POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Graph Neural Network for Event-based Vision

Supervisors
Prof. Luciano Lavagno
Fabrizio Ottati
M. Usman Jamal
Filippo Minnella

Candidate
Daniele Busacca

December 2023

Summary

In recent years, **event cameras**, also known as silicon retinas, have emerged as a novel paradigm for capturing visual information in a sparse and asynchronous way, offering significant advantages in applications such as robotics and computer vision. These cameras differ from standard ones in how they capture visual information. Instead of sampling all pixels simultaneously, as conventional cameras do, they detect changes in brightness for each pixel with microsecond resolution. Consequently, the output from the event cameras is a continuous stream of events. These novel bio-inspired devices offer several key advantages, including low latency, low power consumption, high temporal resolution and high dynamic range. However, to exploit their full potential, the development of innovative algorithms is required. The most effective learning algorithms developed for event cameras typically use Spiking Neural Networks (SNNs) for an event-by-event processing or start by transforming events into dense representations, which are subsequently processed using conventional Convolutional Neural Networks (CNNs). Nonetheless, the SNNs don't provide a back-propagation learning mechanism and the CNNs result in the loss of both the inherent sparsity and the fine-grained temporal resolution of events imposing a substantial computational load and latency introduction. For this regard, this thesis proposes a Machine Learning (ML) algorithm based on **Graph Neural Networks** (**GNNs**) to process event data streams from event cameras. GNNs work on data with irregular shape and dimension and they can process events as spatio-temporal graphs, which are inherently sparse. The primary focus is on event classification, which involves determining the class which input data belongs to based on a model trained on a dataset of event streams. The study employs the IBM **DVS Gesture** dataset, consisting on numerous event sequences representing various hand gestures. Each event stream is associated with a label, for a total eleven distinct classes. Despite being inherently event-based, the dataset is converted into a graph-based format to ensure compatibility with Graph Neural Networks. This conversion involves a preprocessing phase, composed by several sub-steps such as event sub-stream selection, sub-sampling, time normalization and graph creation.

Part of the whole event stream coming from an hand gesture is therefore sampled, discretized in the time domain and then used to create an event-graph using the radius-neighborhood algorithm. Each of these sub-steps is characterized by one or more parameters, which can heavily affect the system performance. The primary objective of this research is therefore to classify event-graphs generated from event sub-streams. The performance of the model is given by its accuracy, evaluated based on the ratio of correctly predicted labels to the total processed data. The model is comprised of four graph convolutional layers, one pooling layer to coarse and convert the irregular structure of the graph into a predefined representation, and a Multi-Layer Perceptron, composed by three linear layers, for classification. The model's parameters are refined through a preliminary learning process which is characterized of several hyperparameters, such as the batch size, the learning rate, the optimizer, among others. The initial preprocessing phase relies on Python libraries such as Numpy, while the subsequent stages, including the GNN model architecture design, training, and evaluation, are conducted using **PyTorch** and **PyTorch Geometric** libraries.

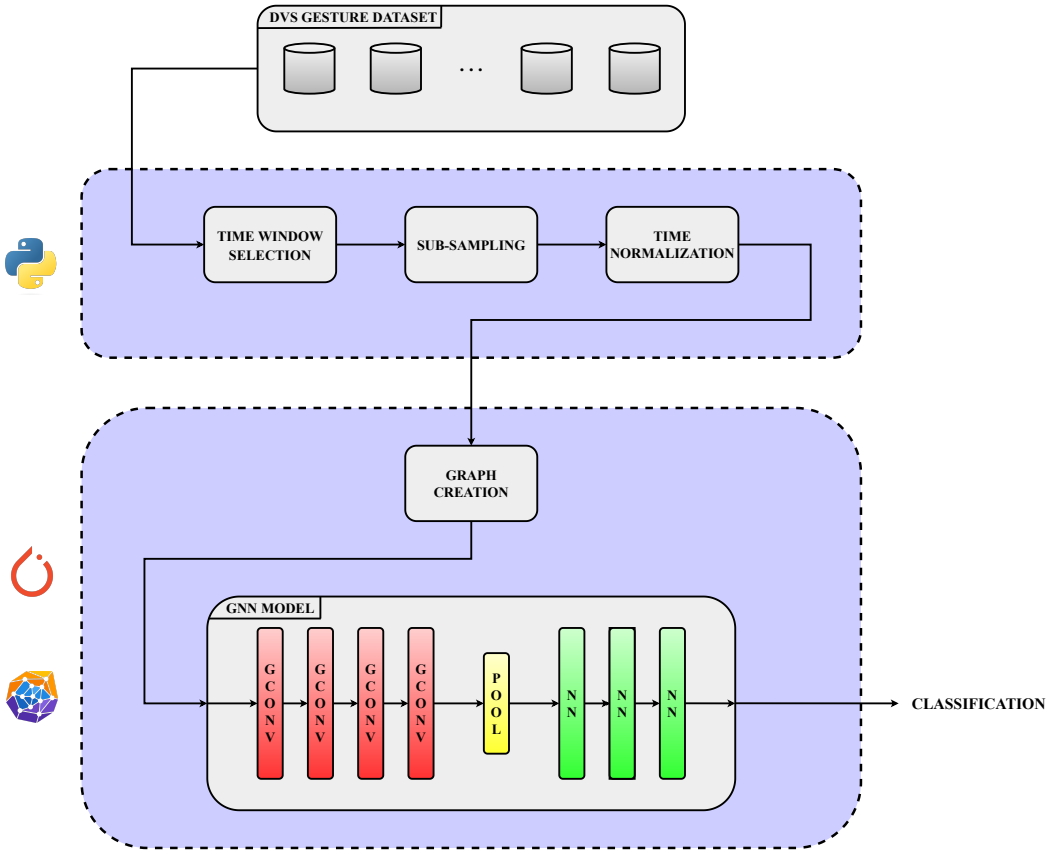

Figure 1: Overview of the thesis approach.

The study demonstrates that the model's performance, referring to the accuracy result, significantly depends on the setup configuration. This configuration includes the values of preprocessing and training hyperparameters, along with the model architecture. Consequently, the aim of the research is to fine-tune these hyperparameters to identify the setup configuration that yields the best solution. The configuration highlighted in Table 1 represents the best setup found in this thesis, showcasing an accuracy of 90.31%.

	TW	**TNR**	**LR**	**LRS**	**BS**	**OPT**	**CONV**	**#CONV**	**#LIN**	**ACC (%)**
Best Setup	1e6	32	1e-3	CAWR	16	AdamW	GATv2	4	3	**90.31**

TW: Time Window, in μs;
TNR: Time Normalization Range;
LR: Initial Learning Rate;
LRS: Learning Rate Scheduler;
BS: Batch Size;
OPT: Optimizer;
CONV: Graph Convolutional Model;
#CONV: Number of Graph Convolutional Layers;
#LIN: Number of Linear Layers;
ACC: Test Accuracy.

Table 1: Setup configuration giving the best accuracy.

In the initial two chapters, an overview of the essential theoretical foundations required for the thesis, the Event-based Vision and Graph Neural Networks, is provided. Moving forward, the third and fourth chapters delves into the details of the DVS Gesture dataset and the associated preprocessing steps needed to transform it into a graph-based representation. The fifth chapter is dedicated to presenting the model's architecture and the corresponding results obtained in the study. The conclusion, suggestions for potential improvements, and future steps are discussed in the sixth chapter. The seventh chapter, serving as the Appendix, contains additional details, including a more comprehensive table of results.

Contents

List of Tables

List of Figures

Part I

Theoretical Foundations

Chapter 1

Event Cameras

Researches on the behavior of human brain and on how we perceive reality has always been a visionary challenge. For this purpose, in 1991, in the cover of the Scientific American the image of cat was acquired by a novel "Silicon Retina", trying to mimic the behavior of the human eye. These new bio-inspired silicon retinas, mostly known as **event cameras**, have opened up new possibilities for performing computations in a more efficient and biologically inspired way, igniting a significant advancement in the field of neuromorphic engineering. This new vision paradigm has became crucial to face different tasks in robotics and computer vision where high-speed motion, low light and high-dynamic-range scenes is present, or in always-on applications where low power is needed.

1.1 Principles

Event cameras are asynchronous sensors that behave differently from standard frame-based cameras. Instead of capturing images at a synchronous rate, event cameras respond asynchronously and independently for every pixel at brightness changes.

Particularly, each pixel memorizes the logarithmic intensity of its photocurrent $L = log(I)$, that can be simply referred as *brightness*. When a change of brightness ΔL exceeds a certain threshold in a pixel location, the camera reports an event. To be more detailed, an event is triggered at pixel $\mathbf{x}_k(x_k, y_k)$ and time $\mathbf{t}_k$ when a brightness increment or decrement overcomes a threshold C.

$$|\Delta L| = |L(x_k, y_k, t_k) - L(x_k, y_k, t_k - \Delta T)| > C \tag{1.1}$$

The polarity $\mathbf{p}_k$ of the event is determined by the triggering of a positive or

negative spike: it's a binary value assuming +1 for photocurrent increasing, -1 otherwise.

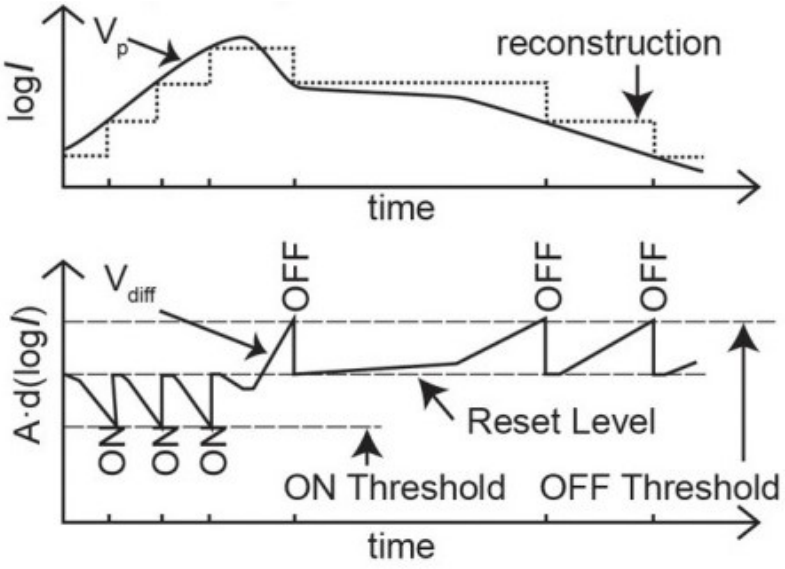

Figure 1.1: Event Camera Operation. Image taken form [1].

Therefore an event can be encoded by a tuple of four elements: the spatial location (x_k, y_k), the timestamp t_k and the polarity p_k.

$$\mathbf{e}_k = (x_k, y_k, t_k, p_k)$$

and the output of event camera is a binary stream of events:

$$\{e_k\}_N = \{x_k, y_k, t_k, p_k\}_N$$

The events are transmitted outside the camera by using an address-event representation (AER) readout protocol that can range from 2MHz to 1200MHz.

1.2 Advantages

Due to their sparse and asynchronous nature, event cameras present several benefits, especially if compared with frame-based cameras.

Thanks to their microsecond time resolution, event cameras are much faster than standard cameras. The **high temporal resolution** allows to capture the motion of very fast-moving objects without suffering from motion blur.

Also, being every pixel completely asynchronous and independent, i.e. there's no need to wait for a global exposure time, event cameras are **low latency** devices: as soon as a brightness change is detected, an event is transmitted ([1]).

Since each pixel respond only to brightness changes, there's no transmission of redundant data, leading to a low power consumption. Therefore event cameras are also **low power** devices. Generally, event-based camera systems can have power consumption ranging around 100mW.

Event cameras, just like biological retinas, can adapt to very dark and light scenarios, due to the logarithmic scale each pixel works with. The **High Dynamic Range (HDR)** can reach values of 120dB which overcomes the 60dB of standard cameras.

1.3 Different event camera designs

The **Dynamic Vision Sensor (DVS)** event camera was one of the first prototypes for this new type of bio-inspired sensors. In these devices a continuous-time photoreceptor is capacitively coupled to a readout circuit which is reset every time a pixel is sampled. Nevertheless, some applications cannot be solved by only relying on DVS events, but they also require an "absolute" brightness. To face this problem, new types of event camera has been developed in order to output both dynamic and static information. In any case, the DVS is always used as sub-module of these new vision sensors.

The pixels of the **Asynchronous Time Based Image Sensor (ATIS)** are composed by two sub-blocks. The DVS sub-pixel, here called *change detector* (*CD*), is responsible of capturing brightness changes. When this happens, another sub-pixel, called *exposure measurement* (*EM*), is triggered to read the absolute intensity. The trigger resets a capacitor to high voltage and this charge is bled away from the capacitor by another photodiode ([1]). The final result is that two more events are transmitted and they code the temporal interval between crossing the two threshold voltages. The larger the interval, the darker the absolute brightness, and vice-versa. The advantage is that it's now possible to reach HDR values greater than 120dB. On the other hand the area needed to build the ATIS sensor is now doubled with respect to the DVS cameras; also, if the reading of the absolute intensity is too long, it can be interrupted by new incoming events.

The **Dynamic and Active Pixel Vision Sensor (DAVIS)** integrates the Dynamic Active Pixel (APS) and the DVS pixel into the same pixel, by having an area overhead of only 5% with respect to the DVS camera. The APS pixel can be triggered both on demand and at constant frame; when the latter readout configuration is applied, there's redundancy if the pixel doesn't change. The only disadvantage involves the High Dynamic Range, now comparable with frame-based cameras ($\sim$55dB).

Anyway, besides more complex vision sensors has been introduced (ATIS and DAVIS), they are generally referred as DVS cameras.

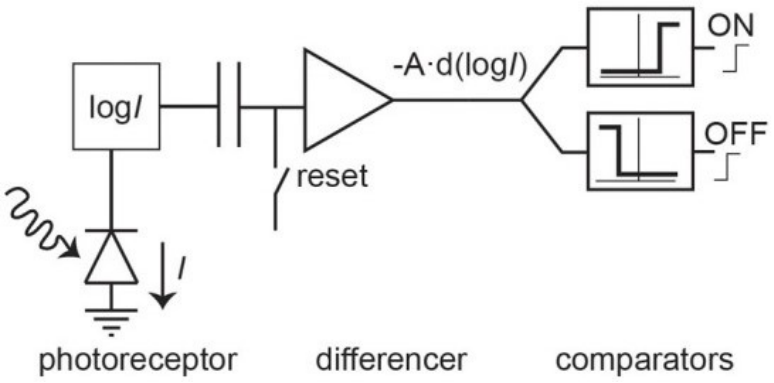

Figure 1.2: DVS Event Camera Schematic. Image taken from [1].

1.4 Event Representation

As already mentioned in section 1.1 and 1.2, the output of an event camera is an asynchronous and sparse stream of events, with very high temporal resolution and low latency. The way in which the events are processed is very important in order to exploit the temporal aspect.

Generally, two types of event representations can be distinguished, depending on how many events are processed simultaneously. The **event-by-event basis** processing can modify the state of the system with the arrival of only one event and it's the one that can achieve the lowest latency; the **groups or packets of events** processing waits for the arrival of a certain amount of events and processes them at the same time, with the introduction of some latency. In the latter, the use of more than one event let the model rely on additional information, i.e. past events or extra knowledge, instead of depending only on one event which could not provide enough information.

To facilitate the extraction of meaningful information, events can be represented in several ways.

Individual Events is a *event-by-event* processing method which takes as input a single event $e_k = (x_k, y_k, t_k, p_k)$. It's mainly used by probabilistic filters and *Spiking Neural Networks* (*SNN*): they contain additional information given by past events that, together with the incoming event, generate an output.

The **Event Packet** method aggregates a certain number N of events $\mathcal{E} = \{e_k\}_{k=1}^{N}$ in a a spatio-temporal neighborhood to produce an output. By grouping events

together, spatial and temporal relationships can be analyzed simultaneously.

Differently, in **Event Frame/Image** or **2D Histogram** events in a spatio-temporal range are shrunk into an image-like representation, where each pixel value represents the number of events or the polarity accumulation in that determined spatial location. This is compatible for classical computer vision algorithms, other than being a simple way to convert streams of events into a 2D representation, containing spatial information about scene edges and to inform about the presence and absence of events. Nevertheless, this kind of representation may discard the sparse and asynchronous nature of event cameras.

Similar to *2D Histogram*, the **Time Surface (TS)** representation also converts an event stream into an image-based structure. In this case each pixel value represents the timestamp of the last event that occurred in that spatial location. Thus, the more recent the event, the higher the "intensity" of that pixel. Only the timestamp of the last event is saved.

