Master Research Project Data Science & AI

Semester 1 2025–2026

When Events Meet Graphs: Evaluating Graph Neural Networks for event based vision with object classification

Final Report

Group Members:

Hannes Friese [i6437274]
Miel Geraats [i6260213]
Beniamin Pasak [i6328603]
Shraddha Pradeep [i6416535]
Jaime Tarazona Querol [i6308284]

Department of Advanced Computing Sciences

Supervisors:

Gaurvi Goyal
Guangzhi Tang
Enrique Hortal

January 20, 2026

Contents

1 Introduction **2**
- 1.1 Background and Motivation 2
- 1.2 Problem Statement 2
- 1.3 Research Objectives 3
- 1.4 Research Questions 3

2 Literature Review **3**
- 2.1 Event-Based Cameras 3
- 2.2 Graph Neural Networks 3
- 2.3 Object Recognition and Detection . . 4
- 2.4 Existing Implementations 4
  - 2.4.1 AEGNN: Asynchronous Event-based Graph Neural Networks 4
  - 2.4.2 EvGNN: Event-driven Graph Neural Network 5
  - 2.4.3 EGSST: Event-based Graph Spatiotemporal Sensitive Transformer 5

3 Methodology **5**
- 3.1 Dataset & Data Preparation 5
- 3.2 Neural Network Standardization . . . 6
  - 3.2.1 EvGNN and AEGNN 6
  - 3.2.2 EGSST 6
- 3.3 Benchmarking 6
  - 3.3.1 Accuracy Metrics 6
  - 3.3.2 Latency Metrics 7
  - 3.3.3 Efficiency Metrics 7
  - 3.3.4 Asynchronous Event-Based Metrics 7

4 Experiments **7**
- 4.1 Datasets and Preprocessing 7
- 4.2 Network Implementations 8
- 4.3 Training Setup 8
- 4.4 Benchmarking Metrics Implementation 8
- 4.5 Reproducibility 9

5 Results and Discussion **9**

6 Conclusion **11**

A Appendix **13**
- A.1 Hyperparameters for Reproducibility . 13
- A.2 Model training performances on N-Caltech101 13
- A.3 Unimplemented Relevant Architectures 13
  - A.3.1 HUGNet: Hemi-Spherical Update Graph Neural Network . . 13
  - A.3.2 DAGR: Dynamic Active Graph Representation 13

Abstract

Event-based cameras capture visual information asynchronously, offering low latency and high temporal resolution for real-time vision tasks such as object recognition and detection. Graph Neural Networks (GNNs) are well suited to this data, as they naturally model spatiotemporal relationships between events and support asynchronous processing. However, existing event-based GNNs are often evaluated on different datasets with inconsistent metrics, making fair comparison difficult. In this work, we present a unified benchmarking framework for evaluating event-based GNN architectures. We standardize and compare AEGNN, EvGNN, and EGSST across the NCars, N-Caltech101, and GEN1 datasets using standard accuracy, latency, efficiency, power consumption, and asynchronous performance metrics. Our results show that performance improvements largely transfer between datasets. We further highlight the importance of combined accuracy and efficiency metrics for evaluation. The code is publicly available under: https://github.com/BPasak/GNNBenchmark

1 Introduction

1.1 Background and Motivation

Conventional frame-based vision systems capture images at fixed intervals, resulting in redundant data, motion blur, and latency when observing fast-moving scenes. Event-based cameras overcome these limitations by asynchronously recording only brightness changes in each individual pixel. This enables microsecond temporal resolution, low latency, and high dynamic range, making event cameras well suited for scenarios involving rapid motion and challenging illumination, such as autonomous driving, robotics, and drone navigation.

Despite their advantages, the asynchronous and sparse nature of event streams introduces new computational challenges. Traditional Convolutional Neural Networks (CNNs), which are designed for dense, grid-based image data, are inefficient when applied to event-based data. They often require reconstructing frames or voxel representations from events, which reintroduces redundancy and increases latency, undermining the very benefits of event-driven vision. Consequently, there is a growing need for learning architectures that can efficiently capture both the spatial and temporal dependencies of asynchronous data while keeping computational costs minimized.

Recent research has explored several strategies to address these challenges, including voxel-based CNNs, spiking neural networks, and graph-based models. Among these, Graph Neural Networks (GNNs) have shown particular promise. By representing each event as a node and connecting it to its spatially and temporally neighboring events, GNNs can directly model the fine-grained dynamics of event streams. Their message-passing mechanism enables asynchronous updates, allowing new events to be integrated without recomputing the entire representation, making them ideal for high-speed, low-latency vision tasks.

Among various event-based vision applications, object recognition and object detection remain some of the most critical for real-time perception in autonomous systems and robotics. Accurate and low-latency inference forms the foundation for higher-level tasks such as tracking, motion planning, and obstacle avoidance. Yet, the use of GNNs for event-based object detection remains relatively unexplored, with many existing methods facing trade-offs between accuracy, scalability, and inference speed. This motivates the present research, which aims to develop a robust set of metrics that will allow for comprehensive analysis of the effectiveness, efficiency, and efficacy of the existing solutions, which should in turn allow for clearer and more transparent research on robust real time object recognition and detection.

