HYPERDIMENSIONAL COMPUTING FOR GESTURE RECOGNITION USING A DYNAMIC VISION SENSOR

by

Abdelrahman Aboeitta

2023

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Artificial Intelligence (AI) has experienced rapid growth in recent years and is receiving unprecedented attention due to the impressive results achieved in various domains, notably computer vision and natural language processing. Two of the key technologies that enabled these groundbreaking results were (1) *Convolutional Neural Networks* (CNNs), designed for image and video processing, and (2) *Backpropagation*, an algorithm for training AI models. However, despite their remarkable capabilities, these technologies face criticism for several reasons. First, CNNs are limited in their applicability to numerous real-world problems, primarily due to their high computational and memory requirements. Furthermore, backpropagation is criticized for being biologically implausible, as it fails to represent how the human brain works.

To address the limitations mentioned above, this thesis introduces a novel neuromorphic approach that exploits two biologically-inspired technologies: (1) *Dynamic Vision Sensor* (DVS), an event-based camera that consumes only 0.1% of the power required by traditional cameras while generating thousands of frames, and (2) *Hyperdimensional Computing* (HDC), a learning algorithm that avoids backpropagation by imitating the brain. By integrating technologies inspired by the human brain, the primary goal of this research is to develop more efficient and adaptable AI systems that can handle various real-world problems, overcoming the constraints faced by CNNs and backpropagation. To demonstrate the effectiveness of this approach, I propose an AI classifier for recognizing two simple hand gestures built using HDC technology and trained using data captured by a DVS. The model managed to achieve a high accuracy rate of 99%. Despite its simplicity, my approach serves as a compelling proof-of-concept, showcasing the potential of combining biologically-inspired technologies (DVS and HDC) to achieve substantial gains in efficiency and performance.

Chapter 1

INTRODUCTION

Recently, AI has made significant leaps in various domains, particularly in developing classifiers for multiple tasks. In addition, the introduction of event-based sensors, such as the Dynamic Vision Sensor (DVS), has given rise to new opportunities for AI to process and interpret data more efficiently and accurately [13]. This honors thesis focuses on developing an AI classifier model built using TorchHD, a high-dimensional computing library that extends PyTorch and is trained using events from a DVS. This model's primary objective is to accurately classify two distinct hand gestures in real-time.

Event-based cameras are biologically inspired neuromorphic systems that generate asynchronous events upon detecting changes in the intensity of the visual input [13]. These sensors offer several advantages, such as reduced redundancy, low latency, and high temporal resolution, making them ideal for real-time applications [12].

Hyperdimensional computing (HDC) is a computing framework that uses high-dimensional vectors, typically 10000 dimensions, to perform various tasks, such as classification, association, and recognition [10]. HDC has demonstrated promising results in multiple domains, including natural language processing, cognitive computing, and robotics [12] [10]. Combining the advantages of event-based sensing and HDC can potentially lead to a more efficient and robust AI classifier model.

This thesis presents a novel AI classifier model that exploits the TorchHD library and the advantages of event-based data from a DVS. The primary objective is to accurately classify two distinct hand gestures in real-time, demonstrating the efficiency and potential of this combination. By incorporating the findings from previous research

on event-based cameras [5] [13] and HDC [6] [10], this thesis aims to contribute to the ongoing efforts in the development of AI classifiers for a variety of tasks, particularly those that demand low latency and high temporal resolution.

Chapter 2

BACKGROUND

2.1 Dynamic Vision Sensors

Neuromorphic Dynamic Vision Sensors, also called event-based cameras, are asynchronous image sensors that offer several advantages over traditional frame-based cameras. DVSs are inspired by the human eye: instead of capturing entire frames at a fixed rate, they asynchronously measure per-pixel changes in brightness, and each individual pixels independently produce an output only if there is a local change in measured brightness [5]. Hence, the output of a DVS is not a sequence of images but a continuous stream of asynchronous events, which allows DVS to capture motion better than a classical camera [8]. Figure 2.1 shows a comparison of the output of a conventional camera and an event-based camera.

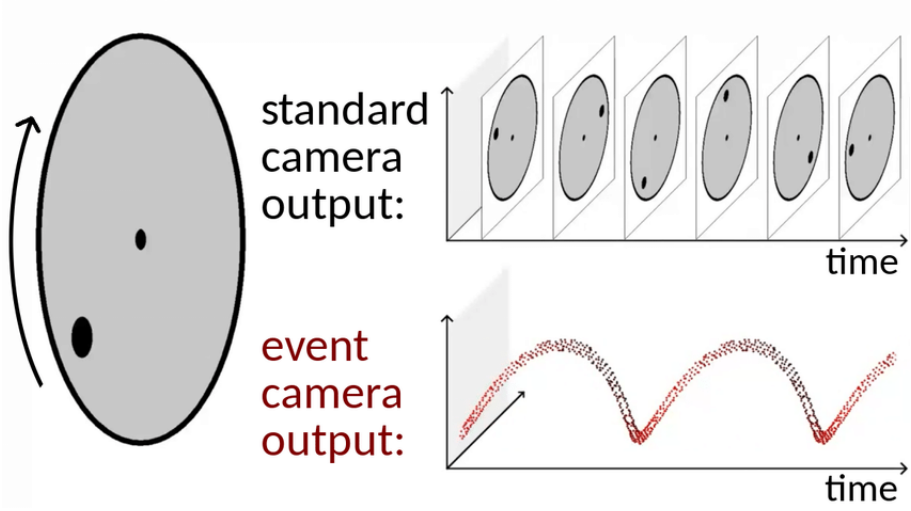

Figure 2.1: Output of a conventional camera vs event-based camera [11]

Figure 2.2 provides a better understanding of the differences between frames and events. On the left, a grayscale output frame from a conventional camera is shown. The output events generated by a DVS imposed on a frame are displayed on the right. The comparison demonstrates that DVS technology can detect and capture changes in brightness at the level of individual pixels. The red indicates off events, and the green indicates on events.

