

# TENNs-PLEIADES: Building Temporal Kernels with Orthogonal Polynomials

Yan Ru Pei, Olivier Coenen,

**Abstract**— We introduce a neural network named PLEIADES (PoLynomial Expansion In Adaptive Distributed Event-based Systems), belonging to the TENNs (Temporal Neural Networks) architecture. We focus on interfacing these networks with event-based data to perform online spatiotemporal classification and detection with low latency. By virtue of using structured temporal kernels and event-based data, we have the freedom to vary the sample rate of the data along with the discretization step-size of the network without additional finetuning. We experimented with three event-based benchmarks and obtained state-of-the-art results on all three by large margins with significantly smaller memory and compute costs. We achieved: 1) 99.59% accuracy with 192K parameters on the DVS128 hand gesture recognition dataset and 100% with a small additional output filter; 2) 99.58% test accuracy with 277K parameters on the AIS 2024 eye tracking challenge; and 3) 0.556 mAP with 576k parameters on the PROPHESEE 1 Megapixel Automotive Detection Dataset.

**Index Terms**—orthogonal polynomials, spatiotemporal, event-based, convolutional neural networks, object detection



## 1 INTRODUCTION

TEMPORAL convolutional networks (TCNs) [18] have been a staple for processing time series data from speech enhancement [22] to action segmentation [17]. However, in most cases, the temporal kernel is very short (usually size of 3), making it difficult for the network to capture long-range temporal correlations. The temporal kernels are intentionally kept short, because keeping a long temporal kernel with a large number of trainable kernel values usually leads to unstable training. In addition, we require a large amount of memory for storing the weights during inference. One popular solution for this has been to parameterize the temporal kernel function with a simple multilayer perceptron (MLP), which promotes stability [28] and more compressed parameters, but it often increases the computational load considerably.

Here, we introduce a method of parameterization of temporal kernels, named PLEIADES (PoLynomial Expansion In Adaptive Distributed Event-based Systems), that can in many cases reduce the memory and computational costs compared to explicit convolutions. The design is fairly modular, and can be used as a drop-in replacement for any 1D-like convolutional layers, allowing them to perform long temporal convolutions effectively. In fact, we augment a previously proposed (1+2)D causal spatiotemporal network [23] by replacing its temporal kernels with this new polynomial parameterization. This new network architecture serves as the backbone for a wide range of online spatiotemporal tasks ranging from action recognition to object detection. This network belongs to a broader class of networks named Temporal Neural Networks (TENNs) developed by Brainchip Inc.

Even though our network can be used for any spatiotemporal data (e.g. videos captured with conventional cameras), in this work, we investigate mainly the performance of our network on event-based data (e.g. data captured by an event camera). Event cameras are sensors that generate outputs events  $\{-1, +1\}$  responding to optical changes in the scene’s luminance [4], and can generate sparse data on an incredibly short time scale, usually at 1 microsecond. Event cameras can produce very rich temporal features capturing subtle motion patterns, and allow for flexible adjustment in the effective FPS into our network by varying the binning window size. This makes it suitable for us to test networks with long temporal kernels sampled at different step sizes.

In Section 2, we discuss earlier and concurrent works that are related to our polynomial parameterization of temporal kernels, including the modern variants of deep state-space models. In Section 3, we discuss how the generation and discretization of temporal kernels are performed, and additionally using the `einsum` notation to perform the optimal order of operations to reduce memory and computational loads. In Section 4, we briefly describe the network backbone architecture, including some architectural novelties besides our polynomial temporal kernels. In Section 5, we run three event-based benchmarks: 1) the IBM DVS128 hand gesture recognition dataset, 2) the CVPR 2024 AIS event-based eye tracking challenge, 3) and the PROPHESEE 1 megapixel automotive detection dataset (Prophesee GEN4 Dataset). We achieved SOTA results on all three benchmarks with significantly fewer parameters.

The code for building the structured temporal kernels, along with a pre-trained PLEIADES network for evaluation on the DVS128 dataset is available here: <https://github.com/PeaBrane/Pleiades>.

- 
- Yan Ru Pei is a Machine Learning Research Engineer at Brainchip Inc. and Olivier Coenen is a Senior Research Scientist at Brainchip Inc. E-mail: [yanrpei@gmail.com](mailto:yanrpei@gmail.com), [ocoenen@brainchip.com](mailto:ocoenen@brainchip.com)

## 2 RELATED WORK

### 2.1 Long Temporal Convolutions and Parameterization of Kernels

When training a neural network containing convolutions with long (temporal) kernels, it is usually not desirable to explicitly parameterize the kernel values for each time step. First of all, the input data may not be uniformly sampled, meaning that the kernels need to be continuous in nature, making explicit parameterization impossible<sup>1</sup>. In this case, the kernel is treated as a mapping from an event timestamp to a kernel value, where its mapping is usually achieved via a simple MLP [25], [26], [28]. In cases where the input features are uniformly sampled, explicit parameterization of the values for each time step becomes possible in theory. However, certain regularization procedures need to be applied [3] in practice, otherwise the training may become unstable due to the large number of trainable weights. Storing all these weights may also be unfavorable in memory-constrained environments (for edge or mobile devices).

The seminal work proposing a memory encoding using orthogonal Legendre polynomials in a recurrent state-space model is the Legendre Memory Unit (LMU) [33], where Legendre polynomials (a special case of Jacobi polynomials) are used. The HiPPO formalism [11] then generalized this to other orthogonal functions including Chebyshev polynomials, Laguerre polynomials, and Fourier modes. Later, this sparked a cornucopia of works interfacing with deep state space models including S4 [12], H3 [2], and Mamba [10], achieving impressive results on a wide range of tasks from audio generation to language modeling. There are several common themes among these networks that PLEIADES differ from. First, these models typically only interface with 1D temporal data, and usually try to flatten high dimensional data into 1D data before processing [12], [37], with some exceptions [21]. Second, instead of explicitly performing finite-window temporal convolutions, a running approximation of the effects of such convolutions are performed, essentially yielding a system with infinite impulse responses where the effective polynomial structures are distorted [11], [31]. And in the more recent works, the polynomial structures are tenuously used only for initialization, but then made fully trainable. Finally, these networks mostly use an underlying depthwise structure [14] for long convolutions, which may limit the network capacity, albeit reducing the compute requirement of the network.

### 2.2 Spatiotemporal Networks

There are several classes of neural networks that can process spatiotemporal data (i.e. videos and event frames). For example, a class of networks combines components from spatial convolutional networks and recurrent networks, with the most prominent network being ConvLSTM [29]. These types of models interface well with streaming spatiotemporal data, but are oftentimes difficult to train (as with recurrent networks in general). On the other hand, we have a class of easily trainable networks that perform (separable) spatiotemporal convolutions such as the R(2+1)D and P3D networks

1. One would require an uncountable-infinite number of “weights” to explicitly parameterize a continuous function.

[27], [32], but they were originally difficult to configure for online inference as they do not assume causality. However, it is easy to configure the temporal convolutional layers as causal during training, such that the network can perform efficient online inference with streaming data via the use of circular buffering [23] or incorporating spike-based [30] or event-based [15] processing.

