AEGNN: Asynchronous Event-based Graph Neural Networks

Simon Schaefer*    Daniel Gehrig*    Davide Scaramuzza

Dept. Informatics, Univ. of Zurich and
Dept. of Neuroinformatics, Univ. of Zurich and ETH Zurich

Abstract

The best performing learning algorithms devised for event cameras work by first converting events into dense representations that are then processed using standard CNNs. However, these steps discard both the sparsity and high temporal resolution of events, leading to high computational burden and latency. For this reason, recent works have adopted Graph Neural Networks (GNNs), which process events as “static” spatio-temporal graphs, which are inherently “sparse”. We take this trend one step further by introducing Asynchronous, Event-based Graph Neural Networks (AEGNNs), a novel event-processing paradigm that generalizes standard GNNs to process events as “evolving” spatio-temporal graphs. AEGNNs follow efficient update rules that restrict recomputation of network activations only to the nodes affected by each new event, thereby significantly reducing both computation and latency for event-by-event processing. AEGNNs are easily trained on synchronous inputs and can be converted to efficient, “asynchronous” networks at test time. We thoroughly validate our method on object classification and detection tasks, where we show an up to a 11-fold reduction in computational complexity (FLOPs), with similar or even better performance than state-of-the-art asynchronous methods. This reduction in computation directly translates to an 8-fold reduction in computational latency when compared to standard GNNs, which opens the door to low-latency event-based processing.

Multimedia Material

For videos, code and more, visit our project page https://uzh-rpg.github.io/aegnn/.

1. Introduction

Compared to standard frame-based cameras, which measure absolute intensity at a synchronous rate, event-cameras only measure changes in intensity, and do this independently for each pixel, resulting in an asynchronous and bi-
nary stream of events (Figure 1 (a)). These events measure a highly compressed representation of the visual signal and are characterized by microsecond-level latency and temporal resolution, a high dynamic range of up to 140 dB, low motion blur, and low power (milliwatts instead of watts). Due to these outstanding properties, event cameras are indispensable sensors in challenging application domains—such as robotics [9, 21, 48, 53], autonomous driving [18, 52, 57], and computational photography [3, 45, 54, 55]—characterized by frequent high-speed motions, low-light and high-dynamic-range scenes, or in always-on applications, where low power is needed, such as IoT video surveillance [24, 37]. A survey about applications and research in event-based vision can be found in [14].

The output of event cameras is inherently sparse and asynchronous, making them incompatible with traditional computer-vision algorithms designed for standard images. This prompts the development of novel algorithms that optimally leverage the sparse and asynchronous nature of events. In doing so, existing algorithms designed for event cameras have traded off latency and prediction performance. Filtering-based [40] [29] approaches process events sequentially, and, thus, can provide low-latency predictions and a high temporal resolution. However, these approaches usually rely on handcrafted filter equations, which do not scale to more complex tasks, such as object detection or classification. Spiking Neural Networks (SNNs) are one instance of filtering-based models, which seek to learn these rules in a data-driven fashion, but are still in their infancy, lacking general and robust learning rules [19, 30, 50]. As a result, SNNs typically fail to solve more complex high-level tasks [2, 40, 42, 52]. Many of the challenges above can be avoided by processing events as batches. In fact, recent progress has been made by converting batches of events into dense, image-like representations and processing them using methods designed for images, such as convolutional neural networks (CNNs). By adopting this paradigm, learning-based methods using CNNs have made significant strides in solving computer vision tasks with events [17, 22, 35, 43, 45, 54, 58, 59].

However, while easy to process, treating events as image-like representations discards their sparse and asynchronous nature and leads to wasteful computation. This wasteful computation directly translates to higher power consumption and latency [1, 24, 38]. A recent line of work [16] showed on an FPGA that by reducing the computational complexity by a factor of 5, they could reduce the latency by a factor of 5 while reducing the power consumption by a factor of 4. Therefore, by eliminating wasteful computation, we can expect significant decreases in the power consumption and latency of learning systems.

Currently, this wasteful computation is caused by two factors: On the one hand, due to the working principle of event cameras, they trigger predominantly at edges, while large texture-less or static regions remain without events. Image representations typically encode these regions as zeros, which are then unnecessarily processed by standard neural networks. On the other hand, for each new event, standard methods would need to recompute all network activations. However, events only measure single pixel changes and, thus, leave most of the activations unchanged, leading to unnecessary recomputation of activations.

