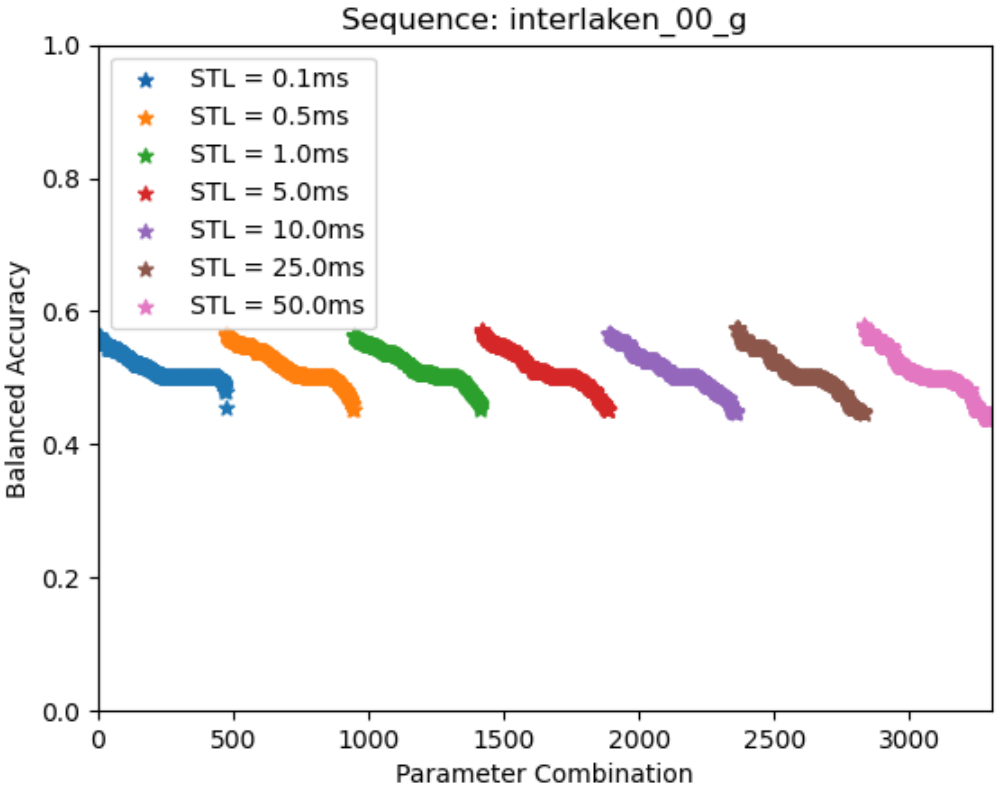


Figure 7.7: Each tukey plot represents a parameter combination of speed filter and/or DBSCAN filter. Each tukey plot is that parameter combination's balanced accuracy score across all ten tested sequences. They are sorted in descending order by median value. The vertical, red dashed line shows the split between the top 20% of performers and the bottom 20% of performers.



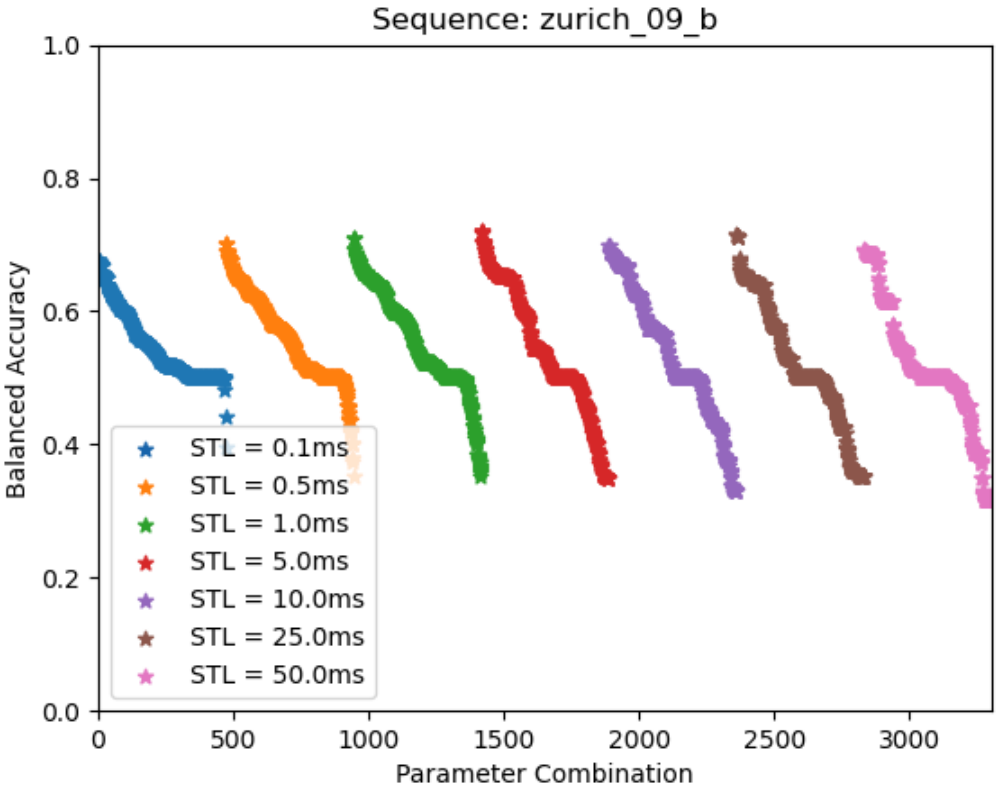
(a) Balanced accuracies of the different parameter combinations on the "thun_00_a" sequence at different STL values.

Figure 7.8



(b) Balanced accuracies of the different parameter combinations on the "interlaken_00_g" sequence at different STL values.

Figure 7.8: (continued)



(c) Balanced accuracies of the different parameter combinations on the "zurich_09_b" sequence at different STL values.

Figure 7.8: (continued)

yields across each sequence for all STL values. This is done to have an apples-to-apples comparison of the networks' performance at different STL values without the percent of events filtered due to batching skewing results.

A balanced accuracy value of 0.5 can indicate that either all events are erroneously filtered out or all events are erroneously recalled as belonging to objects. Figure 7.5 shows tukey plots for three different STL values where each tukey plot corresponds to a parameter combination's performance across all ten sequences. For all sequences, operating at lower STL values yields wider ranges of performance within a specific parameter combination. As the STL value approaches its max at 50ms, parameter combinations' performance range shrinks, and larger swaths of parameters yield neither a negative nor positive impact on balanced accuracy. Additionally as the STL value increases, there is little variation in performance for both the best and worst sets of parameter combinations. Networks operating at lower STL values are more sensitive to parameter changes but also tend to perform slightly better across the board.

Looking at parameter trends more closely, Figure 7.6 shows the parameter combinations that compose the top and bottom 20% of performers when STL is 50ms. For the speed threshold parameters, the better performers filter out events that have less than either 16 or 8 neighboring events that occur across two time segments or in a 100ms period. There is no discernible trend for the DBSCAN filter parameters. Additionally, the top 20% of parameter combinations are composed predominantly of DBSCAN + speed filter or speed + DBSCAN filter network stacks. There are only two outliers where a sole DBSCAN filter or a sole speed filter network performs well enough to be in the top 20%. This means that as the input event stream becomes more dense (resembling a binary image), both a speed filter network and DBSCAN network are likely needed to attain better performance.

Figure 7.7 shows the same as Figure 7.6 but for a STL of 100 microseconds (0.1ms). For the speed filter parameters, the best performers tend to filter out events that have more than 8 neighboring events occur across two time segments or in a 0.2ms period.