### 2.3 Event-based Data and Networks

An event can be succinctly represented as a tuple  $E = (p, x, y, t)$ , where  $p$  denotes the polarity,  $x$  and  $y$  are the horizontal and vertical pixel coordinates, and  $t$  is the time. A collection of events can then be expressed as  $\mathcal{E} = \{E_1, E_2, \dots\}$ . To feed event-based data into conventional neural networks, it is often necessary to bin them into uniform grids, or tensors generally shaped  $(2, H, W, T)$ . Many different event-binning methods have been explored in the past. The simplest approach is to simply count the number in each bin [19]. Other methods include replacing each event with a fixed or trainable kernel [5], [36] before evaluating the contribution of that kernel to a given bin. Here, we only use the direct binning and event-volume binning methods, yielding the  $4d$  tensor  $(2, H, W, T)$  to our network [23], noting that we retain the polarity channel unlike previous works [36].

The most popular class of event-based networks is spiking neural networks, which propagate event or spike signals with continuous timestamps throughout the network, usually under the assumption of some fixed internal dynamics for the neurons [6], [7]. These networks can be efficient during inference, as typically they only need to propagate 1-bit signals, but they are also incredibly difficult to train without specialized techniques to efficiently simulate the neural dynamics and ameliorate the spiking behaviors. The SLAYER model [30] computes the neuron response using the spike-response model (SRM), which can then be used to convolve temporally with the input signal. This impulse-response kernel is typically limited to a non-adaptive shape, such as an exponential decay (leaky integrate-and-fire neuron) or an alpha function. SLAYER then uses customized CUDA kernels to implement the delayed responses. PLEIADES generalizes the impulse-response kernel of SLAYER by making the kernel a trainable convolution kernel. It is a kernel that can take on any shape, thus providing almost any dynamics to the neuron, not limited by a priori assumptions. The kernel is fully adaptive to input data, network connectivity, and cost function used to train the network.<sup>2</sup>

Other works have proposed ameliorating the spiking behaviors with differentiable functions which can interface easier with backpropagation (surrogate gradients) [20], so the spiking network can be trained like a standard neural network. Note that a network or hardware can be fully event-based without necessarily using spike-based processing, with

2. The kernel represented in the expansion is not limited to the temporal domain, it can also be spatial (Zernike polynomials). In general, a spatiotemporal kernel may be composed of products of 2D orthogonal polynomials in space and 1D orthogonal polynomials in time; alternatively, it can directly be built from 3D orthogonal polynomials, albeit at higher computational costs. More generally, the kernel may also process/convolve/integrate over other dimensions, such as the polarization of light or the spin of electrons, to interface with optical or spintronic devices.

a prominent example being Brainchip’s Akida hardware [15]. Such hardware can generally support most modern neural networks, with sparsity-aware processing where only nonzero input features are processed at each layer. The network we propose in this work is a standard neural network (though configurable as an SNN), and can efficiently leverage event-based processing given sufficiently high sparsity<sup>3</sup> (see Appendix B.1.1).

### 3 TEMPORAL CONVOLUTIONS WITH POLYNOMIALS

In this section, we discuss: 1) how the temporal kernels are generated by the weighted sums of polynomial basis functions; 2) how the temporal kernels are discretized to be used in a neural network; 3) how the convolution with the input feature tensor can be optimized with respect to the order of operations. From here on, we will index the input channel with  $c$ , the output channel with  $d$ , the polynomial degree or basis with  $n$ , the spatial dimensions with  $x$  and  $y$ , the input timestamp with  $t$ , the output timestamp with  $t'$ , and the temporal kernel timestamp with  $\tau$ .

#### 3.1 Building temporal kernels from orthogonal polynomials

Jacobi polynomials  $P_n^{(\alpha,\beta)}(\tau)$  are a class of polynomials that are orthogonal in the following sense:

$$\int_{-1}^1 P_n^{(\alpha,\beta)}(\tau) P_m^{(\alpha,\beta)}(\tau) (1-\tau)^\alpha (1+\tau)^\beta d\tau = \delta_{nm} h_n^{(\alpha,\beta)}, \quad (1)$$

where  $\delta_{n,m}$  is the Kronecker delta that equates to 1 if  $n = m$ , and equates to 0 if  $n \neq m$ , hence establishing the orthogonality condition.  $h_n^{(\alpha,\beta)}$  is some normalization constant that is not important for our discussion. A continuous function can be parameterized by taking the weighted sum of these polynomials up to a given degree  $N$ , where the weighting factors (or coefficients)  $\{\gamma_0, \gamma_1, \dots, \gamma_N\}$  are trainable.

When this parameterization is used in an 1D convolutional layer typically involving multiple input and output channels, then naturally we require a set of coefficients for each pairing of input and output channels. More formally, if we index the input channels with  $c$  and the output channels with  $d$ , then the continuous kernel connecting  $c$  to  $d$  can be expressed as

$$k_{cd}(\tau) = \sum_{n=0}^N \gamma_{cd,n} P_n^{(\alpha,\beta)}(\tau). \quad (2)$$

We note that training with such structured temporal kernels will strictly lose us expressivity compared to explicitly parameterized temporal kernels. However, there are several key advantages of using structured kernels which will largely overcompensate for the loss of expressivity. First, using this form of implicit parameterization will allow natural resampling of the kernels during discretization, meaning that the network can interface with data sampled at different rates without additional fine-tuning (see Section 3.2). Second, having a functional basis will allow an intermediate subspace

3. This can be achieved with interfacing with sparse input data (e.g. event-camera data), sparsity promoting activation functions (e.g. ReLU), intermediate loss functions (e.g. L1 activation regularization), and network quantization.

to store feature projection, which can sometimes improve memory and computational efficiency (see Section 3.3). Finally, since a Jacobi polynomial basis is associated with an underlying Sturm-Liouville equation, this will inject good “physical” inductive biases for our network, to make the training more stable and be guided to a better optimum (see Section 5.1 for an empirical proof).

#### 3.2 Discretization of kernels

In the current implementation of our network, which interfaces with inputs that are binned in time, we need to perform discretization of the temporal kernels accordingly. One method is to take the integral of the temporal kernels over the time bins of interest.