The **Voxel Grid** represents instead events in space-time (3D) histograms, where each voxel corresponds to a pixel within a certain time interval. This representation, preserves better the temporal information with respect to *2D Histogram*.

Events are instead represented as points embedded in a spatio-temporal space $(x, y, t) \in \mathbb{R}^3$ in **3D Point Set**: in this way the temporal dimension becomes a geometric one. Similarly, **Point Set on Image Plane** represents events as evolving 2D structures.

The **Motion-compensated event image** representation can give motion estimations. It doesn't rely only on events but on motion hypothesis too. When an edge moves and triggers an event, the motion estimation of the edge can be produced by warping the events into a reference time and by maximizing their alignment. This results in a sharper image that can be better fit a candidate motion.

Instead, a motion-invariant representation can be given by **Reconstructed Images**. Absolute brightness of pixels can be computed by integrating events over time.

These various event representations and processing techniques offer different trade-offs in terms of latency, information utilization, and compatibility with different algorithms and applications. The choice of representation depends on the specific task and requirements of the system which is working at the output of an event camera.

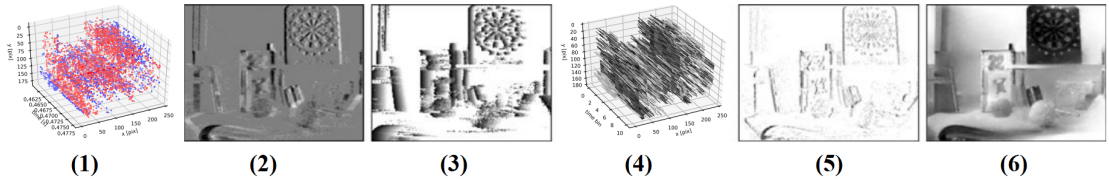

Figure 1.3: Several event representations. (1) Events in a (x,y,t) space: positive events are represented in blue and the negative ones in red. (2) *Event frame*. (3) *Time surface*: each pixel is associated with last timestamp; the darker the pixel, the more recent the time. (4) Interpolated *voxel grid*. (5) *Motion-compensated event image*. (6) *Reconstructed image*. Image taken from [1].

1.5 Event Processing

Event processing methods can mainly follow two directions, depending on the representation choice and available hardware platforms.

Event-by-event-based methods, where events need to be asynchronously processed, are used in order to obtain a minimum latency and an high computational efficiency. Therefore they need dedicate hardware to be able to satisfy these characteristics. Probabilistic filters and Spiking Neural Network are a good choice to process events in a event-by-event fashion since they naturally deal with asynchronous inputs. In any case, these methods rely on additional information to be combined with inputs to produce an output.

Anyway, single events represent only a very small part of the whole event streams and can be subject to noise. **Methods for groups of events** gather events altogether and are capable to get a better signal-to-noise ratio. They don't even require the need of additional data and, since there are several representations as described in section 1.4, many algorithms are available. *Event frames* representations, that convert event streams into image-like structures, can be used in various tasks and are well suitable to be used in many learning methods, such as DNN, SVM or Random Forest. The *time surfaces* are instead used for motion analysis and shape recognition tasks, since they can well detect scene edges and motion flow. They were also recently used as feature extractors in Convolution Neural Network for optical flow estimation. Methods based on *voxel grids* rely on 3D structures, therefore they need a greater computational effort with respect to lower dimensional representation, but at the same time the temporal information is preserved. Voxel grids are often used in Deep Neural Networks as multi-channel input/output.

1.6 Algorithms and Application

Thanks to their advantages, described in section 1.2, DVS devices can be used in many applications such as **feature detection and tracking**. The asynchronicity of event cameras allows them to see in the "blind" time between two subsequent frames of a standard camera, making them faster but also less power consuming. Tracking requires the establishment of correspondences between events (or features built from the events) at different times [1]. For complex algorithms, such as tracking cars on an highway, the shape of the object is user-defined in order to reduce the complexity of the algorithm. For simpler algorithms, a moving object is instead considered as a evolving stream of events with no predefined shape.

In other applications, like the **Optical Flow Estimation** for object velocity detection, the geometry and the motion of a object is not known a priori. In classical algorithms, the problem is faced by analysing consecutive frames but this is not naturally feasible with event cameras. A possible approach can be that one to preprocess events to transform them into image-like representation to be deployed to classical image-based algorithms. Other approaches use instead learning-based methods to predict optical flow estimation by directly using events.

Dynamic Vision Sensors are also applied for **3D Reconstruction**, but the problem of depth estimation can be approached in several ways. *Instantaneous stereo* involves the use of two or more attached and synchronous cameras and the depth estimation is performed in a very narrow amount of time by trying to find a correlation between events. Differently, the *Monocular Depth Estimation* uses a single, moving camera whose events are integrated over time to obtain the 3D map of the scene.

Another application consists on the **Image Reconstruction**. The event stream of the DVS cameras can be seen as way to compress the visual information of a scene since there's removal of redundancy for pixels which don't change their brightness. Therefore, in order to reconstruct (or "decompress") the visual data and to get an absolute brightness of every location in the camera array, a pixel-per-pixel integration is performed: if there's a positive event then the absolute brightness increases, and decreases for a negative event. The reconstructed images can capture high speed motion (the frame rate can range from 2kHz and 5kHz) and HDR scenes, resulting advantageous in many applications.

Chapter 2

Deep Learning for Graphs

In recent years, the advent of *Artificial Intelligence* (*AI*) has brought about significant changes across various disciplines. However, the term "AI" actually covers a wide range of sub-classes or paradigms. One of the most interesting and impactful sub-classes is the **deep learning**, which specifically focuses on training **deep neural networks** (**DNN**). DNNs have a complex structure and are trained on large amount of data to approximate highly intricate functions, enabling accurate predictions or classifications. While DNNs are commonly applied to fixed and grid-like data representations, such as in *Convolutional Neural Networks* (*CNNs*), there have been advancements in learning methods that can handle data with irregular shapes, which can be represented as graphs. This particular branch is known as **Graph Neural Networks** (**GNNs**) or **geometric deep learning**. GNNs have shown promise in various applications, including social network analysis, recommendation systems, molecular chemistry, and computer vision tasks.

2.1 Deep Learning

Learning algorithms are based on networks composed by a certain number of *parameters* (or *weights*) that need to be adjusted and optimized in order to complete a determined task. The powerful characteristics of deep learning is that at the beginning of the training procedure, the model is general, even if has to be specialized in a specific task. In fact, it can be trained to become highly specialized through the iterative optimization of their parameters using input samples. This adaptability and capacity to learn complex representations make deep learning models powerful tools in various fields, including computer vision, natural language processing, and speech recognition.

2.1.1 Principles

To be more detailed, the training process is divided is several steps. First, a dataset of input data and their corresponding desired outputs (*ground truth*) is needed. For example, in a supervised learning task, the input data could be images, and the desired outputs could be their respective labels. Also, the model has learnable parameters called weights: initially, these weights are randomly initialized or set to some predefined values. Then the input data are fed into the model which performs a **forward pass** through its layers. Each layer applies transformations to the input data using the current weights, producing an output. A *loss function* is used to measure the error between the model's output and the desired output. This error represents how well the model is currently performing on the given input data. Then, the gradient of the error with respect to the weights is computed using the chain rule for the derivative of a composite function. This gradient tells us how the error changes with a unit change in each weight. It is calculated by propagating the error gradients in a backward way through the layers of the model. This step is known as **backward pass**. The weights are therefore updated in the direction that minimizes the error. This is done by taking small steps proportional to the negative gradient of the error with respect to the weights. This process is repeated until the loss value decreases to a certain threshold or the accuracy on unseen data overcomes an acceptable level.

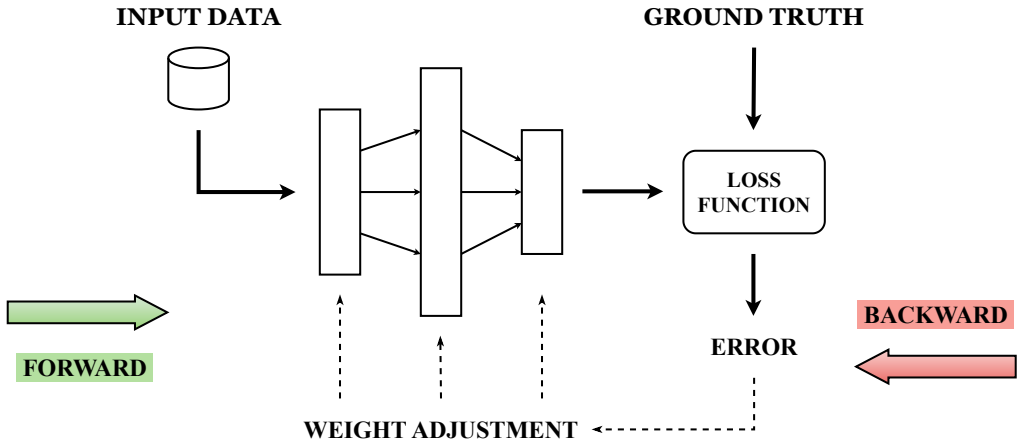

Figure 2.1: Learning process.

This method is iteratively applied for each training sample of the reference dataset. The training iteration in which the parameters are updated for all of the input data is known as an *epoch*. Generally, the training of a neural network is

performed for a certain number of epochs in the *training loop*.

Loss function

The *loss function*, also known as *cost function*, is a function that computes the difference between the output of the model and the ground truth when feeding samples into the neural network. The measured error is a positive value also known as *loss value*. In essence, a loss function serves as a mechanism for prioritizing the errors to address within the training samples. It guides the parameter updates in such a way that the adjustments mainly focus on the outputs with the higher losses. As example, common loss functions include mean squared error or cross-entropy.

Optimization function

The main idea consists on calculating the rate of change of the loss with respect to each parameter, which involves computing the derivatives. Subsequently, the goal is to adjust each parameter in such a way that it moves in the direction of decreasing loss. By iteratively updating the parameters based on their corresponding derivatives, the model aims to minimize the loss value and enhance its performance over time. Thus the concept is to navigate the parameter space towards configurations that result in lower loss values, thereby improving the overall effectiveness of the model. If the change is negative, then we need to increase that particular parameter, and decrease otherwise. The rate of change is actually scaled by a factor known as **learning rate**. The next equation shows the updating parameter function where w represents the weight to be updated, l is the learning rate and L denotes the loss value.

$$w \leftarrow w - l \cdot \frac{\partial L}{\partial w} \tag{2.1}$$

In general, multiple optimization algorithms are available, and each algorithm takes as input a list of parameters that require optimization. These optimization algorithms employ various techniques and strategies to iteratively update the parameter values in order to minimize the loss function. The choice of optimization algorithm depends on factors such as the nature of the problem, the size of the dataset, and computational considerations. Some commonly used optimization algorithms include gradient descent, stochastic gradient descent (SGD), Adam, AdamW and many others. Each algorithm offers different advantages and trade-offs in terms of convergence speed, robustness, memory requirements, and handling of noisy or sparse data.

Train, Validation and Test Sets

To ensure the model's generalization to unseen data during the training process, in addition to the training set, used for adjusting the model parameters through the backward pass, a separate validation set is utilized. This is the concept of *cross-validation* which involves setting aside a portion of the available samples to validate the model's performance on independent data. This approach helps to determine how well the model performs on unseen examples. In fact, while a neural network has the potential to approximate functions of various shapes, training the network using the entire dataset doesn't guarantee good performance on unseen data, leading to the risk of encountering overfitting. Therefore, by employing a separate validation set, the model's performance can be evaluated on independent data, helping to mitigate overfitting and giving an unbiased evaluation.

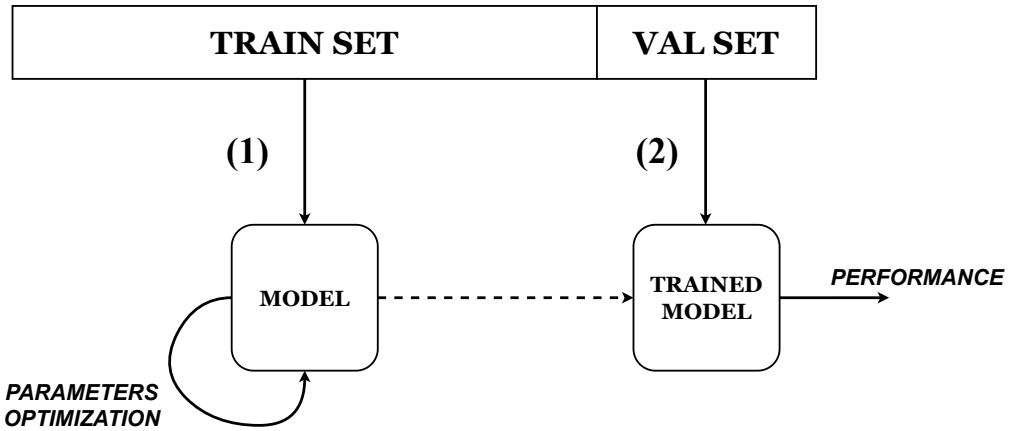

Figure 2.2: The dataset is splitted in two parts. (1) The train set is used for parameter optimization, (2) the validation set is then deployed to give performance evaluation.

In addition to the train and validation sets, a test set can also be utilized. The primary distinction between the validation and test sets lies in their purpose. The validation set is mainly used to fine-tune the hyperparameters of the network, indirectly influencing the model. On the other hand, the test set is employed solely for a golden evaluation.

2.1.2 Neural Networks

The main structure of deep learning algorithms are **neural networks** (**NNs**). These are mathematical entities able to represent complex functions by composing simpler operations.

Neuron

The basic building block of neural networks is a simple mathematical function referred as **neuron**. A neuron takes an input x, applies a linear transformation by multiplying the input by a **weight** w and adding a **bias** b and then passes the result through a non-linear **activation function** σ to produce the output y.

$$y = \sigma(w \cdot x + b) \tag{2.2}$$

The weight w and the bias b are learnable parameters.

The activation function introduces non-linearity to the neuron, enabling it to learn complex relationships between inputs and outputs, instead of applying only a linear transformation.

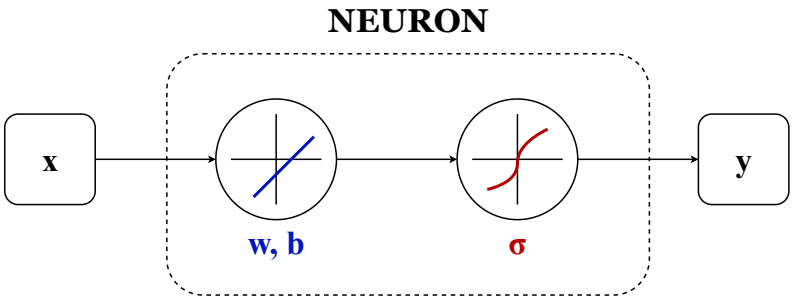

Figure 2.3: Neuron architecture.

When multiple neurons are stacked together, forming a layer, the weights and biases become multidimensional, allowing the layer to represent multiple neurons collectively. Each neuron in the layer performs the same linear transformation and applies the same activation function to its input, but with its own set of weights and biases. Also, the layer is said to be *fully-connected* when each input sample will be transformed by all the neurons of the layer itself. Overall, the neural network's ability to learn and generalize comes from the combination of these basic building blocks, where the neurons collectively process the input data through multiple layers, with each layer performing a non-linear transformation of the preceding layer's output.

In terms of mathematical operations, feeding an M-dimensional vector to a neural network means performing a matrix multiplication by a **weight matrix** $W \in \mathbb{R}^{M,N}$ and a vector summation by a **bias vector** $B \in \mathbb{R}^{N}$, followed by an element-wise activation function, to output an N-dimensional vector.