Figure 2.2: traditional camera frame vs. events on the top of a frame

These events provide a more efficient way to represent visual information and are highly suited for applications that require fast and accurate visual processing. Compared to a traditional camera running at about 30-60 fps, a DVS offers more attractive properties, which are listed below:

- **High dynamic range** (140dB vs. 60dB): DVSs can capture scenes with high contrast ratios, allowing them to operate effectively in diverse lighting conditions [5].

- **Low power consumption:** They only output data whenever there is a change in the pixel brightness, allowing them to achieve ultra-low power consumption (1mW vs. 1W) [5].

- **Low latency:** They capture events with latency as low as 1 microsecond, resulting in reduced motion blur [5].

- **High temporal sampling:** Also in the order of microseconds.

Accordingly, these properties allow the DVS to generate visual data equivalent to thousands of frames per second (fps) while consuming only 0.1% of the power required by traditional digital cameras [5]. This makes a DVS ideal for challenging robotics and wearable applications that require high-speed processing and high dynamic range, such as tracking, navigation, and collision avoidance in autonomous vehicles [5].

2.2 Hyperdimensional Computing

2.2.1 Overview

Hyperdimensional Computing (HDC) [10] is a computing framework that has gained significant interest in recent years due to its potential to outperform conventional computing approaches in terms of robustness and energy efficiency [14]. Inspired by the brain's massive circuit of neurons and synapses, HDC exploits very high-dimensional vectors, called *hypervectors*, attempting to mimic the brain's ability to store and retrieve information [10]. Mathematically speaking, the main idea behind *hypervectors* is to represent information $x \in \mathcal{X}$ onto a *hyperspace* $\mathcal{H}$ with d dimensions, typically the *hyperspace* is binary $\mathcal{H} = \{0, 1\}^d$ or bipolar $\mathcal{H} = \{-1, 1\}^d$ with $d \approx 10,000$ dimensions. [10] [6]. This high-dimensionality characteristic of hypervectors enables combining multiple hypervectors into a single one through vector space operations (discussed in 2.2.2) while retaining the information content from both [14].

In cognitive tasks, the encoding function maps similar objects from input space $\mathcal{X}$ to similar hypervectors in $\mathcal{H}$. Hypervector similarity is measured using cosine similarity, denoted by $\delta : \mathcal{H} \times \mathcal{H} \rightarrow \{\sigma \in \mathbb{R} \mid -1 \leq \sigma \leq 1\}$ [10] [6]. The majority of HDC applications depend on this similarity.

2.2.2 Operations

Unlike conventional neural representations, hypervectors are compositional, allowing calculations in superposition [10]. Furthermore, these composite representations can be combined using HDC operations, yielding hypervectors in the same space, which enables recursive compositions [14] [6]. Arithmetic in HDC is based on three operations:

- **Binding:** The binding function is denoted as $\otimes : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$. This operation is commonly used for *associating* two hypervectors, and it's done by performing element-wise multiplication on two input hypervectors to produce a third vector dissimilar to both operands. [6][14]

- **Bundling:** This operation is denoted as $\oplus : \mathcal{H} \times \mathcal{H} \rightarrow \mathcal{H}$. The bundling operation is a *memorization* function that performs element-wise addition on its inputs to produce a vector that is *maximally similar* to the operands. [6][14]

- **Permutation** It's denoted as $\Pi : \mathcal{H} \rightarrow \mathcal{H}$. The permutation function is used to rearrange the elements of a hypervector, producing an output hypervector that is dissimilar from its input. Consider a hypervector $\mathcal{A}$, for which we aim to perform a cyclic shift of its elements by i positions. This operation can be denoted as $\Pi^i(\mathcal{A})$. Furthermore, the permutation function generates a *reversible* hypervector. This means that applying the inverse permutation to the permuted hypervector $\Pi^i(\mathcal{A})$ will return the original hypervector: $\Pi^{-i}(\Pi^i(\mathcal{A})) = \mathcal{A}$. Consequently, this property allows us to effectively represent *sequences* and their *ordering*. [6][14]

The operations mentioned above are simple and require only trivial element-wise arithmetic. On the other hand, if we were to accomplish the same result using a conventional neural network, we would need to assign new labels to images representing composite classes and train an entirely separate model for making predictions [14].