1.2 Problem Statement

Although event cameras provide an efficient and low-latency alternative to conventional frame-based sensors, existing learning approaches still struggle to fully exploit their asynchronous nature for real-time object detection. Many current methods either reconstruct intermediate frames or require dense updates, leading to unnecessary computation and increased latency. Even graph-based architectures, while promising, present trade-offs between detection accuracy, scalability, and computational cost that re-

main insufficiently understood.

This project therefore investigates how GNNs can be applied and optimized for event-based object detection, with a focus on understanding how dynamic graph representations and message-passing mechanisms affect latency, efficiency, and accuracy. By analyzing and comparing recent GNN architectures designed for event-driven vision, the study aims to identify their respective strengths, limitations, and potential directions for improvement toward more reliable low-latency perception.

1.3 Research Objectives

To address the problem outlined in section 1.2, this project aims to improve the understanding and benchmarking of GNNs for event-based vision. To accomplish this, we investigate existing GNN architectures and identify their key design principles and performance characteristics. Event-to-graph representations, required and suitable for modeling asynchronous data streams in object detection tasks, will be developed. These will aid in evaluating the performance of selected GNN models on benchmark event-based datasets. In doing so, we analyze the trade-offs between computational efficiency, detection accuracy, and inference latency in said models, while outlining potential design improvements and future research directions for scalable, low-latency event-based vision systems.

1.4 Research Questions

To tackle the aforementioned research objectives, this study is guided by the following research questions:

1. Do performance improvements of the GNNs transfer between different datasets?
2. What architectural strategies can further improve the efficiency and scalability of GNNs for real-time event processing?
3. What are fair and interpretable metrics for comparing and benchmarking GNNs in event-based vision tasks?

2 Literature Review

2.1 Event-Based Cameras

Event-based cameras are different from conventional frame-based cameras in that they are not limited to updating full frames when processing new information. Instead of representing scenes in frames, they show the change in a scene through events. These events give information on how much a single pixel changes. More explicitly, each event encodes the time of capture, location of the sensor, and polarity (sign of brightness change) in a given pixel. The polarity in a pixel offers a very high dynamic range, which allows us to detect change in both high- and low contrast environments. Therefore, by only capturing the brightness change in a scene, significantly less data is processed compared to frame-driven cameras; these capture three color channels of a full scene for a set amount of frames per second. Moreover, event data has low latency and high temporal resolution, allowing the capture of data at short intervals. These advantages give event-based cameras many applications, such as in autonomous systems, where changes in a scene often have to be recorded as quickly as possible for safety purposes [1].

A major challenge in this field is the grouping of events, so they can be processed in clusters for use cases like object detection. Since events can arrive at any time, a method is required that can asynchronously process new events efficiently. Although we can use a Convolutional Neural Network (CNN) to recompute our time window of events each time a new event is registered, this method would computationally be very inefficient. A more suitable approach is to connect events in a certain time window, represented as a temporal graph. A Graph Neural Network (GNN) can then be run on this graph data, allowing us to asynchronously append incoming events to the graph without having to reapply convolution to all events. More on this will be explored in the next section.

2.2 Graph Neural Networks

A GNN is a specialized neural network optimized for processing data in graph form. Graph representations allow us to model data that can not easily be fit into traditional tabular structures such as molecular or social data, or in our case, event-based data.

This graph-structured data is internally processed in a GNN architecture using a message passing mechanism, in which all properties of a single node u are

summarized in a node feature vector h_u^0. Here, superscript *0* indicates that no message passing steps have been applied yet. The message passing process aggregates the features in the neighborhood of u and secures them in h_u^k, where k represents the size of the neighborhood we are aggregating from. h_u^0 is therefore the initial feature vector and is updated at each message passing step to incorporate more neighborhood information. The resulting feature vectors h_u^k are called node embeddings. In the network's output layer, these node embeddings represent the class assigned to each node by the algorithm.

The message passing step differs from traditional neural networks, where all data points are processed independently. In GNNs, the set of aggregated neighboring nodes expands with each new embedding layer. Thus, after k iterations, the node embeddings incorporate information from k-hop neighborhoods [2].

With both GNNs and event-based data introduced, we next explore a domain in which combining these methods can provide significant benefits.

2.3 Object Recognition and Detection

Object recognition and detection are the most standard tasks for image data. Object recognition involves classification of the image, whereas object detection involves identifying and localizing objects within visual scenes. In conventional vision systems, these tasks are traditionally performed using CNNs such as YOLO or Faster-RCNN, which operate on dense RGB frames captured at fixed frame rates. However, these frame-based methods struggle with high-speed motion and suffer from motion blur and latency limitations. By contrast, event cameras provide asynchronous brightness changes at microsecond resolution, which enables precise and low-latency perception under dynamic conditions. Among various event-based vision tasks, such as optical flow estimation, motion segmentation, and pose estimation, object detection is particularly critical for autonomous systems and robotics. This is due to the timely recognition and localization of objects, which are essential for decision-making and control. Designing models that can process asynchronous events efficiently and with minimal delay is therefore a fundamental step toward real-time perception in such systems [3].