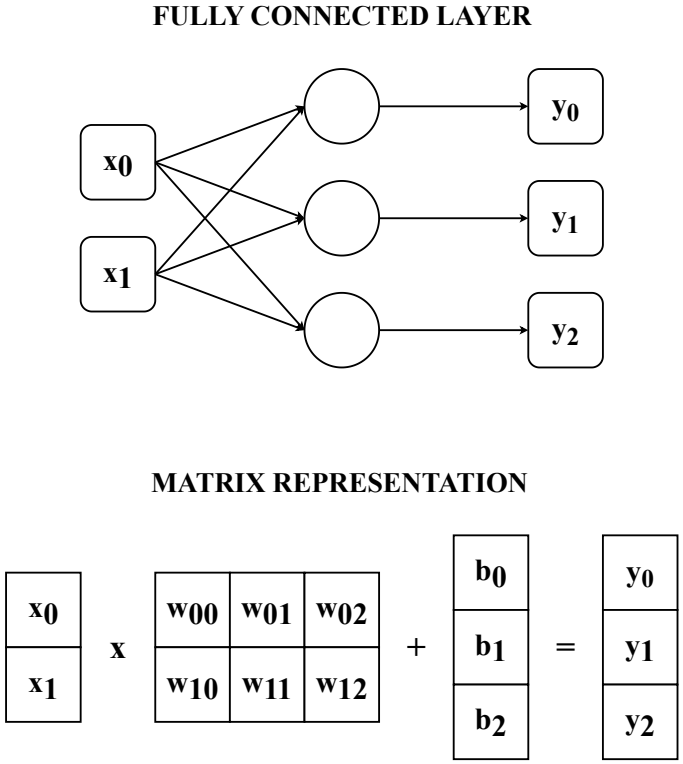

Figure 2.4: Example of Fully-Connected Neural Network Layer and its corresponding matrix representation. As one can see, the 2D input $[x_0, x_1]$ is firstly converted into a 3D vector by means of matrix multiplication by $W \in \mathbb{R}^{2,3}$ and then the bias vector $B \in \mathbb{R}^3$ is added to produce a 3D output vector $[y_0, y_1, y_2]$. This layer has an overall of 9 parameters. The activation function in the matrix representation is here omitted.

Multi-Layered Neural Network

A multi-layered neural network, also known as a **deep neural network**, is a structure consisting on multiple layers of interconnected neurons. Each layer in the network is composed by a set of stacked neurons that perform computations on the input data, exactly as shown in equation 2.2. The output of one layer serves as the input to the next layer, creating a sequential flow of information through the network. Therefore the information moves only in one direction, from the input layer to the output layer, without any cycles or loops. This architecture is also known as a *feedforward neural network* or *multilayer perceptron* (*MLP*). The intermediate layers between the input and output layers are called *hidden layers*. The *depth* of the network refers to the considered number of layers.

$$y = \sigma(w_n \cdot (\dots \sigma(w_2 \cdot (\sigma(w_1 \cdot x + b_1)) + b_2) \cdots + b_n) \tag{2.3}$$

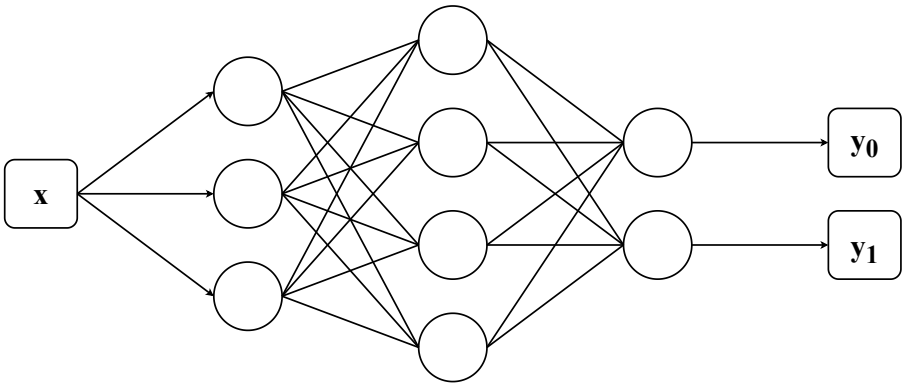

Figure 2.5: Example of Multi-Layered Neural Network with one input, hidden and output layer, thus depth equal to 3, with a 1D vector as input and a 2D vector as output. Each circle represents a neuron.

Activation function

The activation function plays two important roles:

- in the inner parts of the model, it introduces non-linearity to the computations performed by the neurons. A linear function would only be able to represent linear relationships between the inputs and outputs, which limits the expressive power of the network. However, by using activation functions with different slopes, neural networks can approximate more complex and non-linear functions. These activation functions enable the network to capture and represent intricate patterns and relationships in the data.

- at the last layer of the network, it serves the role of mapping the outputs of the preceding linear operations into a specific range or distribution. The choice of activation function at the output layer depends on the nature of the task. For example, in binary classification problems, a *sigmoid* activation function is commonly used to squash the output values between 0 and 1, representing the probability of belonging to a particular class. In multi-class classification problems, the *softmax* activation function is often used to produce a probability distribution over multiple classes.

Activation functions used in neural networks are typically chosen to be nonlinear and differentiable. There are two main reasons for this:

- non-linear activation functions allow neural networks to learn and represent complex and non-linear relationships in the data. In this way it's possible to introduce non-linearity into the network, enabling it to capture and approximate highly nonlinear functions.
- differentiability is crucial for training neural networks using gradient-based optimization algorithms, such as backpropagation. In fact, by using differentiable activation functions, gradients can be computed throughout the network, facilitating the optimization process.

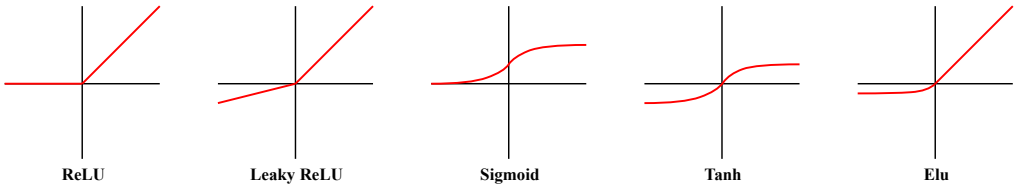

Figure 2.6: Some of the most common activation functions. The *sigmoid*, and also its corresponding multiclass version (the *softmax*), is mostly used to output probabilities since it forces its output to stay in a range (0,1).

As one can see from the image 2.6, activation functions have a sensitive range. The sensitivity in this region ensures that small changes in the input can propagate through the network and contribute to meaningful updates in the weights and biases, enabling effective learning. Also, many of them also have an insensitive or saturated range where there are little changes from input to output or no changes at all. In this range, the activation function becomes flat or approaches a constant value. The saturation of the activation function can cause the gradient to become extremely small or zero, leading to the vanishing gradient problem.

This means that this diversity in response ranges allows the network to capture and represent different aspects or features of the input data. What's more, during the learning process, neurons with activation functions having an higher sensitivity will have larger gradients and, consequently, will experience larger updates.

Learning in Neural Networks

Building models using stacks of linear transformations followed by differentiable activations has proven to be a powerful approach in deep learning. By combining multiple layers of linear transformations and nonlinear activations, deep neural networks can learn hierarchical representations of the input data. The success of deep learning is partly due to the effectiveness of gradient descent optimization

algorithms, such as backpropagation, in estimating the parameters of these models. Through backpropagation, gradients are computed and used to update the model's parameters, iteratively minimizing a loss function. This process allows the model to learn from large-scale data and adapt its parameters to fit the training data well.

Therefore training a neural network involves finding optimal values for its weights and biases so that the network can accurately perform a given task. The objective is to generalize well on unseen data, meaning the network should produce correct outputs on new examples of the same type of that ones which was used during training. During the training process, the network learns to capture the features of the data by adjusting the weights and biases. When the network is successfully trained, its weights and biases reflect the acquired knowledge about the data. This knowledge allows the network to generalize well to previously unseen data samples. This is what *learning* is meant to be, since a general model is specialized in a specific task.

Neural Networks Limitations

As described above, a neural network is composed of one or more layers of interconnected neurons. The network architecture can vary in terms of the size of the input data, the number of layers, the number of neurons in each layer, and the connections between them. However, the neural networks can face the overfitting problem, i.e. the potential of memorizing the training set without generalizing on the validation or test set. Going into the details, there are two main causes that could limit the effectiveness of neural networks. Firstly, depending on the number and dimension of the layers, the model could have too many parameters, hence becoming too complex and learning to fit the data too closely. Secondly, the neural networks, by themselves, are inherently *translation variant*. This means that small changes in the input data, such as shifting an image or text, can lead to significantly different representations.

To combat these problems, it is essential to carefully design the neural network architecture, consider regularization techniques, validate the model's performance on unseen data, and employ strategies such as data augmentation, cross-validation, and model selection based on validation performance.

However, a better method could be instead to address the translation variance issue in neural networks by replacing the dense, fully connected affine transformation with *convolutional layers*.

2.1.3 Convolutional Neural Networks

Convolutional neural networks (**CNNs**) are specifically designed to capture spatial relationships in data, such as images. The main operation of a CNN is the convolution operation which consists on sliding a small window, called *filter* or *kernel*, across each position of the input data and computing element-wise multiplications and summations. The convolution in CNNs is a local and, more important, *translation invariant* linear operator.

More precisely, the convolution consists for a 2D grid-like structure as the scalar product of a kernel function (weight matrix) with the receptive field of the reference pixel, which is determined by the region of the input image that influences the computation at a specific position. For example, a kernel of MxM dimension has a receptive field of MxM pixel neighborhood. In general, the size of the weight matrix can be of any dimension, taking into account that the height and the width are necessarily odd numbers.

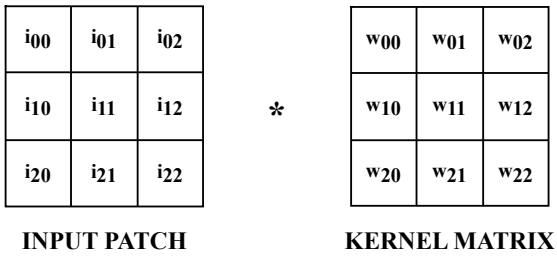

Figure 2.7: Weight matrix and a general patch from an input data.

If considering the figure 2.7, the target pixel of the input patch (i_{11}) is transformed in the output image as:

$$\begin{aligned} o_{11} = & i_{00} \cdot w_{00} + i_{01} \cdot w_{01} + i_{02} \cdot w_{02} + \\ & i_{10} \cdot w_{10} + i_{11} \cdot w_{11} + i_{12} \cdot w_{12} + \\ & i_{20} \cdot w_{20} + i_{21} \cdot w_{21} + i_{22} \cdot w_{22} \end{aligned} \tag{2.4}$$

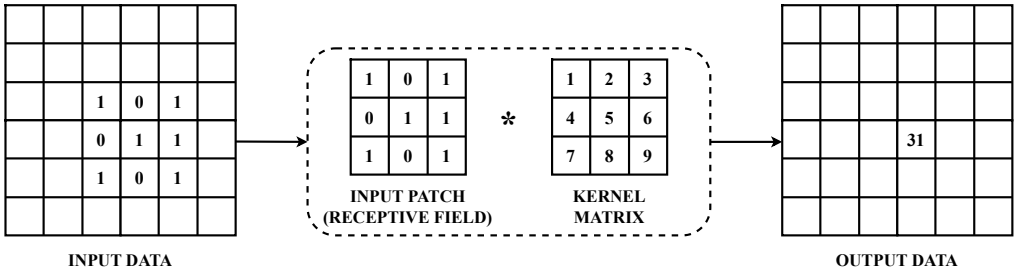

Figure 2.8: Example of convolution where a pixel of interest is updated by means of pixel neighborhood information and kernel matrix.

The kernel matrix is the same for each reference pixel and its weights are learned during the training process, as the same way as the neural network parameters. Therefore, the advantage of convolutional neural network is that this weight sharing property greatly reduces the number of parameters and enables the network to capture spatial patterns more efficiently.

Furthermore, CNNs often incorporate other operations like **pooling** and non-linear activations to further enhance their ability to extract relevant features. Pooling layers downsample the feature maps, reducing their spatial dimensions. Specifically, this is generally done by averaging or summing pixels in a neighborhood or by taking the pixel with maximum features. Non-linear activations introduce non-linearities into the network, allowing it to learn complex and non-linear relationships between the input and output.

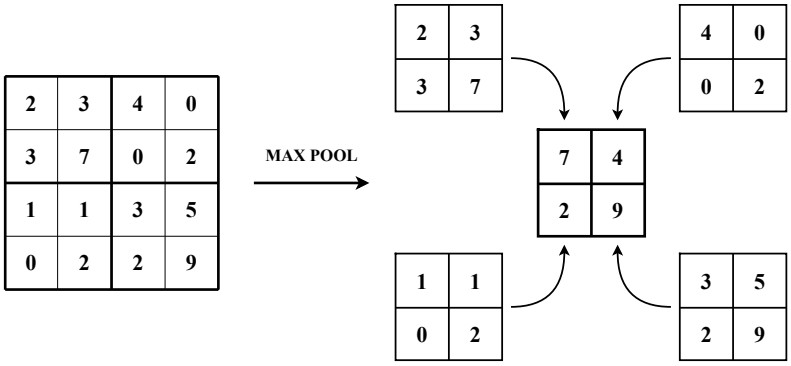

Figure 2.9: Example of max pooling operation. The input data is divided in smaller regions and, for each of them, the max value is selected.

Anyway, reducing the feature map of the input data through convolutional and pooling layers alone does not directly produce probabilities or predictions for a specific task. In fact, convolutional and pooling layers are responsible for feature

extraction and spatial preservation. That's why at least one fully connected layer is needed in the last stage of the model: it takes the *flatten* (or unrolled) 1D output of the last convolutional or pooling layer and transforms it into a suitable format for making predictions.

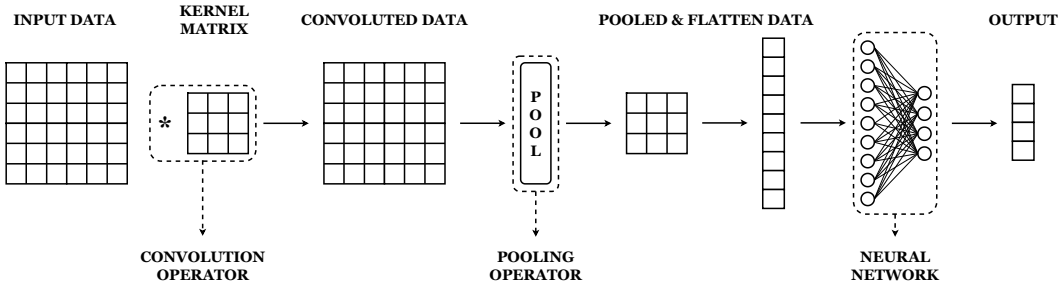

Figure 2.10: Example of a simple CNN composed by one convolutional and pooling layer followed by a 2-layer neural network, giving a 4D output classification or prediction.

2.2 Graph Neural Networks

Graph Neural Networks (**GNNs**) are a generalization of the traditional Convolution Neural Networks to deal with data of arbitrary size and complex topological structures. Unlike the CNN, which uses a kernel matrix to update a pixel by exploiting information of neighborhood pixels, the GNN uses neighborhood node features (can also edge features) to update the attributes of the target node by means of aggregation and updating functions.

2.2.1 Graph Structure

Graph Neural Networks work on graph-structured data. A **graph** G is a network composed by a set of vertices (or nodes) V and a set of edges E.

$$G = (V, E)$$

The set of edges E describes the connectivity between nodes in the graph and has a matrix representation known as **adjacency matrix** $A \in \mathbb{R}^{N,N}$, where N is the number of nodes in the network ($N = |V|$). The entry $A[i,j]$ in the adjacency matrix represents the presence or absence of an edge between nodes i and j. In other words, $A[i,j]$=1 if there is an edge connecting node i and j, and $A[i,j]$=0 otherwise. Also, the adjacency matrix can be further extended to incorporate additional information, such as edge weights or edge types.

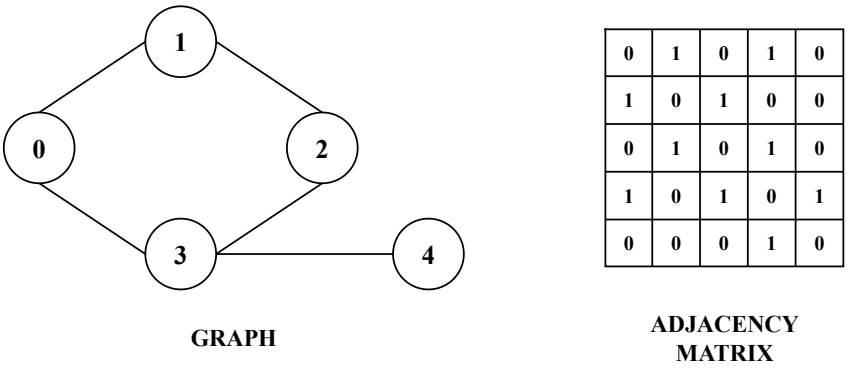

Figure 2.11: Example of a graph with five nodes, with no directed and weighted edges, and its corresponding 5x5 adjacency matrix. Since the graph is undirected, the adjacency matrix is symmetric, because for every couple of connected nodes i and j there's also a couple of edges $e_{i,j}$ and $e_{j,i}$. If there are not self-loops the diagonal of the adjacency matrix will always be composed by zeroes.

