# Object Classification from Event Camera Data

by
Sergio Marin Petersen [5073723]
Serban Blaga [4996666]
Lukas Uptmoor [5011965]
Voxelization, Graph Construction
TU Delft
Code: https://github.com/luptmoor/EV-VGCNN-Reproduction

This blog post is about the reproduction of a script that utilizes a CNN in conjunction with graph construction to classify different points in space and time in a certain manner called ‘events’. The reproduction is based on the article by Y. Deng, H. Chen, H. Liu, and Y. Li titled “A Voxel Graph CNN for Object Classification with Event Cameras”. In this article, the focus is on the principle of event cameras which capture only the changes in brightness at a very high temporal resolution and register each change of brightness as events. The post-processing classification of these events is where a lot of algorithms using artificial intelligence have been developed. This paper explains a method to create a quicker and more precise model which first voxelizes the data and then constructs a graph that is later fed into a convolutional neural network, looking for certain attributes such as position in space and time to fix it into different sections for easier sorting and qualifications.

## Reading Data

The data is recorded using event cameras. Event cameras capture events, which are changes in brightness that occur at a given pixel location, and their corresponding timestamps. This allows event cameras to operate asynchronously, outputting intensity changes as they occur in real-time, with high temporal resolution and low latency. Each pixel in an event camera works as an independent processing unit, allowing it to detect and report changes in brightness asynchronously. When an event occurs, the pixel records the change in brightness, the location of the pixel, and the timestamp of the event. Figure 1 and figure 2 give a representation of how the recorded data looks like using three examples from the Caltech101 dataset.

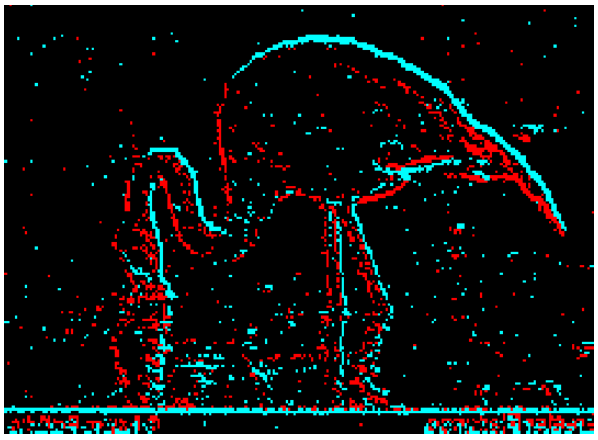


Figure 1: Flamingo

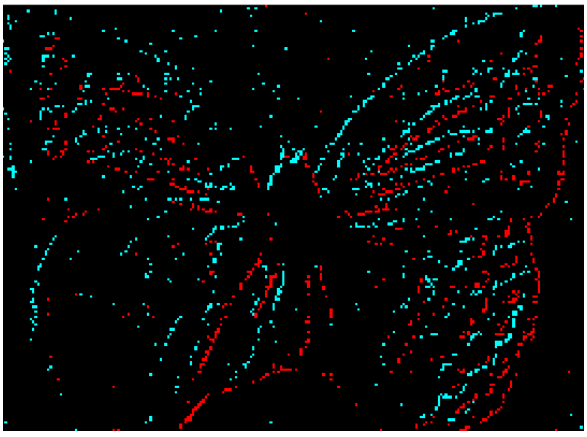


Figure 2: Butterfly

The data is given in a binary format. To properly load this data and transform it into tensors, our code defines a custom PyTorch dataset called *BinaryFileDataset*. This dataset loads binary files from a directory and converts them into PyTorch *Data* objects that can be used as inputs for neural networks.

The *BinaryFileDataset* class takes two arguments: *folder_dir* and *max_length*. *folder_dir* is the path to the directory containing the binary files, and *max_length* is the maximum length of the data sequence that will be returned by the dataset. An *__init__* method of the class sets some instance variables, including the directory path, a list of names which allow us to label each folder of the Caltech101 dataset using one-hot encoding, the maximum sequence length and a list of *Data* objects to hold the data. The class initializes by reading the names of all binary files in the specified directory using the *glob* module, which returns a list of file paths matching the specified pattern (in this case, all files with a .bin extension).

The *__init__* method reads each binary file in the directory and extracts the data points, labels, and timestamps from the file. The data points consist of four arrays: *all_x, all_y, all_p,* and *all_ts*. These arrays contain the x and y coordinates of each data point, a binary flag indicating whether the data point is a pen-up or pen-down event, and a timestamp indicating when the event occurred. These arrays are converted into a PyTorch *Data* object.

The code iterates over each batch in the DataLoader (with a batch size of 16) and pads the tensors in the batch to the same length (the maximum length of any tensor in the batch, which in our case is 5000). It also stacks the padded tensors along the 0th dimension to create a single tensor for the batch. The code prints the shapes of the data and label tensors for each batch.