GNNs offer a promising approach to this challenge. By representing events as nodes in a spatiotemporal graph, GNNs can capture both spatial structure and temporal dynamics without relying on dense frame reconstructions. Their message-passing mechanism allows for asynchronous updates as new events arrive, reducing redundant computation and enabling continuous low-latency inference. Next, we will explore some existing GNN architectures that employ event cameras for object detection and classification tasks.

2.4 Existing Implementations

2.4.1 AEGNN: Asynchronous Event-based Graph Neural Networks

There are multiple existing approaches that use GNNs for processing event driven data. One of the earliest and most well known implementations is AEGNN [4], which tackles this by connecting events in a graph, where connections depend on spatiotemporal dependencies. A new event is added to the graph as soon as it is detected and connected to existing nodes within its spatiotemporal radius R. Since each layer only updates 1-hop neighbors, the update covers the newly added event's entire k-hop neighborhood after k GNN layers.

This approach enables asynchronous processing, which is essential for event-driven data, where events arrive and have to be incorporated into the data model at any arbitrary time. Furthermore, the model can be trained synchronously in batches and later deployed asynchronously, making it suitable for real-time applications. The authors evaluate the model on 3 datasets: NCars, N-Caltech101 and GEN1 and for 2 tasks: object recognition and object detection where relevant. On the NCars dataset their architecture achieves 0.6 MFLOPs/event.

While AEGNN laid the foundation for asynchronous graph-based processing of event-driven data, it has some flaws. Its neighborhood update strategy can lead to extremely rapid growth in highly dynamic scenes, leading to large computational costs.

2.4.2 EvGNN: Event-driven Graph Neural Network

EvGNN extends prior event-based GNN architectures by combining the computational efficiency of HUGNet [5] (explained in the Abstract A.3.1) with the asynchronous graph construction of AEGNN, and incorporates them in object detection tasks. Its graph update strategy is similar to HUGNet, in that it uses a directed graph where node embeddings are only computed for newly added events. This saves memory, since past edges do not have to be stored, and allows EvGNN to calculate GNN layers in parallel for newly added nodes.

These improvements enable EvGNN to be significantly faster than AEGNN in node classification tasks. Using the NCars dataset, it achieves roughly 0.07 MFLOPs/event compared to 0.6 MFLOPs/event for AEGNN, whilst reaching a slightly higher accuracy of 87.8% compared to 86.7% [6].

In contrast to the architectures discussed so far, the following approaches process multiple events in small groups or windows rather than individually, which can improve computational efficiency and detection performance.

2.4.3 EGSST: Event-based Graph Spatiotemporal Sensitive Transformer

EGSST introduces a different strategy from the asynchronous methods previously mentioned. Instead of performing event-by-event updates, it buffers a fixed number of events, constructs a graph from this temporal window, prunes sparsely connected subgraphs, and then performs object detection. This design enables batching, which improves GPU throughput and detection accuracy, but leads to a higher latency.

Evaluation for EGSST was performed on the GEN1 dataset [7].

Building on this, more recent architectures, such as DAGR, explore hybrid strategies that combine asynchronous processing with frame-based context [8]. More information about the DAGR architecture can be found in Appendix A.3.2.

3 Methodology

3.1 Dataset & Data Preparation

When tasked with benchmarking anything, it is essential that the functionalities to be analyzed are compared fairly. In our case of benchmarking GNNs, this means that we need to develop an environment in which different neural network architectures are trained and evaluated using comparable data and methods. Initial research immediately presented problems on this front: almost none of the algorithms we found are trained on the same data, with many algorithms being trained on proprietary data. Training algorithms on different datasets is also non-trivial, especially since the values for training weights and hyperparameters that the algorithms used were rarely explicitly mentioned in the accompanying papers.

This highlights the main challenge we encountered while designing our benchmarking environment: standardization. Many GNN architectures have unique methods for constructing and representing event graphs. This can significantly impact performance, which addresses research question 2, investigating how these architectural differences affect GNN performance. Our framework needs to ensure that these differences are tested fairly, with consistent inputs and interfaces across models. This means that datasets must be downloaded, organized and incorporated into a structure where they can be interchanged. This interchangeability addresses research question 1, allowing us to analyze the transferability of performance improvements between different datasets. Since we require a common way to represent our data, we use the torch_geometric package [9] (also known as PyG), which is integrated within the PyTorch framework [10], providing many useful and generalizable methods and objects. Furthermore, many existing algorithms already use the torch_geometric Data class to represent graphs as data input, making it a natural foundation for our framework. Common datasets were reimplemented using the pattern employed by the torch_geometric library under the Dataset class with torch_geometric Data objects representing the groups of events considered as a graph. This representation ensures that our interface remains consistent across differ-

ent datasets. The implemented and transformable datasets can be found in Table 13.

The Caltech101 dataset comprises images spanning 101 object categories and an additional background class. Its event-based counterpart, N-Caltech101, was generated by recording the original images from an LCD monitor using an ATIS sensor mounted on a motorized pan-tilt unit [11]. Since N-Caltech101 is suitable for both classification and object detection, it serves as a contrast to native datasets such as NCars [12], which are captured in the field rather than through screen-based reproduction. While NCars focuses on single-instance classification, the GEN1 Automotive Detection dataset extends the benchmarking scope to multi-object detection [13]. Captured in diverse urban and highway environments, GEN1 provides natively event-based recordings where each sample contains multiple instances of cars and pedestrians. GEN1 utilizes bounding box annotations to support spatial localization tasks, offering a significantly higher degree of scene complexity and data density for evaluating GNN-based detection architectures.

