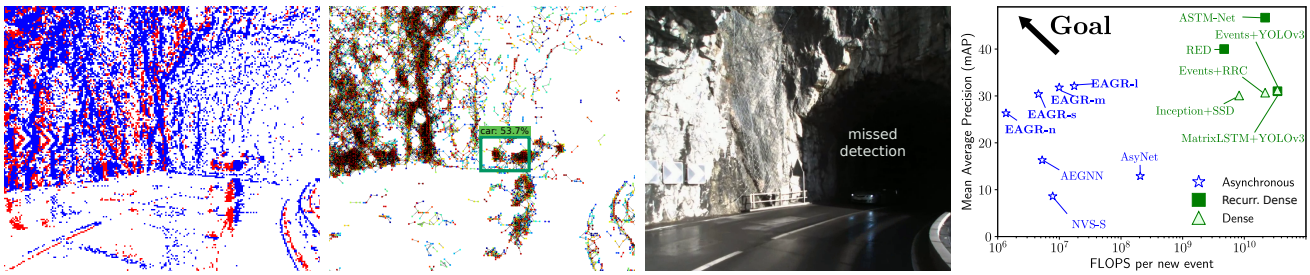
Low-latency Event-based Object detection with Asynchronous Graph Neural Networks

Daniel Gehrig and Davide Scaramuzza

Robotics and Perception Group, University of Zurich



(a) Raw events (b) Graph-based detection (c) YOLOX [14] on image (d) Efficiency vs. Performance

Figure 1. We introduce a new class of event-based object detectors, which we term **E**fficient **A**synchronous **Gr**aph Neural Networks (**EAGR**). They process events (a) as spatiotemporally evolving event graphs (b) and can be deployed in an efficient, asynchronous mode where they only perform local updates for each new event, thus significantly reducing computational complexity. EAGRs have a two times deeper architecture compared with other asynchronous GNNs [27, 45] and are more efficient in event-by-event processing (d, blue box). This opens the door to efficient, and accurate object detection in edge-case scenarios (c) (from [16]), where methods like YOLOX-s [14] based on standard images fail.

Abstract

State-of-the-art machine-learning methods for event cameras treat events as dense representations and process them with conventional deep neural networks. Thus, they fail to maintain the sparsity and asynchronous nature of event data, thereby imposing significant computation and latency constraints on downstream systems. A recent line of work tackles this issue by modeling events as spatiotemporally evolving graphs that can be efficiently and asynchronously processed using graph neural networks. These works showed impressive computation reductions, yet their accuracy is still limited by the small scale and shallow depth of their network, both of which are required to reduce computation. In this work, we break this glass ceiling by introducing several architecture choices which allow us to scale the depth and complexity of such models while maintaining low computation. On object detection tasks, our smallest model shows up to 3.7 times lower computation, while outperforming state-of-the-art asynchronous methods by 7.4 mAP. Even when scaling to larger model sizes, we are 13% more efficient than state-of-the-art while outperforming it by 11.5 mAP. As a result, our method runs 3.7 times faster than a dense graph neural network, taking only 8.4 ms per forward pass. This opens the door to efficient, and accurate object detection in edge-case scenarios.

1. Introduction

Humans can detect fast-moving objects in the blink of an eye thanks to their sophisticated visual cortex originally designed to hunt and spot prey. Today, computer vision researchers try to emulate these systems with data-driven object detection algorithms, which have found widespread application in robotic and automotive settings. However, state-of-the-art approaches operate on data from frame-based sensors like RGB cameras or LiDARs and, for this reason, suffer from a bandwidth-latency tradeoff: at high speeds, they require a high framerate to reduce perceptual latency, but this introduces a significant bandwidth overhead for downstream systems; reducing the framerate reduces the bandwidth requirements but at the cost of missing important scene dynamics, like the ones in Fig. 1, due to increased perceptual latency.

In recent years, event cameras have emerged as alternative sensors that do not suffer from this tradeoff: they are bio-inspired vision sensors that only measure *changes in intensity* that exceed a given threshold. These changes, called *events*, are recorded asynchronously and with microsecond

resolution. Due to their working principle, event cameras output sparse data with only a fraction of the bandwidth and power used by conventional RGB cameras. Also, they can adapt to the scene dynamics, featuring sub-millisecond perceptual latency at all speeds [3, 28]. For an overview of applications and methods for event-based vision see [13].

Despite this promise, state-of-the-art, event-based object detectors still do not leverage the event sparsity and instead, convert them into dense frame-like representations [5, 15, 21, 24, 26, 38, 40, 48–50]. These are then processed with deep Convolutional Neural Networks (CNNs) originally designed to work well for standard images. However, they require a significant amount of computation, most of which is redundant or operates on artificial zeros.

A recent line of work has tried to bring back efficient computation to dense methods by modeling events as spatio-temporal spike trains [7], point clouds [46], or graphs [2, 9, 27, 45], and processing them with corresponding specialized neural network architectures. Among these, graph neural networks (GNNs) have shown the highest efficiency and performance promise, leveraging insights from the fast-growing field of deep learning on graphs. The works in [27, 45] have shown that once trained, such GNNs can be deployed in an asynchronous *event-by-event* processing mode with identical output. In this mode, only local updates are performed for each new event, which are propagated to deeper layers. By limiting the computation to local subgraphs, these methods reduce the processing compared to dense methods by efficiently reusing past computations.

Despite these gains in efficiency compared to dense processing methods, graph neural networks trained on events are still behind in terms of expressiveness and accuracy. This is because current asynchronous GNNs are artificially kept shallow and lightweight to limit the per-event computation. Per-event computation still scales with the feature dimension and size of the subgraphs, which grow as the depth of the network increases. Thus, asynchronous methods hit a glass ceiling, since, as network depth increases their benefits over synchronous processing diminish.