Additionally, likely due to the event sparsity at a lower STL, the best DBSCAN filter network parameters have larger epsilon radius values and min_points values between 4 and 16. However, unlike Figure 7.6, in the top 20% of parameter combinations are several instances where only a properly parameterized DBSCAN filter network is sufficient for comparable performance, without the need for any speed filter. The top performing singular DBSCAN layer is composed of $(2*8+1)^2*2+6 = 584$ neurons, $(2*8+1)^2*3+6 = 873$ synapses, and takes $8+4 = 12$ cycles to process one event. Compared to large and complex convolutional and LSTM networks, this is a very small network. The significance of this is that multiple instances of the network can be loaded onto relatively resource-constrained hardware to parallelize the DBSCAN classification of events.

At the individual sequence level, sometimes the DBSCAN and speed filter network parameters impact balanced accuracy, and other times they do not. Figure 7.8 shows the parameter combinations and their balanced accuracies for each of the seven STL's on three different sequences: "thun_00_a" in Figure 7.8a, "interlaken_00_g" in Figure 7.8b, and "zurich_09_b" in Figure 7.8c. Figure 7.8a shows that on the "thun_00_a" sequence, the hyperparameters tested have little to no effect, regardless of STL value. Figure 7.8b is similar, but shows that there are parameters that result in better performance with respect to others. Alternatively, Figure 7.8c shows an example where the parameter values chosen for the networks drastically impact performance. This indicates that the filter networks' efficacy is extremely environmentally dependent (or sequence dependent in the case of the DSEC dataset), requiring a comprehensive hyperparameter exploration per prospective environment deployment. The average event density per second column in Table 7.1 is included to show that it is not event density alone that indicates whether or not the parameters explored in this study are effective.

7.2.1 Ensemble Experiment

Figure 7.8 shows that for the "thun_00_a" sequence at 25 ms, the parameters made little difference in the overall balanced accuracy, where as for the "zurich_09_b" sequence, the parameter values had a large impact. We hypothesized that the ensemble experiment would identify a combination of network parameters that would overcome the "thun_00_a" sequence's parameter indifference. Additionally, we hypothesized that some combination of one of the best and worst parameter performers on the "zurich_09_b" sequence would result in a much higher balanced accuracy score because of the apparent difference in parameter performance shown in Figure 7.8. Ultimately, we learned that for the "thun_00_a" and "zurich_09_b" sequences at a STL of 25 ms, the best ensembles performed only 0.08% and 0.1% better than their respective best single network configuration performers. This unimpressive result led us to believe that because a 25ms event frame closely resembles a dense binary image with sufficient spatial context, it was possible that the batching filtration was too aggressive to reap the benefit of an ensemble of networks for the parameter values we examined.

We performed one last additional experiment in which we reduced the STL value to 0.5ms and operated only on the first 5 seconds of the "zurich_09_b" sequence. Our new hypothesis was that at a lower STL with less aggressive batching filtration and for a sequence where different parameter combinations bore a large impact on performance (as shown in Figure 7.8) we would have greater success with an ensemble. Technically, we were correct; however, our best performing ensemble performed only 0.4% better than the best single network performer. After a third unimpressive result, we decided not to pursue the ensemble experiment further. We don't show any figures for this additional experiment because they do not show anything different trend-wise from what is already shown in Figure 7.8.

7.2.2 Conclusions

Even though our ensemble experiment did not yield expected results, we hypothesize that a more comprehensive parameter search for the speed filter and DBSCAN networks would boost ensemble performance. Regardless, across all tested STL values, we showed success in processing the events of several DSEC sequences with precisely defined speed filter and DBSCAN spiking neural networks. We presented a low-power method for filtering and preprocessing event data to help reduce event bandwidth and simplify processing. We learned that as the STL value grows, we gain the additional benefit of event filtration via batching when sending the events to the neuroprocessor. Additionally, we demonstrated success at lower STL values where we don't have the benefit of event filtration via batching. However, being able to operate at lower STL values and therefore higher frequencies is necessary for data processing in high-speed or high-frequency situations.

Our best balanced accuracy performance came from operating at the lowest STL of 0.1ms, and because of the large performance variability at that STL, it is warranted to perform an even more comprehensive parameter search for the speed and DBSCAN networks. Specifically, increasing the ϵ_d radius for the DBSCAN networks appears to be worthwhile going forward. We also learned that just a single optimally parameterized DBSCAN network was sufficient to achieve comparable performance in the top 20% of performers for the lower "segment_time_length" values. When the "segment_time_length" was 0.1ms, the best singular DBSCAN network was only composed of roughly 500 neurons and less than 1,000 synapses. The success of this compact network suggests the potential for deploying networks in parallel as an ensemble, tuned towards an array of hyperparameters, to support event recall for objects moving at a range of different speeds while lowering the overall event bandwidth associated with such high volumes of data.

Going forward, a more comprehensive hyperparameter study for the speed filter and DBSCAN networks is justified by the variability of the networks' efficacy across

different sequences. Evaluating on other sequences and even datasets would also be beneficial for determining other environments and situations in which this approach can be successful. Additionally, compiling the networks together and testing it on FPGA hardware is a great next step to realizing the neuromorphic system in the real-world as an event preprocessing module. Continually moving toward leveraging spiking neural networks to process event-based camera data as small, low-power computational elements that can fit on available hardware is the goal. These findings warrant using these hand-tooled spiking neural networks as event denoisers, filters, and spatial feature extractors to aid more heavyweight downstream object classifiers.

Chapter 8

Ongoing/Future Work

This chapter lists some ongoing and future work directions with what I have presented so far.

8.1 Atari Game Agent Training

This is an extension of the Pong experiment presented in Chapter 6. Following the suggestions listed in [77], we have been training and developing SNNs with EONS to play a variety of different Atari games using simple spiking threshold pooling networks for downsampling and simulated event-based camera events (through the ALE_EBC library) for input. We aim to compare the performance results to State-of-the-Art (SOTA) spiking DeepQ trained models and other conventional models that have been developed to play Atari games in the ALE. We plan to evaluate overall scores on each game and power consumption in the form of spike activity.

Some preliminary results on a handful of Atari games are shown in Table 8.1. For reasons we are not yet sure of, EONS easily outperforms other SOTA models on Asteroids specifically. This is more closely shown in Figure 8.1 for 5 uniquely seeded EONS training runs. The dashed line indicates the best performance from Google's DeepMind model. Additionally, we have had great success with the

	EONS	Mnih et al. [81]	Machado et al. [76]	Human [81]
Asteroids	1,995.6 (331.1)	1629 (542)	528.5 (37)	13,157
Atlantis	88,964 (32,945.8)	85,641 (17,600)	232,442.9 (128,678.4)	29,028
Breakout	21.8 (5.5)	401.2 (26.9)	35.1 (22.6)	31.8
Freeway	28.4 (1.1)	30.3 (0.7)	33.0 (0.3)	29.6
Pong	10.9 (10.2)	18.9 (1.3)	15.1 (1.0)	9.3
Seaquest	555.2 (119.3)	5,286 (1,310)	1,485.7 (740.8)	20,182
Space Invaders	1,107.4 (125.1)	1,976 (873)	823.6 (335)	1,652
Video Pinball	145,875.6 (12830.6)	42,684 (16,287)	15,398.5 (2,126.1)	17,298

Table 8.1: Preliminary Atari game training results compared to Google's DeepMind [81] and the Arcade Learning Environment authors' updated DeepMind model [76].