3.2 Neural Network Standardization

3.2.1 EvGNN and AEGNN

EvGNN has a training script and a developed documentation. Furthermore, it is built on the AEGNN architecture and even has the AEGNN architecture as an option for a runnable network in their code, making it a promising improvement. AEGNN, EvGNN, and AEGNN run through EvGNN, which we refer to as Ev-AEGNN, are all implemented, trained, and tested in our benchmarking environment using a standardized interface. Our dataset preparation makes it very easy to interchangeably train and test any network on both NCars and N-Caltech101. For asynchronous processing, the available code from the respective GitHub repositories of AEGNN and EvGNN was used.

3.2.2 EGSST

The source code for EGSST is provided, along with the article [7]. It contains original implementation for most of the developed components, excluding graph preprocessing, Spatiotemporal Sensitivity Module (SSM), and a component that transforms the internal graph embedding into a dense representation used by the network in subsequent stages (further referred to as densification component). Additionally, the network's architecture is not explicitly declared within the source code, but seems to be dynamically constructed using a factory class, which is also missing.

Given that the graph preprocessing and the densification component are vital to the internal dataflow during the inference, and that these components could not be easily reproduced by simply observing the intended dataflow within the network (as it was not explicitly declared), the missing components of the network as well as the explicit declaration were reproduced as faithfully as possible based on the article.

In order to ensure that the performance of the network was not impacted by the necessary modifications, the coupling of the inference module (which was reconstructed), detection head, and evaluation modules was retained.

3.3 Benchmarking

To address research question 3, we designed a framework and a set of metrics used for benchmarking the performance of GNN architectures. These metrics can be divided into 4 distinct categories that cover our areas of interest regarding model comparability.

3.3.1 Accuracy Metrics

Accuracy metrics evaluate the quality of the predictions made by the different GNN architectures on the target tasks.

- Accuracy
- Mean Average Precision (mAP)

Given that we investigate both object recognition and object detection, we use standard metrics to assess the model correctness, which are accuracy and mAP respectively.

mAP is used as an evaluation metric for object detection, since it captures the precision-recall trade-off for given confidence thresholds and is a widely used metric across event-based detection literature. It is

Table 1: Relevant Neural Networks. Implemented dataset are marked in bold

Neural Network	Datasets Used	Source Code	Training Code
AEGNN	**N-Caltech101**, **NCars**, **GEN1**	Yes	No
EvGNN	**NCars**	Yes	Yes
EGSST	**GEN1**, 1Mpx	Yes	Yes

computed using the spatial overlap between predicted and ground-truth bounding boxes, while also evaluating and considering classification performance.

3.3.2 Latency Metrics

Latency metrics measure the time certain stages of the architectures' processing take, which allows us to evaluate the responsiveness of the models and identify potential computational bottlenecks.

- Graph Construction Latency
- Inference Latency

Graph construction latency measures how long it takes the model to construct a graph from the raw event input data, such that it is transformed into a usable representation for the GNN. Inference latency measures the time it takes the model to infer a prediction from an already constructed graph.

3.3.3 Efficiency Metrics

Efficiency metrics represent the computational costs and resources required by the networks. These metrics are relevant because they highlight the hardware requirements, scalability, and energy consumption that these models need.

- Parameter Count
- Memory Footprint
- Power Consumption

Parameter count is used as a measurement of the models' complexity, independent of the hardware used. Memory footprint measures the peak memory that is used while running the model, both for CPU and GPU. Power consumption quantifies the total amount of energy that is used when performing computations, both during training and inference.

3.3.4 Asynchronous Event-Based Metrics

Asynchronous event-based metrics are similar to the previously mentioned metrics, except they evaluate the model performance in cases where input events arrive independently and sequentially, rather than in fixed batches. These metrics are important in order to assess the difference in computational capacity for the model when processing asynchronously

- Per-Event Latency
- Accuracy under asynchronous processing
- Per-Event Power Consumption

Per-event latency evaluates how long it takes to process a single input event. It represents the model's responsiveness in real-time scenarios. Per-event accuracy measures the per-event prediction correctness of the model. This captures the model's performance on a smaller scale in sequential order. Per-event power consumption measures the energy required to process individual events.

4 Experiments

4.1 Datasets and Preprocessing

Experiments were conducted on the N-Caltech101, NCars and GEN1 datasets. All datasets were standardized to a common format, including the timestamp in seconds and the polarity $p \in \{-1, 1\}$. To prepare the samples for inference, a custom method was developed using the existing code where available. This was done for each architecture that transformed the graph in its standardized form to a form that is the same as used in its original paper or source code. For example, for AEGNN the polarities are reduced to $p \in \{0, 1\}$, and for EGSST, the connected components below a certain threshold size are removed.