A recent line of work seeks to address both of these challenges by reducing the computational complexity of learning-based approaches while maintaining the high temporal resolution of events. A key ingredient to keeping high performance in this setting was the adoption of geometric learning methods, such as recursive point-cloud processing [49] or Asynchronous Sparse Convolutions [36]. In both works, standard neural networks were trained using batches of events, leveraging well-established learning techniques such as backpropagation, and then deploying them in an event-by-event fashion at test time, thus minimizing computation. However, both of these methods suffer from limitations: While [49] does not perform hierarchical learning, limiting scalability to complex tasks, [36], relies on a specific type of input representation, which discards the temporal information of events.

In this work, we introduce Asynchronous, Event-based Graph Neural Networks (AEGNN), a neural network architecture geared toward processing events as graphs in a sequential manner (Fig. 1). For each new event, our method only performs local changes to the activations of the GNN, and then deploy them in an asynchronous mode during test-time. These rules are general and can be applied to most existing graph neural network architectures.

(i) We introduce AEGNN, a novel paradigm for processing events sparsely and asynchronously as temporally evolving graphs. This allows us to process events efficiently, without sacrificing their sparsity and high temporal resolution.

(ii) We derive efficient update rules, which allow us to simply train AEGNNs on synchronous event-data, and then deploy them in an asynchronous mode during test-time. These rules are general and can be applied to most existing graph neural network architectures.

(iii) We apply AEGNNs on object recognition and object detection benchmarks. For object detection, we
show similar performance to state-of-the-art methods, while requiring up to 11 times less compute, while for object detection we show a 32% computation reduction with an up to 3.4% increase in terms of mAP.

2. Related Work

Since the advent of deep learning, event-based vision has adopted many of its models. Early models, relying on shallow learning techniques such as SVMs [52] or filtering-based techniques [15, 26, 29, 40], and have gradually shifted to deeper architectures such as CNN’s [17, 35, 45, 58]. While achieving state-of-the-art performance, these types of models do not take into account the sparse and asynchronous nature of events, leading to redundant computation. This prompted the development of sparse network architectures such as SNNs, point cloud methods [49], Submanifold Sparse Convolutions [36] and graph neural networks [4, 5, 32]. Which all seek to reduce computation. While SNNs are traditionally harder to train, due to a lack of efficient learning rules, geometric learning methods such as [4, 5, 32, 36, 49] have gained popularity in recent years, since they are more suited to the asynchronous and sparse nature of events, and are easily trained and implemented thanks to the existence of well-maintained toolboxes.

In particular, graph-based methods such as [4, 5, 32, 47] show a significant reduction in computational complexity compared to dense methods that rely on standard CNNs. This is because, instead of processing events as dense image-like tensors, they only consider sparse connections between events, and confine message passing to these connections. Despite this sparsity, these methods still process events as batches and thus need to recompute all activations, whenever a new event arrives. However, each event only indicates a per-pixel change, and thus recomputing activations leads to the highly redundant computation. To counteract this, a recent line of work has focused on reusing network activations as much as possible between consecutive events, by applying efficient recursive update rules [49] and propagating these to lower layers [36].

These methods, however, do not allow for hierarchical learning [47, 49] or still rely on sparse but image-like input representations, which discard the temporal component of events. These factors either limit the scalability to more complex tasks in the case of [49], or degrade performance while incurring higher computation in the case of [36]. Most similar to our work, [47] learns on dynamic graphs, by performing learned updates each time node events are triggered. However, it also performs shallow learning, i.e. it only computes node embeddings, but does not use them for end-task learning.

In this work, we combine the advantages of graph-based methods with efficient recursive update rules, thus addressing these limitations: Asynchronous Event-based Graph Neural Networks are multi-layered, and can thus learn more complex tasks than [49], and leverage the spatio-temporal sparsity of events better than [36], leading to significant computation reduction.

3. Prerequisites

In this work, we model events as spatio-temporal graphs \( G = \{V, E\} \) with vertices \( V \) and (directed) edges \( E \). In this context, events are represented as nodes within the graph and connections are formed between neighboring events (Fig. 2 (c)). We use a graph neural network to process this graph and generate a prediction \( y \). It can be represented as a function \( f(G) = y \), which executes a set of operations on the graph level. Most common operations consist of graph convolutions and pooling steps, which operate on node features \( x_i \) attached to each node, and edge features \( e_{ij} \) attached to each edge.