## Voxelization

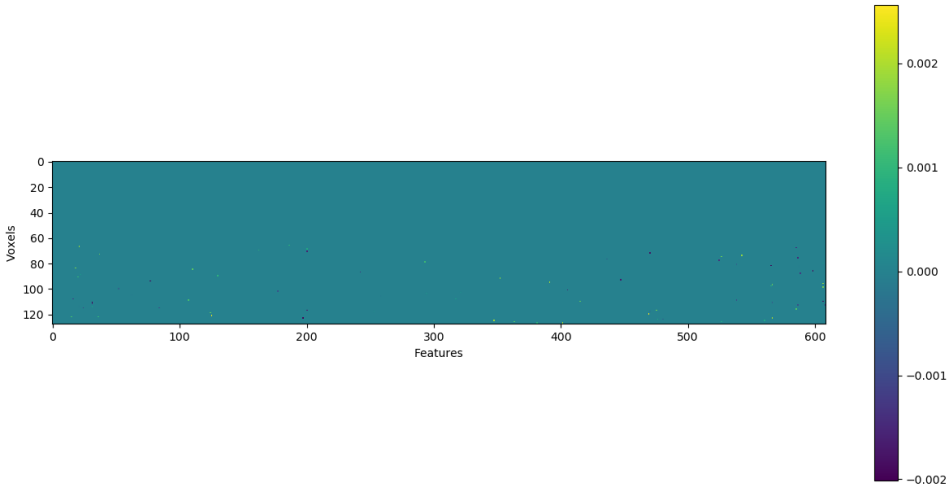
The novelty of the paper is their use of a different graph representation that makes the use of voxel graphs and convolutional neural networks more lightweight. The process initiates with the events registered by the camera and graphed in a three-dimensional way in terms of x, y, and time. Afterwards, this space-time domain is discretized by an evenly-spaced grid, subdividing the domain into voxels, i.e. pixels but in 3D. We have chosen to divide all dimensions into 8 chunks, meaning in total there will be 8x8x8 = 512 voxels. Each voxel has coordinates i, j and k and a list of events that fall within the voxel. The event data originally comes in the form of a point cloud with global coordinates x, y and t, as well as a polarity that is either -1 or 1, depending if the event resembles a decrease or increase in brightness, respectively. The absolute coordinates of the event need to be shifted to the local coordinate system of the respective voxel by subtracting the coordinates of the origin of a voxel.

## Graph Construction

Then, in order to save computational power, a selection of the voxels is made. For this reason, the voxels are sorted by the number of events lying within them. We have chosen to only consider the top 25% most event-rich Np voxels, the rest is dropped. For our case Np = 512/4 =

128. From the remaining voxels, we are now able to construct vertices for a graph. The respective vortex's coordinates are simply the coordinates of the voxel i, j, k and stored in a coordinate vector. The features of each vortex are computed somewhat more complexly, namely by the underlying equation

$$F(x,\ y)\ =\ \sum_{i}^{N_p} p_i^{in}\delta(x\ -\ x_i^{in},\ y\ -\ y_i^{in})t_i^{in} \qquad (1)$$




This 2D feature map is then flattened to obtain a 1D feature vector. Repeating this for all the voxels yields again a 2D map. An example feature map for an image labeled as 'accordion' is given above. The y-axis shows the 128 voxels that have been constructed from the image and along the x-axis the values of the feature vectors are shown. Note that this tensor is very sparse and only occasionally contains a non-zero entry. This results from the nature of equation 1, which returns 0 by default unless a special condition is met (delta function). Furthermore, numbers are relatively small, because polarity and delta function both have an absolute value of 1 for the non-zero case, so the only value determining the scale of the features is the t-coordinate, which is very small since the input videos are far less than one second long.

The coordinate vector is then appended to the feature vector to store both conveniently in a Dataset instance for training. This is possible because both tensors have the same shape in dimension 0 (Np).

## Ground-Truth Labels

To properly classify the data in the dataset, labels are used. The labels are used as ground-truth labels, which represent the true classification of each input sequence. To obtain the labels, a label directory is created in the *BinaryFileDataset* class, where each folder from the Caltech101 dataset was assigned a value from 0 to 100. For example, the folder 'accordion' received the

value 0 since it is the first folder; the folder ‘airplanes’ received the value 1 since it is the second folder and so on. Then, an array of 101 zeros was created for each folder, having a single value of 1 on the position specified by the folder value. Therefore, the labels are one-hot encoded. These labels are then appended to *self.labels* and are converted to tensors by using *‘self.labels = torch.tensor(self.labels)’.* To make sure that, while iterating over the entire dataset, we will not get more labels than needed (since we are working with batches for each folder), we only take into consideration the labels for the 1st batch (so when *batch_idx == 0*).

Together with the pre-processed data, the labels are fed into a DataLoader with a specified batch size of 16. This allows easy iteration during training.

The KNN algorithm is used for classification to sort the data. The function first checks whether *cosine* is True. If it is, it computes the cosine similarity between *xyz* and *new_xyz* using the *cosine_similarity* function. If cosine is *False*, it computes the square distance between *xyz* and *new_xyz* using the *square_distance* function.

The function then finds the *nsample* nearest neighbors for each query point by using the topk function from PyTorch to find the indices of the *nsample* smallest distances in *dist* along the last dimension.

## Network Architecture

After all this pre-processing, the graphed data in the form of feature vector and coordinate vector can actually be fed into the network. After getting in contact with the authors of the paper, they agreed to share their code for this part and we would like to express our gratitude once more here.

The two vectors are fed into five multi-scale feature relational layers (MFRLs) with random pooling in between them. A MFRL is a form of 2D convolutional layer that aims to detect features at different scales by varying the kernel size. Then the feature maps are flattened and fed through a 1D convolutional layer with Batch Normalization and leaky rectified linear unit (leaky ReLU) as activation function. Finally, there is a classifier consisting of three fully-connected layers with Batch Normalization and leaky ReLU. For exact hyperparameters such as layer and channel size, please consult the code. The network is able to distinguish between objects of 101 classes.

## Training

Due to time constraints and problems with setting up the code on Google Cloud, we could not finish training the model on time. For the training loop we recommend a regularization method like weight decay or early stopping to prevent overfitting. Moreover, we suggest using the Adam optimizer for fast and efficient gradient descent.