In this work, we introduce several architecture design choices that allow us to scale the depth and capacity of GNNs significantly while maintaining highly efficient per-event processing. To maintain a low computational complexity, we first investigate effective ways to *prune node updates* caused by new events, so that they do not need to be propagated to lower layers. We find that, when combined with max pooling in early layers and node position rounding, we can skip up to 73% of the computation in lower layers with a minimal performance impact. As a next step, we show the significant benefit of performing *early temporal node aggregation*, which simultaneously boosts performance by 10.6 mAP and allows us to deploy our networks with novel and efficient Look-up-based Spline Convolutions (LUT-SCs). These LUT-SCs are trained as regular Spline Convolutions [12], but are later deployed as efficient look-up tables, which require 4.5 times less computation. Finally, we leverage directed event graphs (DEGs) at the input to boost our model's performance by 1.8 mAP with a minimal computational cost. This is because, the k-hop subgraphs of DEGS maintain a *constant* size, thereby limiting their growth in lower layers. In summary:

- We introduce architecture designs to scale the depth and capacity of asynchronous graph neural networks while maintaining highly efficient processing. We achieve this by leveraging node update pruning, early temporal aggregation through max pooling, novel efficient Look-Up-Spline Convolutions, and directed event graph processing in early layers. Cumulatively, these factors reduce the computation by a factor of 33.1, while boosting the performance by 10.6 mAP.

- We introduce several event-based object detectors, which follow these design principles, and come in four sizes: nano, small, medium, and large. Our nano model uses 3.8 times less computation while outperforming the most accurate asynchronous methods by 7.4 mAP. Simultaneously, our medium model still has a 13% lower computational complexity than the most efficient method, while outperforming it by 11.5 mAP.

- We show that our small model performs on par with dense feed-forward neural networks while outperforming state-of-the-art asynchronous methods in terms of efficiency. Finally, our large model outperforms all feed-forward dense methods, paving the way for both accurate and efficient event-based object detection.

2. Related Work

Since the introduction of powerful object detectors in classical image-based computer vision, such as R-CNN [18, 19, 43], SSD [30] and the YOLO series [14, 41, 42], event-based object detection research has focused on leveraging the available models on dense, "image-like" event representations [5, 6, 21, 24, 26, 38]. This approach allows to use pre-training, and well-established architecture designs and loss functions, while maintaining the advantages of events, such as a high-dynamic range, and negligible motion blur.

Most recent examples of such methods include RED [38] and ASTMNet [26] which operate recurrently on events and have shown high performance on detection tasks in automotive settings. However, due to the nature of their method, these approaches necessarily need to convert events into dense frames. This invariably sacrifices the efficiency and high temporal resolution present in the events, which

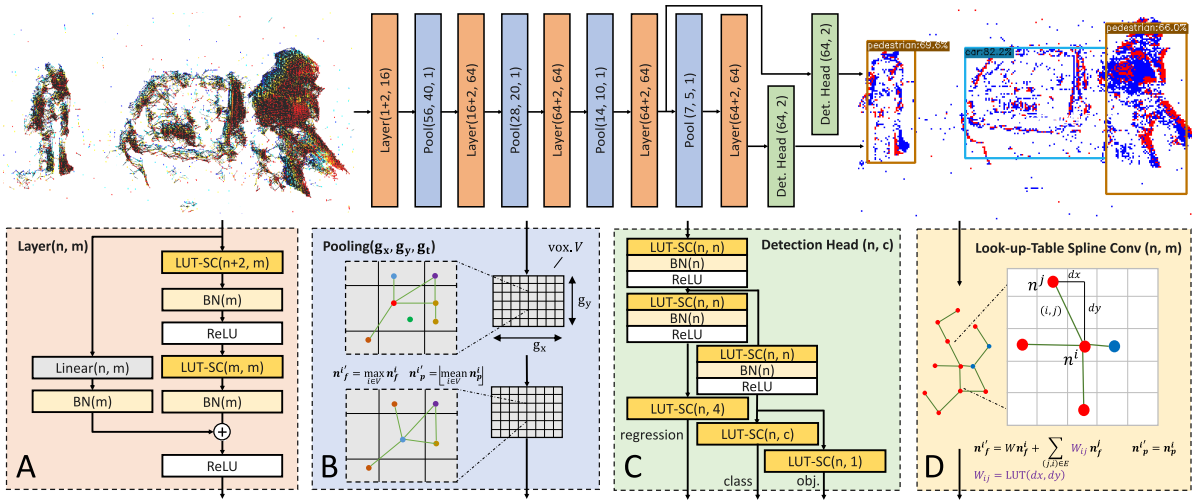


Figure 2. Overview of the network architecture of EAGR. It comprises a series of residual blocks followed by max pooling layers. For the residual blocks, the arguments n and m denote input an output channels dimension. The $+2$ indicates that we concatenate the 2D node position at that scale before processing. For pooling arguments g_x, g_y and g_t denote the number of grid cells in each dimension. Furthermore, we use a multiscale YOLOX-inspired detection head, outputting bounding boxes (*regression*), class scores and object confidence. The basic building block is the Look-up-Table Spline Convolution (LUT-SC). It uses the discrete-valued relative distance between neighboring nodes to look up a weight matrix which is uses to compute the message sent to the center node.

are critical in many application scenarios such as low-power, always-on, surveillance [22, 33], and low-latency, low-power object detection and avoidance [10, 44] .

As a result, a parallel line of research has emerged which tries to reintroduce sparsity into the present models by either adopting spiking neural network architectures [7] or geometric learning approaches [32, 45]. Of these, spiking neural networks are capable of processing raw events asynchronously and are thus closest in spirit to event-based data. However, they lack efficient learning rules and thus do not yet scale to complex tasks and datasets [1, 17, 25, 36, 37, 47]. Recently, geometric learning approaches have filled this gap. They treat events as spatio-temporal point-clouds [46], submanifolds [32] or graphs [2, 27, 34, 45], and process them with specialized neural networks. These methods retain the spatio-temporal sparsity in the events and can be implemented recursively, where single event insertions are highly efficient. Of these, processing events with graph-neural networks has proven to be most scalable, achieving high performance on complex tasks like object recognition [2, 9, 27], object detection [45] and motion segmentation [35]. Simultaneously, they can be updated efficiently and asynchronously, for each new event, by only limiting computation to locally changed subgraphs, and propagating these changes to deeper layers in the network.

However, they are still far from achieving the same level of accuracy as dense methods. Due to efficiency requirements, current asynchronous graph neural networks are limited in terms of capacity and depth of the networks [9, 27, 45]. This is because asynchronous methods become less efficient as the depth of the network increases, and their complexity still scales with the number of network parameters. In this work, we address this limitation by introducing a class of graph neural networks, which simultaneously has deep and high-capacity networks, but only has low computational complexity.

3. Method

In Sec. 3.1 we start by reviewing the generation and data structure of events, followed by a description of how to create a graph from events. Next, in Sec. 3.2 we introduce our neural network, and finally describe the proposed asynchronous operation in Sec. 3.3.