We start by defining the antiderivative of the temporal kernels as

$$\begin{aligned} K_{cd}(\tau) &= \int_{-1}^{\tau} k_{cd}(\tau') d\tau' = \int_{-1}^{\tau} \sum_{n=0}^N \gamma_{cd,n} P_n^{(\alpha,\beta)}(\tau') d\tau' \\ &= \sum_{n=0}^N \frac{\gamma_{cd,n}}{(n+1)!} P_{n+1}^{(\alpha,\beta)}(\tau) - const, \end{aligned} \quad (3)$$

where the constant term does not depend on  $\tau$  and can be ignored. If we need to now evaluate the integral of  $k_{cd}(\tau)$  in the time bin  $[\tau_0, \tau_0 + \Delta\tau]$ , which we can denote as the discrete  $\bar{k}_{cd}[\tau_0]$ , we can simply take the difference

$$\begin{aligned} \bar{k}_{cd}[\tau_0] &= K_{cd}(\tau_0 + \Delta\tau) - K_{cd}(\tau_0) \\ &= \sum_{n=0}^N \gamma_{cd,n} \frac{P_{n+1}^{(\alpha,\beta)}(\tau_0 + \Delta\tau) - P_{n+1}^{(\alpha,\beta)}(\tau_0)}{(n+1)!} \\ &= \sum_{n=0}^N \gamma_{cd,n} \bar{P}_n^{(\alpha,\beta)}[\tau_0], \end{aligned} \quad (4)$$

where  $\bar{P}$  is the appropriately defined discrete polynomials, recovering the same form as Eq. 2. Note that Eq. 4 can be considered a generalized matrix multiplication operation where the dimension  $n$  (the polynomial basis dimension) is contracted, discussed further in Section 3.3. See Fig. 1 for a schematic representation of the operations of generating temporal kernels for multiple channels.

Under this discretization scheme, it is very easy to resample the temporal kernels (either downsampling or upsampling), to interface with data sampled at arbitrary rates, i.e. arbitrary bin-sizes for event-based data. This means that the network can be trained at a given step-size  $\Delta\tau$ , but adapted to perform inference at different rates (either faster or slower), without any additional tuning. One simply has to regenerate the discretized basis polynomials using the equations above with the new  $\Delta\tau$ , and keep everything else in the network unchanged<sup>4</sup>.

4. This is true if the scale of the input data is invariant under resampling. For event-based data accumulated into bins, this means the bin values have to be rescaled by a factor reciprocal to the size of the new bin size relative to the original one. For example, if the bin size is doubled, then the bin values need to be appropriately halved to maintain the same scale.

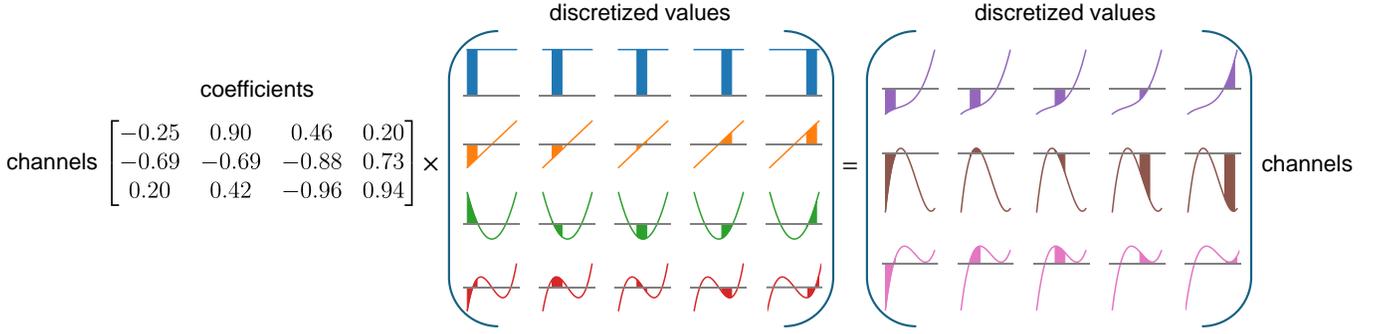


Fig. 1. An example of generating discrete temporal kernels for multiple channels, based on trainable coefficients and fixed basis orthogonal polynomials. Here, we are considering (depthwise) convolution with 3 channels, 4 basis polynomials, and a kernel size of 5. The shaded areas can be interpreted as discretized values. The coefficients can be organized as a  $3 \times 4$  matrix, and the discretized basis polynomials can be organized as a  $4 \times 5$  matrix. The matrix multiplication of the two then yields the final discretized kernels for the channels as a  $3 \times 5$  matrix.

### 3.3 Optimal order of operations

Now that the kernels are discretized, to lessen the burden of notation, we can employ the Einstein summation notation, or `einsum`, to reduce the above equation to

$$\bar{k}_{cd\tau} = \gamma_{cdn} \bar{P}_{n\tau}, \quad (5)$$

where the repeating index  $n$  is assumed to be summed over (or contracted) on the right-hand side, corresponding to summing over the polynomial basis. See Appendix A.1 for a detailed description of the contraction rules. If we now wish to convolve the temporal kernel  $\bar{k}_{ij\tau}$  with a spatiotemporal input feature tensor  $u$  which gives us the output  $y$ , the operation becomes

$$y_{dxyt'} = u_{cxyt} M_{ntt'}(\bar{P}) \gamma_{cdn}, \quad (6)$$

where  $M(\bar{P})$  is the convolution operator matrix, a sparse Toeplitz matrix generated from  $\bar{P}$  (see Appendix A.3). If a depthwise convolution is performed [14], then the equation simplifies to

$$y_{cxyt} = u_{cxyt} M_{ntt'}(\bar{P}) \gamma_{cn}, \quad (7)$$

as we only have parallel connections between input and output channels (both denoted by  $c$ ). Note that the temporal kernels do not interact with the spatial indices  $x$  and  $y$ , meaning that each temporal kernel is applied separately to every spatial pixel.

All the `einsum` operations are associative and commutative<sup>5</sup>, so we have full freedom over the order of contractions. For example, we can first generate the temporal kernels from the orthogonal polynomials, then perform the convolutions with the input features (the order of operations we assumed by default). But equally valid, we can also first project the input features onto the basis polynomials separately, then weigh and accumulate these results using the polynomial coefficients. This can be written as  $(u_{cxyt} M_{ntt'}) \gamma_{cdn} = x_{cxynt} \gamma_{cdn} = y_{dxyt'}$  in `einsum` form, where  $x$  represents the intermediate projections. Note that this contraction ordering freedom is not allowed for unstructured temporal kernels, as there is no intermediate basis  $n$  to project anything onto.

5. Matrix multiplications, or tensor contractions in general, are not commutative. However, the `einsum` notation restores the commutativity by explicitly keeping track of the contraction indices, so that the contraction operations are invariant under commutation.

In practice, we select the contraction path based on optimizing memory or computational usage [9], depending on the training hardware and cost limitations. This is possible because the memory and computational costs can be calculated for any contraction path, given the dimensions of the contraction indices (tensor shapes). See Appendix A.2 for how these costs can be calculated. The choice of the optimal contraction path can be automatically selected using the `opt_einsum` library<sup>6</sup>, under certain modifications of the cost estimation rules (see Appendix A.3).

## 4 NETWORK ARCHITECTURE

The main network block is a spatiotemporal convolution block, factorized as a (1+2)D convolution. In other words, we perform a temporal convolution on each spatial pixel followed by a spatial convolution on each temporal frame, similar in form to the R(2+1)D convolution block [32], or a previously proposed (1+2)D network for online eye tracking [23]. Furthermore, each of the temporal and spatial convolutional layers can be additionally factorized as a depthwise-separable (DWS) layer [14], to further reduce the computational costs. For every temporal kernel (for every channel pairing, every layer, and every network variant), we use  $\alpha = -0.25$  and  $\beta = -0.25$  for the Jacobi polynomial basis functions, with degrees up to 4.