2.2.3 HDC vs. Gradient Descent

In this section, I present a comparison between HDC and gradient descent, the standard algorithm for training neural networks, focusing on efficiency, computation, and accuracy. Heddes et al. conducted an experiment to compare HDC and gradient descent by developing a hybrid HDC architecture that incorporates both techniques, where the α parameter sets the fraction of gradient descent [6]. In the context of efficiency, HD computing offers several benefits, such as:

- **Reduced training time:** As shown in Figure 2.3, HDC can have much faster training times compared to traditional neural networks employing gradient descent. Figure 2.3 shows that HDC had the highest accuracy in the first epoch among all models. This is mainly because HDC is based on trivial vector operations that can be performed in parallel. In contrast, gradient descent involves iterative weight updates, which can be computationally expensive [6].

- **Energy efficiency:** HDC systems are known to be more energy-efficient than gradient descent-based neural networks. This is because HDC requires much fewer epochs to train than gradient descent by a factor of 10x [6]. The energy consumption associated with training dominates the overall energy consumption in most machine learning systems.

- **Biological plausibility:** Since the invention of modern neural nets in the late 1980s, neuroscientists have criticized the backpropagation algorithm as biologically implausible [2]. Hence, a learning algorithm that avoids backpropagation (and gradient descent in general) has a better chance of shedding light on the way that actual brains perform these tasks.

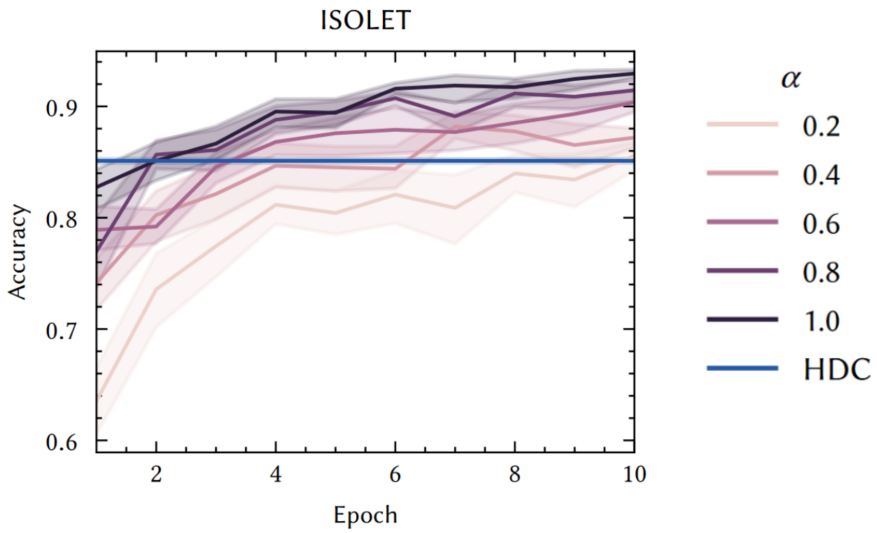

Figure 2.3: Accuracy calculated by training the hybrid model on ISOLET dataset [1]. The α parameter sets the fraction of gradient descent, where $\alpha = 1.0$ is equal to traditional gradient descent [6].

HDC also has its limitations. As we can see in Figure 2.3, the accuracy of the HDC model didn't improve with more epochs [6]. While gradient descent outperformed HDC's performance, this came at the cost of more training time by a factor of 10x. Gradient descent also comes with its own set of challenges in terms of efficiency and performance:

- **Convergence Speed:** Gradient descent may take significant time to converge to an optimal solution, as shown in Figure 2.3.

- **Local Minima:** Gradient descent is vulnerable to getting trapped in local minima, which can lead to suboptimal solutions.

In conclusion, HDC and gradient descent each possess unique strengths and weaknesses. Although HDC is not intended to replace traditional neural networks for all classification tasks entirely, it demonstrates remarkable performance in lightweight tasks and is well-suited for highly resource-constrained devices [4]. This compatibility aligns perfectly with the proposed AI model for hand gesture recognition.

Chapter 3

APPROACH

In this chapter of the thesis, I explain the methods used to create the hand gesture recognition AI model. The primary goal is to showcase the strength and efficiency of the proposed brain-inspired approach over traditional neural networks. To accomplish this, I will explain the specific techniques and procedures implemented while developing the model. I will also discuss the datasets used for training and validation, the algorithms integrated into the model, and the software and hardware tools employed.

Figure 3.1 demonstrates an overview of all the steps taken to build this model. First, the approach begins with data acquisition by employing a DVS to capture raw event data related to hand gestures. The DVS generates a continuous stream of event data corresponding to the captured motions. The next step is data preprocessing. The raw data is transformed into 2 types of data necessary for AI model training: *input features* and *output labels*. Next, the transformed data is fed into an AI model built using the TorchHD framework, facilitating gesture recognition and analysis. Finally, the model learns to map the events data captured by the DVS to the appropriate hand gesture classifications, effectively recognizing the different gestures. Notably, this learning is done by a novel, single-pass algorithm [7] that avoids the aforementioned issues associated with back-propagation.