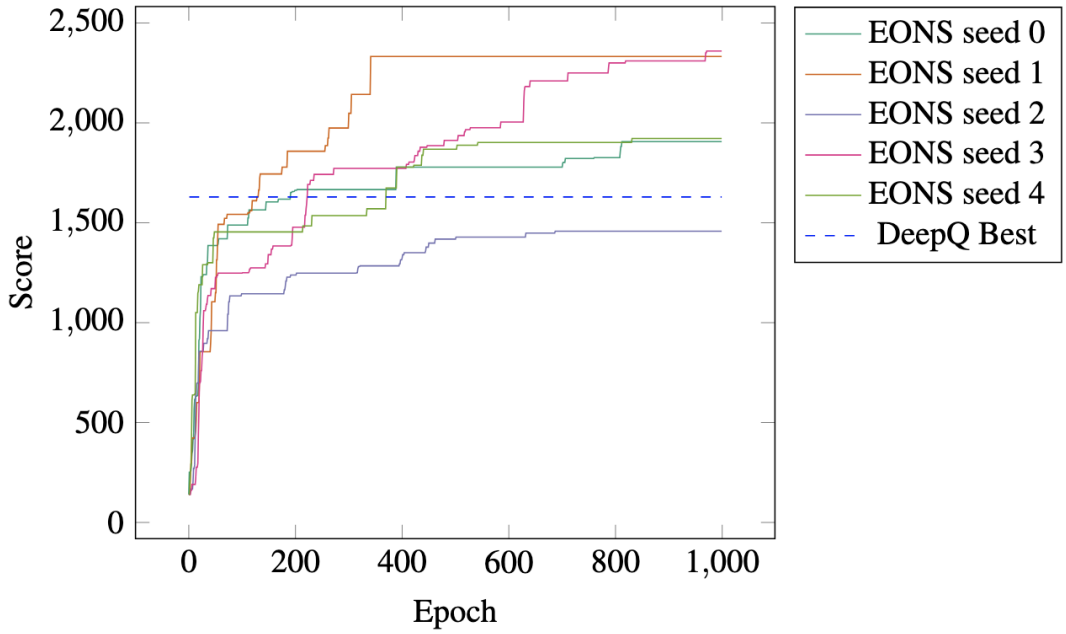


Figure 8.1: Five uniquely seeded Asteroids episodes' scores compared to SOTA spiking DeepQ model.

Video Pinball game. For every other tested game, we have had comparable performance with Seaquest being the only outlier that currently performs much more poorly.

Given that one of the main selling points of neuromorphic computing is its low power consumption, we have replicated a couple of Atari DeepQ SNN workflows from other literature in order to collect their neuron firing information for power estimates. In [132], Tan, Patel, and Kozma train DeepQ ANN's using PyTorch [97] and then convert those ANNs to SNNs using BindsNET [54]. We have duplicated their setup and are able to train and test their SNNs on different Atari games. Additionally in [131], Sun, Zeng, and Li leverage PyTorch with BrainCog [143] to train SNNs "natively". They train using the Tianshou Reinforcement Learning library [140]. We have also duplicated this workflow for developing spiking neural networks to play different Atari games. For both workflows, upon evaluating the SNNs, we print out neuron activity to a file so that we can count neuron firings, which we link directly to power consumption for our hypothetical neuromorphic hardware system.

We will compare our small EONS trained neural networks (order of magnitude of hundreds of neurons) and their activity to these much larger DeepQ models (which are all derivatives architecturally of Google's DeepMind model that is roughly composed of over 21,000 neurons containing over 1,600,000 trainable parameters) in two ways. First, we will collect the firings of the small EONS trained agent SNN which we can compare to the larger DeepQ models' neuron firings that occur in their final fully connected layers. Second, for the spiking threshold pooling network components that are responsible for downsampling the Atari observation, we will argue that we can implement the networks on bi-level memristive hardware. We can count the spikes necessary for downsampling the original Atari resolution down to what EONS uses to train on and assign smaller power-per-spike values to this subset of spikes. We can compare this value to the spikes that occur in the spiking DeepQ models' convolutional and pooling layers. In this hypothetical system, we leverage memristive

hardware to perform the downsampling or spiking threshold pooling and then a taped-out RAVENS ASIC to run the small agent SNN developed by EONS. We hope to show comparable or better performance and less power consumption through lesser spike activity when compared to the larger spiking DeepQ models.

8.2 Embedded Neuromorphic Atari Demo

The goal of this project is to realize the developed Atari agents in a real-time embedded system, where a spiking neural network is receiving real-time observations from a TV screen showing an Atari game and producing actions that control the Atari game via a Digital-to-Analog (DAC) controller. This is a closed-loop hardware demo that connects a DVS346 camera to neuromorphic hardware to an actual Atari system. The camera will be aimed at a TV showing an Atari game, and a network running on the neuromorphic kit will use camera's events as input and directly play the Atari game. Figure 8.2 shows a system diagram of the setup.

I am co-mentoring the NeuroPong senior design team (with Bryson Gullet) that is helping with this project. The current progress is a hardware system that connects a DVS346 camera to a Raspberry Pi4 which receives events from the camera, formats them, and sends them to a Neuromorphic Starter Kit [101] (NSK) via the SPI interface. The Neuromorphic Starter Kit is a hardware system composed of a base printed circuit board (PCB), a spike encoder unit, a neuroprocessor unit, a spike decoder unit, and power to the system. Bryson Gullet is the leader developer of the NSK project. For our experimentation, the spike encoder unit, the spike decoder unit, and the neuroprocessor unit are all Raspberry Pi PICOs. The neuroprocessor Raspberry Pi PICO is running the microcontroller implementation of the RAVENS neuroprocessor. A long-term goal is to replace the neurorprocessor microcontroller unit with an FPGA (like the CMOD Artix-A7). One of the major benefits of the kit is that this hot swapping of components (assuming some common form factor) is supported and developed for.

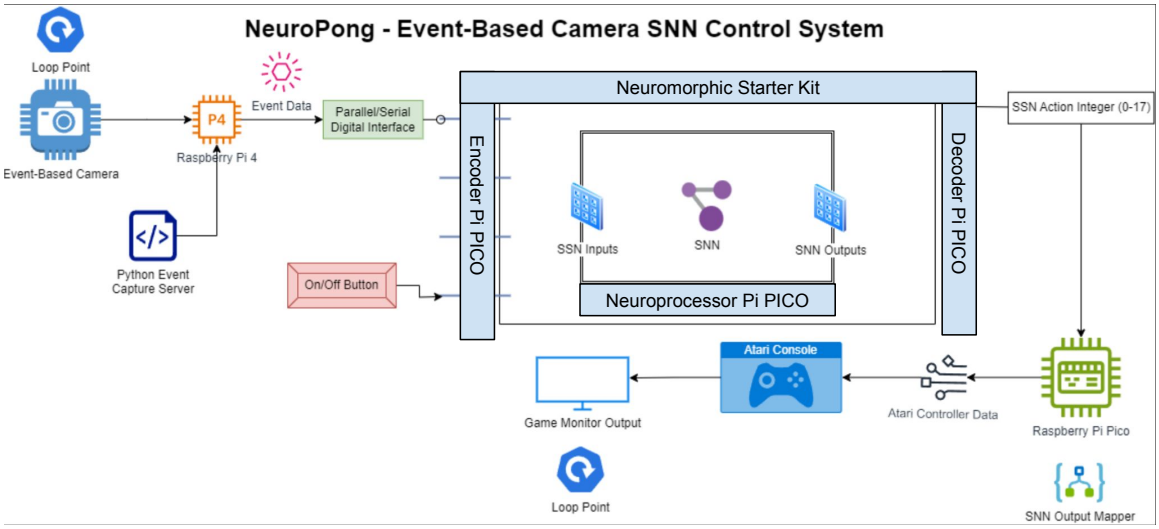


Figure 8.2: Figure showing the system diagram of the NeuroPong hardware setup. This figure was provided courtesy of the NeuroPong Senior Design Team: Alex Crumley, Mason Hyman, Julia Steed, Maxwell Marcum, Frank Standaert, and Carter Earheart-Brown.