3.1. Event Generation and Graph Construction

Event cameras have independent pixels which respond asynchronously to changes in logarithmic brightness $\mathbf{L}$. Whenever the magnitude of this change exceeds the contrast threshold C, that pixel triggers an event $e_i = (\mathbf{x}_i, t_i, p_i)$ characterized by the position $\mathbf{x}_i$, timestamp t_i with microsecond resolution and polarity (sign) $p_i \in \{-1, 1\}$ of the change. An event is triggered when

$$p_i[\mathbf{L}(\mathbf{x}_i, t_i) - \mathbf{L}(\mathbf{x}_i, t_i - \Delta t_i)] > C. \quad (1)$$

The event camera thus outputs a sparse stream of events $\mathcal{E} = \{e_i\}_{i=0}^{N-1}$. As in [2, 9, 27, 35, 45], we interpret events as a 3D point cloud, connected via spatio-temporal edges.

Our event graph $\mathcal{G} = \{\mathcal{V}, E\}$ consists of nodes $\mathcal{V}$ and edges E. Each event e_i corresponds to a node. These nodes $\mathbf{n}^i \in \mathcal{V}$ are characterized by their position $\mathbf{n}_p^i = (\hat{\mathbf{x}}_i, \beta t_i) \in$

$\mathbb{R}^3$ and node features $\mathbf{n}_f^i = p_i \in \mathbb{R}$. Here $\hat{\mathbf{x}}_i$ is the event pixel coordinate, normalized by the height and width, and t_i and p_i are taken from the corresponding event. To map t_i into the same range as $\mathbf{x}_i$ we rescale it by a factor of $\beta = 10^{-6}$. These nodes are connected via edges, $(i,j) \in E$, connecting nodes $\mathbf{n}_i$ and $\mathbf{n}_j$, each with edge attributes $e_{ij} \in \mathbb{R}^{d_e}$. We connect nodes that are within a spatio-temporal distance from each other and temporally ordered

$$(i,j) \in E \quad \text{if} \quad \|\mathbf{n}_p^i - \mathbf{n}_p^j\|_\infty < R \text{ and } t_i < t_j \tag{2}$$

Here $\|.\|_\infty$ denotes the Manhattan distance, which returns the absolute value of the largest component. Constructing the graph in this way gives us several advantages: First, we can leverage the queue-based graph construction method in [27] to implement a highly parallel graph construction algorithm on GPU. Our implementation constructs full event graphs with 50'000 nodes in 1.75 ms, and inserts single events in 0.3 ms on a Quadro RTX 4000 laptop GPU. Secondly, the temporal ordering constraint above, makes the event graph directed [27, 35] which will enable high efficiency in early layers before pooling (see Sec. 3.3). In this work, we select $R = 0.01$ and limit the number of neighbors of each node to 16.

3.2. Efficient Asynchronous Graph Neural Network

An overview of our neural network architecture, which we term **E**fficient **A**synchronous **Gr**aph Neural Network, is shown in Fig. 2. It processes the spatio-temporal graphs from Sec. 3.1 (Fig. 2, top left) and outputs object detection at multiple scales (top right). It consists of five alternating residual layers (Fig. 2 A) and max pooling blocks (Fig. 2 B), followed by a YOLOX-inspired detection head at two scales (Fig. 2 C). Crucially, our network has a total of 13 convolution layers. By contrast, the methods in [27] and [45] feature only 5 and 7 layers respectively, making our network almost twice as deep as previous methods. Before each residual layer, we concatenate the x and y coordinates of the node position onto the node feature, which is indicated by +2 at the residual layer input. Residual layers and the detection head use the Look-up-Table Spline Convolutions (LUT-SC) as the basic building block (Fig. 2 B). These LUT-SC are trained as a standard Spline Convolution [12, 45] and later deployed as an efficient LUT (Sec. 3.3).

Spline Convolutions Spline Convolutions update the node features by aggregating messages from neighboring nodes:

$$\mathbf{n}_f'^i = W\mathbf{n}_f^i + \sum_{(j,i)\in E} W(e_{ij})\mathbf{n}_f^j, \quad \text{and} \quad \mathbf{n}_p'^i = \mathbf{n}_p^i \tag{3}$$

Here $\mathbf{n}_f'^i$ is the updated feature at node $\mathbf{n}_j$, $W \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}$ is a matrix that maps the current node feature $\mathbf{n}_f^i$ to the output, and $W(e_{ij}) \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}$ a matrix that maps neighboring node features $\mathbf{n}_f^j$ to the output. In [12], $W(e_{ij})$ is a matrix-valued smooth function of the edge feature e_{ij}. We set the edge feature to be $e_{ij} = |\mathbf{n}_{xy}^i - \mathbf{n}_{xy}^j|/2r + 1/2$, where $\mathbf{n}_{xy}$ denotes only the x and y component of the node position, and r is chosen such that $e_{ij} \in [0,1]^2$. The function $W(e_{ij})$ is modeled by a d-order B-Spline in $m = 2$ dimensions with $k \times k$ learnable weight matrices equally spaced in $[0,1]^2$. During the evaluation, the function interpolates between these learnable weights according to the value of e_{ij}. In this work, we choose $d = 1$ and $k = 5$.

Max Pooling Max pooling splits the input space into $g_x \times g_y \times g_t$ voxels V, and clusters nodes in the same voxel. At the output, each non-empty voxel has a node, located at the rounded mean of the input node positions, and with its feature equal to the maximum of the input nodes features.

$$\mathbf{n}_f'^i = \max_{\mathbf{n}\in V_i} \mathbf{n}_f \quad \text{and} \quad \mathbf{n}_p'^i = \frac{1}{\alpha}\left\lfloor \frac{\alpha}{|V_i|} \sum_{\mathbf{n}\in V_i} \mathbf{n}_p \right\rfloor \tag{4}$$

Here multiplying by $\alpha = [H, W, \frac{1}{\beta}]^T$ scales the mean to the original resolution. To compute the new edges, it forms a union of all edges connecting the cluster centers and removes duplicates. This operation can result in bi-directional edges between output nodes if at least one node from voxel A connected to one of voxel B and vice versa. The combination of max-pooling and position rounding has two main benefits: First, it allows the implementation of highly efficient LUT-SC convolutions, and second, it enables update pruning, which further reduces computation, discussed in Sec. 4.1. For our pooling layers we select $(g_x, g_y, g_t)_i = (56/2^i, 40/2^i, 1)$, where i is the index of the pooling layer. As seen in Sec. 4.1, selecting $g_t = 1$ is crucial to obtain high performance, since it accelerates the information mixing in the network.