4.2 Network Implementations

Experiments included the AEGNN, EvGNN and EGSST architectures, all of which were adapted to fit a standardized interface. This allows scalability of our benchmarking framework, as well as consistency in the input and output formats the models use across the different datasets. The implementation of the interface further ensures that datasets can easily be interchanged without causing any errors or requiring manual changes to the datasets or networks. When implementing these networks, several challenges arose that impacted implementation and reproducibility:

- AEGNN: Missing training script and hyperparameters, asynchronous evaluation only measured in FLOPs / event
- EvGNN: Heavily reliant on CUDA for HUGNet graph construction
- Ev-AEGNN: asynchronous processing for EvGNN not applicable for AEGNN
- EGSST: Crucial components for dataflow were missing, the network was not explicitly declared, appears to be dynamically constructed in a missing factory class, code is extremely coupled: coupling logic for training, evaluation, and postprocessing together.

On top of this, the asynchronous module in both AEGNN and EvGNN serves as a wrapper around the forward function of a trained model, changing how newly added events to a graph are processed. Here, EvGNN and AEGNN differ, meaning that for both of these networks, we had to extract the given asynchronous modules from their respective source code. Although the EvGNN repository includes a proper asynchronous evaluation script with hyperparameters present, these were both missing in the AEGNN repository.

Despite these challenges, all architectures were implemented into our benchmarking environment, with our reimplementations of the models staying true to the original designs as much as possible.

4.3 Training Setup

Each network was trained in a standardized environment using PyTorch and the torch_geometric library. The experiments were conducted on a Linux-based system with an AMD Ryzen 7 260 CPU, 16GB of RAM, and a single NVIDIA GeForce RTX 5070 Laptop GPU. Hyperparameters were kept consistent wherever possible and can be found in Table 9 in Appendix A.1, including both general preprocessing parameters used during graph construction and internal architectural parameters that are unique to specific highlighted models.

To tackle our research questions, we aim to train every model we have implemented on every available compatible dataset. This way, we can analyze whether performance improvements transfer between datasets, answering research question 1. Furthermore, any architectural strategies found that provide more efficiency or scalability can help answer research question 2.

For object recognition, all models were trained for 500 epochs using a learning rate of 4e-3 and an AdamW optimizer. For each model-dataset pair, 10 individual, independent experiments were conducted, ensuring that the results are averaged and not prone to outliers. An exception is AEGNN on NCars, where instabilities caused by bugs in dependencies resulted in conflicts with CUDA after around 150 epochs, meaning it was only run on 100 epochs.

For object detection, all models were trained for 300 epochs using a learning rate of 1e-3 and an AdamW optimizer. For each model-dataset pair, 3 individual, independent experiments were conducted, with an exception for AEGNN on GEN1 due to similar dependency instabilities.

To dynamically decrease the learning rate over the iterations, the ReduceLROnPlateau scheduler was used with a patience of 50 and a factor of 0.5.

To assess the inference and graph construction latency as well as inference power consumption, a 50 iteration test was run for each of the batch sizes of: 1, 2, 4, 8.

4.4 Benchmarking Metrics Implementation

To compute the latencies, we pass the standardized models with example samples to a function that randomly samples the specified samples to construct a batch, and proceeds to perform appropriate inference/construction for 100 iterations. We time each

Table 2: mAP values on GEN1 and N-Caltech101 datasets.

Method	Metric	GEN1	N-Caltech101
AEGNN	map@0.9	0.00%	0.00%
	map@0.5	0.00%	0.00%
	map@0.1	0.00%	1.33%
EGSST	map@0.9	0.00%	0.00%
	map@0.5	2.00%	2.00%
	map@0.1	13.33%	3.00%

inference/construction using Python's perf_counter.

To track the power consumption of the CPU and GPU, as well as memory foothold we employ a library called AIPowerMeter [14].

4.5 Reproducibility

Reproducing documented results from existing event-based GNN architectures provided several challenges, as highlighted prior in section 4.2. These issues resulted in the complete reimplementation of the AEGNN training and the development of custom evaluations for the AEGNN latency, accuracy, and power metrics. Furthermore, missing methods in the EGSST repository required complete manual implementation, and CUDA-specific hardware and conflicting library dependencies caused many further reproducibility problems.

To ensure better reproducibility of our own work, we standardized the datasets and preprocessing into a unified framework, including network interfaces and consistent hyperparameters across all models. Multiple runs were conducted with different random seeds, randomly initializing the neural networks. To account for this randomness, the standard deviation is calculated over all training runs.

Our implemented framework allows repeatable synchronous and asynchronous performance measurement in a controlled environment, ensuring that our reported results are fair, interpretable, and reproducible.

5 Results and Discussion

As seen in Figure 1, we have successfully reproduced the results of the EvGNN paper both for their original architecture and for AEGNN when it comes to synchronous inference. Additionally, EvGNN outperformed the other networks on the N-Caltech101 dataset, as can be seen in Figure 2, where it scored an accuracy of 47,6% while the original paper did not train or test on N-Caltech-101.