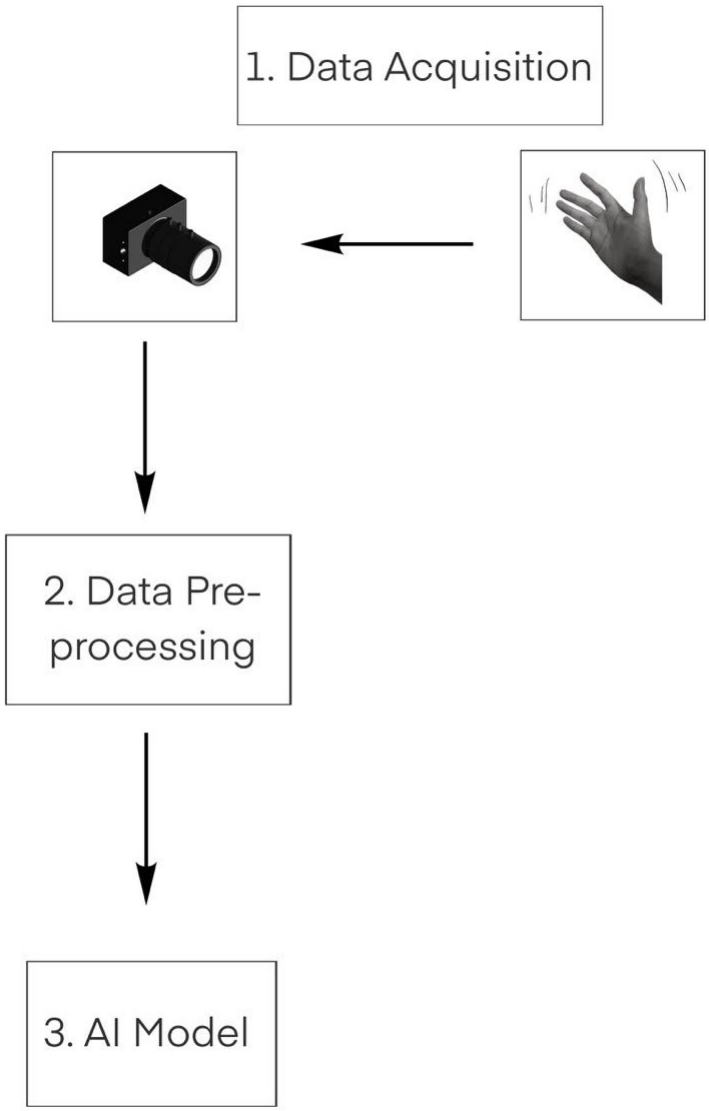

Figure 3.1: Overview of the steps to building the AI model

3.1 Data Acquisition

3.1.1 Overview

This AI model is trained using labeled data, which consists of both *input features* (discussed in Section 3.2.2) and the corresponding correct *output labels* (elaborated in Section 3.2.4). Figure 3.2 demonstrates the difference between *input features* and *output labels* using the famous hotdog/no hotdog example. In this example, the food image serves as the input feature, while the hotdog or no hotdog designation represents the output label. These input-output pairs are essential for training an AI model. They enable the model to learn the underlying patterns and relationships that allow it to accurately map inputs to their corresponding outputs.

For our hand recognition model, the process begins with capturing raw events data using a Dynamic Vision Sensor, followed by extracting the *input features* and *output labels* through *preprocessing methods*, which is discussed in Section 3.2. Finally, the model learns to make predictions by identifying patterns in the training data and generalizes these patterns to make accurate predictions on unseen data.