**Graph Convolutions:** Graph convolutions generally consist of three distinct steps which are repeated for each node \( i \) in the graph: First the function \( \psi \) computes messages based on pairs of neighbors \( (i, j) \), where \( i \) is fixed and \( j \in \mathcal{N}(i) \) is in the neighborhood of \( i \). These messages depend on the node features at these nodes, the edge feature but also on the spatial arrangement of nodes \( i \) and \( j \). Next, all messages are aggregated through summation\(^1\), and followed by a function \( \gamma_\Theta \), which computes the new value for node \( i \). These steps are summarized in the equations below:

\[
\begin{align*}
  z_i &= \sum_{j \in \mathcal{N}(i)} \psi_\Theta(x_i, x_j, e_{ij}) \\
  \hat{x}_i &= \gamma_\Theta(x_i, z_i)
\end{align*}
\]

Both \( \psi \) and \( \gamma \) denote differentiable functions such as a multi-layer perceptron, parametrized by \( \Theta = \{\theta_\psi, \theta_\gamma\} \).

**Graph Pooling** Graph pooling operations transform a graph \( G \) to a more coarse graph \( G_c \). For an overview of the different types of graph pooling, we refer to [56]. Within this work, we will focus on cluster-based pooling methods, which aggregate the graph nodes into clusters \( C_k \) with cluster centers \( k \in \mathcal{V}_c \) which form a subset of \( \mathcal{V} \). The new features at these cluster centers are computed by aggregating features in each cluster:

\[
x_k = \max_{i \in C_k} x_i
\]

Since clustering reduces the number of nodes, the original edges need to be reconnected, and this is performed with the

\(^1\)While summation is the most common form of aggregation, any function which is symmetric in its inputs can be used, such as max and min or \( \Sigma \).
Figure 2. Overview of the processing steps in our method. The event stream (a) is first subsampled using uniform sampling (b). The subsampled events are used to generate a sparse spatio-temporal graph (c), which is processed by a graph neural network (GNN)(d), which generates a bounding-box prediction (d). Although our method works for any task, here we illustrate our method for the task of object detection. In the figure below, we show an overview of the used network architecture. It combines Graph Convolutions (here Spline Convolutions) with pooling layers, followed by a prediction head. Each graph convolution block consists of several graph convolutions followed by ELU and Batch Normalization.

By embedding these events in a spatio-temporal space $\mathbb{R}^3$ we thus can see that they are inherently sparse and asynchronous (Fig. 2 (a,b)).

For the sake of computational efficiency, we first sub-sample the events uniformly by a factor $K$ (Fig. 2 (b)). In this work, we select $K = 10$. While this preprocessing step removes events, we found that it is critical to combat over-fitting, since the network learns to consider larger contexts, focusing on more informative events. In contrast to other representations of event data such as event histograms [36] or event volumes [3, 43], the full temporal resolution of the event stream is preserved. This high temporal resolution is crucial in robotic applications like obstacle avoidance [9, 34, 48].

We use the remaining events to form an event graph $\mathcal{G}$, where each event is a node (Fig. 2 (c)). Inspired by [4] the event’s temporal position is normalized by a factor $\beta$ to map it to a similar range as the spatial coordinates. The position of each vertex is then denoted as $X_i = (x_i, y_i, t^*_i)$ with $t^*_i = \beta t_i$.

By embedding these events in a spatio-temporal space $\mathbb{R}^3$ we thus can see that they are inherently sparse and asynchronous (Fig. 2 (a,b)).

For the sake of computational efficiency, we first sub-sample the events uniformly by a factor $K$ (Fig. 2 (b)). In this work, we select $K = 10$. While this preprocessing step removes events, we found that it is critical to combat over-fitting, since the network learns to consider larger contexts, focusing on more informative events. In contrast to other representations of event data such as event histograms [36] or event volumes [3, 43], the full temporal resolution of the event stream is preserved. This high temporal resolution is crucial in robotic applications like obstacle avoidance [9, 34, 48].

We use the remaining events to form an event graph $\mathcal{G}$, where each event is a node (Fig. 2 (c)). Inspired by [4] the event’s temporal position is normalized by a factor $\beta$ to map it to a similar range as the spatial coordinates. The position of each vertex is then denoted as $X_i = (x_i, y_i, t^*_i)$ with $t^*_i = \beta t_i$.

For each pair of nodes $i$ and $j$, an edge $e_{ij}$ between them is generated if they are within spatio-temporal distance $R$, i.e. $R \geq \|X_i - X_j\|$ from each other. To reduce computation and regularize the graph, we limit the maximal number of neighborhood nodes to $D_{max}$, i.e. $|N(i)| \leq D_{max}$. Finally, we assign initial node features, $x_i = p_i$ and edge features corresponding to the relative position between the

function $\pi$:

$$E_c = \pi(E, C)$$

resulting in the final coarse graph.

Stacking these operations as layers enables rich, and high-level feature computation, making these models more powerful than the point cloud method in [49] or shallow features computed in [52].