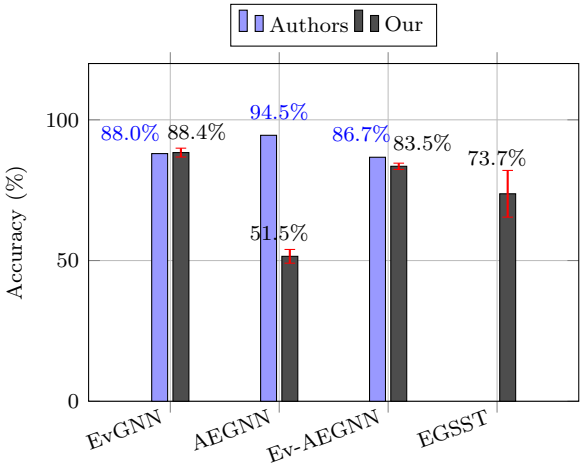

Figure 1: Accuracy comparison on NCars between authors and our reconstruction for object recognition

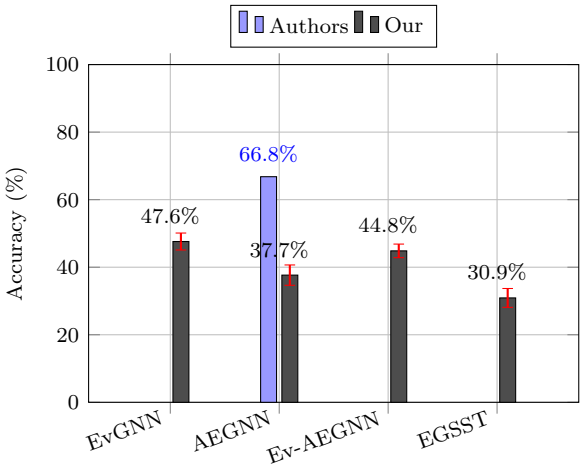

Figure 2: Accuracy comparison on N-Caltech101 dataset between authors and our reconstruction for object recognition.

Table 3: NCars: Time, Power Consumption and Memory Foothold during Training. Values are reported as mean ± std across runs.

Model	Dur (s)	CPU (J)	GPU (J)	Max VRAM (Gb)
AEGNN	*30.3 ± 8.6*	*29.1 ± 2.7*	*1214 ± 92*	*3.15 ± 0.64*
Ev-AEGNN	*139.6 ± 3.4*	*166 ± 10.4*	*2753 ± 125.6*	*4.01 ± 0.84*
EvGNN	*89 ± 1.6*	*157 ± 2.4*	*794 ± 54*	*5.84 ± 0.87*
EGSST	*914*	*457*	*2217*	*0.72*

Table 4: Model Complexity: Parameter counts for N-Cars and N-Caltech101 datasets.

Model	NCars (Params)	N-Caltech101 (Params)
AEGNN	*20,198,368*	*20,401,120*
Ev-AEGNN	*30,448*	*232,816*
EvGNN	*6,608*	*584,784*
EGSST	*3,830,062*	*3,855,505*

Unfortunately, the reproduced AEGNN seems to produce a random classifier for the NCars dataset. This would suggest that we have failed to integrate AEGNN properly into our codebase. Nevertheless, Figure 2 shows that our implementation of AEGNN is not fundamentally faulty, with a very solid accuracy of 37.7% on N-Caltech101, nearly the 44.8% of Ev-AEGNN. This suggests that the issue likely lies in how our implementation interacts with the NCars dataset, and that with more time and resources, a full reproduction of the original AEGNN is plausible.

Analyzing Figures 1, 2 and Table 2 we can also identify that EGSST works well with object recognition, having a comparable result to EvGNN. Unfortunately, we failed to obtain meaningful results for object detection.

Taking into account these results, we believe that current research into the application of GNNs for event-based vision requires significant improvements in the areas of reproducibility and openness. Most researchers in the field do not share the source code related to their work, and source code that is publicly available often lacks key parts necessary for proper reproduction. Within the 4 codebases we analyzed, EGSST is the worst offender in this sense. While AEGNN lacked training code, the model's architecture itself was clearly declared. Mournfully, EGSST lacked components that were declared by the authors as key aspects of their research (SSM and graph processing), as well as a key step within their architecture in the form of densification component.

Table 5: Asynchronicity power consumption on N-Caltech101 for 499,850 events.

Metric	CPU (J)	GPU (J)
Total Consumption	293.39	33,326.92
Average Consumption/Event	0.000587	0.06667

For each of the models, we have tracked the power consumption and memory foothold associated with the model's training, as well as recorded their parameter count. Tables 3 and 4 show a few insights into the considered architectures: EGSST is the slowest of all the architectures to train, which is expected given its high parameter count. Surprisingly, the memory foothold of EGSST is the lowest out of all the models, requiring only 0.72 Gb VRAM. Interestingly, AEGNN has 20M parameters while its training time is far below that of any other architecture. [7] also recorded that AEGNN contains over 20M parameters, which would suggest that it is not an issue within our codebase. We suspect that due to the complicated dependencies of AEGNN, there are many parameters that are unnecessarily registered by PyTorch. These parameters likely lead to the numerical instability that we experienced. Importantly, implementation by the EvGNN authors appears to have resolved this issue, decreasing the parameter count to 200,000 without significant loss in the accuracy and speed of the network, which seems to condone our belief.