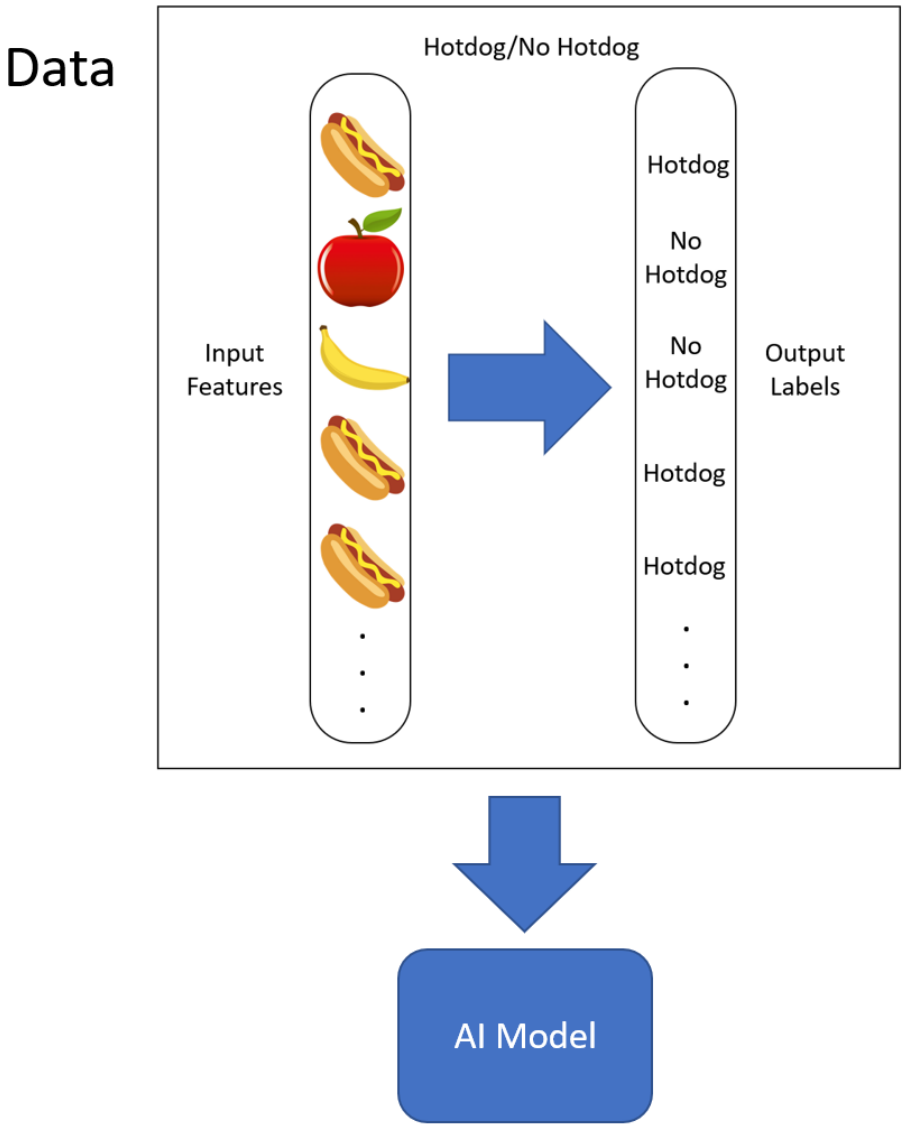

Figure 3.2: Hotdog/No Hotdog example to explain the difference between input features and output labels.

3.1.2 Capturing Raw Events using DVS

In this subsection, I will outline the process of capturing motion data using the DAVIS346 sensor. I coded two Python programs to allow for two data-capturing methods: realtime_playback.py, which captures data in real-time, and file_playback.py, which retrieves data from a previously recorded file. Although the real-time playback

method operates similarly, this section primarily focuses on file playback.

First, I record a file using the DV software provided by the DAVIS346 creators, saving it in aedat4 format. Then, I use a Python module called *dv_processing*, also supplied by the DAVIS346 creators, to access the recording and extract raw events. The raw events consist of 4 data types, but we are only interested in three specific types:

- **X:** an array of arrays, with each array containing the x-axis coordinates of all pixels showing a brightness change at a particular frame or timestamp.

- **Y:** an array of arrays, with each array containing the y-axis coordinates of all pixels showing a brightness change at a particular frame or timestamp.

- **Polarity:** an array of arrays, with each array containing the polarities (on or off) of all pixels showing a brightness change at a particular frame or timestamp.

For example, as shown in Figure 3.3, the illuminated pixels (either red or green) at a specific frame or timestamp indicate where the DVS detected changes in brightness. Therefore, we extract three NumPy arrays from this frame, containing the x-axis position, y-axis position, and polarity (red for off, green for on) of the illuminated pixels. This extraction process is repeated for each frame in the aedat4 recording, enabling us to extract the raw data: X, Y, and Polarity, as described above.

3.1.3 Frame Division into Tiles for Enhanced Data Extraction

In this section, I discuss an approach for enhancing data extraction by dividing the frame into smaller tiles, allowing for more detailed information to be gathered from each frame, ultimately benefiting the AI model's performance.

AI models often require substantial data to achieve optimal performance. Therefore, I developed a Python module that divides a given frame into N x N tiles. This division results in each tile functioning as an individual frame, from which we can extract the three types of raw data mentioned earlier: X, Y, and Polarity. Figure 3.3