Spike raster-level verification is currently being performed by the senior design group to verify that the hardware system produces the same spike activity from the same observation spikes as it does in software simulation. Additionally, they are working to rectify differences in frequencies between various system components. In other words, the Arcade Learning Environment operates at 60Hz, the actual Atari hardware produces observations at approximately 30-50Hz, the camera produces events at 1MHz, and the TV has some refresh rate estimated to be between 60-120Hz. By synchronizing the system to one frequency, we hypothesize that we can then tweak the ALE training FPS to mimic what the hardware can support in real-time without much latency. Our goal is to run the entire system at 30FPS with negligible latency. Additionally, the senior design team has made it possible for a human to play against the neuromorphic agent.

This project will stress test the Neuromorphic Starter Kit, introduce an embedded neuromorphic control application running in real-time that can leverage the spiking threshold pooling techniques, and highlight coupling a neuromorphic camera with neuromorphic hardware. Figure 8.3 shows the current hardware setup and its connectivity in the laboratory environment.

8.3 Publish the DBSCAN Filtration Work

The work that was initially presented was rejected, so I need to rework the results to be more compelling for resubmission at another venue. I need to get back on the proverbial horse.

8.4 EONS Modification

While genetic algorithms are not my specialty, I am proposing a modification to the EONS training algorithm. Currently, the way EONS works is that it allows users to develop spiking neural network agents without regard to parameter initialization

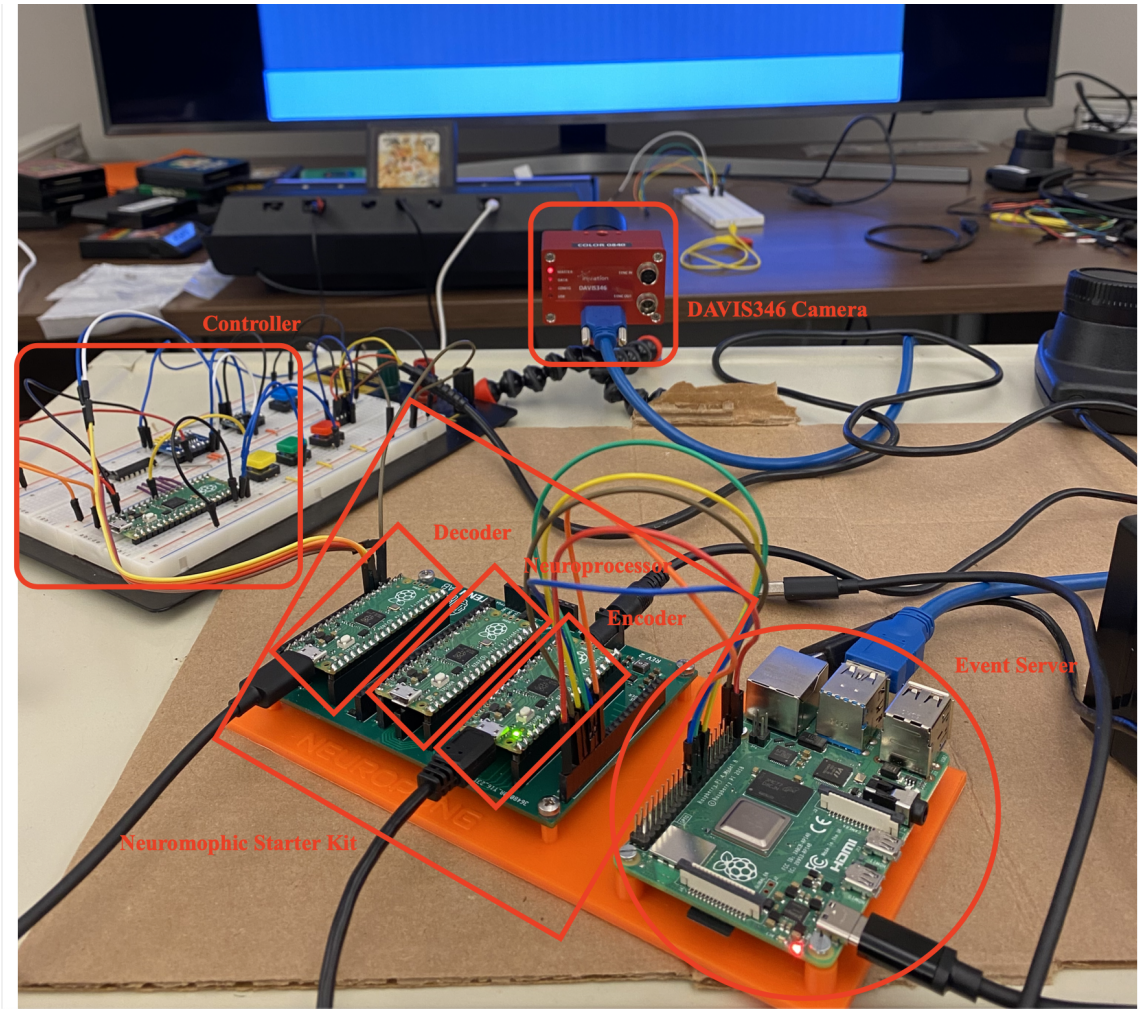


Figure 8.3: Figure showing the current hardware setup in the lab.

or architectural information. It would be advantageous, as it relates to my work, to define some structure of neurons and synapses that EONS does not evolve. In this way, I can define a mesh input grid of neurons and synapses succeeded by other networks like the spiking threshold pooling networks or the DBSCAN networks. EONS can then train and develop a spiking neural network to either some downsampled representation of the observation or even a denoised and clustered representation of the observation. The benefit is that EONS won't be bogged down by upwards of $640 * 480 = 307,200$ neurons to optimize and connect. Instead, EONS can focus on developing a smaller SNN that has recurrence built in to represent memory in the network. This could also be a possible path toward eliminating having to process events in batches and truly asynchronously process the events event-by-event.

8.5 Memristive Implementation

All of the presented networks in Chapter 3 are elegant and parameterically require little resolution. All networks' synaptic weight values are binary (either -1 or 1) and delay values are typically only 1 cycle. This is ideal for RAVENS where higher delay values on hardware require more resources. Because these networks are so simple and straightforward, they are amenable to a bi-level memristive implementation of RAVENS. Using memristors as synapses in neuromorphic devices facilitates more efficient computing due to their ability to offer high connectivity, high density, and low power consumption [58]. Specifics about memristors are outside the scope of this work, so readers are referred to [68] for more information about memristors, their applicability in neuromorphic computing, and the challenges and compatibility of different material stacks with CMOS processes.

Chapter 9

Conclusion

In this body of work, I have introduced a collection of hand-tooled spiking neural networks that are designed to work directly with event-based cameras. The networks are customizable, scalable, and precisely defined. They are capable of processing regions of the camera or events on an event-by-event basis. Additionally, I have created and discussed the `ebc_processing_tool` which allows users to leverage the network techniques on EBC files. The tool allows users to experiment with event-based camera data by playing with different segment or batch lengths, preprocessing techniques, etc. Furthermore, the tool will generate the RAVENS networks that correspond to the user's choices of preprocessing techniques without the need for any training.

I have demonstrated a classification experiment using the DVSGesture dataset, the regional event processing networks, and conventional random forest models that can be converted to spiking neural networks in the TENNLab software framework. Overall, the results were positive and showed that both preprocessing networks, when parameterized correctly, yielded F1-Scores over 80% for a tough, real-time minded experiment.

Similarly, I have demonstrated a control theory experiment that leverages the Arcade Learning Environment to develop agents that play Atari games. I have

produced the open source ALE_EBC library that converts Atari frames to events so that online learning of machine learning agents on Atari games with simulated event-based camera data is now possible. Furthermore, I leveraged the spiking threshold pooling networks to reduce the observation space down to a more manageable size for EONS to develop an SNN agent for. The result of doing so was a resounding success and total SNN agent dominance over the CPU player. I generated the large tiled network that was able to play Pong as the artifact from that project.