Detection head Inspired by the YOLOX detection head, we design a series of (LUT-SC, BN, ReLU) blocks which progressively compute a bounding box regression $\mathbf{f}_{\text{reg}} \in \mathbb{R}^4$, class score $\mathbf{f}_{\text{cls}} \in \mathbb{R}^{n_{\text{cls}}}$ and object score $\mathbf{f}_{\text{obj}} \in \mathbb{R}$ for each output node. We then decode the bounding box location as in [14], but relative to the voxel location in which the node resides. This results in a sparse set of output detections.

3.3. Asynchronous Operation

As in [27, 32, 45], after training, we deploy our network in an asynchronous mode. The conversion to asynchronous mode happens in three steps: (i) look-up-table spline convolution caching and batch norm fusing, (ii) network activation initialization, and (iii) update propagation and pruning.

Look-up-Table Spline Convolution Caching Spline Convolutions generate the highest computational burden in our method since they involve evaluating a multi-variate, matrix-valued function, and performing a matrix-vector multiplication. Following the implementation in [12], com-

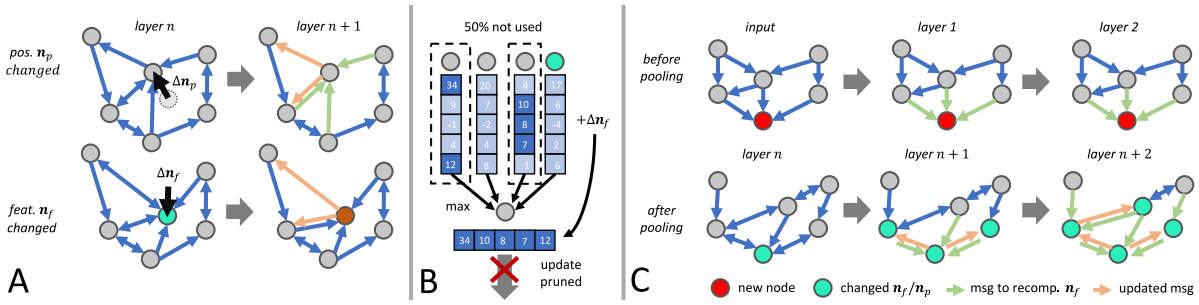


Figure 3. Overview of update propagation rules for a single new event. For convolution layers (A), we update the messages sent after a node position of feature change. If either of them change, we send update messages from the changed node (orange arrows). When the position changes, recompute messages are also sent to the changed node (green messages). In pooling layers (B), output features originating from changed input nodes are recomputed. If the changed node is in the currently unused (grayed out) set, it does not have a feature higher than the current output and it does not change the output node sufficiently to change rounding, the update is pruned. C shows the application to multiple layers. Before pooling, edges are directed, so the number of computed messages remains constant with network depth. After pooling, bidirectional edges may appear, leading to a growth in the number of computed messages in lower layers.

puting a single message between neighbors requires

$$C_{\text{msg}} = (2[d+1]^m - 1)c_{\text{in}}c_{\text{out}} + (2c_{\text{in}} - 1)c_{\text{out}}, \quad (5)$$

floating point operations (FLOPS), where the first term computes the interpolation of the weight matrix, and the second computes the matrix-vector product. Here the first term dominates due to the highly superlinear dependence on d and m. Our LUT-SC eliminates this term. We recognize that the edge attributes e_{ij}, only depend on the relative *spatial* node positions. Since events are triggered on a grid, and the distance between neighbors is bounded, these edge attributes can only take on a *finite* number of possible values. Therefore, instead of recomputing the interpolated weight at each step, we can precompute all weight matrices once and store them in a look-up table. This table stores the relative offsets of nodes together with their weight matrix. We thus replace the message propagation equation with

$$\mathbf{n}_f'^i = W\mathbf{n}_f^i + \sum_{(j,i)\in E} W_{ij}\mathbf{n}_f^j \quad (6)$$

$$W_{ij} = LUT(dx, dy), \quad (7)$$

where dx and dy are the relative 2D positions of nodes i and j. Note that this transformation reduces the complexity of our convolution operation to $C_{\text{msg}} = (2c_{\text{in}} - 1)c_{\text{out}}$ which is on the level of the classical graph convolution (GC) used in [27]. However, crucially, LUT-SC still retains the relative spatial awareness of Spline Convolutions, since W_{ij} changes with the relative position and is thus more expressive than GCs. After caching, we fuse the weights computed above with the batch norm layer immediately following each convolution, thereby eliminating its computation from the tally. After pooling, ordinarily, node positions would not have the property that they lie on a grid anymore, as their coordinates get set to the centroid location. However, since we apply position rounding, we can apply LUT-SC caching in all layers of the network.

Network Activation Initialization Before asynchronous processing, we pass a dense graph through our network and cache the intermediate activations at each layer. While in convolution layers we cache the activation, i.e., the results of sums computed from Eq. (6), in max pooling layers we cache (i) the indices of input nodes used to compute the output feature for each voxel, (ii) a list of currently occupied output voxels, and (iii) a partial sum of node positions and node counts per voxel, to efficiently update output node positions after pooling.

Update Propagation and Pruning When a new event is inserted, we recompute messages in all layers of the network. We do this to achieve an output identical to the output the network would have computed if the complete graph with one event added was processed from scratch. The propagation rules are outlined in Fig. 3.

Input layer: In the early layers of the network before pooling (C, top row), we only need to compute messages (green arrow) which are required to compute the feature of the new node in the network. Moreover, as the network depth increases, the number of messages stays constant, allowing us to stack multiple layers at the input with minimal computation increase. After pooling (C bottom row), bidirectional edges may be encountered, and thus update propagation follows the rules outlined in A and B. This is because, it may happen that for voxels V_i and V_j one edge goes from V_i and V_j and vice versa, with both edges being temporally ordered. The union operation of max pooling would then form a bidirectional edge between the output clusters. At each layer, we maintain a running list of unchanged nodes (gray) and changed nodes (cyan), and whether their position has changed, the feature has changed, or both.