Several other minor design choices we made for our networks (besides polynomial kernels) include:

- We keep every operation in our network fully causal, such that the network can be easily adapted for online inference with minimal latency. Importantly, we perform only causal temporal convolutions.
- After every temporal convolution, we perform a causal Group Normalization [23] with groups = 4. And after every spatial convolution, we perform a Batch Normalization. This strategy of using a mixture of static and dynamic is shown to improve performance [8].

6. This library is well-integrated with PyTorch so can be easily called. However, a caveat here is that the optimal contraction path in reality also depends on software support and hardware architecture, and there are scenarios where choosing the most compute-efficient contraction path does not necessarily lead to speedups, especially if the operations are memory-bound.

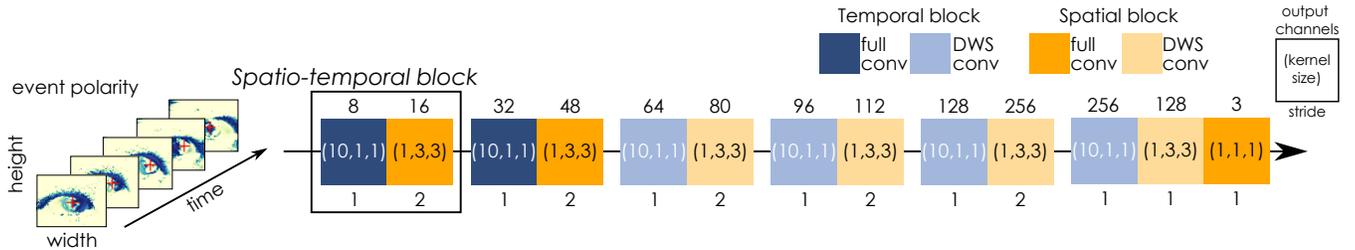


Fig. 2. A representative network used for eye tracking. The backbone consists of 5 spatiotemporal blocks. *full conv* is short for full convolution and denoted by darker blocks, and *DWS conv* is short for depthwise-separable convolution and denoted by lighter blocks. The detection head is inspired by CenterNet, with the modification that the  $3 \times 3$  convolution is made depthwise-separable and a temporal layer is prepended to it.

- We apply a ReLU activation after every convolution layers, and also within every DWS layer. The activation function is intentionally kept simple, for ease of implementation on mobile or edge devices, and to promote activation sparsity in the network.

For tasks requiring object tracking or detection (see Sections 5.2 and 5.3), we attach a temporally smoothed CenterNet detection head to the backbone (see Fig. 2), consisting of a DWS temporal layer, a  $3 \times 3$  DWS spatial layer, and a final pointwise layer [35], with ReLU activations in between. Since our backbone is already spatiotemporal in nature and capable of capturing long-range temporal correlations, we do not use any additional recurrent heads (e.g. ConvLSTMs) or temporal-based loss functions [24].

## 5 EXPERIMENTS

We conduct experiments on standard computer vision tasks with event-based datasets. For all baseline experiments, we preprocess the event data into  $4d$  tensors of shape  $(2, H, W, T)$ , with the 2 polarity channels retained. General details of data and training pipelines are given in Appendix B.

### 5.1 DVS128 Hand Gesture Recognition

The DVS128 dataset contains recordings of 10 hand gesture classes performed by different subjects [1], recorded with a  $128 \times 128$  dynamic vision sensor (DVS) camera. We use a simple backbone consisting of 5 spatiotemporal blocks. The network architecture is almost the same as that shown in Fig. 2 with the exception that the detection head is replaced by a spatial global-average pooling layer followed by a simple 2-layer MLP to produce classification logits (technically a pointwise Conv1D layer during training). This means that the output produces raw predictions at 10 ms intervals, which already by themselves are surprisingly high-quality. With additional output filtering on the network predictions, the test accuracy can be pushed to 100% (see Table 1). In addition, we compare the PLEIADES network with a counterpart that uses unstructured temporal kernels, or simply a Conv(1+2)D network [23], and find that PLEIADES has better performance with a smaller number of parameters (due to the polynomial compression).

Unfortunately, previous studies lacked a unified standard for performing the evaluations on the dataset, so it is not entirely clear the metrics being reported. In particular, some networks perform online inference, and the others process

TABLE 1

The raw 10-class test accuracy of several networks on the DVS128 dataset. With the exception of models marked with an asterisk, no output filtering is performed on the networks. PLEIADES is evaluated on output predictions where all temporal layers process nonzero valid frames, which incurs a natural warm-up latency of 0.44 seconds (see Section 5.1). Additionally, a majority filter of window 0.15 seconds can be applied to the raw PLEIADES predictions to reach 100% accuracy.

Model	Accuracy (%)	Parameters
PLEIADES + filtering*	100.00	192K
PLEIADES	99.59	192K
Conv(1+2)D	99.17	196K
ANN-Rollouts [16]	97.16	500K
TrueNorth CNN* [1]	96.59	18M
SLAYER [30]	93.64	-

entire recording segments before producing a prediction. Here, we produce a more general accuracy vs. latency relationship for our network variants, as to establish multiple Pareto frontiers for comparisons. In the context of this experiment, the latency is simply the number of event frames (from the beginning of the recording) multiplied by the bin size, that the network has “seen” before making a prediction.

Note that if we wish to guarantee that every temporal layer is working with nonzero valid input features, then the network will have a natural latency of equal to  $(\text{number of temporal layers}) \times (\text{temporal kernel size} - 1)$ , meaning that the baseline network would have a latency of 440 ms. However, if we relax this requirement, by assuming implicit zero paddings for missing data (or equivalently zero initialization of the buffers), we can then allow the network to perform inference at much lower latencies. On the contrary, if latency is not of primary concern, then we can also apply output filtering to the network [1] to boost performance at the cost of higher system latencies. See Appendix B.1 for details on how the network can be configured to run at different latencies.