Using the per-event processing networks, I proposed a real-time driving experiment and system that used the networks to denoise and cluster objects of interest in the DSEC driving dataset. The results showed that the networks, when properly parameterized, do a reasonable job of denoising and localizing objects of interest on a tough dataset. This proposed system further justified pairing an event-based camera with neuromorphic techniques for real-world, high fidelity applications.

Lastly, I have discussed and proposed avenues of ongoing and future work that seek to further establish the benefit of small, sparsely connected EONS trained spiking neural networks that operate on either real or simulated event-based camera data. I have outlined the ongoing work in producing an embedded system to play on an Atari system in real-time using a real EBC as the input sensor. Overall, I have demonstrated several applications for successfully pairing event-based cameras with spiking neural networks leveraging the TENNLab software framework and the RAVENS neurorprocessor. This work reinforces the pairing of event-based cameras with SNNs when the preprocessing of the event output can be performed in network.

Bibliography

[1] Akolkar, H., Meyer, C., Clady, X., Marre, O., Bartolozzi, C., Panzeri, S., and Benosman, R. (2015a). What can neuromorphic event-driven precise timing add to spike-based pattern recognition? *Neural computation*, 27(3):561–593. 10

[2] Akolkar, H., Panzeri, S., and Bartolozzi, C. (2015b). Spike time based unsupervised learning of receptive fields for event-driven vision. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4258–4264. IEEE. 10

[3] Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., et al. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557. 1, 6, 65, 80

[4] Ameli, S., Foshie, A., Friend, D., Plank, J., Rose, G., and Schuman, C. (2023). Algorithm and application impacts of programmable plasticity in spiking neuromorphic hardware. In *Proceedings of the 2023 International Conference on Neuromorphic Systems*, pages 1–6. 7

[5] Amir, A., Taba, B., Berg, D., Melano, T., McKinstry, J., di Nolfo, C., Nayak, T., Andreopoulos, A., Garreau, G., Mendoza, M., Kusnitz, J., Debole, M., Esser, S., Delbruck, T., Flickner, M., and Modha, D. (2017). A low power, fully event-based gesture recognition system. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 9, 27, 56, 64, 65, 67

[6] Barbier, T., Teulière, C., and Triesch, J. (2021). Spike timing-based unsupervised learning of orientation, disparity, and motion representations in a spiking neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1377–1386. 11

[7] Bardow, P., Davison, A. J., and Leutenegger, S. (2016). Simultaneous optical flow and intensity estimation from an event camera. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 884–892. 10

[8] Barua, S., Miyatani, Y., and Veeraraghavan, A. (2016). Direct face detection and video reconstruction from event cameras. In *2016 IEEE winter conference on applications of computer vision (WACV)*, pages 1–9. IEEE. 10

[9] Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A. R., and Eliasmith, C. (2014). Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7:48. 13

[10] Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., and Maass, W. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature communications*, 11(1):3625. 2

[11] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279. 81, 91

[12] Berlincioni, L., Cultrera, L., Albisani, C., Cresti, L., Leonardo, A., Picchioni, S., Becattini, F., and Del Bimbo, A. (2023). Neuromorphic event-based facial expression recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4108–4118. 9

[13] Bi, Y., Chadha, A., Abbas, A., Bourtsoulatze, E., and Andreopoulos, Y. (2019). Graph-based object classification for neuromorphic vision sensing. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 491–501. 9

[14] Binas, J., Neil, D., Liu, S.-C., and Delbruck, T. (2017). Ddd17: End-to-end davis driving dataset. *arXiv preprint arXiv:1711.01458*. 9

[15] Bradski, G. (2000). The opencv library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25(11):120–123. 61

[16] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. 82

[17] Brosch, T., Tschechne, S., and Neumann, H. (2015). On event-based optical flow detection. *Frontiers in neuroscience*, 9:133662. 10

[18] Buckley, S., McCaughan, A. N., Chiles, J., Mirin, R. P., Nam, S. W., Shainline, J. M., Bruer, G., Plank, J. S., and Schuman, C. D. (2018). Design of superconducting optoelectronic networks for neuromorphic computing. In *IEEE International Conference on Rebooting Computing*, pages 36–42, Tysons, VA. 13

[19] Cady, N. C., Beckmann, K., Olin-Ammentorp, W., Chakma, G., Amer, S., Weiss, R., Sayyaparaju, S., Adnan, M., Murray, J., Dean, M., Plank, J., Rose, G., and Van Nostrand, J. (2018). Full CMOS-memristor implementation of a dynamic neuromorphic architecture. In *2018 GOMACTech Conference*, Miami. 13

[20] Calabrese, E., Taverni, G., Awai Easthope, C., Skriabine, S., Corradi, F., Longinotti, L., Eng, K., and Delbruck, T. (2019). Dhp19: Dynamic vision sensor

3d human pose dataset. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 0–0. 9

[21] Camunas-Mesa, L., Acosta-Jimenez, A., Serrano-Gotarredona, T., and Linares-Barranco, B. (2005). A digital pixel cell for address event representation image convolution processing. In *Bioengineered and Bioinspired Systems II*, volume 5839, pages 160–171. SPIE. 65

[22] Censi, A. and Scaramuzza, D. (2014). Low-latency event-based visual odometry. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 703–710. IEEE. 10

[23] Cheng, R. and Nikolic, K. (2023). System integration of neuromorphic visual (dvs), processing (spinnaker) and servomotor modules into an autonomous robot controlled by a spiking neural network. *Authorea Preprints*. 11

[24] Cheng, W., Luo, H., Yang, W., Yu, L., Chen, S., and Li, W. (2019). Det: A high-resolution dvs dataset for lane extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0. 9

[25] Cohen, G. (2022). Gooaall!!!: Why we built a neuromorphic robot to play foosball. *IEEE Spectrum*, 59(3):44–50. 16

[26] Coletti, M. A., Scott, E. O., and Bassett, J. K. (2020). Library for evolutionary algorithms in python (leap). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1571–1579. 2

[27] Corradi, F., Pande, S., Stuijt, J., Qiao, N., Schaafsma, S., Indiveri, G., and Catthoor, F. (2019). Ecg-based heartbeat classification in neuromorphic hardware. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE. 7

[28] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99. 1, 6

[29] DeWolf, T., Jaworski, P., and Eliasmith, C. (2020). Nengo and low-power ai hardware for robust, embedded neurorobotics. *Frontiers in Neurorobotics*, 14:568359. 7

[30] DeWolf, T., Patel, K., Jaworski, P., Leontie, R., Hays, J., and Eliasmith, C. (2023). Neuromorphic control of a simulated 7-dof arm using loihi. *Neuromorphic Computing and Engineering*, 3(1):014007. 7

[31] Dey, S., Mukherjee, A., Bzard, G., and McLelland, D. (2019). Human gesture recognition using spiking input on akida neuromorphic platform. *Neural Information Processing Systems (NeurIPS)*. 7

[32] Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C., and Pfeiffer, M. (2015). Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International joint conference on neural networks (IJCNN)*, pages 1–8. ieee. 10

[33] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16. 8

[34] Ehrlich, M. and Tsur, E. E. (2021). Neuromorphic adaptive body leveling in a bioinspired hexapod walking robot. In *2021 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 01–04. IEEE. 7

[35] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231. 41

[36] Fayyazi, A., Kundu, S., Nazarian, S., Beerel, P. A., and Pedram, M. (2019). Csrram: Area-efficient low-power ex-situ training framework for memristive neuromorphic circuits based on clustered sparsity. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 465–470. IEEE. 7

[37] Fischer, T., Pang, J., Huang, T. E., Qiu, L., Chen, H., Darrell, T., and Yu, F. (2022). Qdtrack: Quasi-dense similarity learning for appearance-only multiple object tracking. *arXiv preprint arXiv:2210.06984*. 104