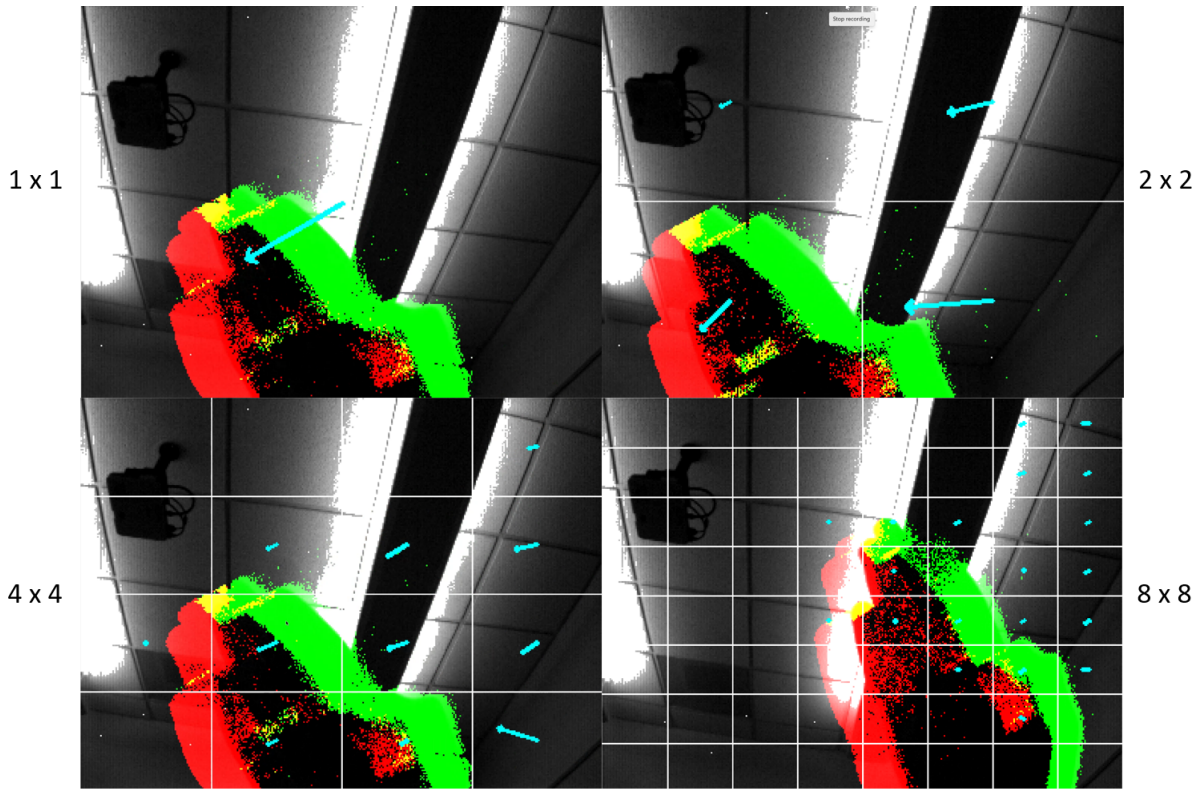

Figure 3.3: An example to show a comparison between different configurations of tiles as I move my hand from top right to bottom left.

offers a visualization of four different tile configurations: $N = 1, 2, 4, 8$ as I move my hand from the top right corner to the bottom left corner. The 1 x 1 configuration outputs three streams of raw data, while the 2 x 2 configuration yields 12 streams of raw data. The 4 x 4 configuration generates 48 streams of raw data, and so on. By capturing this hand movement and increasing the number of raw data streams, the AI model gains access to a more comprehensive dataset, potentially improving its overall performance.

Furthermore, utilizing tiles offers a significant advantage by enabling the simultaneous detection and classification of multiple hand gestures within a single frame. For instance, in a 1 x 1 configuration, the raw event data from two hands within the

same frame may overlap, resulting in incorrect data. However, in an 8 x 8 configuration, the presence of two hands in the frame does not cause interference because each tile functions as an independent frame. In conclusion, increasing the number of tiles leads to more accurate and reliable data at the cost of higher computational and memory demands. The complexity of this approach scales at a rate of $O(N^2)$.

In addition to the tile-based data extraction method, I have implemented an arrow that tracks the hand movement direction within the frame, as shown in Figure 3.3. The arrows follow my hand, pointing towards the bottom left corner as I move my hand in that direction. The arrows implementation is discussed in detail in section 3.2.3.

3.2 Data Preprocessing

In the previous section, I discussed the extraction of raw event data, returning **X**, **Y**, and **Polarity** NumPy arrays. In this section, I will elaborate on the preprocessing techniques applied to this data, which are essential in preparing it for the subsequent training phase.

3.2.1 Encoding Raw Events using "motion" algorithm

The raw events data obtained is structured as an array of arrays, which we can't use to train our model. Our goal is to convert this data into an array of values. To achieve this, I developed an encoding function called "motion" that accepts raw data (in the form of an array of values) as input and encodes it into a single value, which is then returned as the output.

The Python code is as follows:

```
1 def motion(self, axis, pp):
2     green_position = axis[pp == 1]
3     red_position = axis[pp == 0]
4     if len(green_position) < Tiles.THRESHOLD or len(
          red_position) < Tiles.THRESHOLD:
```

```
            return 0
    green_position_mean = axis[pp == 1].mean()
    red_position_mean = axis[pp == 0].mean()
    return red_position_mean - green_position_mean
```

The function requires two parameters:

- **axis:** either the X-axis or Y-axis array.
- **pp:** the polarity array, where 0 denotes off pixels (red) and 1 denotes on pixels (green).

The algorithm operates as follows. First, it calculates the mean position of all on and off pixels, resulting in green_position_mean and red_position_mean, respectively, as shown in the Python code. Next, the function returns the difference between these two mean values. Finally, by employing the "motion" function, we can effectively preprocess the raw data into a more manageable format, enabling further analysis and model training.

3.2.2 Extracting Input Features using "motion"

In the previous subsection, I provided an overview of the "motion" function and its implementation. In this subsection, I will explain how to extract *input features* using the "motion" function to prepare the data for model training.

The extraction of input features involves applying the "motion" function to every array in the **X** dataset and its corresponding array in the **Polarity** dataset, resulting in a new dataset called **X_motion**. Similarly, we perform this process for every array in the **Y** dataset and its corresponding array in the polarity dataset, yielding another new dataset called **Y_motion**.

Figure 3.4 and Figure 3.5 illustrate the entire process. Figure 3.4 depicts the raw events data before the application of the "motion" function. In this figure, there are three streams of data, with each stream represented as an array of arrays. After

applying the "motion" function, the transformed output is shown in Figure 3.5. In this figure, there are two streams of data, also referred to as channels. Each channel consists of an array of values.

By using the "motion" function to extract *input features*, we can effectively prepare the data for training the AI model, enhancing its performance and accuracy in hand gesture recognition tasks.

```
X                          Y                          Polarity
array([array([...])        array([array([...])        array([array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])               array([...])               array([...])
       array([...])])             array([...])])             array([...])])
```

Figure 3.4: Raw Events: Three data streams before preprocessing. Each stream is an array of arrays.