4. Approach

Representing event data as spatio-temporal graphs allows us to efficiently process incoming events by performing sparse but complete graph updates. In the following, we show how a graph can be constructed from an event stream (Sec. 4.1), and we demonstrate how it can be used for efficient and asynchronous computations (Sec. 4.2). An overview of the full method is illustrated in Figure 2.

4.1. Graph Construction

Event cameras have independent pixels which each trigger events, whenever they perceive a brightness change. Each event encodes the pixel position $(x_i, y_i)$, time $t_i$ with microsecond level resolution and polarity (sign) $p_i \in \{-1, 1\}$ of the change. A group of event in a time window $\Delta T$, can thus be represented as an ordered list of tuples

$${e_i}_N = {e_{i,i = 1}}^N \quad \text{with} \quad e_i = (x_i, y_i, t_i, p_i)$$

By embedding these events in a spatio-temporal space $\mathbb{R}^3$ we thus can see that they are inherently sparse and asynchronous (Fig. 2 (a,b)).

For the sake of computational efficiency, we first sub-sample the events uniformly by a factor $K$ (Fig. 2 (b)). In this work, we select $K = 10$. While this preprocessing step removes events, we found that it is critical to combat over-fitting, since the network learns to consider larger contexts, focusing on more informative events. In contrast to other representations of event data such as event histograms [36] or event volumes [3, 43], the full temporal resolution of the event stream is preserved. This high temporal resolution is crucial in robotic applications like obstacle avoidance [9, 34, 48].

We use the remaining events to form an event graph $\mathcal{G}$, where each event is a node (Fig. 2 (c)). Inspired by [4] the event’s temporal position is normalized by a factor $\beta$ to map it to a similar range as the spatial coordinates. The position of each vertex is then denoted as $X_i = (x_i, y_i, t^*_i)$ with $t^*_i = \beta t_i$.

For each pair of nodes $i$ and $j$, an edge $e_{ij}$ between them is generated if they are within spatio-temporal distance $R$, i.e. $R \geq \|X_i - X_j\|$ from each other. To reduce computation and regularize the graph, we limit the maximal number of neighborhood nodes to $D_{max}$, i.e. $|N(i)| \leq D_{max}$. Finally, we assign initial node features, $x_i = p_i$ and edge features corresponding to the relative position between the
connected vertices, normalized by $R$.

4.2. Asynchronous Processing

As we slide the time window $ΔT$, new events enter this window, and old events leave the window. While traditional methods would need to recompute all activations once this happens, here we present a recursive formulation that incorporates new events with minimal computational effort.

As a new event arrives, a new node is added to the graph, together with new edges connecting this node to existing vertices. The new connections are sparse, affecting only neighboring events. In fact, in the first layer, a new event only affects the state of its 1-hop subgraph (Fig. 3, Layer 1), corresponding with the neighborhood of the new node $i'$. Therefore, activations in the next layer need to only be recomputed for this subgraph via Eq. (2).

$$
\tilde{z}_i = \sum_{j \in \mathcal{N}(i)} \psi_\theta(x_i, x_j, e_{ij})
$$

(6)

$$
\tilde{x}_i = \gamma(x_i, z_i)
$$

(7)

As deeper layers are reached, this subgraph expands, hopping one node after each layer step, until at layer $N$ the nodes in $\mathcal{H}_{N}(i')$ need to be updated. $\mathcal{H}_{N}(i')$ denotes the $N$-hop subgraph which contains all nodes $j$ such that $j$ could be reached from $i'$ using $N$ hops or fewer. We visualize this hopping behavior in Fig. 3. Instead of processing the whole graph, only this subgraph has to be processed to obtain the same resulting graph activations as Eq. 2. By iteratively applying this concept to each graph-convolution layer of a graph neural network, its forward pass can be formulated sparsely, which significantly reduces the computational effort. At each layer, the necessary computation is proportional to the number of nodes in the respective subgraph. This number is known in the graph-theory literature as neighborhood function [6], and is influenced by the average and variance of the connectivity of the graph, which together forms the index of dispersion [6].

Graph Convolutions Our sparse update rules for graph convolutions are agnostic to the choice of functions $\psi$ and $\gamma$ (Eq. 2) and are therefore applicable to arbitrary types of graph convolution. It consists of two steps: During the initialization the convolution is applied to the full graph, while the resulting graph, i.e., the vertices and edges as well as their attributes, are stored. We perform this step at the beginning and whenever the camera is stationary and mostly noise events enter the sliding window. Thereafter, in the processing step, every time a new vertex is inserted into the graph, the graph only changes locally. Therefore, a full graph update is equivalent to updating the 1-hop subgraph starting from the new vertex, by applying Eq. 2 to its 1-hop subgraph only. Thereby, the subgraph can be efficiently obtained, as the graph’s edges are known from the initialization and updated with every subsequent forward pass. The same procedure can be applied to every subsequent computational layer. Hence, the update of the $k$th layer is limited to the $k$-hop subgraph of the new vertex. These steps lead to significant computational savings, as demonstrated in Sec. 5.