[38] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47. 102

[39] Foshie, A. Z., Plank, J. S., Rose, G. S., and Schuman, C. D. (2023). Functional specification of the RAVENS neuroprocessor. arXiv:2307.15232. 13

[40] Foshie, A. Z., Rizzo, C., Das, H., Zheng, C., Plank, J. S., and Rose, G. S. (2022). Benchmark comparisons of spike-based reconfigurable neuroprocessor architectures for control applications. In *GLSVLSI - Great Lakes Symposium on VLSI*, pages 383–386. ACM. 13

[41] Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665. 1, 6

[42] Gallego, G., Delbrück, T., Orchard, G., Bartolozzi, C., Taba, B., Censi, A., Leutenegger, S., Davison, A. J., Conradt, J., Daniilidis, K., et al. (2020). Event-based vision: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(1):154–180. 7, 10

[43] Gallego, G., Lund, J. E., Mueggler, E., Rebecq, H., Delbruck, T., and Scaramuzza, D. (2017). Event-based, 6-dof camera tracking from photometric depth maps. *IEEE transactions on pattern analysis and machine intelligence*, 40(10):2402–2412. 10

[44] Gao, Y., Lu, J., Li, S., Ma, N., Du, S., Li, Y., and Dai, Q. (2023). Action recognition and benchmark using event cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 9

[45] Gehrig, D. and Scaramuzza, D. (2022). Pushing the limits of asynchronous graph-based object detection with event cameras. *arXiv*. 104

[46] Gehrig, M., Aarents, W., Gehrig, D., and Scaramuzza, D. (2021). Dsec: A stereo event camera dataset for driving scenarios. *IEEE Robotics and Automation Letters*. 9, 41, 102

[47] George, A. M., Banerjee, D., Dey, S., Mukherjee, A., and Balamurali, P. (2020). A reservoir-based convolutional spiking neural network for gesture recognition from dvs input. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. 27, 65, 66, 67, 79, 80

[48] Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.". 7

[49] Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4):1430. 13

[50] Goodman, D. F. and Brette, R. (2009). The brian simulator. *Frontiers in neuroscience*, 3:643. 13

[51] Guo, S., Kang, Z., Wang, L., Zhang, L., Chen, X., Li, S., and Xu, W. (2020). A noise filter for dynamic vision sensors based on global space and time information. *arXiv preprint arXiv:2004.04079*. 10

[52] Hagenaars, J., Paredes-Vallés, F., and De Croon, G. (2021). Self-supervised learning of event-based optical flow with spiking neural networks. *Advances in Neural Information Processing Systems*, 34:7167–7179. 11

[53] Hajizada, E., Berggold, P., Iacono, M., Glover, A., and Sandamirskaya, Y. (2022). Interactive continual learning for robots: A neuromorphic approach. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, ICONS '22, New York, NY, USA. Association for Computing Machinery. 7

[54] Hazan, H., Saunders, D. J., Khan, H., Patel, D., Sanghavi, D. T., Siegelmann, H. T., and Kozma, R. (2018). Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in neuroinformatics*, 12:89. 127

[55] Hu, Y., Liu, S.-C., and Delbruck, T. (2021). v2e: From video frames to realistic dvs events. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1312–1321. 8, 92

[56] Inc., P. T. (2015). Collaborative data science. 61, 88, 95

[57] Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80. 7

[58] Jo, S. H., Chang, T., Ebong, I., Bhadviya, B. B., Mazumder, P., and Lu, W. (2010). Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, 10(4):1297–1301. 132

[59] Joubert, D., Marcireau, A., Ralph, N., Jolley, A., van Schaik, A., and Cohen, G. (2021a). Event camera simulator improvements via characterized parameters. *Frontiers in Neuroscience*, 15. 8

[60] Joubert, D., Marcireau, A., Ralph, N., Jolley, A., van Schaik, A., and Cohen, G. (2021b). Event camera simulator improvements via characterized parameters. *Frontiers in Neuroscience*, page 910. 85

[61] Kaiser, J., Tieck, J. C. V., Hubschneider, C., Wolf, P., Weber, M., Hoff, M., Friedrich, A., Wojtasik, K., Roennau, A., Kohlhaas, R., et al. (2016). Towards a framework for end-to-end control of a simulated vehicle with spiking neural

networks. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 127–134. IEEE. 8, 84, 85, 86, 89

[62] Khodamoradi, A. and Kastner, R. (2018). $o(n)$ o (n)-space spatiotemporal filter for reducing noise in neuromorphic vision sensors. *IEEE Transactions on Emerging Topics in Computing*, 9(1):15–23. 10

[63] Kim, H., Handa, A., Benosman, R., Ieng, S.-H., and Davison, A. J. (2014). Simultaneous mosaicing and tracking with an event camera. In *BMVC*. 10

[64] Kim, H., Leutenegger, S., and Davison, A. J. (2016). Real-time 3d reconstruction and 6-dof tracking with an event camera. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI 14*, pages 349–364. Springer. 10

[65] Kim, J., Bae, J., Park, G., Zhang, D., and Kim, Y. M. (2021). N-imagenet: Towards robust, fine-grained object recognition with event cameras. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2146–2156. 8

[66] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444. 1

[67] Li, H., Liu, H., Ji, X., Li, G., and Shi, L. (2017). Cifar10-dvs: an event-stream dataset for object classification. *Frontiers in neuroscience*, 11:309. 8

[68] Li, Y., Wang, Z., Midya, R., Xia, Q., and Yang, J. J. (2018). Review of memristor devices in neuromorphic computing: materials sciences and device challenges. *Journal of Physics D: Applied Physics*, 51(50):503002. 132

[69] Lichtsteiner, P., Posch, C., and Delbruck, T. (2006). A 128 x 128 120db 30mw asynchronous vision sensor that responds to relative intensity change. In *2006*

IEEE International Solid State Circuits Conference-Digest of Technical Papers, pages 2060–2069. IEEE. 64

[70] Lichtsteiner, P., Posch, C., and Delbruck, T. (2008). A 128 × 128 120 db 15 μs latency asynchronous temporal contrast vision sensor. *IEEE journal of solid-state circuits*, 43(2):566–576. 64

[71] Liu, Q., Ruan, H., Xing, D., Tang, H., and Pan, G. (2020). Effective aer object classification using segmented probability-maximization learning in spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1308–1315. 10

[72] Liu, Q., Xing, D., Tang, H., Ma, D., and Pan, G. (2021). Event-based action recognition using motion information and spiking neural networks. In *IJCAI*, pages 1743–1749. 9, 10

[73] Lohmann, N. (2022). JSON for Modern C++. 82

[74] Lu, J., Dong, J., Yan, R., and Tang, H. (2020). An event-based categorization model using spatio-temporal features in a spiking neural network. In *2020 12th International Conference on Advanced Computational Intelligence (ICACI)*, pages 385–390. IEEE. 10

[75] Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560. 7

[76] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. (2018a). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562. xiii, 126

[77] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. (2018b). Revisiting the arcade learning environment: Evaluation

protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562. 81, 91, 96, 125

[78] Maqueda, A. I., Loquercio, A., Gallego, G., García, N., and Scaramuzza, D. (2018). Event-based vision meets deep learning on steering prediction for self-driving cars. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5419–5427. 10

[79] Markram, H., Gerstner, W., and Sjöström, P. J. (2012). Spike-timing-dependent plasticity: a comprehensive overview. *Frontiers in synaptic neuroscience*, 4:2. 7

[80] Mitchell, J. P. (2018). DANNA2: Dynamic adaptive neural network arrays. Masters Thesis, University of Tennessee. 13