Convolution Layers: In a convolution layer (A), if the node has a different position (top), we recompute that node's feature and resend a message from that node to all

2D Conv	LUT-SC	Pruning	Pos. Rounding	mAP↑	MFLOPS/ev↓
✗	✗	✗	✗	31.84	150.87
✓	✗	✗	✗	31.90	79.6
✓	✓	✗	✗	31.90	17.3
✓	✓	✓	✗	**31.90**	16.3
✓	✓	✓	✓	31.79	**4.58**

Table 1. Features affecting computational complexity.

early aggregation	multi-layer input	deep network	mAP	MFLOPS/ev
✗	✗	✗	15.8	2.02
✓	✗	✗	22.5	**1.94**
✗	✓	✓	21.2	6.27
✓	✗	✓	30.0	4.56
✓	✓	✓	**31.8**	4.58

Table 2. Features affecting accuracy.

its neighbors. These are marked as green and orange arrows in the top row of Fig. 3 (c). If instead only the node's feature changed (bottom), we only update the messages sent from that node to its neighbors. We can gain an intuition for these rules from Eq. 6. A change in the node feature $\mathbf{n}_f$ only changes one term in the sum which has to be recomputed. Instead, a node position change causes all weight matrices W_{ij} to change and thus a recomputation of the entire sum.

Pooling Layers and Pruning: During pooling, update pruning can occur. When an input node has a changed position or feature we check if (i) the changed node is currently in the set of unused nodes (grayed out in Fig. 3), (ii) the changed feature of the node does not beat the current maximum at any feature index, and (iii) its position change did not deflect the average output node position sufficiently to change rounding. If not all three conditions are met, we recompute the output feature for that node, otherwise, we prune the update and skip the computation in the lower layers. Skipping happens surprisingly often. In our case, we found that 73% of updates are skipped due to this mechanism. This also motivated us to place the max pooling layer in the early layers, since then it has the highest potential to save computation. In the next section, we will show the impact these features have on the computation of the method.

4. Experiments

Training Details In all our experiments, we use the AdamW optimizer [31] with a learning rate of 0.01 and weight decay of 10^{-5}. We train each model for 150'000 iterations with a batch size of 64. We randomly crop the events to 75% of the full resolution, and randomly translate them by up to 10% of the full resolution. We use the YOLOX loss [14], which includes an IOU loss, class loss, and a regression loss, discussed in [14]. To stabilize training, we also use exponential model averaging (EMA) [23].

Datasets We evaluate our method on the N-Caltech101 detection [36], and the Gen1 Detection Dataset [8]. N-Caltech101 consists of recording by a DAVIS240 [3] undergoing a saccadic motion in front of a projector, projecting samples of Caltech101 [11] on a wall. In post-processing, bounding boxes around the visible boxes were hand placed. The Gen1 Driving Dataset is a more challenging, large-scale detection dataset targeting an automotive setting. It was recorded with an ATIS sensor [39] with a resolution of 304×240, two classes, 228,123 annotated cars, and 27,658 annotated pedestrians. As [38] we remove bounding boxes diagonals below 30 and sides below 20 pixels from Gen1.

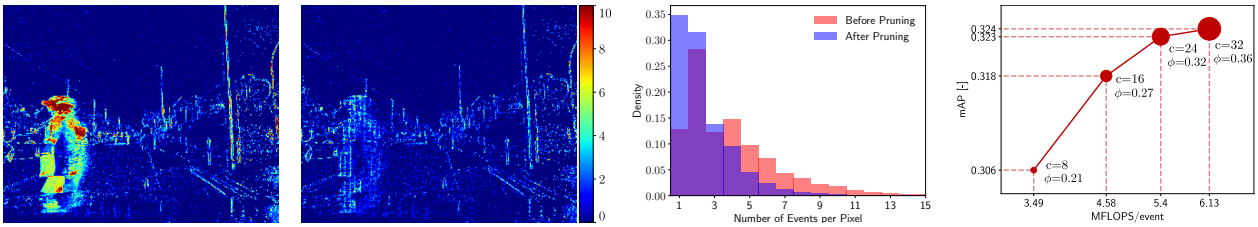
4.1. Ablation Studies

Here we motivate the use of the features discussed previously. We split our ablation studies into two parts, i.e. those targeting the efficiency (Tab. 1), or the accuracy (Tab. 2) of the method. For all experiments, we use the model illustrated in Fig. 2 as a baseline, and report means average precision (mAP) scores [29] on the validation set of Gen1 [8].

Ablations on Efficiency A key enabling factor for using 2D LUT-SCs lies in transitioning from 2D to 3D convolutions, which we investigate by training a model with 3D Spline Convolutions (row 1 in Tab. 1). This result does not yet take into account update pruning, which is discussed later. With an mAP of 31.84, it achieves a 0.05 higher mAP than our baseline (bottom row). Using 3D convolutions yields a slight improvement in accuracy since it uses more information, but does not allow us to perform an efficient lookup, yielding 150.87 MFLOPS per new event. Using 2D convolutions (row 2) reduces the computation to 79.6 MFLOPS/ev due to the dependence on the dimension d in Eq. (5), which is further reduced to 17.3 MFLOPS/ev after implementing LUT-SCs (row 3). Despite the small decrease in performance due to 2D convolutions, we gain a factor of 8.7 in terms of FLOPS per event.

Next, we investigate pruning. We recompute the FLOPS of the previous model by terminating update propagation after max-pooling layers, illustrated in Fig. 3 (c), and reported in row 4 of Tab. 1. We find that this reduces the computational complexity from 17.3 to 16.3 MFLOPS/ev. This reduction comes from removing the orange messages in Fig. 3 A (bottom). Implementing node position in Eq. 4 (row 5), allows us to fully prune updates. The final method only requires 4.56 MFLOPS/ev. Node position rounding reduces mAP only by 0.01, justifying its use.

Ablations on Accuracy We found that two features of our network had a major impact on performance: First, we applied early temporal aggregation, i.e., using $g_t = 1$, which sped up training and led to higher accuracy. We train another model which pools the temporal dimension more gradually by setting $g_t = 8/2^i$, where i is the index of the pooling layer. This model only reached an mAP of 21.2 (Tab. 2, first row), after reducing the learning rate to 0.002 to enable stable training. This highlights that early pooling plays an important role since it improves our result by 10.6 mAP. We believe that it is important for mixing features quickly so that they can be used in lower layers.