Each node in V carries information which can be represented by a 1-dimensional array called **node feature vector** $\vec{x} \in \mathbb{R}^F$, where F represents the number of features of every node in the network. All feature vectors can be stacked together to form the **node feature matrix** $X \in \mathbb{R}^{N,F}$ of the whole graph. Therefore, the k^{th} row of the node feature matrix corresponds to the feature vector of the k^{th} node of the graph.

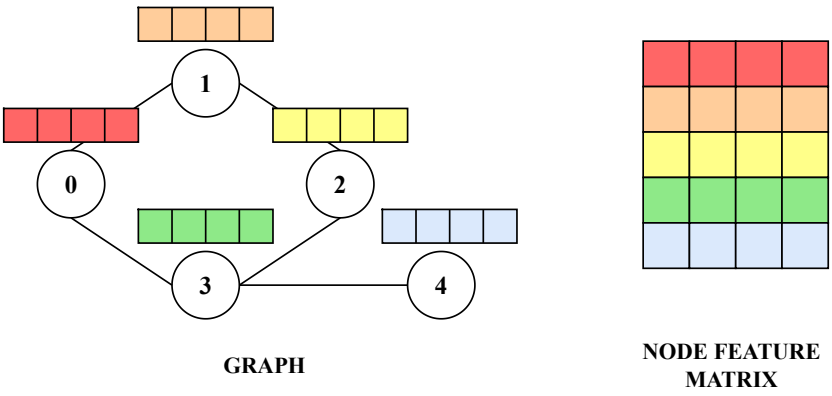

Figure 2.12: Example of graph with five nodes and 4D node feature vectors and its corresponding 5x4 node feature matrix.

The same accounts for the edges: they can give information on the relationship or properties between connected nodes, rather than only connectivity information. The **edge feature vector** for an edge linking nodes i and j can be denoted as $e_{ij} \in \mathbb{R}^{F_e}$, where F_e represents the number of features associated to each edge. All

edge feature vectors can be stacked together to form the **edge feature matrix** $E \in \mathbb{R}^{N,N,F_e}$.

2.2.2 Network Architecture

The top-level structure of a Graph Neural Network shares similarities with Convolutional Neural Networks. GNNs also employ convolutional, pooling, and fully-connected layers, but the implementation of these layers is different in GNNs, while still having similar concepts. In fact, in CNNs, convolutional layers operate on regular grid-like structures such as images, where filters slide over local receptive fields to extract local patterns. In GNNs, the convolutional operation is adapted to work on graph-structured data. Instead of applying convolution on regular grids, GNNs perform convolution on the graph's nodes and their local neighborhoods. For what concerns pooling layers, in CNNs they are typically used to downsample the input and reduce its spatial dimensions. In GNNs, the concept of pooling is different because graphs don't have a fixed spatial structure. However, in GNNs, fully-connected layers are often used after the graph-based operations to transform the node features into the desired output format for the given task. GNNs are in fact capable of making predictions at three different levels within a graph: node-level, edge-level and graph-level predictions. It's possible to classify individual nodes and edges in the graph or to capture global properties of the entire graph. In the latter case, a *readout* operation is needed in order to aggregate node-level embeddings and therefore get a fixed-size representation of the whole graph which may be deployed into a multilayer perceptron.

Graph Convolutional Layer

A Graph Convolutional Layer is mainly composed by two operations: **aggregation** and **updating**.

The *aggregation* operation involves gathering information from the 1-hop neighborhood of each node in the input graph. This is achieved through **message passing**, where the node features of neighboring nodes, and also the features of the target node, are exchanged and aggregated to create a representation of the neighborhood. The purpose of aggregation is to capture the collective information from neighboring nodes and incorporate it into the target node's representation. It's very important to ensure the **permutation invariance**, i.e. the final result must not change by reordering the node messages, and a common approach is to perform a permutation-invariant aggregation operation such as summation or averaging. In these operations, the order of the neighboring nodes does not affect the

result.

$$z_i = f_{AGG}(x_i, f_{MSG}(x_j, e_{ij})), \quad \forall j \in N(i) \tag{2.5}$$

The *updating* operation consists on applying a transformation to the aggregated information to obtain the updated node features, also known as **embeddings**. The update function can be implemented as a neural network layer, where the aggregated information is passed through a set of learnable parameters. According to the graph convolutional layer design, the dimensionality of the embeddings can change when applying the neural network to the input node feature vector. Generally a weight matrix $W^{(l)} \in \mathbb{R}^{F,F'}$, where F is the dimensionality of the input embeddings, F' is the dimensionality of the output emmbeddings and l denotes the layer index, is used to alter the embedding dimensionality. This weight matrix is the same for all nodes in the graph within a layer. The output of the update function is therefore the updated node feature vector, which could then be used as input for the subsequent layers.

$$x'_i = f_{UPD}(z_i) \tag{2.6}$$

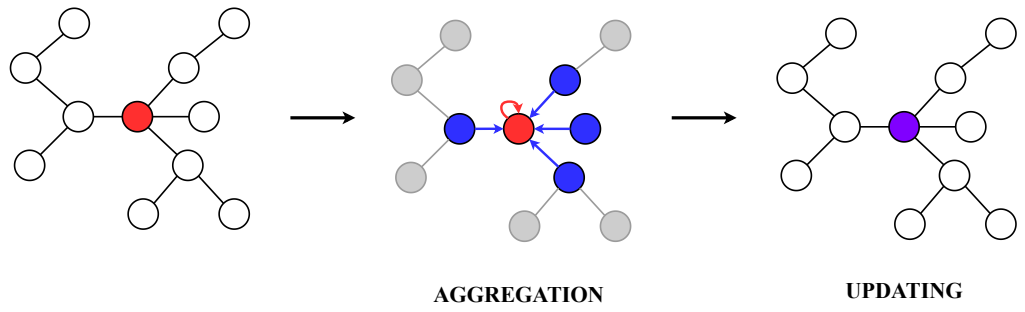

Figure 2.13: The target node (in red) gathers information from its 1-hop neighborhood (in blue) to get its updated version (in purple). The same procedure is applied for the nodes in the graph.

By iteratively applying aggregation and updating operations in multiple GNN layers, the network can propagate and refine information across the graph, enabling it to capture both local and global dependencies in the data, similarly to the case of CNNs.

Pooling

Pooling in Graph Neural Networks refers to the process of aggregating information from multiple nodes or sub-graphs to create a coarser representation of the graph.

Pooling is often used in GNNs to reduce the size or resolution of the graph while retaining important structural and semantic information in order to produce graph-level representations. Specifically, given a graph G, its nodes are clustered in smaller sub-graphs G_1, G_2, ..., $G_N \in G$. The clustering can be based on various criteria such as node similarity, node features, or graph structure. Then, for each sub-graph G_i, the node embeddings are aggregated using pair-wise operations to produce a single node representation at coarsen level. Common pooling functions are the sum, the average and the maximum operations. Finally, the pooled nodes need to be connected to create the output graph. This step involves determining the edges between the pooled nodes that can depend on several criteria such as the connectivity of the original graph or learned weights.

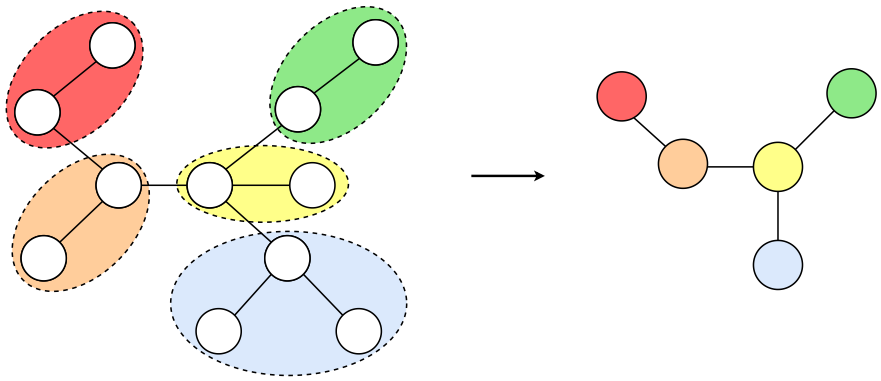

Figure 2.14: Example of graph pooling. Nodes within clusters are aggregated together to obtain a single node. Here the connectivity of the pooled graph follows that of the original graph, meaning that if there are edges connecting two clusters then the respective pooled nodes will also be connected.

Readout

A readout operation refers to the process of generating a fixed-size representation of the entire graph based on the information learned from its nodes and edges. The readout step is typically performed after the last graph convolutional or pooling layer and it aims to capture global properties and patterns in the graph. To be more detailed, the readout process can be seen as a **global pooling** where the embeddings of every node in the graph are gathered to obtain a graph-level representation of the network. The readout can then be used for various downstream tasks such as graph classification. The choice of readout operation depends on the specific problem and the desired properties to be captured. As in the case of pooling, common readout operation are the sum, the average and the maximum. Often,

after the readout, a fully-connected (or multi-layered) neural network is applied to its output to increase the expressiveness of the Graph Neural Network.

Overall architecture

Summarizing, the basic approach of a Graph Neural Network is to take as input the node feature matrix containing all the input node features vectors:

$$h_i^{(0)} = x_i \tag{2.7}$$

then apply graph convolutional layers to get higher-level feature representations:

$$h_i^{(l+1)} = GConv^{(l+1)}(h_i^{(l)}), \quad \forall l \in \{0, ..., L-1\} \tag{2.8}$$

to get the final node embeddings after the L^{th} layer of neighborhood aggregation:

$$z_i = h_i^{(L)} \tag{2.9}$$

If graph-level predictions has to be performed, graph pooling operations can be used between convolutional layers to reduce the graph dimension while keeping its structural information. Then a readout function is used to represent the entire graph with a fixed size representation that can be also delivered to a fully-connected neural network to increase the expressiveness power of the GNN.

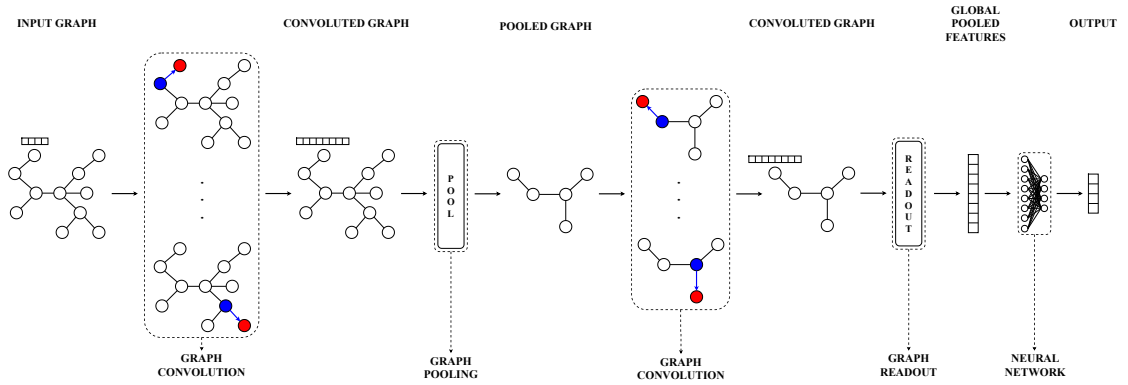

Figure 2.15: Example of a simple Graph Neural Network for graph-level classification composed by two graph convolutional and one pooling layers, followed by a readout operation and a 2-layer neural network, producing a 4D output vector. Here every node of the input graph is characterized by a 4-dimensional node features vector. The dimensionality of the embeddings increases to eight after the first graph convolutional layer and still remains the same after the second graph convolution. The red nodes in the graph represent the target nodes, whereas the blue ones represent the 1-hop neighborhood. Then the pooling operator reduces the size of the graph and the readout function outputs a vector with dimensionality equal to the final node embeddings length.

Part II

Event-based GNN

Chapter 3

DVS Gesture Dataset

The **DVS Gesture Dataset** is a collection of event streams representing hand gestures, presented in 2017 by *Amir et Al.* in [3]. The events are generated by using a DVS128 camera which is characterized by a 128x128 grid structure.

3.1 Structure

The *DVS Gesture Dataset* contains 1342 event streams of a set of 11 hand gestures. The samples are collected by 29 people under 3 different light conditions combining natural light, fluorescent light and LED light. Each trial involves one subject standing against a stationary background and sequentially performing all 11 gestures under the same light condition. The gestures include hand waving (both arms), large straight arm rotations (both arms, clockwise and counterclockwise), forearm rolling (forward and backward), air guitar, air drums, and a last gesture created by the subject ([3]). The duration of an event stream can range between about 3 to 8 seconds for an overall average of approximately 6 seconds. For cross-validation purposes, the dataset is divided in training and test sets (80% and 20% respectively): 23 subjects are selected for the training set and remaining 6 subjects for the test set.

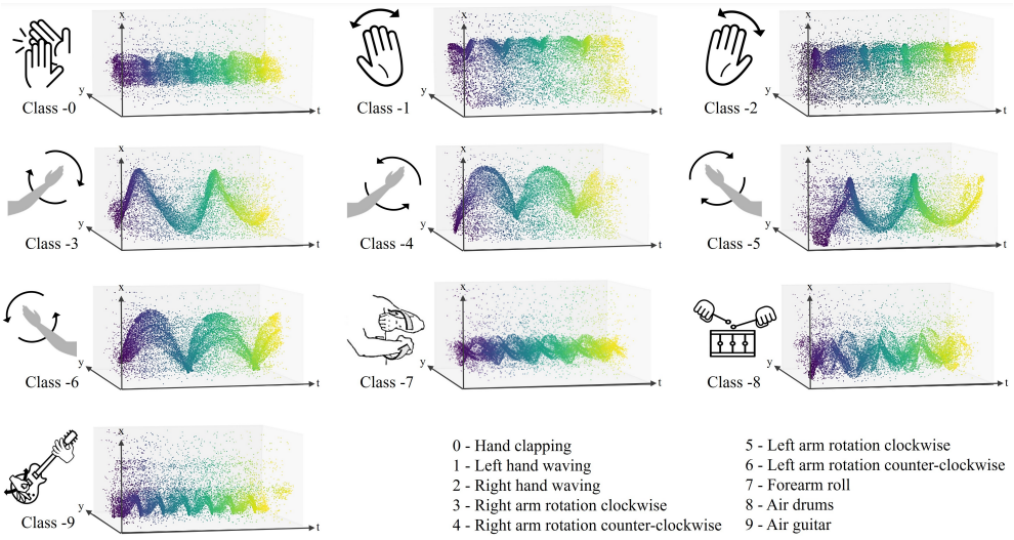

Figure 3.1: Two-seconds snippets of the first ten classes. The eleventh class is not present because it depends from the subject. Image taken from [2].

Chapter 4

Event2Graph

The DVS Gesture dataset, described in chapter 3, provides samples which are nothing but streams of events. Nevertheless, the Graph Neural Network model used in this work, and presented in chapter 5, accepts only graph-based structures at its input. Thus, before effectively training the model, it's necessary to apply a **preprocessing** step in order to treat events as graphs in a three-dimensional space $(x, y, t) \in \mathbb{R}^3$. This preprocessing phase involves several sub-steps which gather a certain number of events from a event stream and process them to finally obtain the *event graph*. This procedure is applied to each sample of the dataset. The final results consists on translating an event-based dataset to a graph-based dataset.

Anyway, the DVS Gesture dataset is partitioned in training and test sets with a percentage of 80% and 20% respectively (1078 samples for training and 264 for testing), and there's not a validation set. Therefore, the training set is further sliced in order to create the validation set: the ratio between the number of samples per label in a set and the overall number of samples per set is, in any case, the same for the training, test and validation sets, accounting now for the 60%, 20% and 20% respectively (814 samples for training, 264 for testing and 264 for validation).

4.1 Preprocessing Phase

The *preprocessing phase* is needed to process events in order to finally output them in a graph-based representation. A sliding time window is used to select batches of events from a sample, then for each of them a series of transformations are applied to remodel events in a suitable form for being applied to a Graph Neural Network.

Each sample of the event dataset is defined by an integer label, i.e. the ground truth, ranging from 0 to 10 accounting for a total of 11 classes. Since the goal

of this work refers to a classification task, the labels are converted into a one-hot encoding representation. Precisely, each event stream is now associated with a 11D vector containing all 0s, except 1 in the location pointed by the original integer label.

Old Label	New Label
0	100000000000
1	01000000000
2	00100000000
3	00010000000
4	00001000000
5	00000100000
6	00000010000
7	00000001000
8	00000000100
9	00000000010
10	00000000001

Table 4.1: One-Hot Encoded labels.

All the graphs generated by the same event sequence share the same label because they represent a part of the same gesture.