In Tables 5 and 6, given that each sample is made up of 10000 events, we find that the power consumption per event is significantly smaller when doing batch processing. Interestingly, we can identify that EGSST has the lowest power consumption per sample across all of the models, suggesting that the solutions that it employs, including connected component filtering, significantly reduce the amount of unnecessary computations. The tables also show that AEGNN has a significantly higher power consumption than Ev-AEGNN. Given that the implementation of Ev-AEGNN by the EvGNN authors does not

Table 6: Power consumption on N-Caltech101 in Joules per batch of x samples.

Method	**Batch 1**		**Batch 2**		**Batch 4**		**Batch 8**	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
AEGNN	0.018	0.86	0.024	2.45	0.12	7.72	0.182	17.21
EV-AEGNN	0.0098	0.18	0.0038	0.255	0.0095	1.12	0.073	3.56
EvGNN	0.0028	0.148	0.011	0.109	0.021	0.37	0.0109	1.55
EGSST	0.0039	0.138	0.0053	0.198	0.0105	0.814	0.016	1.612

Table 7: Graph construction latency (ms).

Dataset	**Model**	**Latency (ms)**
N-Caltech101	AEGNN	*18.82 ± 3.59*
N-Caltech101	Ev-AEGNN	*20.34 ± 3.33*
N-Caltech101	EvGNN	*22.00 ± 3.52*
N-Caltech101	EGSST	*371.32 ± 49.28*
NCars	AEGNN	*6.65 ± 9.01*
NCars	Ev-AEGNN	*10.86 ± 13.69*
NCars	EvGNN	*11.69 ± 12.37*
NCars	EGSST	*129.06 ± 124.38*

Table 8: Inference latency for batch size = 8 (ms)

Dataset	**Model**	**Latency (ms)**
N-Caltech101	AEGNN	*275.04 ± 12.88*
N-Caltech101	Ev-AEGNN	*76.56 ± 8.20*
N-Caltech101	EvGNN	*40.97 ± 24.72*
N-Caltech101	EGSST	*43.53 ± 4.11*
NCars	AEGNN	*42.11 ± 14.05*
NCars	Ev-AEGNN	*23.77 ± 4.92*
NCars	EvGNN	*14.94 ± 3.54*
NCars	EGSST	*26.04 ± 3.22*

suffer from this issue, it is likely caused by the overhead due to additional parameters.

Tables 7 and 8 show that graph construction is the cause of the high latency of EGSST, since the filtering of graphs based on the size of the connected components is the main difference in the implementation of EGSST and the remaining architectures. There is a significant lack in existing solutions to effectively parallelize this process, which might be a leading issue in employment of this technique in other architectures. Nevertheless, given it has shown effectiveness in increasing the efficiency of inference, it might find significant uses in the applications where a high-speed GPU is difficult to access or where very large DNN are employed.

6 Conclusion

This study brings together multiple GNNs designed for object classification and/or detection on event driven data, with different datasets on different unified metrics. We present a benchmarking framework with a working interface, where networks, datasets, and metrics are easily interchangeable to flexibly test for performance. Ideally, this allows for a rigid comparison of networks on the same datasets with the same metrics, with users being able to integrate their own datasets, networks, or metrics to benchmark and compare.

In practice, the current state of the repository has room for future improvements. The incompleteness and lack of reproducibility of the implemented architectures' repositories caused many challenges, meaning that perfect reproduction of the results found in the AEGNN and EGSST papers was not feasible. Future work could look more closely at how documentation and code implementations could be improved and made more adoptable, to be able to fully reproduce the original results.

Nevertheless, with the data gathered, our study answers our research questions and can make the following conclusions:

Do performance improvements of the GNNs transfer between different datasets? Improve-

ments within GNNs transfer between datasets. This is especially visible in the performances of our networks on NCars compared to N-Caltech101, where a model performing better than another on one dataset generally resulted in it also performing better on the other dataset.

What architectural strategies can further improve the efficiency and scalability of GNNs for real-time event processing? Architectural strategies like EvGNN's directed or AEGNN's k-hop radius graph processing can be made to work asynchronously, which can reduce the latency for processing events, allowing for real time operation. Additionally, the filtering of connected components below a certain size might find significant uses for very large neural networks or for neural networks which are to be deployed in environments with severely limited computational capabilities.

What are fair and interpretable metrics for comparing and benchmarking GNNs in event-based vision tasks? We looked at different benchmarks to compare the performance of investigated architectures. We found that the parameter count may not be a metric that effectively informs of practical application of the model. Power consumption, combined with maximum required memory and latency, proved to be a very strong combination capable of showing unexpected strengths and weaknesses of models.

References

[1] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, *et al.*, "Event-based vision: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 1, pp. 154–180, 2020.

[2] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.

[3] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A survey of modern deep learning based object detection models," *Digital Signal Processing*, vol. 126, p. 103514, 2022.

[4] S. Schaefer, D. Gehrig, and D. Scaramuzza, "Aegnn: Asynchronous event-based graph neural networks," in *CVPR*, 2022.

[5] T. Dalgaty, T. Mesquida, D. Joubert, A. Sironi, P. Vivet, and C. Posch, "Hugnet: Hemispherical update graph neural network applied to low-latency event-based optical flow," in *CVPR*, 2023.