(a) events without pruning (b) events after pruning (c) num. events per pixel (d) effect of num. channels

Figure 4. Effect of update pruning due to max pooling. We interpret max pooling as a kind of event filter. In (a-b) we show an example of aggregated events before (a) and after (b) filtering. This filter acts as a saliency detector, only letting through events with "new information", and removing redundant events in high event rate regions. This results in a more uniform distribution of events (c). We can control the filter strength by modulating the number of output features, c. As seen in (d), increasing c increases both computation and mAP. However, mAP growth drastically reduces in slope after $c = 24$. The dot size is proportional to c, and ϕ measures the proportion of updates that pass through the filter. In our baseline setting with $c = 16$, we see that only 27% of updates pass the first max pooling layer.

Next, we investigate using multiple layers before the max-pooling layer. We train another model which only has a single input layer, replacing the Layer in Fig. 2 with a (LUT-SC, BN, ReLU) block. This yielded a performance of 30.0 mAP which is 1.8 mAP lower than the baseline. The computational complexity is only marginally lower, which is explained by Fig. 3 C (top). We see that adding layers at the input only generates few additional messages. This highlights the benefits of using a directed event graph. Finally, our model with fewer layers (7 instead of 13) experiences a 7.5 mAP highlighting the benefit of a deep network.

4.2. Max Pooling

In this section, we take a closer look at the pruning mechanism. We find that almost all pruning happens in the very first max pooling layer. This motivates the placement of the pooling layer at the early stages of the network, which allows us to skip most computations when pruning happens. Also, since the subgraph is still small in the early layers, it is easy to prune the entire update tree. We interpret this case as "event filtering" and investigate this filter in Fig. 4.

When applied to raw events (Fig. (a)) we obtain filtered events (Fig. (b)), i.e., events that passed through the first max pooling layer. We observe that max-pooling makes the events more uniformly distributed over the image plane. This is also supported by the density plot in Fig. 4 (c), which shows that the distribution of the number of events per pixel shifts to the left after filtering, removing events in regions where there are too many. This behavior can be explained by the pigeon-hole principle when applied to max-pooling layers. Max-pooling usually only uses a fraction of its input nodes to compute the output feature. The number of input nodes used by the max-pooling layer is upper bounded by its output channel dimension, c_{out}, since it could at maximum only use one feature from each input node. As a result, max-pooling selects at most c_{out} nodes for each voxel, resulting in more uniformly sampled events.

To study the effect of the output channel dimension on filtering, we train four models with $c_{\text{out}} \in \{8, 16, 24, 32\}$, where our baseline model had $c_{\text{out}} = 16$. We report the mAP, MFLOPS/ev, and fraction of events after filtering, ϕ averaged over Gen1, in Fig. 4 (d). As predicted, we find that increasing c_{out}, increases mAP, MFLOPS, and ϕ. However, increase happens at different rates. While MFLOPS and ϕ grow roughly linearly, mAP growth slows down signficantly after $c = 24$. Interestingly, by selecting $c_{\text{out}} = 8$ we still achieve an mAP of 30.6, while only using 21% of events. This type of filtering has interesting implications for future work. An interesting question would be whether events that are not pruned carry salient and interpretable information.

4.3. Comparison with State of the Art

Finally, we compare our method against state-of-the-art dense and asynchronous methods and report results in Tab. 3 on the N-Caltech101 and Gen1 test sets. We evaluate four versions of our model: nano (-N), small (-S), medium (-M), and large (-L). These differ in the number of features in the layer blocks 3,4 and 5 and in the detection heads, and have 32, 64, 92, and 128 channels in these layers respectively. Our baseline from before has 64. We compare against the following state-of-the-art methods:

Dense Recurrent Methods In this category, RED [38] and ASTM [26] are the state-of-the-art, and feature recurrent architectures. We also include MatrixLSTM+YOLOv3 [5] which features a recurrent, learnable representation and a YOLOv3 detection head.

Dense Feed-forward Methods The work in [26] provides results on Gen1 for the dense feed-forward methods which we term Events+RRC [6], Inception+SDD [21] and Events+YOLOv3 [24]. These use dense event representations with the RRC, SSD, or YOLOv3 detection head.

Spiking Methods We compare against the spiking network Spiking DenseNet [7], which uses an SSD detection head.

Asynchronous Methods Here we compare against state-of-the-art methods AEGNN [45] and NVS-S [27], both graph-based, AsyNet [32] which uses submanifold sparse convolutions [20] and YOLE [4], which uses an asynchronous CNN. All of these methods deploy their networks in an

Method	Async.	Gen1		N-Caltech101	
		mAP↑	MFLOPS/ev↓	mAP↑	MFLOPS/ev↓
Inception+SSD [21]	✗	30.1	>8'245*	-	-
Events+RRC [6]	✗	30.7	>21'758	-	-
MatrixLSTM+YOLOv3 [5]	✗	31.0	>34'519*	-	-
Events+YOLOv3 [24]	✗	31.2	>34'518*	-	-
RED [38]	✗	40.0	4'712	-	-
ASTM-Net [26]	✗	46.7	>21'758*	-	-
NVS-S [27]	✓	8.60	7.80	34.6	7.80
AsyNet [32]	✓	14.5	205	64.3	200
AEGNN [45]	✓	16.3	5.26	59.5	7.41
Spiking DenseNet [7]	✓	18.9	N/A	-	-
YOLE [4]	✓	-	-	39.8	3682
EAGR-N (ours)	✓	26.3	**1.36**	62.9	**2.28**
EAGR-S (ours)	✓	30.4	4.58	70.2	6.85
EAGR-M (ours)	✓	31.8	9.94	72.7	12.2
EAGR-L (ours)	✓	**32.1**	17.4	**73.2**	18.9

* lower bound from network backbone
N/A: FLOPS are undefined due to spike-based computation.

Table 3. Comparison against state of the art methods on the Gen1 detection dataset [8] and N-Caltech101 [36].

asynchronous mode during testing. Due to missing implementation details for Events+RRC [6], Inception+SDD [21] and Events+YOLOv3 [24], MatrixLSTM+YOLOv3 [5] and ASTM-Net [26] we find a lower bound on the per-event computation necessary to update their network based on the complexity of their detection backbone. While for Events+YOLOv3, and MatrixLSTM+YOLOv3 we use the DarkNet-53 backbone, for ASTM-Net and Events+RRC we use the VGG11 backbone, and for Inception+SDD the Inception v2 backbone. Since Spiking DenseNet uses spike-based computation, we do not report FLOPS since they are undefined and mark that entry with N/A.