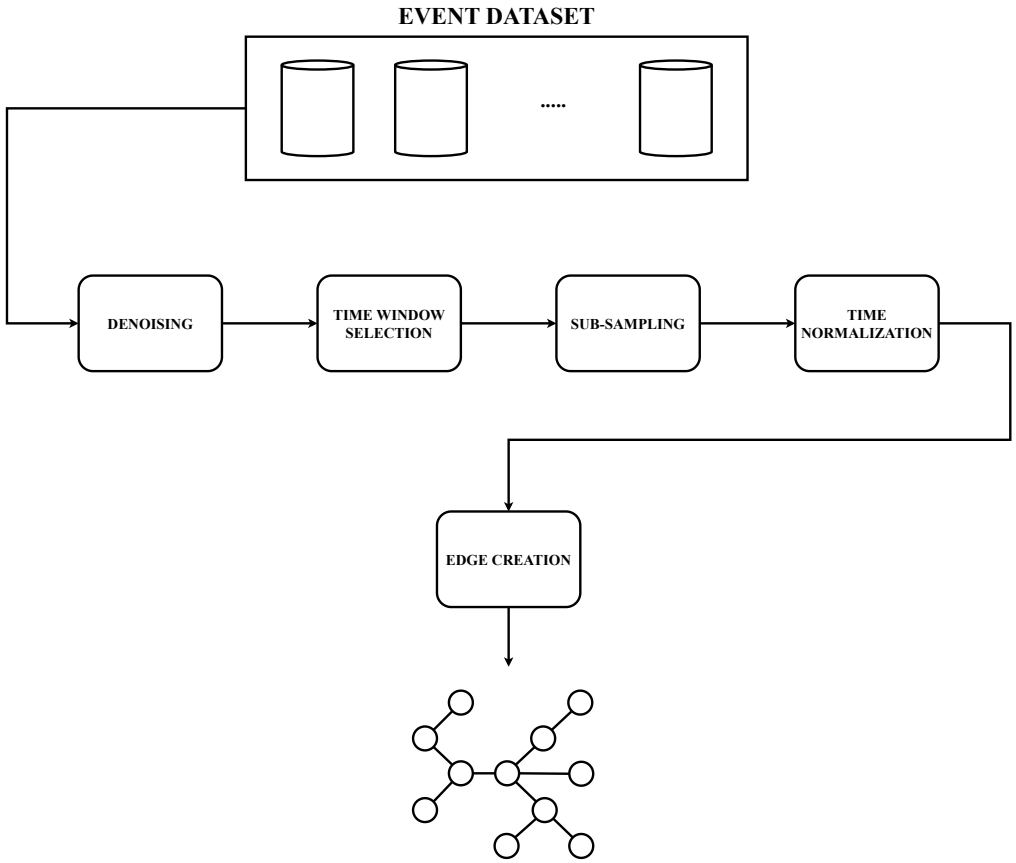

Figure 4.1: Preprocessing Scheme.

Figure 4.1 represents the block scheme of the preprocessing phase. An event-sample at the time is withdrew from the dataset and it's passed through several sub-steps, including *denoising*, *time window selection*, *sub-sampling*, *time normalization* and *edge creation*. This preprocessing is iteratively executed for a certain number of times for each event stream, according to the number of consecutive time windows, as better explained in section 4.1.2. Multiple graph-based structures are therefore generated from a single event sequence, each representing part of the input event-based sample.

Algorithm 1 Event2Graph

Input: ED ▷ $ED : EventDataset \rightarrow [ED_{train}, ED_{test}, ED_{val}]$

procedure EVENT2GRAPH(ED)
▷ $GD : GraphDataset \rightarrow [GD_{train}, GD_{test}, GD_{val}]$
for *set* in [*train, test, val*] **do**
for e_{in} in ED_{set} **do** ▷ e_{in}: input sample
▷ $e_{\{in\},i}[t]$: temporal information of the i^{th} input event
$t_{start} \leftarrow 0$
$t_{end} \leftarrow TW$
$t_{last} \leftarrow e_{in,N-1}[t]$
while $t_{end} < t_{last}$ **do**
$e_{tw} \leftarrow$ TWSELECTION$(e_{in}, t_{start}, t_{end})$
$e_{ss} \leftarrow$ SUBSAMPLING (e_{tw}, SSF)
$e_{tn} \leftarrow$ TIMENORM (e_{ss}, TNR)
$G \leftarrow$ EDGECREATION (e_{tn}, R, D_{max})
$push(G, GD_{set})$
$t_{start} \leftarrow t_{start} + TS$
$t_{end} \leftarrow t_{end} + TS$
end while
end for
end for
end procedure

4.1.1 Denoising

Event cameras can be affected by noisy events which are generally isolated by the main flow of the sequence. Therefore, before effectively preprocessing the event streams, a transformation to the dataset is applied in order to reduce the noise. In this case, **denoising** the samples means performing a space-time filtering. Specifically, an event triggered at a certain pixel is dropped if none of the neighborhood pixels is triggered within a *filter time*.

In general, this operation may prevent the model to learn also the noise, if too powerful, thus not generalizing and causing overfitting.

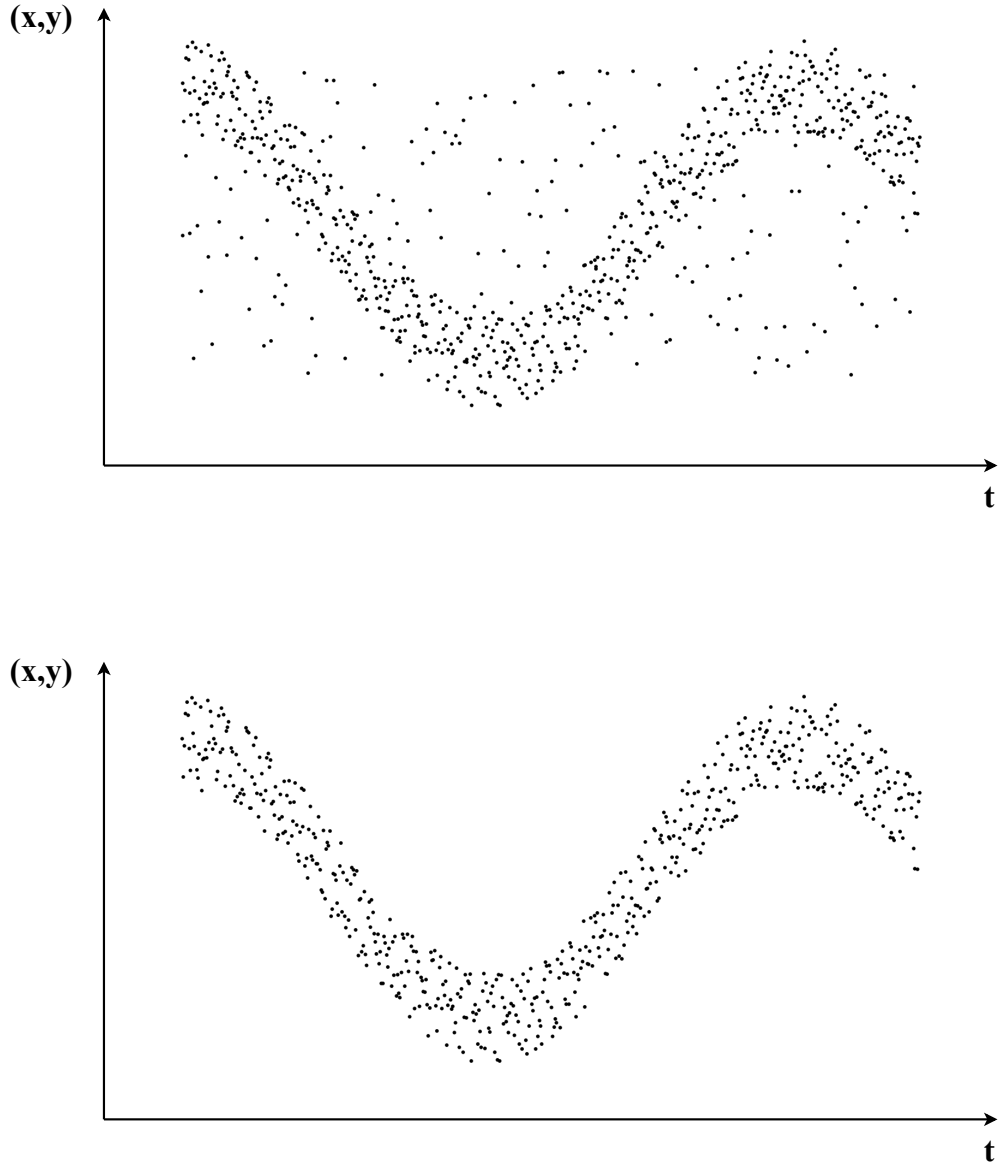

Figure 4.2: Denoising process. Original event stream (on the top) vs denoised event stream (on the bottom).

4.1.2 Time Window Selection

After withdrawing a sample from the dataset and applying transformations to it, the next step is to select a sub-set of the entire event stream and process it to create a graph. In fact, not the whole event sequence is used to create a single graph, but the sample is sliced in smaller sequences which are then converted in graph-based structures. In order to accomplish that, a fixed length **time window**

slides through the temporal dimension of the event stream and pick only the events within the current start and end values of the time window itself.

However, the sliding window doesn't move in a event-by-event fashion, but it slides by a fixed size **time step**. If the dimension of the time step is smaller than the time window, then two or more successive sliding windows are *overlapped*: in this case, one or more events can be part of two different graphs. The solution of sliding the time window by a time step factor may be executed only for training purposes, so that the training times are reduced.

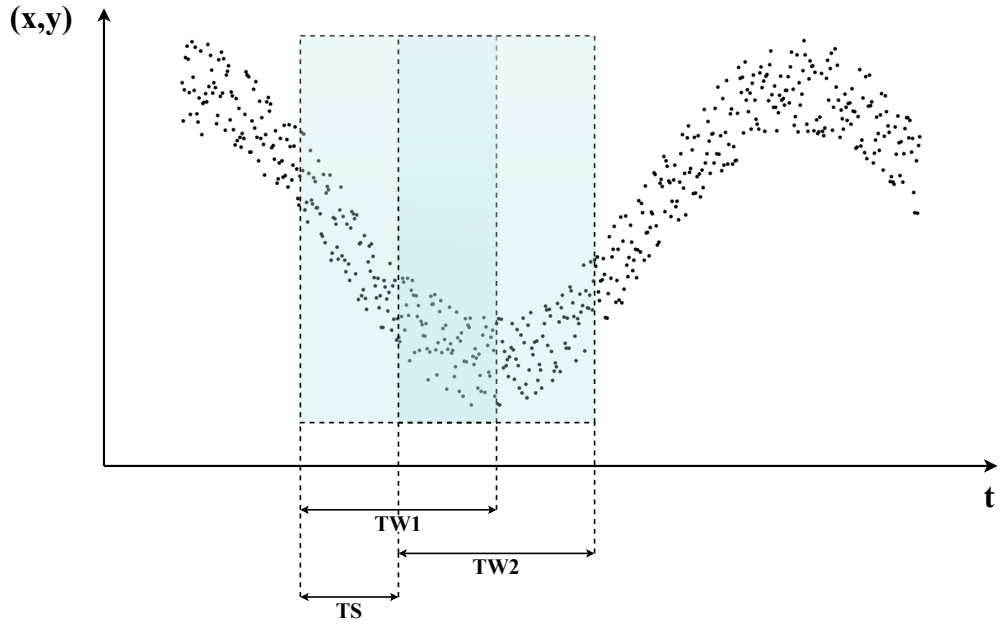

Figure 4.3: Time Window Selection with overlapping time windows. *TW1* stands for *Time Window 1*, and *TW2* means *Time Window 2*. They are two successive time windows. *TS* is the *Time Step*. In this case, $TS < TW$.

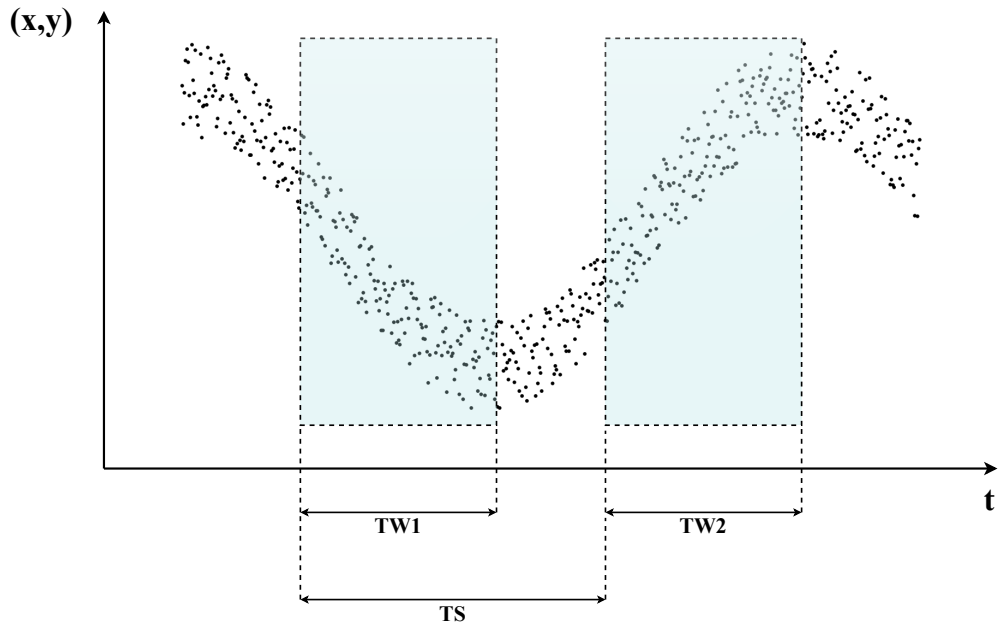

Figure 4.4: Time Window Selection with not overlapping time windows. In this case, $TS > TW$.

The choice of the time window length can influence the performances of the model in terms of reactivity and accuracy results; it's therefore a trade-off between:

- latency: a smaller time window leads to a more reactive system because less events are captured, thus less computation effort is required;
- accuracy: a larger time window allows to explore in a more complete way the temporal evolution of the event stream to better understand its features, thus probably giving better classifications.

The dimension of the time step is choosed to have a reasonable amount of samples for the graph-based dataset. The smaller the time step, the greater the number of slices of the event stream caused by the sliding window; thus the greater the number of graphs and the larger the *event-graph dataset*. The relation between the final size of the graph dataset is inversely proportional to the time step value. Actually, also the size of the time window influences the dimension of the final number of graphs.

$$n^{\circ}graphs = \frac{len(sample) - TimeWindow}{TimeStep} \tag{4.1}$$

Algorithm 2 Time Window Selection

Input: $\{e_{in}\}_N$ ▷ N: number of input events
Output: $\{e_{out}\}_M$ ▷ M: number of output events

function TWSELECTION($e_{in}, t_{start}, t_{end}$)
▷ $e_{\{in,out\},i}[t]$: temporal information of the i^{th} input or output event
$j \leftarrow 0$
for $i \leftarrow 0$ to N-1 **do**
if $e_{in,i}[t] > t_{start}$ **and** $e_{in,i}[t] < t_{end}$ **then**
$e_{out,j} \leftarrow e_{in,i}$
$j \leftarrow j + 1$
end if
end for
$M \leftarrow j$
return e_{out}
end function

4.1.3 Sub-Sampling

After selecting part of the event stream by means of a time window, it's generally necessary to reduce the number of events by **sub-sampling** operations. The large amount of events coming out a DVS sensor, in fact, could be difficult to process if not handled in a determined way. Basically, from a set of N events $\{e_i\}_N$, a sub-set of M of representative events $\{e_i\}_M$ is sampled, with $M \ll N$. In general, two sub-sampling methods exist. The *uniform sub-sampling* consists on selecting an event at regular intervals or with equal spacing; the *non-uniform sub-sampling* involves selecting events in a non-regular manner. In this work, the events are uniformly sampled by an integer factor *SSF*, which stands for *Sub-Sampling Factor*.

There are several reasons for the application of the sub-sampling step. First of all, reducing the number of events also reduces the computational cost to process them, because less number of operations are needed, as well as the storage to save them. Additionaly, although it removes events, it may be critical to prevent overfitting since the model will focus on more informative events, instead of trying to learn on less important or isolated parts of the input data. It also helps to reduce the noise of the event stream, since event cameras are naturally noisy; nevertheless, in this work a denoising transformation is performed over the whole dataset before the sub-sampling operation.