[6] Y. Yang, A. Kneip, and C. Frenkel, "Evgnn: An event-driven graph neural network accelerator for edge vision," *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, 2024.

[7] S. Wu, H. Sheng, H. Feng, and B. Hu, "Egsst: Event-based graph spatiotemporal sensitive transformer for object detection," *Advances in Neural Information Processing Systems*, vol. 37, pp. 120526–120548, 2024.

[8] D. Gehrig and D. Scaramuzza, "Low-latency automotive vision with event cameras," *Nature*, vol. 629, pp. 1034–1040, 2024.

[9] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[11] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, p. 437, 11 2015.

[12] A. Sironi, M. Brambilla, N. Bourdis, X. Lagorce, and R. Benosman, "HATS: Histograms of Averaged Time Surfaces for Robust Event-Based Object Classification," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018*, pp. 1731–1740, 6 2018.

[13] D. T. Pierre, D. Nitti, E. Perot, D. Migliore, and A. Sironi, "A large scale event-based detection dataset for automotive," *arXiv (Cornell University)*, 1 2020.

[14] GreenAI-Uppa, "GitHub - GreenAI-Uppa/AIPowerMeter."

A Appendix

A.1 Hyperparameters for Reproducibility

Table 9: Values for all common and model-specific hyperparameters

Networks	Hyperparameter	Value
All	inference_event_count	10000
All	beta	0.0001
All	radius	5.0
All	sampling	True
Ev-AEGNN & AEGNN	max_neighbours	32
AEGNN	kernel_size	8
AEGNN	pooling_outputs	128
EGSST	min_nodes_subgraph	1000
EGSST	ecnn_flag	True
EGSST	ti_flag	True

A.2 Model training performances on N-Caltech101

Table 10: N-Caltech101: Time, Power Consumption and Memory Foothold during Training. Values are reported as mean ± std across runs.

Model	Dur (s)	CPU (J)	GPU (J)	Max VRAM (Gb)
AEGNN	*673 ± 10*	*285 ± 7*	*31572 ± 402*	*6.8 ± 0.15*
Ev-AEGNN	*271 ± 5*	*219 ± 9*	*9068 ± 290*	*6.32 ± 0.52*
EvGNN	*148 ± 3*	*224 ± 25*	*1719 ± 102*	*5.83 ± 0.79*
EGSST	*1555*	*547*	*6238*	*1.4*

A.3 Unimplemented Relevant Architectures

A.3.1 HUGNet: Hemi-Spherical Update Graph Neural Network

HUGNet primarily differs from AEGNN when it comes to connecting new events to a graph: only the embedding of newly added nodes is computed, without updating the nodes' neighborhoods. Moreover, the aim of HUGNet is optical flow estimation rather than object detection. Its quick calculation of only newly added node embeddings results in very low latency, reduced computational cost, and lower memory usage compared to AEGNN.

However, HUGNet has some notable drawbacks: it is not designed for object detection, and its update strategy can cause problems in tasks where bidirectional temporal information is required [5]. The next architecture we will examine, EvGNN, adapts the computational efficiency of HUGNet while extending its functionality to object detection tasks.

A.3.2 DAGR: Dynamic Active Graph Representation

Recent advances in event-based perception have sought to exploit the microsecond-level temporal precision of event cameras for high-speed tasks such as object detection and motion estimation. The DAGR architecture [8] shows that event cameras can deliver low-latency, high-frequency object detection suitable for real-world automotive applications. It combines asynchronous event processing with conventional frame-based CNN features in a hybrid architecture. Its design combines graph maintenance, which involves only updating the affected subset of nodes when new nodes arrive, with a hybrid design that leverages both the dense semantic context of frames and the high temporal resolution of events. Its advanced design and computational efficiency establish DAGr as the current state-of-the-art framework for low-latency event-driven object detection.

Problems with the DAGR architecture we ran into:

Table 11: Power consumption on NCars in Joules per batch of x samples.

Method	**Batch 1**		**Batch 2**		**Batch 4**		**Batch 8**	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
AEGNN	0.0036	0.104	0.0031	0.18	0.0066	0.36	0.0067	1.1274
AEGNN-EvGNN	0.0030	0.065	0.0060	0.1654	0.00722	0.22	0.0082	0.995
EvGNN	0.0047	0.061	0.0027	0.086	0.0065	0.155	0.0074	0.20
EGSST	0.0073	0.042	0.0042	0.096	0.0059	0.15	0.0076	0.538

Table 12: Model Complexity: Parameter counts for GEN1 and N-Caltech101 datasets.

Model	**GEN1 (Params)**	**N-Caltech101 (Params)**
AEGNN	*20980704*	*30614496*
EGSST	*22283798*	*22489298*

- DAGR: CUDA files were required for the setup, problematic YOLOX installation, the architecture was very heavy on how much GPU memory it took when using limited VRAM (6GB), dimensionality issues with the preprocessing of N-Caltech101.

This lead to us not having a working DAGR version implemented.

Table 13: Datasets implemented within the framework.

Dataset	**Task**	**Native?**	**Description**
N-Caltech101	Classification and Detection	No	Neuromorphic version of 101 object classes.
NCars	Classification	Yes	Event-based recordings of cars.
GEN1	Detection	Yes	Event-based recordings of cars.