Graph Pooling Similar to sparse graph convolutions, sparse graph pooling operations are composed of an initialization and a processing step. During initialization, the procedure described in Sec. 3 is applied to the dense input graph $G$, which results in the coarse output graph $G_c$. Subsequently, in the processing stage, we assign events to the respective voxels where they are triggered, connecting them with nodes in the input graph, and then perform the max operation again for that specific voxel. If a node attribute is changed, we similarly perform the max operation again at the respective voxel. Finally, the output graph $G'_c$ can be efficiently computed by applying Eqs. 3 and 4 on $G'_c$.

Other Layers Non-graph-based layers such as linear or batch normalization can be sparsely updated similarly, by storing the results of the dense update during initialization and only processing the subset of the input, which changes from the previous input, as described in [36]. However, since these layers are applied at the lowest level, most nodes need to be updated, leading to only small gains in computational efficiency.

4.3. Network Details

While the method described in Sec. 4 would allow to sparsely update any kind of graph convolution, we found that spline convolutions [13] find a balance between computational complexity and predictive accuracy. In contrast to the standard graph convolutions [28] used in [32], spline convolutions maintain spatial information in the encoding by using a B-spline-based kernel function in the positional vertex space. This means that spline convolutions also take the relative position of neighboring nodes into account, a feature which is ignored in standard GNN-based methods like [32]. We use voxel-grid-based max-pooling [51] due


5. Experiments

All experiments within this work have been conducted using the PyG library [12] in the Torch framework [41]. For training, we use the Lightning framework [10].

Implementation Details: We used Adam [27] with batch size 16 and an initial learning rate $10^{-3}$, which decreases by a factor of 10 after 20 epochs. We apply AEGNN to the tasks of object recognition and object detection.

We have analytically deduced the computational complexity of a forward pass of our model by adding up the computational complexity of each layer. A detailed derivation can be found in the supplementary material.

5.1. Object Recognition

Event-based object recognition tackles the problem of predicting an object category from the event stream and is an important application of event cameras. Due to their high dynamic range and high temporal resolution, event cameras have the potential to detect objects, that would otherwise be undetectable by frame-based methods, especially in low-light conditions, or in conditions with severe motion blur. We demonstrate that our approach is capable of solving this task very efficiently while achieving state-of-the-art recognition performance. The model is evaluated on two diverse datasets: The Neuromorphic N-Caltech101 dataset [39] contains event streams recorded with a real event camera representing 101 object categories in 8,246 event sequences, each 300 ms long, mirroring the well-known Caltech101 dataset [11] for images. The N-Cars dataset [52] has real events, assigned to either a car or the background. It has 24,029 event sequences, each being 100 ms long. For training, we use the cross-entropy loss with batch-size 64 (N-Cars) and 16 (N-Caltech101).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Representation</th>
<th>Async.</th>
<th>N-Caltech101 Accuracy ↑</th>
<th>MFLOP/ev ↓</th>
<th>N-Cars Accuracy ↑</th>
<th>MFLOP/ev ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-First [40]</td>
<td>Spike</td>
<td>✓</td>
<td>0.054</td>
<td>-</td>
<td>0.561</td>
<td>-</td>
</tr>
<tr>
<td>HOTS [29]</td>
<td>Time-Surface</td>
<td>✓</td>
<td>0.210</td>
<td>54.0</td>
<td>0.624</td>
<td>14.0</td>
</tr>
<tr>
<td>HATS [52]</td>
<td>Time-Surface</td>
<td>✓</td>
<td>0.642</td>
<td>4.3</td>
<td>0.902</td>
<td>0.03</td>
</tr>
<tr>
<td>DART [44]</td>
<td>Time-Surface</td>
<td>✓</td>
<td>0.664</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>YOLE [7]</td>
<td>Event-Histogram</td>
<td>✓</td>
<td>0.702</td>
<td>3659</td>
<td>0.927</td>
<td>328.16</td>
</tr>
<tr>
<td>EST [17]</td>
<td>Event-Histogram</td>
<td>×</td>
<td>0.817</td>
<td>4150</td>
<td>0.925</td>
<td>1050</td>
</tr>
<tr>
<td>SSC [20]</td>
<td>Event-Histogram</td>
<td>×</td>
<td>0.761</td>
<td>1621</td>
<td>0.945</td>
<td>321</td>
</tr>
<tr>
<td>AsyNet [36]</td>
<td>Event-Histogram</td>
<td>✓</td>
<td>0.745</td>
<td>202</td>
<td>0.944</td>
<td>21.5</td>
</tr>
<tr>
<td>NVS-S [32]</td>
<td>Graph</td>
<td>✓</td>
<td>0.670</td>
<td>7.8</td>
<td>0.915</td>
<td>5.2</td>
</tr>
<tr>
<td>EvS-S [32]</td>
<td>Graph</td>
<td>✓</td>
<td>0.761</td>
<td>11.5</td>
<td>0.931</td>
<td>6.1</td>
</tr>
<tr>
<td><strong>Ours</strong></td>
<td>Graph</td>
<td>✓</td>
<td>0.668</td>
<td>7.31</td>
<td><strong>0.945</strong></td>
<td><strong>0.47</strong></td>
</tr>
</tbody>
</table>