There are several ways to “force” the network to respond faster. The natural way is to simply use a smaller kernel size or binning window (temporal step size). Here, we test two model variants with a binning window of 5 ms and 10 ms, keeping the temporal kernel size fixed at 10 (meaning that the 5 ms variant has a shorter effective temporal window). Another way is to randomly mask out frames starting from the beginning of the input segment, to force the network to “respond” only to the more recent input frames, such that the

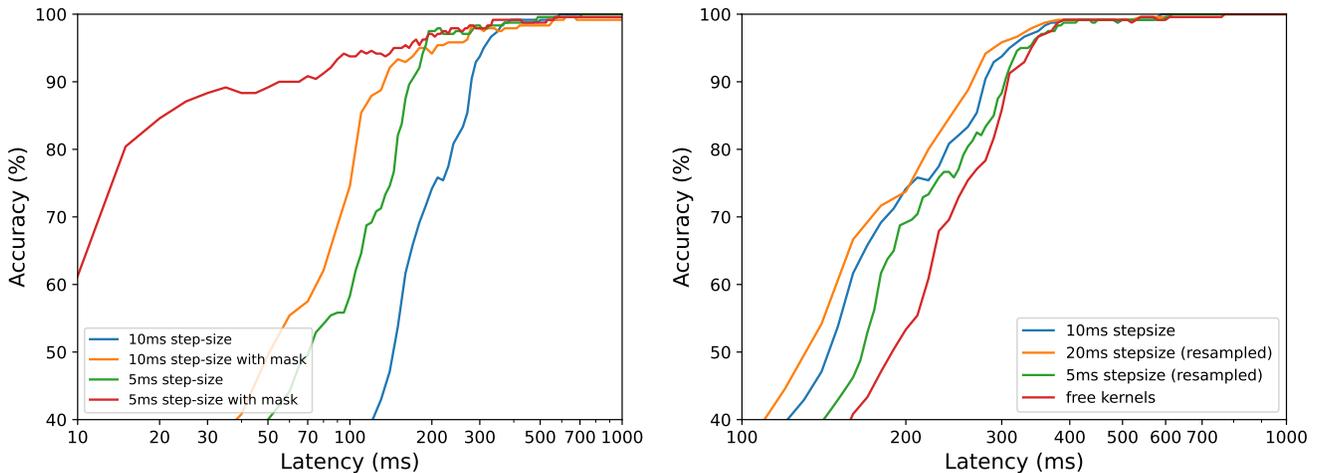


Fig. 3. (Left) The accuracy vs. latency curves for different PLEIADES variants with a kernel size of 10 but different step sizes on the DVS128 dataset. A masking augmentation is optionally used to randomly mask out the starting frames of dataset segments during training, in order to stimulate faster responses in the network. (Right) The accuracy vs. latency curves for different PLEIADES variants with an effective temporal window of 100 ms for each temporal layer, but having different step sizes. The benchmark network is trained with a kernel size of 10 and a step size of 10 ms, and the other variants are resampled without additional fine-tuning. A network variant trained without any structured temporal kernel is also displayed as a baseline reference.

effective response window of the network is shorter<sup>7</sup>. See the left plot of Fig. 3 of the effects of these two approaches for stimulating faster network response times. See Appendix B.1 for details on the random masking augmentation.

The benchmark network is trained with step-size  $\Delta\tau$  of 10 ms, which is also the event data bin-size. Here, we change the step-sizes to 5 ms and 20 ms (upsampling and downsampling) without fine-tuning the network, and simply re-discretize the basis polynomials under the new step-sizes (see Section 3.2) and re-bin the event data. To keep the effective temporal window (thus the network behavior) the same, the 5 ms step-size network would have a kernel size of 20, and the 20 ms step-size network would have a kernel size of 5. We see from the right plot of Fig. 3 that the accuracy vs. latency curve does not vary much under the time-step resampling. In the same plot, we also show the performance for the Conv(1+2)D baseline network with unstructured kernels, denoted as “free kernels”, to compare the PLEIADES variants against.

## 5.2 AIS2024 Event-based Eye Tracking

We use the same backbone as the network for the DVS128 hand gesture recognition, but with a temporal step-size of 5 ms. We simply replace the 2-layer MLP classification head with the CenterNet detection head and loss adapted from [23]. Note that we omit predictions of the bounding box sizes, and only predict center points of pupils for this challenge. See Fig. 2 for a drawing of the network architecture.

## 5.3 Prophesee GEN4 Roadscene Object Detection

The Prophesee GEN4 Dataset is a road-scene object detection dataset collected with a megapixel event camera [24]. The

7. Note this does not decrease the theoretical latency of the network, but rather improves the prediction accuracy of the network when fed with fewer input frames (hence having less “effective” latency for a given accuracy). However, it is likely this augmentation will degrade accuracy when more frames are fed to the network, when compared to a counterpart not trained with this masking augmentation.

TABLE 2

The 10-pixel, 5-pixel, and 3-pixel tolerances for the CVPR 2024 AIS eye tracking challenge. The performances of other models are extracted from [34].

Model	p10	p5	p3	Parameters
PLEIADES + CenterNet	99.58	97.95	94.94	277K
MambaPupil	99.42	97.05	90.73	-
CETM	99.26	96.31	83.83	7.1M
Conv(1+2)D	99.00	97.97	94.58	1.1M
ERVT	98.21	94.94	87.26	150K
PEPNet	97.95	80.67	49.08	640K

dataset has around 14 hours of recording with both daytime and night time, and both rural and urban driving scenarios. It contains 7 classes, but we evaluate the mAP only on 2 classes: cars and pedestrians, to be consistent with the original evaluation pipeline [24]. See Appendix B.2 for details on the model architecture used and the training pipeline. The backbone network is an hourglass network built from a stack of spatiotemporal blocks with a temporal step size of 10 ms. The detection head is again the CenterNet detection head as described in Section 4. We do not use any non-max suppression on the bounding box outputs, as suggested in the original CenterNet pipeline being robust against spurious bounding box predictions, which is further augmented by the implicit temporal consistency of our network. See Appendix B.2 for details of the network architecture.

TABLE 3

The performance of PLEIADES with a CenterNet detector compared to the models introduced in the original benchmark paper.

Model	mAP	Parameters
PLEIADES + CenterNet	0.556	0.576 M
RED	0.43	24.1 M
Gray-RetinaNet	0.43	32.8 M

## 6 LIMITATIONS

Since the temporal layers of our network work like a finite-window filter with circular buffers, a practical limitation may be the high memory cost to explicitly buffer the moving window of recent input features, which is worsened if the temporal kernel size or spatial dimensions are large. However, by virtue of the polynomial structures of our temporal kernels, we can derive estimates of an online running projection of the past input features onto the fixed polynomial basis functions [11], [31], an idea briefly discussed in Section 3.3 also.

These compressed coefficients are analogous to internal states in recurrent networks, which we can then perform a pointwise operation with our pre-trained kernel coefficients to estimate the would-be output of the original finite-window convolution<sup>8</sup>. This is similar in spirit to deep state-space modeling [12], [13], and there may be a way to potentially integrate such developments into our network architecture. In other words, we can potentially convert our convolutional network into a recurrent network for even better online inference efficiency.

## 7 CONCLUSION

We introduced PLEIADES, a spatiotemporal network with temporal kernels built from orthogonal polynomials. The network achieved state-of-the-art results on all the event-based benchmarks we tested, and its performance is shown to be stable under temporal resampling without additional fine-tuning. Currently, the network is configured as a standard neural network, which by itself is already ultra-light in memory and computational costs. To truly leverage the full advantage of event-based processing, we can consider using intermediate loss functions to promote activation sparsity [23]. Another direction is to adapt/convert this architecture into a spiking system by leveraging the structure of the polynomial kernels to provide richer dynamics to neurons beyond the typical and arbitrary leaky integrate-and-fire neurons, without adding computational complexity in inference or training. It offers the possibility for such spiking systems to be trained as a convolutional network without having to simulate any differential equation of the internal neural dynamics.