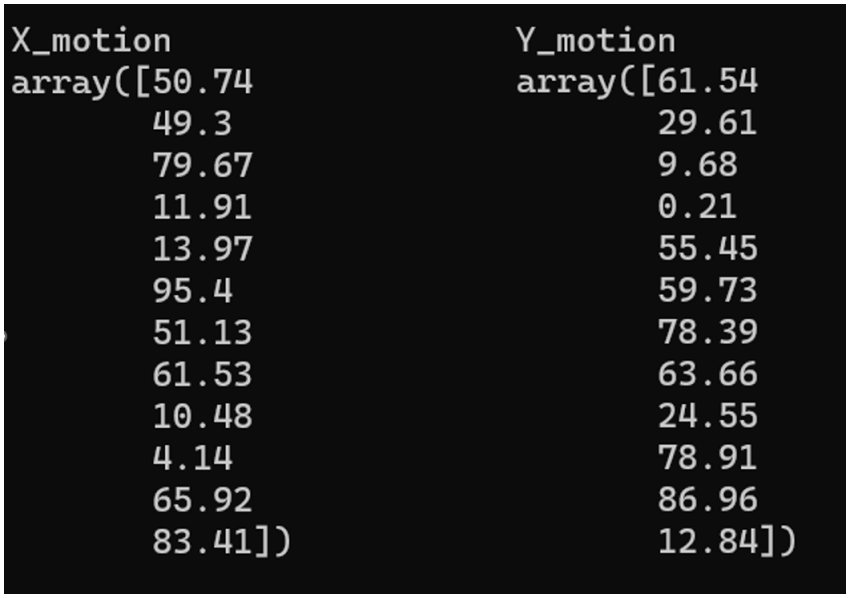

Figure 3.5: The Input Features: two streams (also called channels) of preprocessed data. Each channel is an array of values.

3.2.3 Computing "Motion Direction" for Arrow Visualization

In this subsection, I explain how I implemented the arrow, shown in previous figures, that visualizes the direction of motion of an object. As mentioned in section 2.1, a DVS sensor detects changes in brightness, which can be either positive (on) or negative (off). If the amount of light falling on a specific pixel decrease, the DVS marks the polarity of that pixel as off, indicating a negative change. However, if the amount of light increases, the polarity is marked as on, representing a positive change. On pixels are marked as green, while off pixels are marked as red.

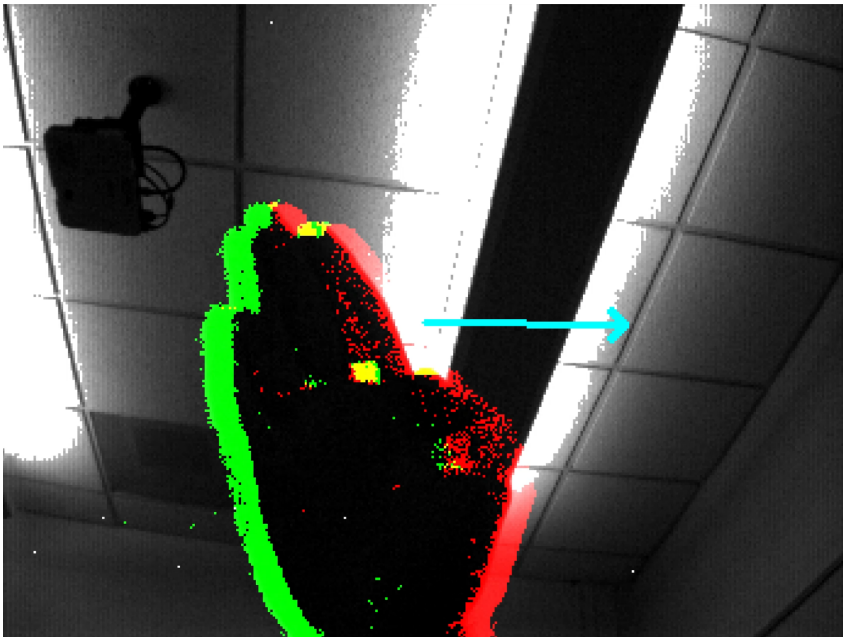

Figure 3.6: Moving my hand from left to right. The right side of my hand blocks light, causing the sensor to detect less light (red). The left side of my hand moves away, causing more light to fall on the DVS (green).

For example, as shown in Figure 3.6, I move my hand from left to right. The right side of my hand blocks light from reaching the DVS, causing the sensor to detect less light and mark these pixels as off (red). As the left side of my hand, which was previously blocking light, moves away, more light falls on the DVS, and the pixels are marked as on (green).

From this observation, we can deduce that if the red (off) pixels are ahead of the green (on) pixels, it means the object is moving in the same direction as the red pixels. In other words, if the returned **X_motion** value is positive, the hand is moving to the right, while a negative value indicates movement to the left. Similarly, if the **Y_motion** value is positive, the object is moving down, and if it is negative, the object is moving up.