Table 1. Comparison with several asynchronous and dense methods for object recognition. Our graph-based method has the lowest computational complexity overall while achieving state-of-the-art performance. Especially, it obtains the best accuracy on N-Cars [52] with 20 times lower computational complexity, compared to the second-best asynchronous method.
Finally, we demonstrate the use of our method for early prediction (c). Although the model was trained with dense GNN, we see that our method saves most of the computation in the early and middle layers, where high feature dimensions are used. This results in a computation growth much more slowly than for other methods. We show in (b) the FLOPS saved per layer, compared to a dense CNN. By additionally using our asynchronous formulation, we further reduce this number by a factor of 30. Additionally, for the cumulative FLOPS for processing events in sequence (a), it is visible that already using a GNN reduces the number of FLOPS by a factor of 10. We demonstrate the use of our method for early prediction (c). Although the model was trained with 10,000 events, merely 2,500 events are required to achieve over 90% accuracy.

cannot be formulated asynchronously. Thus, our method is 3,000 times more efficient while achieving a similar predictive performance on N-Cars. AsyNet [36] proposes an asynchronous, sparse network based on event-histograms. Hence, it does not explicitly account for the event’s temporal component. Lastly, NVS-S and EvS-B [32] also use a graph-based event representation. In contrast to the standard graph convolutions used in EvS-B, the spline convolutions AEGNN encode spatial information. Consequently, our method is 21 times more efficient while achieving a similar accuracy, in comparison to [32].

Scalability While previously assuming a constant number of input events, in the following, we analyze the impact the number of events has on both the computational complexity and the recognition accuracy to determine the viability of our method for low-latency prediction. To do this, we compare our model’s test set accuracy on N-Cars for different numbers of events, and plot the accuracy and required cumulative computation in Figs. (a) and (c). To highlight the efficiency of our method, we also plot the required number of FLOPs for the dense GNN, the asynchronous method [36] and its dense, synchronous variant. Our proposed method outperforms [36] in terms of accuracy (Tab.1) and in terms of FLOPs (Fig. 4 (a)), showing a computation reduction by a factor of 300. The computational savings come from the comparably flat architecture and sparse graph representation. Notably, our model does not require the full event stream, that it was trained on, for a correct prediction. As demonstrated in Fig 4 (c), only 5,000 events are required to achieve state-of-the-art recognition accuracy, further improving the computational efficiency of our method. Moreover, our method takes $30 \pm 4.8$ kFLOPS/ev for 25'000 events, averaged over all sequences. The low variance indicates a high level of stability.

<table>
<thead>
<tr>
<th>Model</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNN</td>
<td>$1.72 \cdot 10^4$</td>
<td>$4.68 \cdot 10^4$</td>
<td>$7.89 \cdot 10^4$</td>
<td>$11.2 \cdot 10^4$</td>
</tr>
<tr>
<td>AEGNN (ours)</td>
<td>4.29</td>
<td>5.56</td>
<td>5.94</td>
<td>6.11</td>
</tr>
</tbody>
</table>

Table 2. Computational effort in MFLOPs per event of our sparse method compared to its dense equivalent, evaluated on NCALtech101. With a higher number of events, and thus increasing complexity of the event graph, the computational gap becomes larger.