## ACKNOWLEDGEMENT

We would like to acknowledge Nolan Ardolino, Kristofor Carlson, M. Anthony Lewis, and Anup Varase (listed in alphabetical order) for discussing ideas and offering insights for this project. We would also like to thank Daniel Endraws for performing quantization studies on the PLEIADES network, and Sasskia Brüers for help with producing the figures.

<sup>8</sup>. This is analogous to the convolution theorem (at least in the forward direction), which states that a convolution in the original domain becomes a pointwise product of the coefficients in an appropriately transformed domain.

## REFERENCES

- [1] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey McKinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, et al. A low power, fully event-based gesture recognition system. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7243–7252, 2017.
- [2] Daniel Y Fu, Tri Dao, Khaled K Saab, Armin W Thomas, Atri Rudra, and Christopher Ré. Hungry Hungry Hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.
- [3] Daniel Y Fu, Elliot L Epstein, Eric Nguyen, Armin W Thomas, Michael Zhang, Tri Dao, Atri Rudra, and Christopher Ré. Simple hardware-efficient long convolutions for sequence modeling. In *International Conference on Machine Learning*, pages 10373–10391. PMLR, 2023.
- [4] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J Davison, Jörg Conradt, Kostas Daniilidis, et al. Event-based vision: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(1):154–180, 2020.
- [5] Daniel Gehrig, Antonio Loquercio, Konstantinos G Derpanis, and Davide Scaramuzza. End-to-end learning of representations for asynchronous event-based data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5633–5643, 2019.
- [6] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
- [7] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.
- [8] Ariel Gordon, Hanhan Li, Rico Jonschkowski, and Anelia Angelova. Depth from videos in the wild: Unsupervised monocular depth learning from unknown cameras. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8977–8986, 2019.
- [9] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021.
- [10] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.
- [11] Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *Advances in neural information processing systems*, 33:1474–1487, 2020.
- [12] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.
- [13] Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in neural information processing systems*, 34:572–585, 2021.
- [14] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [15] Dmitry Ivanov, Aleksandr Chezhegov, and Denis Larionov. Neuronomorphic artificial intelligence systems. *Frontiers in Neuroscience*, 16:959626, 2022.
- [16] Alexander Kugele, Thomas Pfeil, Michael Pfeiffer, and Elisabetta Chicca. Efficient processing of spatio-temporal data streams with spiking neural networks. *Frontiers in neuroscience*, 14:512192, 2020.
- [17] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017.
- [18] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks: A unified approach to action segmentation. In *Computer Vision—ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8–10 and 15–16, 2016, Proceedings, Part III 14*, pages 47–54. Springer, 2016.
- [19] Ana I Maqueda, Antonio Loquercio, Guillermo Gallego, Narciso García, and Davide Scaramuzza. 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, 2018.
- [20] Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

- [21] Eric Nguyen, Karan Goel, Albert Gu, Gordon W Downs, Preey Shah, Tri Dao, Stephen A Baccus, and Christopher Ré. S4ND: Modeling images and videos as multidimensional signals using state spaces. *arXiv preprint arXiv:2210.06583*, 2022.
- [22] Ashutosh Pandey and DeLiang Wang. Tcnn: Temporal convolutional neural network for real-time speech enhancement in the time domain. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6875–6879. IEEE, 2019.
- [23] Yan Ru Pei, Sasskia Brüers, Sébastien Crouzet, Douglas McLelland, and Olivier Coenen. A Lightweight Spatiotemporal Network for Online Eye Tracking with Event Camera. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2024.
- [24] Etienne Perot, Pierre De Tournemire, Davide Nitti, Jonathan Masci, and Amos Sironi. Learning to detect objects with a 1 megapixel event camera. *Advances in Neural Information Processing Systems*, 33:16639–16652, 2020.
- [25] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pages 28043–28078. PMLR, 2023.
- [26] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.
- [27] Zhaofan Qiu, Ting Yao, and Tao Mei. Learning spatio-temporal representation with pseudo-3d residual networks. In *proceedings of the IEEE International Conference on Computer Vision*, pages 5533–5541, 2017.
- [28] David W Romero, Anna Kuzina, Erik J Bekkers, Jakub M Tomczak, and Mark Hoogendoorn. CKConv: Continuous kernel convolution for sequential data. *arXiv preprint arXiv:2102.02611*, 2021.
- [29] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.
- [30] Sumit B Shrestha and Garrick Orchard. SLAYER: Spike layer error reassignment in time. *Advances in neural information processing systems*, 31, 2018.
- [31] Andreas Stöckel. Discrete function bases and convolutional neural networks. *arXiv preprint arXiv:2103.05609*, 2021.
- [32] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. A closer look at spatiotemporal convolutions for action recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 6450–6459, 2018.
- [33] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre Memory Units: Continuous-time representation in recurrent neural networks. *Advances in neural information processing systems*, 32, 2019.
- [34] Zuowen Wang, Chang Gao, Zongwei Wu, Marcos V. Conde, Radu Timofte, Shih-Chii Liu, Qinyu Chen, et al. Event-Based Eye Tracking. AIS 2024 Challenge Survey. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2024.
- [35] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.
- [36] Alex Zihao Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. Unsupervised event-based learning of optical flow, depth, and egomotion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 989–997, 2019.
- [37] Lianghui Zhu, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. Vision Mamba: Efficient visual representation learning with bidirectional state space model. *arXiv preprint arXiv:2401.09417*, 2024.

## APPENDIX A

### OPTIMAL CONTRACTION ORDER MEMORY AND COMPUTE

#### A.1 The Rules of Einsum

The rules of contracting an einsum expression can be summarized as follows:

- At every contraction step, any two operands can be contracted together, as the einsum operation is associative and commutative.
- For any indices appearing in the two contracted operands but not the output and other operands, these indices can be summed away for the intermediate contraction results.

Any ordering of contractions (or a contraction path) following these rules are guaranteed to yield equivalent results.

A simple example is multiplying three matrices together, or  $D_{il} = A_{ij}B_{jk}C_{kl}$ . In the first stage, we can first choose to contract  $A_{ij}$  and  $B_{jk}$ , which would yield an intermediate result of  $M_{ik}$ , where the index  $j$  is contracted away as it does not appear in the output  $D_{il}$ . In the second stage, we then contract  $M_{ik}$  and  $C_{kl}$  to arrive at the output  $D_{il}$ .

We can also choose to do the contractions in any other order, and the result will remain the same. As a more extreme example, we can even perform an outer product first  $M_{ijkl} = A_{ij}C_{kl}$ , noting that we cannot contract away the  $j$  and  $k$  indices yet as they appear in  $B_{jk}$  still. The contractions of  $j$  and  $k$  then need to be left to the second stage contraction,  $D_{il} = M_{ijkl}B_{jk}$ . Intuitively, we feel that this is a very suboptimal way of doing multiplication of three matrices, and we can formalize why this is by looking at the memory and compute complexities of performing a contraction.