[81] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533. xiii, 126

[82] Mollica, G., Felicioni, S., Legittimo, M., Meli, L., Costante, G., and Valigi, P. (2023). Ma-vied: A multisensor automotive visual inertial event dataset. *IEEE Transactions on Intelligent Transportation Systems*. 9

[83] Moradi, S., Qiao, N., Stefanini, F., and Indiveri, G. (2017). A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE transactions on biomedical circuits and systems*, 12(1):106–122. 6

[84] Mueggler, E., Bartolozzi, C., and Scaramuzza, D. (2017a). Fast event-based corner detection. 10

[85] Mueggler, E., Rebecq, H., Gallego, G., Delbruck, T., and Scaramuzza, D. (2017b). The event-camera dataset and simulator: Event-based data for pose

estimation, visual odometry, and slam. *The International Journal of Robotics Research*, 36(2):142–149. 9

[86] Munda, G., Reinbacher, C., and Pock, T. (2018). Real-time intensity-image reconstruction for event cameras using manifold regularisation. *International Journal of Computer Vision*, 126(12):1381–1393. 10

[87] Nagaraj, M., Liyanagedera, C. M., and Roy, K. (2023). Dotie-detecting objects through temporal isolation of events using a spiking architecture. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4858–4864. IEEE. 33

[88] Neckar, A., Fok, S., Benjamin, B. V., Stewart, T. C., Oza, N. N., Voelker, A. R., Eliasmith, C., Manohar, R., and Boahen, K. (2018). Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proceedings of the IEEE*, 107(1):144–164. 7

[89] Nguyen, A., Do, T.-T., Caldwell, D. G., and Tsagarakis, N. G. (2019). Real-time 6dof pose relocalization for event cameras with stacked spatial lstm networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0. 10

[90] O'Connor, P., Neil, D., Liu, S.-C., Delbruck, T., and Pfeiffer, M. (2013). Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in neuroscience*, 7:58710. 10

[91] Orchard, G., Benosman, R., Etienne-Cummings, R., and Thakor, N. V. (2013). A spiking neural network architecture for visual motion estimation. In *2013 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 298–301. IEEE. 11

[92] Orchard, G., Jayawant, A., Cohen, G. K., and Thakor, N. (2015). Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in neuroscience*, 9:159859. 8

[93] Palinauskas, G., Amaya, C., Eames, E., Neumeier, M., and Von Arnim, A. (2023). Generating event-based datasets for robotic applications using mujoco-esim. In *Proceedings of the 2023 International Conference on Neuromorphic Systems*, pages 1–7. 8

[94] Panda, P. and Srinivasa, N. (2018). Learning to recognize actions from limited training examples using a recurrent spiking neural model. *Frontiers in neuroscience*, 12:126. 7

[95] Pang, J., Qiu, L., Li, X., Chen, H., Li, Q., Darrell, T., and Yu, F. (2021). Quasi-dense similarity learning for multiple object tracking. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 104

[96] Paredes-Vallés, F., Scheper, K. Y., and De Croon, G. C. (2019). Unsupervised learning of a hierarchical spiking neural network for optical flow estimation: From events to global motion perception. *IEEE transactions on pattern analysis and machine intelligence*, 42(8):2051–2064. 11

[97] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32. 127

[98] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830. 41, 71

[99] Pérez-Carrasco, J. A., Zhao, B., Serrano, C., Acha, B., Serrano-Gotarredona, T., Chen, S., and Linares-Barranco, B. (2013). Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets. *IEEE transactions on pattern analysis and*

machine intelligence, 35(11):2706–2719. 10

[100] Perot, E., De Tournemire, P., Nitti, D., Masci, J., and Sironi, A. (2020). Learning to detect objects with a 1 megapixel event camera. *Advances in Neural Information Processing Systems*, 33:16639–16652. 9

[101] Plank, J. S., Gullett, B., Foshie, A. Z., Rose, G. S., and Schuman, C. D. (2022). Disclosure of a neuromorphic starter kit. arXiv:2211.04526. 128

[102] Plank, J. S., Rizzo, C., Shahat, K., Bruer, G., Dixon, T., Goin, M., Zhao, G., Anantharaj, J., Schuman, C. D., Dean, M. E., Rose, G. S., Cady, N. C., and Van Nostrand, J. (2019). The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems. In *44th Annual GOMACTech Conference*, Albuquerque. 11, 92

[103] Plank, J. S., Schuman, C. D., Bruer, G., Dean, M. E., and Rose, G. S. (2018). The TENNLab exploratory neuromorphic computing framework. *IEEE Letters of the Computer Society*, 1(2):17–20. 11, 91

[104] Plank, J. S., Zhao, J., and Hurst, B. (2020). Reducing the size of spiking convolutional neural networks by trading time for space. In *IEEE International Conference on Rebooting Computing (ICRC)*, pages 115–126. IEEE. 2

[105] Plank, J. S., Zheng, C., Schuman, C. D., and Dean, C. (2021). Spiking neuromorphic networks for binary tasks. In *International Conference on Neuromorphic Computing Systems (ICONS)*, pages 1–8. ACM. 2, 17

[106] Rebecq, H., Gehrig, D., and Scaramuzza, D. (2018). Esim: an open event camera simulator. In *Conference on Robot Learning*, pages 969–982. PMLR. 8, 86, 89

[107] Rebecq, H., Ranftl, R., Koltun, V., and Scaramuzza, D. (2019). High speed and high dynamic range video with an event camera. *IEEE transactions on pattern analysis and machine intelligence*, 43(6):1964–1980. 10

[108] Reverter Valeiras, D., Orchard, G., Ieng, S.-H., and Benosman, R. B. (2016). Neuromorphic event-based 3d pose estimation. *Frontiers in neuroscience*, 9:522. 9

[109] Reynolds, J. J. M., Plank, J. S., and Schuman, C. D. (2019). Intelligent reservoir generation for liquid state machines using evolutionary optimization. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. 11

[110] Rizzo, C., McCombs, L., Haynie, B., Schuman, C. D., and Plank, J. S. (2023a). DVSGesture recognition with neuromorphic observation space reduction techniques. In *International Conference on Neuromorphic Computing Systems (ICONS)*. ACM. xii, xv, xvi, 16, 28, 29, 64, 68, 70, 72, 74, 75, 76, 77

[111] Rizzo, C. P., Schuman, C. D., and Plank, J. S. (2022). Event-based camera simulation wrapper for Arcade Learning Environment. In *International Conference on Neuromorphic Computing Systems (ICONS)*, pages 1–5. ACM. xii, xvi, 81, 82, 87, 88, 90, 91, 95

[112] Rizzo, C. P., Schuman, C. D., and Plank, J. S. (2023b). Neuromorphic downsampling of event-based camera output. In *Proceedings of the 2023 Annual Neuro-Inspired Computational Elements Conference*. ACM. xiv, xv, xvi, 16, 19, 22, 23, 26, 91, 93, 95, 99

[113] Rizzo, C. P., Schuman, C. D., and Plank, J. S. (2024). Speed-based filtration and dbscan of event-based camera data with neuromorphic computing. *arXiv preprint arXiv:2401.15212*. x, xi, xii, xv, 32, 35, 36, 38, 40, 43, 46, 47, 48, 49

[114] Schuman, C., Plank, J., Patton, R., Potok, T., and Rose, G. (2022a). A framework to enable top-down co-design of neuromorphic systems for real-world applications. In *Proceedings of the 2022 Annual Neuro-Inspired Computational Elements Conference*, NICE '22, page 84–85, New York, NY, USA. Association for Computing Machinery. xiv, 12

[115] Schuman, C., Rizzo, C., McDonald-Carmack, J., Skuda, N., and Plank, J. (2022b). Evaluating encoding and decoding approaches for spiking neuromorphic systems. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, pages 1–9. 7

[116] Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Kay, B., et al. (2022c). Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science*, 2(1):10–19. 11