5.2. Object Detection

Event-based object detection seeks to classify and detect object bounding boxes from an event stream and is an emerging topic in event-based vision. Especially in nighttime scenarios or when objects travel at high speeds, frame-based object detection degrades due to image degradation, caused by underexposure or severe motion blur. Event cameras by contrast do not suffer from these issues and are thus viable alternatives in these cases. We apply our framework to this task and validate our approach on two challenging datasets: the N-Caltech101 dataset [39], see Sec. 5.1, and the Gen1 dataset [8]. While N-Caltech101 contains only one bounding box per sample, it contains 101 classes, making it a difficult classification task. By contrast, Gen1 targets an automotive scenario in an urban environment with annotated pedestrians and cars. With 228,123 bounding boxes for cars and 27,658 for pedestrians, the Gen1 dataset is much larger. To avoid the well-known over-smoothing problem of GNNs [31], we adopt the same backbone as for the recognition task but use a YOLO-based object detection head [46], as illustrated in Fig. 2. Similar to [46] we use a weighted sum of class, bounding box offset and shape as well as prediction confidence losses.

Detection Performance To evaluate the performance of our model, we use the eleven-point mean average precision (mAP) [33] score as well as the computational complexity per event, as described in Sec. 5.1. We compare...
with synchronous and asynchronous state-of-the-art methods and present the results in Tab.3. Qualitative results of our object detector on N-Caltech101 and the Gen1 dataset are shown in Fig.5. We reimplement NVS-S [32], as open-source code is not available.

Our method outperforms NVS-S [32] by 7.7%, while using 21 times less computation. This is because NVS-S uses standard graph convolutions, and thus has a receptive field that is limited to their direct neighborhood, which deteriorates detection performance. Compared to RED [43], we achieve a lower accuracy but outperform the method by a significant margin: While our method uses 0.39 MFLOPs/ev, [43] uses 4712 MFLOPs/ev. This is because [43] uses a dense, synchronous recurrent network, and it is thus not capable of event-by-event processing. Finally, AsyNet [36] outperforms AEGNN on N-Caltech101 by 4.8 mAP, but we show a 3.4 mAP higher performance on Gen1. While performances are comparable, we achieve this with 520-540 times fewer MFLOPs per event.

Timing Experiments

We timed our method, implemented in Python and CUDA, on an Nvidia Quadro RTX. To construct the graph we implemented the radius search algorithm in [32] in CUDA, which takes 2 ms to generate a graph with 2,500 nodes. For processing one event in an event graph of 4,000 from N-Caltech101, the dense update requires 167ms, our sparse method 92ms. For 25,000 events, the dense GNN needs 1014ms, our sparse method 129ms, an improvement by a factor of 8. A dense CNN with the same input requires 202ms. While our method is only 1.5 times faster than a CNN, we point out here that CNNs have highly optimized implementations in the PyTorch Library [41]. However, we expect that if implemented on suitable hardware, such as FPGA or IPU [25] processors, the reported computation reduction will lead to significant reductions in latency and power consumption, as was already demonstrated in [16].

6. Conclusion

While event-based vision has made significant strides by adopting standard learning-based methods based on CNNs, these discard the spatio-temporal sparsity of events, which leads to wasteful computation. For this reason, geometric-learning approaches for event-based vision have gained in popularity. In this work, we introduced AEGNNs, which model events as evolving spatio-temporal graphs and formulate efficient update rules for each new event that restrict recomputation of network activations only to a few nodes, which are propagated to lower layers. We applied AEGNNs to the tasks of object recognition and detection. While in object recognition we achieved an up to a 11-fold reduction in computational complexity (FLOPs), for object detection we achieved an up to 32% reduction, while outperforming asynchronous methods by 3.4% mAP. We showed that this computation reduction speeds up processing latency by a factor of 8 compared to dense GNNs. We believe that, if our method is implemented on specialized hardware such as FPGA or IPUs [25], we will see additional reductions in
latex and a significant reduction in power consumption.

7. Acknowledgment

This work was supported Huawei, and as a part of NCCR Robotics, a National Centre of Competence in Research, funded by the Swiss National Science Foundation (grant number 51NF40_185543).

8. Appendix

Here we report additional information to support the main manuscript. In what follows, we will refer to figures, tables, sections, and equations from the manuscript by prepending ”M-“. We start by providing a detailed network overview in Sec.8.1. We then show a derivation for the number of FLOPS required to compute the Spline Convolution in [13] for a single node in Sec.8.2. We finally list the licenses of the datasets used in this submission in Sec. 8.3.

8.1. Network Details

We use two network architectures in this work, one for object recognition (Sec. M-5.2) and one for object detection (Sec. M-5.3). Both networks consist of convolutional blocks, each containing a SplineConv [13], defined by the number of output channels $M_{out}$ and kernel size $k$, an ELU activation function, and a batch norm. As shown in Figure M-2, max graph pooling layers after the fifth and seventh convolution, as well as skip connections after the fourth and fifth convolution, are used. Also, a fully connected layer maps the extracted feature maps to the network outputs. The recognition network has convolutions with kernel size $k = 2$ and output channels $M_{out} = (1, 8, 16, 16, 16, 32, 32, 32)$. The convolutions in the detection network have a much larger kernel size $k = 8$ and more output channels $M'_{out} = (1, 16, 32, 32, 32, 128, 128, 128).