#### A.2 Memory and Compute Requirements of a Contraction

If we assume that we are not performing any kernel fusion, and explicitly materializing and saving all intermediate tensors for backpropagation, then the extra memory and compute incurred by each contraction step is as follows:

- The memory needed to store the intermediate result is simply the size of the tensor, or equivalently the product of the sizes of its indices.
- The compute needed to evaluate the intermediate result is the product of the sizes of all indices involved in the contraction (repeated indices are counted only once).

Again, we can use the  $D_{il} = A_{ij}B_{jk}C_{kl}$ , where we assume that the index sizes to be<sup>9</sup>  $i$ ,  $j$ ,  $k$ , and  $l$ . Doing the first stage contraction  $M_{ik} = A_{ij}B_{jk}$  will require  $ik$  units of extra memory and  $ijk$  units of compute, and doing the second stage contraction  $D_{il} = M_{ik}C_{kl}$  will require no extra memory besides that for storing the output and  $ikl$  units of compute. This gives us a total extra memory requirement of  $ik$  units and a total compute requirement of  $ijk + ikl$  units.

<sup>9</sup> From here on, we will consistently use this abuse of notation where the same letter will be used to denote both the index and the corresponding dimensional size of that index.

On the other hand, if we perform the outer product  $M_{ijkl} = A_{ij}C_{kl}$  first, this will require  $ijkl$  units of extra memory and  $ijkl$  units of compute. The second stage contraction  $D_{il} = M_{ijkl}C_{kl}$  will require  $ijkl$  units of compute. Therefore, the total memory requirement of this contraction path is  $ijkl$  units and the total compute requirement is  $2ijkl$  units, both being significantly worse than the first contraction path, regardless of the sizes of the tensors.

A more subtle example (the only remaining contraction path) is contracting the operands from back to front, which we can verify requires a total memory of  $jl$  units and a total compute of  $jkl + ijl$  units. So this is only more memory optimal than the first contraction path if  $jl < ik$ , and more compute optimal if  $jkl + ijl < ijk + ikl$ , which may not be immediately obvious from inspection as the optimality now depends on the sizes of the tensors.

Note that it is assumed that every tensor involved in the einsum expression requires gradient from backpropagation, in the context of neural network training. This is why we identify the size of each intermediate result as “additional memory”, as they need to be stored as tensors used for gradient computation. In addition, it is not difficult to see that for einsum operations, the computational costs required for backpropagation is exactly double that of the forward computation. Therefore, we only need to consider the forward pass of the einsum expression, which is what we have been doing.

Importantly, note that this argument for memory and computational costs of gradient computations is only true under the assumption of reverse-mode automatic differentiation (backpropagation), which is what is used in almost all modern machine learning frameworks. In other words, we do not consider more general forms of automatic differentiation such as the forward-mode variant. Another important note is that in practice if the operations are memory-bound, then the computational cost estimates may not be useful for training time estimation.

### A.3 Convolution with a Parameterized Temporal Kernel

Recall in the main text that the equation for performing a full convolution with a polynomially parameterized temporal kernel is

$$y_{dxyt} = u_{cxyt} \gamma_{dnc} M_{nt't}(\bar{P}), \quad (8)$$

where the convolution operator tensor  $M(\bar{P})$  based on the discretized polynomial basis functions  $\bar{P}$  is given by the following Toeplitz matrix for each degree or basis  $n$  (assuming that kernel size is 5 with the discretized timestamps being  $\{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ ):

$$M(\bar{P})_n = \begin{bmatrix} \bar{P}[\tau_0] & 0 & 0 & 0 & 0 & \dots & 0 \\ \bar{P}[\tau_1] & \bar{P}[\tau_0] & 0 & 0 & 0 & \dots & 0 \\ \bar{P}[\tau_2] & \bar{P}[\tau_1] & \bar{P}[\tau_0] & 0 & 0 & \dots & 0 \\ \bar{P}[\tau_3] & \bar{P}[\tau_2] & \bar{P}[\tau_1] & \bar{P}[\tau_0] & 0 & \dots & 0 \\ \bar{P}[\tau_4] & \bar{P}[\tau_3] & \bar{P}[\tau_2] & \bar{P}[\tau_1] & \bar{P}[\tau_0] & \dots & 0 \\ 0 & \dots & \bar{P}[\tau_4] & \bar{P}[\tau_3] & \bar{P}[\tau_2] & \bar{P}[\tau_1] & \bar{P}[\tau_0] \end{bmatrix}, \quad (9)$$

where for a valid-type convolution we omit the first four rows of the matrix.

Note that we need to make two modifications to the memory and compute calculation rules in Section A.2 to adapt for the sparse and Toeplitz structure of the convolution matrix  $M$ . First is that the memory required for storing any tensor containing both  $t, t'$  is guaranteed to be some form of convolution kernel, so it should only contribute a memory factor of  $N_\tau$  (the kernel size) instead of  $N_t N_{t'}$ . Second is that any contraction of two tensors with one containing  $t$  and the other containing  $t, t'$  is guaranteed to be a temporal convolution, so should similarly contribute a compute factor of  $N_{t'} N_\tau$  for valid-type convolutions and  $N_t N_\tau$  for same-type convolutions. For our implementation, we monkey patch these modifications into the `opt_einsum` package used to provide memory and FLOP estimations of einsum expressions.

TABLE 4

The memory and compute requirements for each possible contraction path, where we are using a slight abuse of notation by allowing the index to represent the dimensional size of that index in the “extra memory” and “total compute” columns. The initial equation `cxyt, dnc, nt't` is always assumed. We assume here that  $N_t = N_{t'}$  for simplicity (equivalent to performing same-type convolutions).

Contraction Path	Extra Memory	Total Compute
<code>-&gt; dnxyt, nt't -&gt; dxyt'</code>	$dnxyt$	$nxyt(dc + d\tau)$
<code>-&gt; ncxyt', dnc -&gt; dxyt'</code>	$ncxyt$	$nxyt(c\tau + dc)$
<code>-&gt; cxyt, dct't -&gt; dxyt'</code>	$cxyt$	$dnc\tau + dcxyt\tau$

Following the prescription given above for calculating the memory and compute requirements for performing contractions, we summarize the requirements of each contraction path for the temporal convolution in Table 4. We only consider the case for full convolutions, but the case for depthwise convolutions is analogous.

The first contraction path first contracts the input with the polynomial coefficients, then convolves the intermediate result with the basis functions. The second contraction path first convolves the input with the basis functions, then contracts the intermediate result with the polynomial coefficients. The third contraction path first generates the temporal kernels from the polynomial coefficients and basis functions, then convolves them with the input features. In most cases, we see that the last contraction path is most memory efficient in typical cases, or when  $c < dn$ . However, the optimality for computational efficiency is more subtle and requires comparison of  $dn(c + \tau)$ ,  $nc(\tau + d)$ , and  $dct\tau$ .