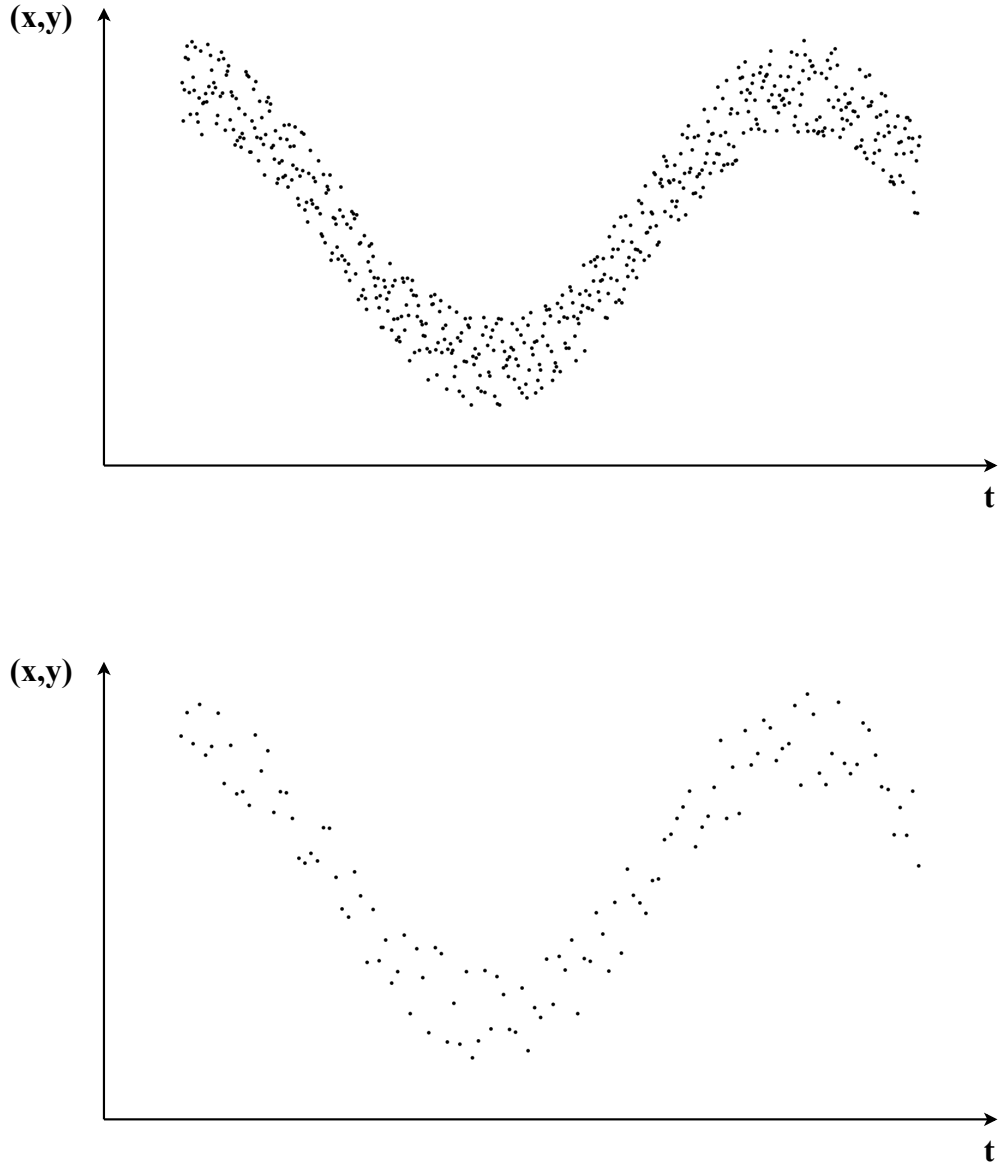

Figure 4.5: Sub-sampling process. Original event stream (on the top) vs sub-sampled event stream (on the bottom) with $SSF = 5$.

Algorithm 3 Sub-Sampling Algorithm

Input: $\{e_{in}\}_N$ ▷ N: number of input events
Output: $\{e_{out}\}_M$ ▷ M: number of output events

```
function SubSampling(e_in, SSF)
    ▷ e_{in,out},i: i^th input or output event
    j ← 0
    for i ← 0 to N-1 do
        if i % SSF == 0 then
            e_out,j ← e_in,i
            j ← j + 1
        end if
    end for
    M ← j
    return e_out
end function
```

4.1.4 Time Normalization

The sub-sampled events are used to create a graph. Nonetheless, the resolution of the temporal dimension ($1\mu s$) differs from that of spatial dimension (1 pixel) by several order of magnitude, i.e. timing accuracy is much higher than spatial grid resolution. Therefore, in order to compensate the temporal and spatial resolution and use the edge creation algorithm described in 4.1.5, a **time normalization** sub-step is needed. Basically, to map the temporal dimension in a similar range as the spatial coordinates, the timing information of the sub-sampled events is discretized by shrinking it to stay in a range [0, *TNR*), where *TNR* it's an integer number representing the *Time Normalization Range*, that is the discretized temporal range. The main concept is to divide the sub-sampled events into *TNR* successive slices and, for each of them, replace their high resolution temporal information with a discretized temporal information.

The choice of the Time Normalization Range can influence the training phase. In fact, a less extended normalized temporal dimension, i.e. lower *TNR* factor, may lead to a better temporal correlation between events (when creating edges between events) because of the shorter time-distance, thus eventually better accuracy results. However, reducing the temporal normalization factor too much could produce the opposite effect to that for which this sub-step is performed, i.e. the resolution of the spatial dimension becomes increasingly greater than the temporal dimension, and this would become very similar to the event-frame representation (section 1.4).

Algorithm 4 Time Normalization Algorithm

Input: $\{e_{in}\}_N$ ▷ N: number of input and output events
Output: $\{e_{out}\}_N$

function TIME NORMALIZATION(e_{in}, TNR)
▷ $e_{\{in,out\},i}[t]$: temporal information of the i^{th} input or output event
$t_{min} \leftarrow e_{in,0}[t]$
$t_{max} \leftarrow e_{in,N-1}[t]$
$t_{range} \leftarrow t_{max} - t_{min}$
for $i \leftarrow 0$ to N-1 **do**
$t_{shift} \leftarrow e_{in,i}[t] - t_{min}$
$t_{norm} \leftarrow t_{shift} \cdot \frac{TNR}{t_{range}}$
$e_{out,i}[t] \leftarrow t_{norm}$
end for
return e_{out}
end function

4.1.5 Edge creation

Events are now ready to be converted in a graph-based representation, where each event can be considered as a node in the output graph. Anyway, a graph-based structure is composed by both nodes and edges, therefore a method to define the network connectivity is needed. Being the event stream treated as a 3D Point Set representation (see 1.4), a reasonable strategy to be used is the **Radius-Neighborhood Algorithm** which allows to create edges within a determined spatio-temporal range. This allows to exploit both the spatial and temporal correlation between nodes. To be more detailed, two nodes i and j are connected by an edge e_{ij} if their Euclidean distance d_{ij} is less than a radius distance R.

$$d_{ij} = \sqrt{|x_i - x_j|^2 + |y_i - y_j|^2 + |t_i - t_j|^2} < R \tag{4.2}$$

In addition, in order to limit the size of the graph, the maximum connectivity degree is set to a value D_{max}. This is mainly performed for computational efficiency since for highly event-dense regions an huge amount of edges may be created and this would lead to an high computational load when performing the graph convolution.

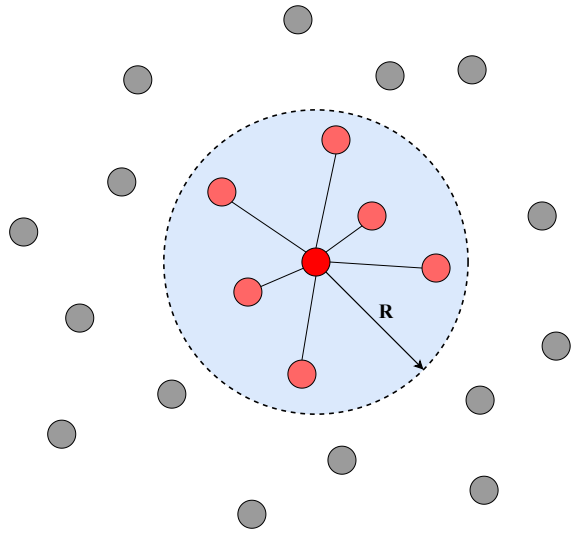

Figure 4.6: Edge Creation with Radius Neighborhood Algorithm.

Algorithm 5 Edge Creation Algorithm

Input: $\{e_{in}\}_N$ ▷ N: number of input events
Output: $G(V, E)$

function EDGE CREATION(e_{in}, R, D_{max})
▷ $e_{in,i}[x, y, t]$: spatio-temporal information of the i^{th} input event
$V \leftarrow e_{in}$ ▷ events are the nodes of the graph
for $i \leftarrow 0$ to N-1 **do**
$d \leftarrow 0$ ▷ d: node degree
for $j \leftarrow 0$ to N-1 **do**
if $i \neq j$ **then**
if $(norm(e_{in,i}[x, y, t], e_{in,j}[x, y, t]) < R)$ **and** $(d < D_{max})$ **then**
$connect(V_i, V_j)$ ▷ E structure is updated
$d \leftarrow d + 1$
end if
end if
end for
end for
return $G(V, E)$
end function

4.1.6 Node and Edge features

The edge creation step creates an *event graph*. However, it's necessary to associate **node and edge features** to each node and edge in the network.

Each event in the network is represented by a four-dimensional tuple (x, y, t^*, p), where t^* refers to the normalized time. The (x, y, t^*) information collocates an event in a determined position of the spatio-temporal space. The remaining value p is assigned as the initial node feature vector, i.e. $\vec{x} = [p]$. This means that each node has a feature vector outlined by a one-dimensional value representing the polarity of the associated event, thus it can only assume {-1,+1} values. It follows that the node feature matrix is a one-dimensional array.

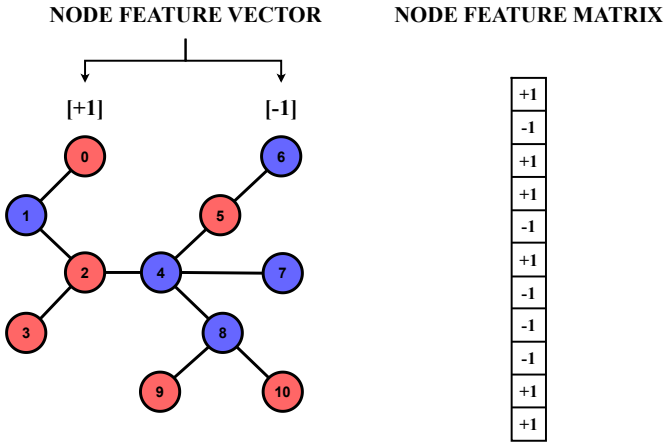

Figure 4.7: Node feature vector and node feature matrix representation. The red nodes are positive events, therefore their node feature vectors are equal to [+1]; the blue nodes are negative events associated to [-1]. The node feature matrix is a 1-dimensional array.

The connectivity of the nodes is assigned by exploiting the spatio-temporal position between nodes in a 3D space (x, y, t^*). For this reason, edge features can be defined as the relative distance of two nodes i and j in each spatio-temporal dimension, i.e. $e_{ij} = [\Delta x, \Delta y, \Delta t]$.

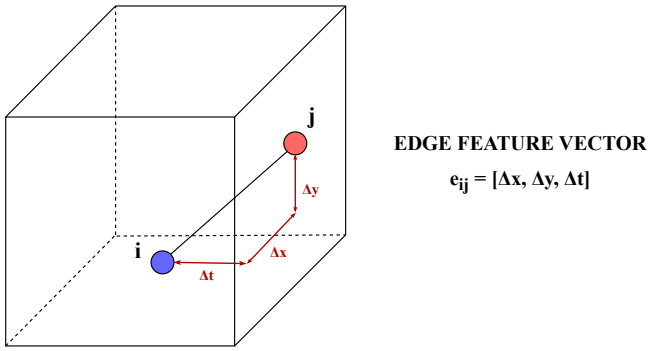

Figure 4.8: Edge Features Representation. Node i and j are two events within a spatio-temporal range defined by the radius value R.

4.1.7 Output graph structure

The *event graphs* are composed by a variable number of nodes, or events, depending both on the composition of the input event stream and the preprocessing parameters.

The structure of these graph-based networks can be described by several matrix structures (see 2.2.1). Being N the number of nodes (or events), the connectivity is determined by the *adjacency matrix* $A \in \mathbb{R}^{N,N}$, the *node feature matrix* $X \in \mathbb{R}^{N,1}$ is composed by bipolar values ± 1, whereas the *edge feature matrix* $E \in \mathbb{R}^{N,N,3}$ can be seen as an extension of the adjacency matrix representing edge features instead of just giving information on connectivity. PyTorch Geometric allows to describe A and E in a COO format which is used to represent sparse matrices; therefore, $A_{COO} \in \mathbb{R}^{2|e|,2}$ and $E_{COO} \in \mathbb{R}^{2|e|,3}$. In addition, the *position matrix* $P \in \mathbb{R}^{N,3}$ gives information about the location of each node embedded in the spatio-temporal space. Also, a one-hot encoded vector $L \in \mathbb{R}^{11}$ is used as label of the event graph.

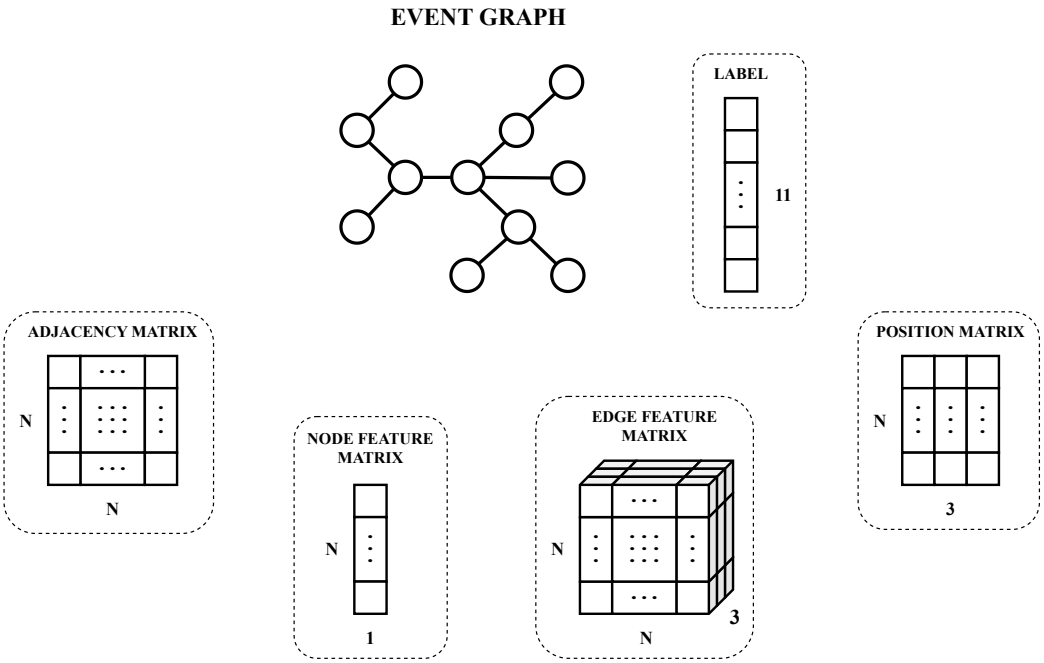

Figure 4.9: Event-based graph and its embedding matrix structures.

4.2 Preprocessing Hyperparameters

The preprocessing phase is characterized by a certain number of steps and each of them is characterized by one or two parameters. The *Time Window Selection* step involves the use of the TW and TS parameters, the *Sub-Sampling* and the *Time Normalization* operations uses the SSF and the TNR factors respectively, and the *Edge Creation* block utilizes the R and D_{max} values. The choice of those parameters can heavily influence the performance of the system, therefore they can be considered as **preprocessing hyperparameters**.

Expanding the time window allows for a more comprehensive analysis of the temporal progression of the event stream. However, this extension can potentially introduce latency in the architecture due to the greater number of events to be processed.

The sub-sampling factor serves for the purpose of enhancing computational efficiency and mitigating noise. It should be fine-tuned in conjunction with the time window value. When adjusting the time window size, it's essential to correspondingly modify the sub-sampling factor to maintain a roughly constant number of events.

The time step parameter (TS) primarily defines the dataset dimensions. A

larger TS value results in a smaller graph-based dataset, while a smaller TS value yields a larger dataset. Excessively high TS values may lead to a limited number of samples, compromising the model's ability to generalize during testing.

The time normalization factor (TNR) effectively compresses events along the temporal dimension, aligning it more closely with the spatial dimension. This parameter should be chosen along-side the radius value in the edge generation step. Lower TNR values allow the linking of temporally distant events without altering the R parameter, thereby increasing temporal correlation and potentially improving accuracy. In addition, a larger radius value facilitates the connection of events that are more distant in the spatio-temporal space. However, if the temporal dimension becomes significantly smaller than the spatial dimension (i.e., lower TNR values), the resulting event graph will exhibit much higher temporal correlation than spatial correlation.