[117] Schuman, C. D., Mitchell, J. P., Patton, R. M., Potok, T. E., and Plank, J. S. (2020a). Evolutionary optimization for neuromorphic systems. In *NICE: Neuro-Inspired Computational Elements Workshop*. 2, 11, 91

[118] Schuman, C. D., Mitchell, J. P., Patton, R. M., Potok, T. E., and Plank, J. S. (2020b). Evolutionary optimization for neuromorphic systems. In *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*, pages 1–9. 7

[119] Schuman, C. D., Plank, J. S., Bruer, G., and Anantharaj, J. (2019). Non-traditional input encoding schemes for spiking neuromorphic systems. In *IJCNN: The International Joint Conference on Neural Networks*, pages 1–10, Budapest. 92

[120] Schuman, C. D., Plank, J. S., Parsa, M., Kulkarni, S. R., Skuda, N., and Mitchell, J. P. (2021). A software framework for comparing training approaches for spiking neuromorphic systems. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. 11, 73

[121] Severa, W., Parekh, O., Carlson, K. D., James, C. D., and Aimone, J. B. (2016). Spiking network algorithms for scientific computing. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. 2, 79

[122] Severa, W., Vineyard, C. M., Dellana, R., Verzi, S. J., and Aimone, J. B. (2019). Training deep neural networks for binary communication with the whetstone method. *Nature Machine Intelligence*, 1(2):86–94. 2, 11

[123] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2017). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*. 8, 84, 86

[124] Shrestha, S. B. and Orchard, G. (2018). Slayer: Spike layer error reassignment in time. *Advances in neural information processing systems*, 31. 2, 7

[125] Sironi, A., Brambilla, M., Bourdis, N., Lagorce, X., and Benosman, R. (2018). Hats: Histograms of averaged time surfaces for robust event-based object classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1731–1740. 9

[126] Son, B., Suh, Y., Kim, S., Jung, H., Kim, J.-S., Shin, C., Park, K., Lee, K., Park, J., Woo, J., et al. (2017). 4.1 a 640× 480 dynamic vision sensor with a 9μm pixel and 300meps address-event representation. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 66–67. IEEE. 3

[127] Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature Neuroscience*, 3(9):919–926. 11, 13

[128] Soures, N. and Kudithipudi, D. (2019). Deep liquid state machines with neural plasticity for video activity recognition. *Frontiers in neuroscience*, 13:686. 7

[129] Stagsted, R., Vitale, A., Binz, J., Bonde Larsen, L., Sandamirskaya, Y., et al. (2020). Towards neuromorphic control: A spiking neural network based pid controller for uav. RSS. 11

[130] Stroobants, S., Dupeyroux, J., and De Croon, G. (2022). Design and implementation of a parsimonious neuromorphic pid for onboard altitude control for mavs using neuromorphic processors. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, ICONS '22, New York, NY, USA. Association for Computing Machinery. 7

[131] Sun, Y., Zeng, Y., and Li, Y. (2022). Solving the spike feature information vanishing problem in spiking deep q network with potential based normalization. *Frontiers in Neuroscience*, 16:953368. 127

[132] Tan, W., Patel, D., and Kozma, R. (2021). Strategy and benchmark for converting deep q-networks to event-driven spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 9816–9824. 127

[133] Taverni, G., Moeys, D. P., Li, C., Cavaco, C., Motsnyi, V., Bello, D. S. S., and Delbruck, T. (2018). Front and back illuminated dynamic and active pixel vision sensors comparison. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):677–681. 3

[134] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE. 7

[135] Vasco, V., Glover, A., and Bartolozzi, C. (2016). Fast event-based harris corner detection exploiting the advantages of event-driven cameras. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 4144–4149. IEEE. 10

[136] Vasudevan, A., Negri, P., Linares-Barranco, B., and Serrano-Gotarredona, T. (2020). Introduction and analysis of an event-based sign language dataset. In *2020 15th IEEE International Conference on Automatic Face and Gesture Recognition (FG 2020)*, pages 675–682. IEEE. 9

[137] Viale, A., Marchisio, A., Martina, M., Masera, G., and Shafique, M. (2022). Lanesnns: Spiking neural networks for lane detection on the loihi neuromorphic processor. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 79–86. IEEE. 9

[138] Wang, L., Ho, Y.-S., Yoon, K.-J., et al. (2019). Event-based high dynamic range image and very high frame rate video generation using conditional generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10081–10090. 10, 20

[139] Watkins, Y., Thresher, A., Mascarenas, D., and Kenyon, G. T. (2018). Sparse coding enables the reconstruction of high-fidelity images and video from retinal spike trains. In *Proceedings of the International Conference on Neuromorphic Systems*, pages 1–5. 10

[140] Weng, J., Chen, H., Yan, D., You, K., Duburcq, A., Zhang, M., Su, Y., Su, H., and Zhu, J. (2022). Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6. 127

[141] Xing, Y., Di Caterina, G., and Soraghan, J. (2020). A new spiking convolutional recurrent neural network (scrnn) with applications to event-based hand gesture recognition. *Frontiers in neuroscience*, 14:590164. 27, 65, 66, 67, 79, 80

[142] Yang, L., Zeng, Z., and Huang, Y. (2019). An associative-memory-based reconfigurable memristive neuromorphic system with synchronous weight training. *IEEE Transactions on Cognitive and Developmental Systems*, 12(3):529–540. 7

[143] Zeng, Y., Zhao, D., Zhao, F., Shen, G., Dong, Y., Lu, E., Zhang, Q., Sun, Y., Liang, Q., Zhao, Y., et al. (2023). Braincog: A spiking neural network based, brain-inspired cognitive intelligence engine for brain-inspired ai and brain simulation. *Patterns*, 4(8). 127

[144] Zhang, Z.-Z., Zhang, J., Niu, C.-X. M., Hao, M.-Z., and Lan, N. (2021). An integrated virtual hand platform for evaluation of model-based control of hand prosthesis. In *2021 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 287–291. IEEE. 7

[145] Zhou, Y., Gallego, G., Rebecq, H., Kneip, L., Li, H., and Scaramuzza, D. (2018). Semi-dense 3d reconstruction with a stereo event camera. In *Proceedings of the European conference on computer vision (ECCV)*, pages 235–251. 10

[146] Zhu, A. Z., Thakur, D., Özaslan, T., Pfrommer, B., Kumar, V., and Daniilidis, K. (2018a). The multivehicle stereo event camera dataset: An event camera dataset for 3d perception. *IEEE Robotics and Automation Letters*, 3(3):2032–2039. 9

[147] Zhu, A. Z., Yuan, L., Chaney, K., and Daniilidis, K. (2018b). Ev-flownet: Self-supervised optical flow estimation for event-based cameras. *arXiv preprint arXiv:1802.06898*. 10

Vita

Hailing from New Jersey, Charles Rizzo was born to an immigrant household and was raised by his mother, aunt, and grandmother, all of whom were born in Italy. With his passionate Italian caretakers, he moved to Chattanooga, Tennessee, and after completing high school at Silverdale Baptist Academy, attended the University of Tennessee for his Bachelor's degree in computer science with a minor in cybersecurity. Toward the end of his undergraduate years, and with no experience in machine learning whatsoever, he joined the TENNLab neuromorphic computing group. Even though he joined the group on a whim, he enjoyed his time with it and so decided he would stay with the group at UT to pursue a Doctor of Philosophy degree with a concentration in neuromorphic computing. His research interests include creating embedded systems leveraging event-based cameras as input sensors and spiking neural networks as processing components. Perhaps it's Stockholm Syndrome, but after graduation, he will remain with the TENNLab group at UT in pursuance of a research professorship with the long term goal of creating a start up out of the research. He is incredibly grateful to all of his friends and family for their support throughout his decade of schooling and as he begins his new career as a researcher.