## APPENDIX B DETAILS OF EXPERIMENTS

To convert events into frames, we choose the binning window to be 10 ms, unless otherwise specified. This time step is kept fixed throughout our network, as we do not perform any temporal resampling through the network. For the DVS128 and AIS2024 eye tracking experiments, we perform simple direct binning along with random affine augmentations (with rotation angles up to 10 degrees, translation factors up to 0.1, and spatial scaling factors up to 1.1). For the Prophesee roadscene detection, we perform event-volume binning (analogous to bilinear interpolation), with augmentations consisting of horizontal flips at 0.5 probability and random scaling with factors from 0.9 to 1.1.

Recall that our network performs valid-type causal temporal convolutions which reduces the number of frames by  $(\text{kernel size} - 1)$  per temporal convolution. To avoid introducing any implicit temporal paddings to our network, we prepend extra frames (relative to the labels) to the beginning of the input segment. The total number of extra frames is then  $(\text{number of temporal layers}) \times (\text{kernel size} - 1)$ .

For all training runs, we use the AdamW optimizer with a learning rate of 0.001 and weight decay of 0.001 (with PyTorch default keywords), along with the cosine decay learning rate scheduler (updated every step) with a warmup period of around 0.01 of the total training steps. The runs are performed with automatic mixed precision (float 16) with the model `torch.compile`. All training jobs are done on a single NVIDIA A30 GPU.

## B.1 DVS Hand Gesture Recognition

Following the standard benchmarking procedure on this dataset, we only train and evaluate on the first 1.5 seconds of each trial, and filter out the “other” class where the subject performs random gestures not falling into the 10 predefined classes.

As mentioned, the network requires at least  $(\text{number of temporal layers}) \times (\text{kernel size} - 1) + 1$  frames of inputs to guarantee that every temporal layer is processing “valid” nonzero input features. To generate output predictions with less than this number of frames, we can prepend zeros to layer inputs where needed to match the kernel size. This simulates the behavior of initializing the buffers of the temporal layers with zeros during online inference.

If the number of input frames is greater than  $(\text{number of temporal layers}) \times (\text{kernel size} - 1) + 1$ , then the network will produce more than one output predictions. If the latency budget allows, we can apply a majority filter to the classification predictions of the network, such that there is more confidence in the predictions.

To force the network to respond faster, we apply a custom random temporal masking augmentation sample-wise with 1/2 probability. The random masking operation works by selecting a frame uniformly random from the first frame to the middle frame of the segment, then the selected frame and every frame preceding it is completely set to zero. This means that the network will be artificially biased to respond to more recent input features during inference, thereby effectively decreasing its response latency.

### B.1.1 Input Sparsity

We perform 4-bit quantization (with quantization aware training) on the gesture recognition network, and find that the network can achieve very high sparsity even without applying any regularization loss, given that it interfaces with event-based data and uses ReLU activations (which is sparsity promoting).

## B.2 Prophesee GEN4 Roadscene Object Detection

Following a recipe similar to the original paper, we remove bounding boxes that are less than 60 pixels in the diagonal. In addition, we perform event-volume binning which simultaneously performs spatial resizing from (720, 1280)

TABLE 5  
Input sparsity for each layer of the gesture recognition network backbone under 4-bit quantization.

Layer	Input Sparsity
Conv(1+2)D	0.99
Conv(1+2)D	0.94
Conv(1+2)D	0.94
Conv(1+2)D	0.79
Conv(1+2)D	0.68

to (160, 320) and temporal binning of 10 ms. For data augmentations, we perform horizontal flips at 0.5 probability and random scaling with factors from 0.9 to 1.1.

The CenterNet detection head produces feature frames where each frame is spatially shaped (40, 80). Each pixel contains  $7 + 2 + 2$  outputs containing 7 class logits (center point heatmaps), the bounding box height and width scales, and the bounding box center point  $x_c$  and  $y_c$  offsets. We perform evaluations directly on these raw predictions, without any output filtering (e.g. no non-max suppression). The network is trained on the full 7 road-scene classes of the dataset, and the mAP is evaluated on the cars and pedestrians classes, at confidence thresholds from 0.05 to 0.95 in steps of 0.05 and averaged using trapezoid integration.

See Table. 6 for details on the model architecture, which resembles an hourglass structure. Unless otherwise indicated, the temporal kernel size is assumed to be 10, causal and valid-type. The spatial kernel size is assumed to be  $3 \times 3$ , where the spatial stride can be inferred from the output shape of the layer. DWS denotes both the temporal and spatial layers in the Conv(1+2)D block as depthwise-separable. The BottleNeck block is similar (but not identical) to the IRB block in MobileNetV2; it is a residual block with the residual path containing three Conv2D layers with ReLU activations in between: a depthwise  $3 \times 3$  Conv2D followed by a pointwise Conv2D quadrupling the channels followed by a pointwise Conv2D quartering the channels.

Before each decoder layer, the input feature is first upsampled spatially by a factor of  $2 \times 2$ . It is then summed with an intermediate output feature from an encoder layer that has the same spatial shape. To match the temporal shapes, the beginning frames are truncated if necessary. The remaining frames are projected with a pointwise convolutional layer (a long-range skip connection).

TABLE 6  
The PLEIADES + CenterNet architecture used for the Prophesee dataset.

Layer	Output Shape	Channels
Input	$(2, T, 160, 320)$	
<b>Encoder</b>		
Conv(1+2)D	$(32, T - 9, 80, 160)$	$2 \rightarrow 16 \rightarrow 32$
BottleNeck 2D	$(32, T - 9, 80, 160)$	$32 \rightarrow 32 \rightarrow 128 \rightarrow 32$
DWS Conv(1+2)D	$(64, T - 18, 40, 80)$	$32 \rightarrow 48 \rightarrow 64$
BottleNeck 2D	$(64, T - 18, 40, 80)$	$64 \rightarrow 64 \rightarrow 256 \rightarrow 64$
DWS Conv(1+2)D	$(96, T - 27, 20, 40)$	$64 \rightarrow 80 \rightarrow 96$
DWS Conv2D	$(128, T - 27, 10, 20)$	$96 \rightarrow 128$
DWS Conv2D	$(256, T - 27, 5, 10)$	$128 \rightarrow 256$
<b>Decoder</b>		
Upsample	$(256, T - 27, 10, 20)$	
DWS Conv2D	$(256, T - 27, 10, 20)$	$256 \rightarrow 256$
Upsample	$(256, T - 27, 20, 40)$	
DWS Conv2D	$(256, T - 27, 20, 40)$	$256 \rightarrow 256$
Upsample	$(256, T - 27, 40, 80)$	
DWS Conv2D	$(256, T - 27, 40, 80)$	$256 \rightarrow 256$
<b>CenterNet Head</b>		
DWS Conv(1+2)D	$(128, T - 27, 40, 80)$	$256 \rightarrow 256 \rightarrow 128$
pointwise Conv	$(11, T - 27, 40, 80)$	$128 \rightarrow 11$