The node maximum degree D_{max} is instead choosed to a reasonable maximum number of neighboring nodes.

The *preprocessing hyperparameters* values used in this work are:

- TW: 1000000 (μs)
- TS: 20000 (μs)
- SSF: 20
- TNR: 32
- R: 10
- D_{max}: 32

Those parameters allow to embed the output event-graph, whose events are taken by a time window of $1s$, into a 128x128x32 (x,y,t) space.

4.3 Graph Dataset Structure

The choice of the time window and time step hyperparameters determines the dimension of the graph-based dataset, as described before. According to the values pointed in 4.2, the preprocessing phase is able to generate 372227 graph samples complessively, divided in 222373, 76866 and 72988 samples for train, test and validation sets respectively, accounting for a percentage of about 59.74%, 20.68% and 19.61%.

The length of an event stream and its number of events is not constant and it changes sample by sample. This means that the event-graphs generated by the pre-processing phase don't contain the same number of nodes. Furthermore, being the time window value constant and the temporal length of the event stream variable, it follows that the label distribution of the graph-based dataset is unbalanced, thus the number of graphs differ according to the label. That could mean that the GNN model may be more specialized to learn from a certain graph structure than other ones.

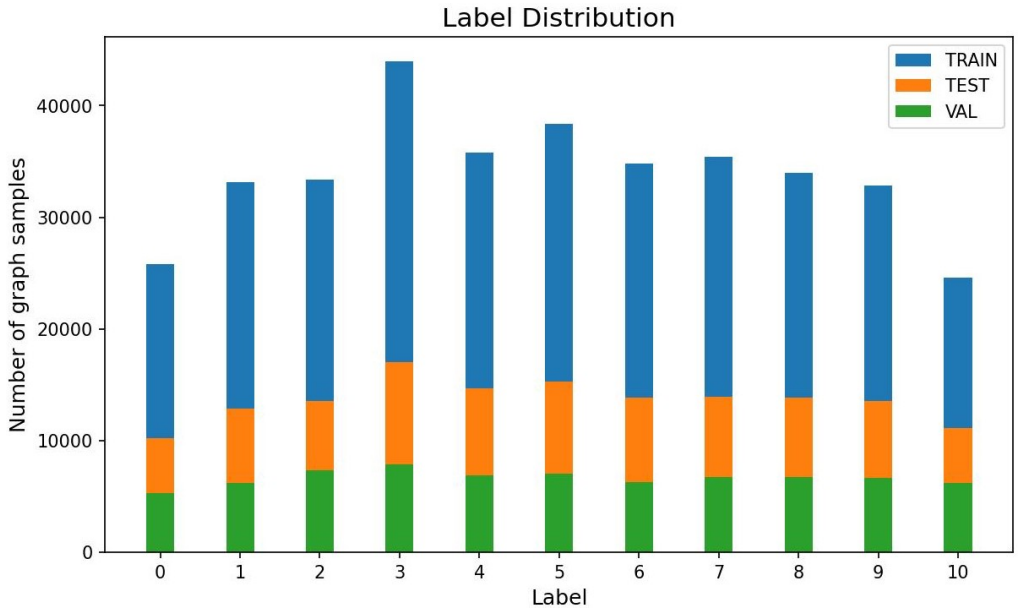

Figure 4.10: Distribution of the Graph Dataset.

		Set			
		Train	Test	Val	Tot
Label	0	15596	4933	5271	25800
	1	20296	6692	6165	33153
	2	19847	6257	7300	33404
	3	26943	9164	7872	43979
	4	21126	7821	6869	35816
	5	23104	8265	7008	38377
	6	20991	7573	6245	34809
	7	21505	7191	6732	35428
	8	20141	7174	6698	34013
	9	19345	6898	6614	32857
	10	13479	4898	6214	24591
	Tot	222373	76866	72988	372227

Table 4.2: Number of graphs per label and set.

Chapter 5

Graph Neural Network Model

The preprocessing step, as elaborated in chapter 4, involves the creation of graph-based structures derived from an event-based dataset. A portion of these generated graphs serves the purpose of fine-tuning model parameters through a learning process, while the remaining subset is reserved for testing and validation. To effectively classify any graph, representing an aspect of the hand gesture, a Graph Neural Network (GNN) model becomes essential. The model's performance is determined by its accuracy, defined as the ratio of correct classifications to the total number of graphs. It's important to note that the training process, like the preprocessing phase, is also characterized by several hyperparameters, each of which can exert significant influence on the model's overall performance.

5.1 Model architecture

The architecture of the Graph Neural Network model consists of four graph convolutional layers, each followed by a batch normalization layer and a ReLU activation function. These are succeeded by a pooling layer and a Multi-Layer Perceptron (MLP) comprising three linear layers. The output of the MLP layer produces an 11-element vector, which represents the graph classification.

The next figure shows the block diagram of the model architecture.

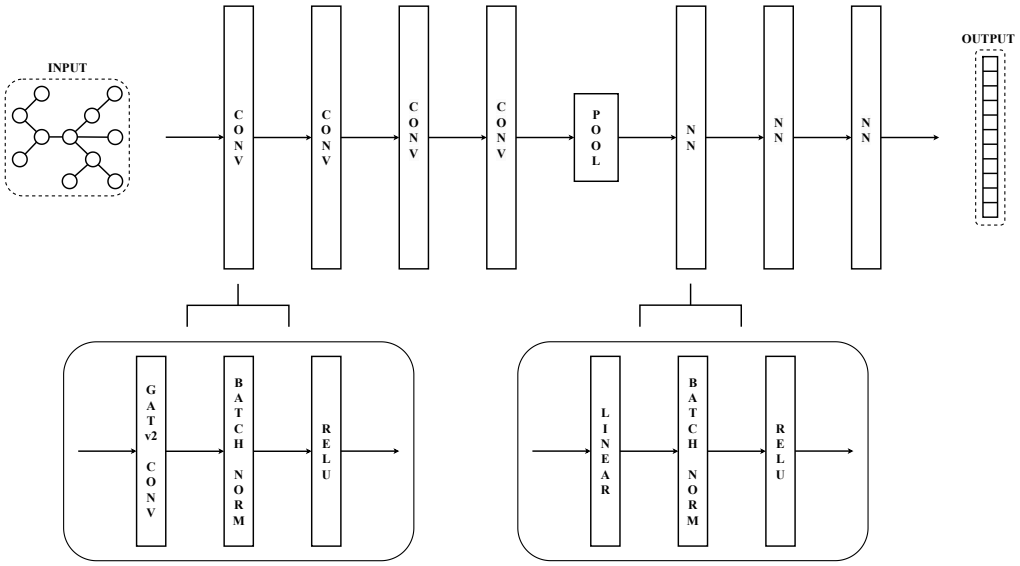

Figure 5.1: Model Architecture.

The input node feature vector corresponds to the polarity of each event, resulting in a dimensionality of 1. The first graph convolutional layer enhances the feature dimension of each node from 1 to 32, maintaining this dimension throughout the subsequent layers. Consequently, the second, third, and fourth convolutional layers receive and output node embeddings with a dimensionality of 32.

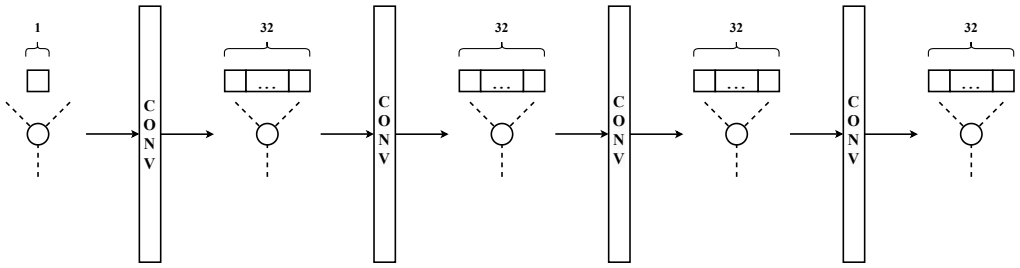

Figure 5.2: Node feature vector dimensionality through convolutional layers.

5.1.1 GATv2 Convolution

This work uses the **GATv2** (Graph Attention Network v2) graph convolutional layer presented by Brody et Al. in 2021 ([4]), which is an improvement of the original GAT graph convolution introduced by Veličković et Al. in 2017 ([5]). An **attention mechanism** enables the model to focus on the most relevant aspects of the input data by assigning varying weights to neighboring nodes: this means that nodes within a neighborhood can possess varying degrees of importance. According

to this, the adjacency matrix can be seen as a weighted structure, where connections between two nodes are not necessarily equal to 1.

The convolutional model can be applicable for both transductive and inductive problems, thus including tasks where the model works on data which are completely unseen during the training process.

The GATv2 node-wise formulation is presented in equation 5.1.

$$x_i^{(l+1)} = \sigma(\sum_{j \in \tilde{N}} \alpha_{ij} W^{(l)} x_j^{(l)}), \quad \tilde{N} = N(i) \cup \{i\} \tag{5.1}$$

The α_{ij} parameter indicates the importance of the message exchanged between nodes i and j and it's determined by a learnable attention mechanism represented by a single-layer feed-forward neural network parameterized by a weight vector $a \in \mathbb{R}^{F'}$.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(a^T LeakyReLU(W^{(l)} \cdot [x_i || x_j || e_{ij}]))}{\sum_{k \in \tilde{N}} exp(a^T LeakyReLU(W^{(l)} \cdot [x_i || x_k || e_{ik}]))} \tag{5.2}$$

The GATv2 convolution, as shown in equation 5.2, enables the incorporation of edge features, enhancing its overall capabilities. In this case, an additional weight matrix for the edge features becomes necessary to align the edge features dimensionality with that one of the output node features.

Furthermore, PyTorch Geometric provides the flexibility to choose whether shared weight matrices should be employed. In the convolutional model utilized in this study, shared matrices are not used. Consequently, two distinct matrices are applied, one to the source node and the other one to the target node for each edge. In this configuration, the equation 5.1 changes as follows:

$$x_i^{(l+1)} = \sigma(\alpha_{ii} W_{ii}^{(l)} x_j^{(l)} + \sum_{j \in N} \alpha_{ij} W_{ij}^{(l)} x_j^{(l)}) \tag{5.3}$$

It's important to note that along each weight structure, i.e. the weight matrices for node, edge features and the attention mechanism, also a bias vector is being used by the GATv2 convolution operator.

5.1.2 Pooling Method

The goal of the convolutional layers is to update the node features by gathering information from the k-hop neighborhood, also changing the dimensionality of the node feature vector, as previously described. Anyway, the data structure after the last convolutional layer is still a graph having an arbitrary shape and dimension

and it cannot be delivered to a MLP layer which instead only accepts an input fixed-size structure. Consequently, a pooling layer is essential to both limit the size of the output graph and transform its irregular structure into a vector with predefined dimensions.

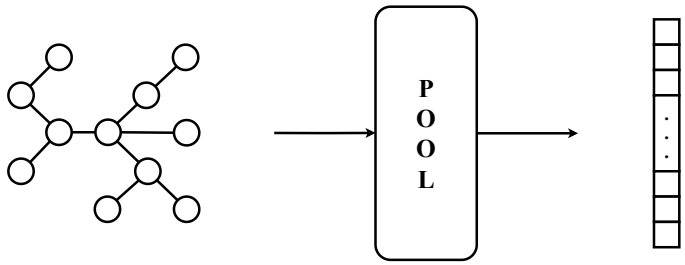

Figure 5.3: Input and Output of the pooling layer.

The chosen pooling method consists on segmenting the spatio-temporal space, in which the graph is embedded, into uniformly sized 3D **voxels**; each one represents a distinct sub-space within the original spatio-temporal domain. These voxels can cluster a specific number of nodes and, if any, a **max-pooling** operation is executed. Specifically, when one or multiple nodes are situated within the same sub-space, a maximum-wise operation is performed across all node embeddings within that sub-space.

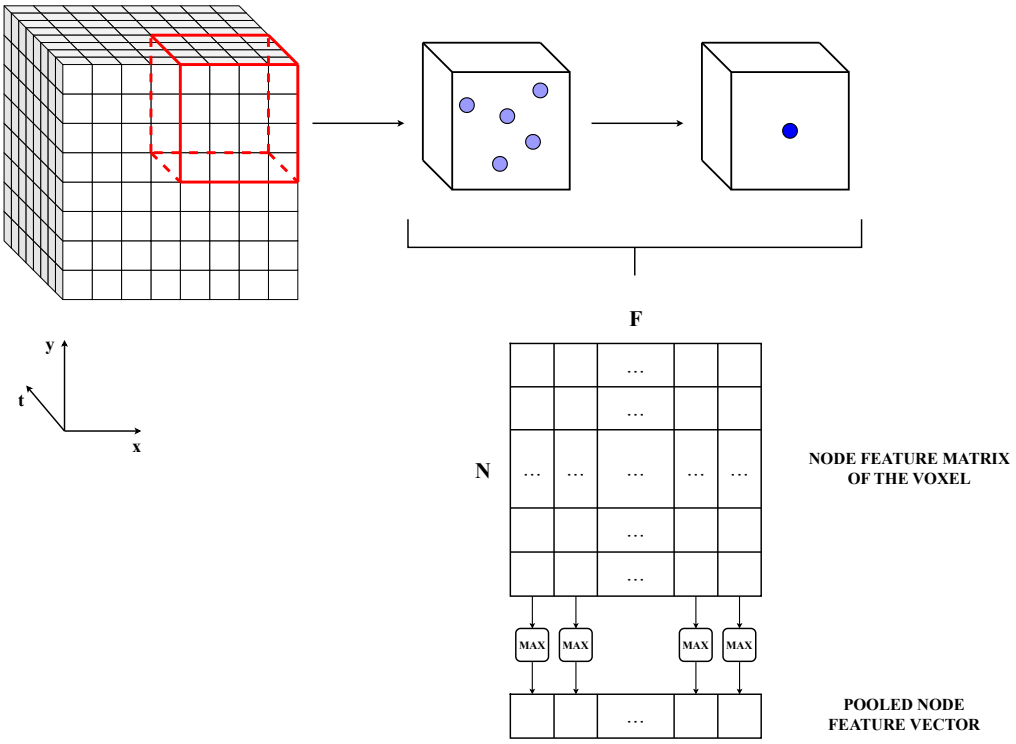

Figure 5.4: Pooling Method. N represents the number of nodes inside a voxel, whereas F is the embedding dimensionality.

Hence, each voxel outputs a vector with the same node feature dimensionality as the final convolutional layer, which is 32. Subsequently, the outputs coming from all voxels are concatenated to produce an one-dimensional array. This implies that the output vector dimensionality after the pooling layer is equal to the product of the number of voxels and the embedding dimensionality.

$$n_{out} = n_{voxels} \cdot n_{features} \tag{5.4}$$

The spatio-temporal space that embeds the event-graph measures 128x128x32, while the chosen voxel size is 32x32x8: this results in a total of 64 voxels. Given that the embedding dimensionality is set to 32, the output of the pooling layer consists of $64 \cdot 32 = 2048$ elements.

It's important to note that this number can change by varying the voxel dimension and/or the embedding dimensionality of the last convolutional layer.

5.1.3 MLP

The pooling layer outputs a fixed-size vector to be delivered to a Multi-Layer Perceptron. The purpose of the MLP is to compress the output of the pooling layer into a vector corresponding to the graph classification. Specifically, it takes the 2048-dimensional vector from the pooling layer and reduces its dimensionality to eleven, which matches the number of classes.

The MLP is composed by three linear layers: the first one performs a compression from 2048 to 512 dimensions, the second one from 512 to 128, while the third one further reduces it from 128 to 11 dimensions. This compression process is achieved using three weight matrices and three bias vectors.

Actually, the first and the second linear layers of the MLP are both followed by a batch normalization layer and a ReLU activation function (see figure 5.1). The third layer doesn't need to be followed by any of those layers since it directly outputs the final classification.