From Tab. 3 we see first that, recurrent dense methods RED and ASTM net outperform our L model by 7.9 mAP and 14.6 mAP respectively but use significantly higher computation compared to our method (4712 vs. 1.36 for our N model). We believe that deeper networks and recurrence are two major factors that help performance in their methods. By contrast, our large model with 32.1 mAP outperforms the recurrent method MatrixLSTM [5] by 1.1 mAP and also has 120 times fewer FLOPS. We also find that our large model with an mAP of 32.1 outperforms dense feed-forward methods Events+RRC [6] (30.7), Inception+SSD [21] (30.1) and Events+YOLOv3 [24] (31.2). When compared to the spiking network [7] we find that our method has a 13.1 mAP higher score. The low performance of the SNN is expected to increase as better learning strategies become available to the community. Finally, we compare against sparse methods. We find that our small model outperforms all methods in terms of computation, with around 13% times fewer MFLOPS/ev than the runner-up AEGNN [45]. It also achieves a 14.1 mAP higher performance than AEGNN. Our smallest network, nano, is even 3.8 times more efficient while still outperforming AEGNN by 10 mAP.

On N-Caltech101, EAGR-S outperforms state-of-the-art dense and sparse methods, with 70.2 mAp, 5.9 mAP higher than the runner-up AsyNet [32]. Moreover, this model uses less computation than state-of-the-art AEGNN [45]. EAGR-L achieves the highest score with 73.2 mAP. EAGR-N achieves the lowest computation of 2.28 MFLOPS/ev, 3.25 times lower than that of AEGNN and 3.4 mAP higher.

Timing Experiments We compare the time it takes for our dense GNN to process a batch of 50'000 events averaged over Gen1, and compare it against our asynchronous implementation on a Quadro RTX 4000 laptop GPU. We found that our dense network takes 30.8 ms, while the asynchronous method requires 8.46 ms, a 3.7-fold reduction. We believe that with further optimizations, and when deployed on potentially spiking hardware, this method can reduce power and latency by additional factors.

5. Limitations and Future Work

Methods that currently outperform our event-based detector are all recurrent, a feature that was not studied in this work, but could bring benefits to our method. With it our method could overcome issues especially when few events are present. Combining this approach with additional sensors, such as LiDARs or RGB cameras can provide strong priors, which may increase its performance and reduce its complexity if shallower networks can be utilized.

6. Conclusion

Data-driven event-based methods have relied on converting events into dense image-like representations, but these still do not effectively model their asynchronous and sparse nature, which increases computation. Despite progress in reducing computation, asynchronous methods have not yet delivered high accuracy due to their shallow networks. In this work, we introduced a series of network design choices that allow us to design deeper neural networks without sacrificing complexity. We achieve this by introducing novel look-up-based convolutions, performing update pruning, and using directed event graphs in early layers which reduce the complexity of our approach by a factor of 33.1 compared to the baseline. Thus, on the Gen1 and N-Caltech101 datasets, our method achieves a 11.5 point higher mAP than state-of-the-art asynchronous methods, with higher efficiency, and it even outperforms several dense, feed-forward methods. This opens the door to efficient, and accurate object detection in edge-case scenarios.

7. Acknowledgement

This work was supported by Huawei Zurich, the Swiss National Science Foundation through the National Centre of Competence in Research (NCCR) Robotics (grant number 51NF40_185543), and the European Research Council (ERC) under grant agreement No. 864042 (AGILEFLIGHT).

References

[1] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey McKinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, Jeff Kusnitz, Michael Debole, Steve Esser, Tobi Delbruck, Myron Flickner, and Dharmendra Modha. A low power, fully event-based gesture recognition system. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 7388–7397, 2017. 3

[2] Yin Bi, Aaron Chadha, Alhabib Abbas, Eirina Bourtsoulatze, and Yiannis Andreopoulos. Graph-based object classification for neuromorphic vision sensing. *IEEE Int. Conf. Comput. Vis. (ICCV)*, 2019. 2, 3

[3] Christian Brandli, Raphael Berner, Minhao Yang, Shih-Chii Liu, and Tobi Delbruck. A 240x180 130dB 3μs latency global shutter spatiotemporal vision sensor. *IEEE JSSC*, 49(10):2333–2341, 2014. 2, 6

[4] Marco Cannici, Marco Ciccone, Andrea Romanoni, and Matteo Matteucci. Asynchronous convolutional networks for object detection in neuromorphic cameras. In *CVPRW*, 2019. 7, 8

[5] Marco Cannici, Marco Ciccone, Andrea Romanoni, and Matteo Matteucci. A differentiable recurrent surface for asynchronous event-based data. 2020. 2, 7, 8

[6] Nicholas F. Y. Chen. Pseudo-labels for supervised learning on dynamic vision sensor data, applied to object detection under ego-motion. In *CVPRW*, 2018. 2, 7, 8

[7] Loic Cordone, Benoit Miramond, and Philippe Thierion. Object detection with spiking neural networks on automotive event data. *International Joint Conference on Neural Networks (IJCNN)*, 2022. 2, 3, 7, 8

[8] Pierre de Tournemire, Davide Nitti, Etienne Perot, Davide Migliore, and Amos Sironi. A large scale event-based detection dataset for automotive. *arXiv e-prints*, abs/2001.08499, 2020. 6, 8

[9] Yongjian Deng, Hao Chen, Hai Liu, and Youfu Li. A voxel graph cnn for object classification with event cameras. 2022. 2, 3

[10] Davide Falanga, Kevin Kleber, and Davide Scaramuzza. Dynamic obstacle avoidance for quadrotors with event cameras. *Science Robotics*, 5(40):eaaz9712, 2020. 3

[11] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *CVPRW*, 2004. 6

[12] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. Splinecnn: Fast geometric deep learning with continuous b-spline kernels. *Conference of Computer Vision and Pattern Recognition (CVPR)*, 2018. 2, 4

[13] Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew Davison, Jörg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based vision: A survey. *IEEE T-PAMI.*, 2020. 2

[14] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. 2021. 1, 2, 4, 6

[15] Daniel Gehrig, Antonio Loquercio, Konstantinos G. Derpanis, and Davide Scaramuzza. End-to-end learning of representations for asynchronous event-based data. In *IEEE Int. Conf. Comput. Vis. (ICCV)*, 2019. 2

[16] Mathias Gehrig, Willem Aarents, Daniel Gehrig, and Davide Scaramuzza. Dsec: A stereo event camera dataset for driving scenarios. 1