8.2. Spline Convolutions Complexity

In this work, we make heavy use of Spline Convolutions [13]. Compared to standard GNN layers which only aggregate features over layers, Spline Convolutions also take into account the spatial arrangement of these neighbors and thus produce richer features. Here we will give a summary of spline convolutions and refer the reader to [13] for more details. Given nodes $i$ with features $f(i)$ we define convolution kernels $g_n$, with $n = 1, ..., M_{out}$ the index of the output feature. They act as

$$ (g_n * f)(i) = \frac{1}{|N(i)|} \sum_{l=1}^{M_{in}} \sum_{j \in N(i)} f_l(j) g_{n,l}(u(i,j)) $$

where $f_l(j)$ is the input feature with index $j$, $N(i)$ counts the number of neighbors of node $i$, and $u(i,j)$ are pseudo coordinates. These are defined as the normalized distance vector between nodes $i$ and $j$.

The function $g$ is expanded as

$$ g_{n,l}(u) = \sum_{p \in P} w_{p,l,n} B_p^m(u) $$

Here $P$ denotes an index set, which is a regular grid in 3 dimensions. It has two elements in each direction, resulting in $2^3 = 8$ elements (tuples). For each coordinate tuple a learnable weight $w_{p,l,n}$ is stored and multiplied by a B-Spline basis $B_p^m(u)$ in three dimensions. Each B-Spline basis is computed by forming the product of three splines as

$$ B_p^m(u) = \prod_{s=1}^{3} N_{p,s}^m(u_s), $$

i.e. one for each dimension. Here $m$ is the degree of the B-Spline, and in this work we use $m = 3$. Each function $N_{p,s}^m(u_s)$ can thus be written $N_{p,s}^m(u_s) = \sum_{j=0}^{m-1} a_j u_s^j$.

8.2.1 FLOPS Computation

Here we count the FLOPS necessary. In what follows we define $N_i = |N(i)|$ and $N_p = |P|$ and will proceed in a series of steps. To evaluate Eq. 8 we first compute $u$ for all neighbors and dimensions, resulting in:

$$ FLOPS(u(i,j)) = N_i d $$

then we compute the FLOPS to evaluate $B_p^m(u) as

$$ FLOPS(B_p^m(u)) = 2m N_i d + N_i (d - 1) $$

For the first term we make use of Horner’s method [23], which states the optimal number of additions and multiplications for a polynomial of degree $m$ as $2m$. For the second term we count the FLOPS to compute the product. Each operation needs to be repeated for each neighbor. Next we compute $g_{n,l}(u)$ as the sum of products over elements of $P$, input features and output features.

$$ FLOPS(g_{n,l}(u)) = (2N_p - 1) M_{out} M_{in} N_i $$

Finally we aggregate these terms, first over neighbors and then over input features

$$ FLOPS \left( \sum_{l=1}^{M_{in}} \sum_{j \in N(i)} f_l(j) g_{n,l}(u(i,j)) \right) = $$$$ (2N_i - 1) M_{out} M_{in} + (M_{in} - 1) M_{out} $$

Where the first term counts products and summation over neighbors, and the second counts summation over input features. Finally, we divide all output features by $N_i$, adding
additional $M_{out}$ FLOPS. We thus have
\[
C_{tot} = N_i d + 2mN_i d + N_i (d - 1) \\
+ (2N_p - 1)M_{out}M_{in}N_i \\
+ (2N_i - 1)M_{out}M_{in} \\
+ (M_{in} - 1)M_{out} + M_{out} \\
= N_i M_{out} M_{in} (1 + 2N_p) + N_i (2d + 2md - 1).
\]

### 8.3. Licenses

We use the Prophesee Gen1 dataset [8] under the “PROPHESEE GEN1 AUTOMOTIVE DETECTION DATASET LICENSE TERMS AND CONDITIONS” found at the URL https://www.prophesee.ai/2020/01/24/prophesee-gen1-automotive-detection-dataset/. N-Caltech101 [39] is used under the “Creative Commons Attribution 4.0 license” and downloaded at https://www.garrickorchard.com/datasets/n-caltech101. Finally, the N-Cars dataset [52] is also released by Prophesee under https://www.prophesee.ai/2018/03/13/dataset-n-cars/.

### References