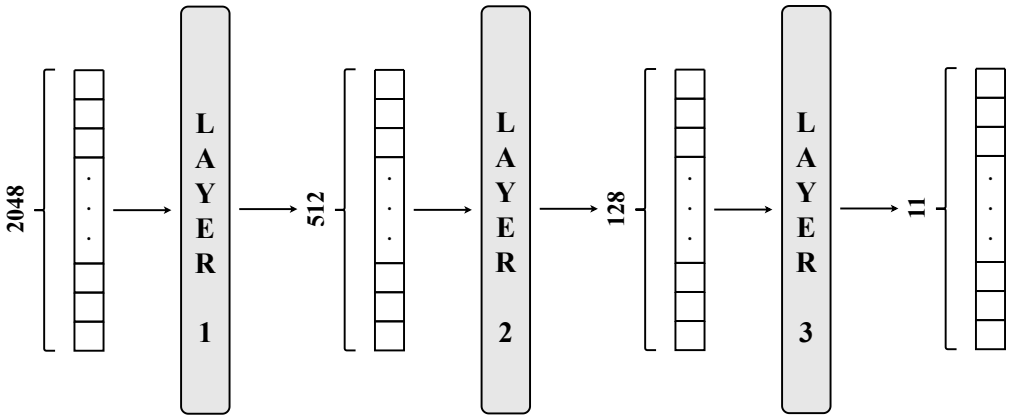

Figure 5.5: MLP dimensionality through the linear layers.

5.1.4 Model Size

The model architecture is composed by several layers and each of them is characterized by more structures, as described in the previous sections. These structure contain a certain number of parameters (or weights), which are optimized during the learning process. The pooling layer, differently from the convolutional, batch normalization and MPL layers, doesn't require any learning parameters.

The GATv2 convolution layer comprises the following components, each associated with its respective bias vector:

- $W_{ii} \in \mathbb{R}^{F,F'}$: weight matrix for self loop message;

- $W_{ij} \in \mathbb{R}^{F,F'}$: weight matrix for neighboring messages;
- $W_e \in \mathbb{R}^{F_e,F'}$: weight matrix for aligning edge features dimension to the node features dimension during the attention mechanism;
- $a \in \mathbb{R}^{F'}$: attention vector;
- $B_{ii} \in \mathbb{R}^{F'}$: bias vector associated to W_{ii};
- $B_{ij} \in \mathbb{R}^{F'}$: bias vector associated to W_{ij};
- $B_e \in \mathbb{R}^{F'}$: bias vector associated to W_e;
- $a_b \in \mathbb{R}^{F'}$: bias vector associated to a.

However, it's important to note that the dimensions of the embeddings vary throughout the network. Consequently, the size of the arrays differs for each convolution layer. Specifically:

- for the first layer: $F = 1$, $F' = 32$, $F_e = 3$;
- for the three subsequent layers: $F' = 32$, $F' = 32$, $F_e = 3$.

This leads to a total of 3360 parameters only needed for convolutional layers.

The batch normalization layers, placed after each convolutional layer, normalizes the convoluted data. This is done by computing the mean and variance of each mini-batch and applying a scaling and a shifting to get zero mean and unitary variance in that mini-batch. The parameters to be learned are therefore $\gamma \in \mathbb{R}^{32}$ and $\beta \in \mathbb{R}^{32}$, corresponding to the scaling and shifting vectors. Given four batch normalization layers, a total of 256 parameters are involved.

The MLP layer is instead composed by the weight matrices $W_1 \in \mathbb{R}^{2048,512}$, $W_2 \in \mathbb{R}^{512,128}$, $W_3 \in \mathbb{R}^{128,11}$, along with the bias vectors $B_1 \in \mathbb{R}^{512}$, $B_2 \in \mathbb{R}^{128}$, $B_3 \in \mathbb{R}^{11}$. Also the batch normalization layers through the MLP linear layers must be taken into account. It follows that there are additional parameters described by the matrices $\gamma_1 \in \mathbb{R}^{512}$, $\gamma_2 \in \mathbb{R}^{128}$ and $\beta_1 \in \mathbb{R}^{512}$, $\beta_2 \in \mathbb{R}^{128}$. This results in combined total of 1117451 parameters.

Finally, the network size accounts for 1121067 parameters.

5.2 Training Hyperparameters

The model is tuned through a training process. This training loop, as briefly described in section 2.1.1, is responsible for optimizing the model parameters and is

also characterized by several hyperparameters that influence the weight optimization.

One crucial aspect is the selection of the **loss function**, which determines the error to be back-propagated for parameter adjustment. In this work, the **Cross Entropy** (**CE**) loss function is employed. CE is commonly used when dealing with multi-dimensional arrays that require comparison. The model's output, in this case, consists of an 11-dimensional vector, which is compared with an array of the same dimension, that is the graph label, to compute the error value.

$$\begin{aligned} l(x,y) &= \frac{\sum_{n=1}^{N} l_n}{N} \\ l_n &= -\sum_{c=1}^{C} log \frac{exp(x_{n,c})}{\sum_{i=1}^{C} exp(x_{n,i})} y_{n,c} \end{aligned} \tag{5.5}$$

The two equations in 5.5 illustrate how PyTorch performs the Cross Entropy loss function. In these equations, x represents the model's classification, y corresponds to the label, C denotes the number of classes, and N stands for the number of training samples per batch.

In fact, during the training process, the model does not receive one sample at a time. Instead, it randomly selects a certain number of samples, often referred to as a **batch** or **mini-batch**, from the dataset. Consequently, the loss value is calculated not on an individual sample but for the entire batch. This batch-wise approach aids in improving generalization. The specific size of a batch is known as the **batch size**. In this work, a batch size of 16 is employed.

Another important parameter is the **learning rate** which, as highlighted in 2.1, is a value proportional to the gradient of the loss. Initially, a common learning rate of 1e-3 is employed throughout the training process. However, a widely adopted technique involves the use of a **learning rate scheduler**, which dynamically adjusts the learning rate value during training, epoch per epoch. Several types of learning rate schedulers are available and, in this study, the **Cosine Annealing Warm Restart** ([6]) method is utilized.

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \left(1 + cos\left(\frac{T_{cur}}{T_i}\right)\right) \tag{5.6}$$

The equation 5.6 describe the learning rate behavior. The η_t, η_{max} and η_{min} parameters refer to the current, initial and minimum learning rate, whereas T_{cur} is the number of epochs since the last restart and T_i is the number of epochs between two warm restarts. This approach schedules the learning rate to gradually decrease it with each epoch, exhibiting a cosine-like behavior until it reaches a minimum value, set to 1e-6. After a defined number of iterations, or epochs, the learning rate

resets to its initial value. The number of epochs is set to 200, meaning that the model is trained, tested and validated over the whole dataset for 200 times. The number of epochs needed to make the learning rate restart from its initial value is set to 15.

The used **optimizer** is the **AdamW** ([7]), which is an updated version of the common Adam optimizer ([8]). A **weight decay**, consisting in a regularization method that penalizes the loss, is set to a common value of 5e-4.

Actually, another regularization technique is used during the training loop: the **dropout**. It's applied to the MLP layer when dealing only with training samples and consists on randomly removing neurons from each layer with a certain probability. In this work, the dropout probability is set to 0.5.

Summarizing:

- loss function: Cross Entropy;
- batch size: 16;
- learning rate: 1e-3, with Cosine Annealing scheduler;
- number of epochs: 200;
- optimizer: AdamW;
- weight decay: 5e-4;
- dropout: 0.5.

5.3 Results

All experiments within this work have been conducted using the GeForce RTX 4090 GPU. The model definition and the training loop have been elaborated mostly with the contribution of two Python libraries: **PyTorch** and **PyTorch Geometric** (**PyG**), both renowned machine learning frameworks for constructing and training deep neural networks. Specifically, PyG serves as an extension of PyTorch for dealing with Graph Neural Networks.

Throughout the experiments, various configurations were employed, varying across preprocessing and training hyperparameters, neural network depth, the selected convolutional model, and other setup parameters. The objective was a gradual improvement in accuracy on the test set. Therefore, the experimental processes involved fine-tuning the model configuration, preprocessing, and training hyperparameters. The final setup values are highlighted in section 4.2 and 5.2, as well as

final model architecture in section 5.1, and in the table below are represented by *Setup 4*.

	TW	**TNR**	**LR**	**LRS**	**BS**	**OPT**	**CONV**	**#CONV**	**#LIN**	**ACC (%)**
Setup 1	8e5	128	1e-3	Poly2	128	Adam	GCN	4	2	73.87
Setup 2	1e6	128	1e-3	Poly2	128	Adam	GAT	3	2	80.39
Setup 3	1e6	32	1e-3	CAWR	16	AdamW	GATv2	3	2	87.50
Setup 4	1e6	32	1e-3	CAWR	16	AdamW	GATv2	4	3	**90.31**

TW: Time Window, in μs;
TNR: Time Normalization Range;
LR: Initial Learning Rate;
LRS: Learning Rate Scheduler, Poly2: Polynomial of 2^{nd} order;
BS: Batch Size;
OPT: Optimizer;
CONV: Graph Convolutional Model;
#CONV: Number of Graph Convolutional Layers;
#LIN: Number of Linear Layers;
ACC: Test Accuracy.

Table 5.1: Best test accuracy values for each graph convolutional model.

Table 5.1 shows the best test accuracy results obtained by using three different graph convolutional models: GCN ([9]), GAT and GATv2. It's crucial to highlight that not only a change in the graph convolution impacts the final accuracy, but the variation in setup, including preprocessing and training hyperparameters, also plays a significant role. While not all setup parameters are listed in the table, only those with the most substantial impact on the final result are included. A more comprehensive table detailing how accuracy results fluctuate with the tuning of such hyperparameters is provided in section 7 (Table 7.1).

In any case, the experiments reveal a superior performance of the system when utilizing GAT convolution (*Setup 2*) over GCN (*Setup 1*), and notably, its enhanced version, GATv2 (*Setup 3* and *Setup 4*). Furthermore, the model employing GATv2 demonstrates improved results with an increased depth in the neural network, both in the graph convolutional and the MLP layers. Transitioning from 3 to 4 graph convolutional layers and from 2 to 3 linear layers, in fact, enables the system to achieve a **test accuracy** of **90.31%**, that is the highest result attained in this study.

Part III

Conclusion

Chapter 6

Conclusion

In the course of this study, a comprehensive exploration of Graph Neural Network models for Neuromorphic Vision has been conducted. The primary focus was on developing a method working on the DVS Gesture dataset for gesture classification. The research journey mainly faced three phases: the conversion of events to graphs, the definition of the model architecture, and the tuning of hyperparameters, all aimed at enhancing the overall system performance. Therefore the core objective of this thesis has been to identify the optimal setup configuration that lead to the highest test accuracy. This involved carefully looking at different setups, each playing a part in how the model works together. The culmination of these efforts resulted in the identification of the most effective setup, achieving a test accuracy of 90.31%.

This work underscores the significance of thoughtful setup configuration and hyperparameter tuning in optimizing the performance of Graph Neural Network models for event stream classification.

6.1 Possible improvements

Design Space Exploration In this study, various setup configurations were explored. Numerous hyperparameters were adjusted to enhance the system's performance, but this process may not have yielded the optimal solution. A more effective strategy involves exploring the preprocessing space. This can be accomplished by fixing the model and systematically varying each preprocessing hyperparameter to identify the values that yield the best accuracy. Once the optimal preprocessing configuration is determined, a similar process can be applied to training hyperparameters. This method can also be extended to the model architecture

by experimenting with the number of convolutional and linear layers, as well as the embedding dimensionality across layer or the pooling size dimensions.

Implementing such an approach is expected to result in optimal accuracy, surpassing the performance achieved in this work.

6.2 Future steps

Validation on other datasets This work has focused on evaluating the process from event to graph conversion and model architecture using the DVS Gesture dataset exclusively. However, it's important to note that the effectiveness of this approach can be further validated by considering additional datasets. Several neuromorphic datasets are accessible, such as N-Cars, N-Caltech101, Gen1 for object detection and recognition, N-Mnist for digit recognition, among others. Expanding the evaluation to these datasets would provide a broader understanding of the method's generalizability and effectiveness across different tasks and scenarios.

Hardware implementation This study primarily focuses on the training aspect of a Graph Neural Network model, serving as the initial phase in a broader project aimed at realizing a hardware implementation of the neural network. However, before effectively transitioning to the digital representation, another crucial step needs consideration, that is the quantization. During the training process, PyTorch utilizes 32-bit values to represent the model's parameters. To optimize the area and power consumption of the subsequent digital circuit, it becomes essential to reduce the bit-length of the model's weight. This reduction can be achieved through two main approaches: post-training quantization and quantization-aware training. It's essential to note that this quantization process may impact accuracy, prompting the use of specialized frameworks to mitigate performance losses. Once the quantized parameters are obtained, the hardware implementation of the neural network can be realized. Given the dataflow nature of this system, High-Level Synthesis tools emerge as the optimal choice for hardware implementation.

Chapter 7

Appendix

7.1 Comparison between different setups

	TW	TS	TNR	LR	LRS	BS	OPT	CONV	#CONV	#LIN	ACC (%)
Setup 1	1e5	1e5	128	1e-3	Poly2	256	Adam	GCN	4	2	55.42
Setup 2	1.5e5	7.5e5	128	1e-3	Poly2	256	Adam	GCN	4	2	59.20
Setup 3	5e5	5e5	128	1e-3	Poly2	256	Adam	GCN	4	2	72.14
Setup 4	8e5	7.5e4	128	1e-3	Poly2	128	Adam	GCN	4	2	73.87
Setup 5	8e5	7.5e4	128	1e-3	Poly2	128	Adam	GAT	4	2	74.60
Setup 6	1e6	5e4	128	1e-3	Poly2	128	Adam	GAT	3	2	80.39
Setup 7	1e6	5e4	128	1e-3	Poly2	128	AdamW	GATv2	3	2	81.28
Setup 8	1e6	5e4	64	1e-3	Poly2	128	AdamW	GATv2	3	2	82.67
Setup 9	1e6	5e4	32	1e-3	Poly2	128	AdamW	GATv2	3	2	83.20
Setup 10	1e6	5e4	32	1e-3	Poly2	16	AdamW	GATv2	3	2	85.52
Setup 11	1e6	5e4	32	1e-3	Poly1	16	SGD	GATv2	3	2	71.65
Setup 12	1e6	5e4	32	1e-3	CAWR	16	AdamW	GATv2	3	2	87.50
Setup 13	1e6	5e4	32	1e-2	CAWR	16	AdamW	GATv2	3	2	80.52
Setup 14	1e6	5e4	32	1e-3	Multistep	16	AdamW	GATv2	3	2	86.79
Setup 15	1e6	5e4	32	1e-3	CAWR	128	AdamW	GATv2	3	2	84.53
Setup 16	1e6	5e4	32	1e-3	CAWR	16	AdamW	GATv2	4	2	88.05
Setup 17	1e6	5e4	32	1e-3	CAWR	16	AdamW	GATv2	4	3	88.97
Setup 18	1e6	2e4	32	1e-3	CAWR	16	AdamW	GATv2	4	3	**90.31**

TW: Time Window, in μs;
TS: Time Step, in μs;
TNR: Time Normalization Range;
LR: Initial Learning Rate;
LRS: Learning Rate Scheduler, Poly<n>: Polynomial of $< n >^{nd}$ order;
BS: Batch Size;
OPT: Optimizer;
CONV: Graph Convolutional Model;
#CONV: Number of Graph Convolutional Layers;
#LIN: Number of Linear Layers;
ACC: Test Accuracy.

Table 7.1: Comparison between different setups.

Various configurations were explored, as outlined in Section 5.3. Table 7.1 illustrates how distinct setups can impact the system's performance. In each column of the table, highlighted cells signify a change in that particular hyperparameter compared to another setup sharing the same highlighted column. Upon initial observation, it becomes apparent that variations in the time window value have a significant impact on the final accuracy, with higher values correlating to improved accuracy. Generally, optimizing results involve reducing the time normalization range and batch size values, opting for the Cosine Annealing learning rate scheduler over Polynomial or Multi-Step alternatives, and selecting the AdamW optimizer instead of Adam and SGD (Stochastic Gradient Descent). The initial learning rate plays also an important role in the training process; a higher value compared to the used one results in a noticeable percentage decrease. The last two setups in the table highlight that decreasing the Time Step value, effectively increasing the number of samples in the training process, contributes to improved outcomes. The effect of the graph convolutional model, as well as depth of the graph neural network, has been already outlined in section 5.3.

Bibliography

[1] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jörg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based vision: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(1):154–180, 2022.

[2] Qinyi Wang, Yexin Zhang, Junsong Yuan, and Yilong Lu. Space-time event clouds for gesture recognition: From rgb cameras to event cameras. pages 1826–1835, 01 2019.

[3] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[4] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*, 2021.

[5] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[6] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[7] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[10] Simon Schaefer, Daniel Gehrig, and Davide Scaramuzza. Aegnn: Asynchronous event-based graph neural networks. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12361–12371, 2022.

[11] 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 *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 934–943, 2021.

[12] Yin Bi, Aaron Chadha, Alhabib Abbas, Eirina Bourtsoulatze, and Yiannis Andreopoulos. Graph-based object classification for neuromorphic vision sensing. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 491–501, 2019.