[17] Mathias Gehrig, Sumit Bam Shrestha, Daniel Mouritzen, and Davide Scaramuzza. Event-based angular velocity regression with spiking networks. *IEEE International Conference on Robotics and Automation (ICRA)*, 2020. 3

[18] Ross Girshick. Fast R-CNN. In *IEEE Int. Conf. Comput. Vis. (ICCV)*, 2015. 2

[19] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 580–587, 2014. 2

[20] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3D semantic segmentation with submanifold sparse convolutional networks. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 9224–9232, 2018. 7

[21] Massimiliano Iacono, Stefan Weber, Arren Glover, and Chiara Bartolozzi. Towards event-driven object detection with off-the-shelf deep learning. In *IROS*, 2018. 2, 7, 8

[22] Giacomo Indiveri, Bernabe Linares-Barranco, Tara Hamilton, André van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, Johannes Schemmel, Gert Cauwenberghs, John Arthur, Kai Hynna, Fopefolu Folowosele, Sylvain SAÏGHI, Teresa Serrano-Gotarredona, Jayawan Wijekoon, Yingxue Wang, and Kwabena Boahen. Neuromorphic silicon neuron circuits. *Frontiers in Neuroscience*, 5:73, 2011. 3

[23] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. 2018. 6

[24] Zhuangyi Jian, Pengfei Xia, Kai Huang, Walter Stechele, Guang Chen, Zhenshan Bing, and Alois Knoll. Mixed frame-/event-driven fast pedestrian detection. 2019. 2, 7, 8

[25] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Front. Neurosci.*, 10:508, 2016. 3

[26] Jianing Li, Jia Li, Lin Zhu, Xijie Xiang, Tiejun Huang, and Yonghong Tian. Asynchronous spatio-temporal memory network for continuous event-based object detection. 2022. 2, 7, 8

[27] Yijin Li, Han Zhou, Bangbang Yang, Ye Zhang, Zhaopeng Cui, Hujun Bao, and Guofeng Zhang. Graph-based asynchronous event processing for rapid object recognition. In *IEEE Int. Conf. Comput. Vis. (ICCV)*, 2021. 1, 2, 3, 4, 5, 7, 8

[28] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128$\times$128 120 dB 15 μs latency asynchronous temporal contrast vision sensor. *IEEE JSSC*, 43(2):566–576, 2008. 2

[29] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference of Computer Vision (ECCV)*, pages 740–755. 2014. 6

[30] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *European Conference of Computer Vision (ECCV)*, 2016. 2

[31] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. 2019. 6

[32] Nico A. Messikommer, Daniel Gehrig, Antonio Loquercio, and Davide Scaramuzza. Event-based asynchronous sparse convolutional networks. In *European Conference of Computer Vision (ECCV)*, 2020. 3, 4, 7, 8

[33] Srinjoy Mitra, Stefano Fusi, and Giacomo Indiveri. Real-time classification of complex patterns using spike-based learning in neuromorphic vlsi. *IEEE Transactions on Biomedical Circuits and Systems*, 3(1):32–42, 2009. 3

[34] Anton Mitrokhin, Cornelia Fermuller, Chethan Parameshwara, and Yiannis Aloimonos. Event-based moving object detection and tracking. In *IROS*, 2018. 3

[35] Anton Mitrokhin, Zhiyuan Hua, Cornelia Fermuller, and Yiannis Aloimonos. Learning visual motion segmentation using event surfaces. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, June 2020. 3, 4

[36] Garrick Orchard, Ajinkya Jayawant, Gregory K. Cohen, and Nitish Thakor. Converting static image datasets to spiking neuromorphic datasets using saccades. *Front. Neurosci.*, 9:437, 2015. 3, 6, 8

[37] José A. Perez-Carrasco, Bo Zhao, Carmen Serrano, Begoña Acha, Teresa Serrano-Gotarredona, Shouchun Chen, and Bernabé Linares-Barranco. Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward ConvNets. *IEEE T-PAMI.*, 35(11):2706–2719, Nov. 2013. 3

[38] Etienne Perot, Pierre de Tournemire, Davide Nitti, Jonathan Masci, and Amos Sironi. Learning to detect objects with a 1 megapixel event camera. *Adv. Neural Inf. Process. Syst.*, 2020. 2, 6, 7, 8

[39] Christoph Posch, Daniel Matolin, and Rainer Wohlgenannt. A QVGA 143 dB dynamic range frame-free PWM image sensor with lossless pixel-level video compression and time-domain CDS. *IEEE JSSC*, 46(1):259–275, Jan. 2011. 6

[40] Henri Rebecq, Rene Ranftl, Vladlen Koltun, and Davide Scaramuzza. Events-to-video: Bringing modern computer vision to event cameras. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, 2019. 2

[41] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, 2016. 2

[42] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. 2018. 2

[43] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Adv. Neural Inf. Process. Syst.*, volume 28. Curran Associates, Inc., 2015. 2

[44] Nitin Sanket, Chethan M. Parameshwara, Chahat Singh, Ashwin Varghese Kuruttukulam, Cornelia Fermüller, Davide Scaramuzza, and Yiannis Aloimonos. Evdodgenet: Deep dynamic obstacle dodging with event cameras. pages 10651–10657, 05 2020. 3

[45] Simon Schaefer, Daniel Gehrig, and Davide Scaramuzza. AEGNN: Asynchronous event-based graph neural networks. 2022. 1, 2, 3, 4, 7, 8

[46] Yusuke Sekikawa, Kosuke Hara, and Hideo Saito. EventNet: Asynchronous recursive event processing. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, 2019. 2, 3

[47] Amos Sironi, Manuele Brambilla, Nicolas Bourdis, Xavier Lagorce, and Ryad Benosman. HATS: Histograms of averaged time surfaces for robust event-based object classification. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 1731–1740, 2018. 3

[48] Stepan Tulyakov, Francois Fleuret, Martin Kiefel, Peter Gehler, and Michael Hirsch. Learning an event sequence embedding for dense event-based deep stereo. In *IEEE Int. Conf. Comput. Vis. (ICCV)*, October 2019. 2

[49] Stepan Tulyakov, Daniel Gehrig, Stamatios Georgoulis, Julius Erbach, Mathias Gehrig, Yuanyou Li, and Davide Scaramuzza. Time lens: Event-based video frame interpolation. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, pages 16155–16164, 2021. 2

[50] Alex Zihao Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. Unsupervised event-based learning of optical flow, depth, and egomotion. In *Conference of Computer Vision and Pattern Recognition (CVPR)*, 2019. 2