By combining the directional changes using **X_motion** and **Y_motion**, I computed the "Motion Direction", represented as a tuple containing two values: Δx and

Δy. These two values signify the extent of change in direction along the x-axis and y-axis, respectively. Using "Motion Direction", I created an arrow that accurately tracks the motion direction of the detected object. This visualization aids in understanding and interpreting the movement of objects detected by the DVS sensor.

3.2.4 Extracting Output Labels using "Motion Direction"

The final stage in the data preprocessing phase is computing and extracting *output labels*, represented as numerical values, to simplify the categorical data representation within the model. Each class (or category) is assigned a number, with 0 consistently denoting the hand rest position. For instance, if we are developing a model to classify two hand gestures, such as waving and pinching, there would be three numerical classes representing these gestures: 0 for the hand rest position, 1 for waving, and 2 for pinching. After the model is trained, the numerical labels can be easily mapped back to their original categorical labels, enabling the final output to be presented as hand gestures.

There are two approaches to extracting output labels, depending on the complexity of the hand gestures involved. For simple and easy hand gestures, such as moving a hand in one direction, output labels can be automatically computed using the "Motion Direction" method, as previously discussed in section 3.2.3. However, when dealing with more complex hand gestures, such as those used in sign language, manual labeling becomes necessary. In this case, an individual would have to manually label tens of thousands of input features, which is an intensive and time-consuming process.

3.3 AI Model

For the AI model, it is important to note that I did not develop my own AI model. Instead, I utilized a pre-existing model designed and developed by Rahimi et al. Their model exploits HDC and it is specifically designed for hand gesture recognition from a stream of Electromyography (EMG) signals [6] [12]. Moreover, their model achieves an impressive classification accuracy of 97.8% while requiring only a third of the training data needed by the state-of-the-art Support Vector Machine (SVM) for the same task. Therefore, I used their model as a foundation for my project and adapted the dataset and preprocessing steps methods to ensure that the data format is consistent with their model. By utilizing the HDC model from Rahimi et al., I could concentrate more on the previous steps while benefiting from the robust and efficient processing capabilities that HDC offers.

Chapter 4

EXPERIMENT AND RESULTS

4.1 Experimental Setup

In this section, I describe the experiment conducted to evaluate the performance of my biologically-inspired approach. The primary goal of this experiment is to develop an AI model capable of accurately classifying two basic hand gestures: moving the hand to the right or left direction. I chose to focus on this relatively simple model for two primary reasons: (1) Given that the brain-inspired approach is still relatively new, it is essential to start with a simple application of the approach and gradually progress to more complex models; and (2) It's tough and time-consuming to label the data of complex hand gestures (as discussed in section 3.2.4). Therefore, in the following section, I will outline the steps undertaken to develop this particular model, detailing the 3 phases discussed in the Approach section: Data Acquisition, Data Preprocessing, and Developing the AI Model.

4.1.1 Data Acquisition

To develop this model, a substantial amount of data is required. Collecting this data through manual hand movements would be very time-consuming for any individual. Therefore, to facilitate data acquisition, an Arduino and a servo motor were used to oscillate continuously between right and left hand gestures. Figure 4.1 illustrates the setup for capturing raw data. A straw was attached to the servo motor and pointed towards the Dynamic Vision Sensor (DVS). Then, the servo motor moved at a rate of one second per direction, enabling the DVS to capture one second of each hand gesture before alternating to the other.

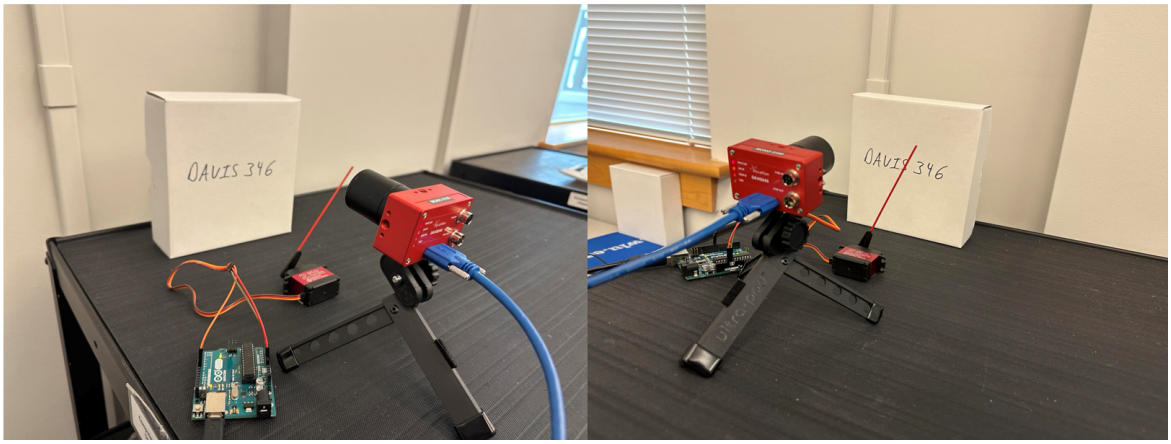

Figure 4.1: Data Acquisition Setup.

Furthermore, I improved the data extraction process by employing the tiles method, described in Section 3.1.3. As shown in Figure 1, I set the parameter N equal to 2, resulting in an output image featuring a 2 x 2 grid, which helps with data representation and analysis.

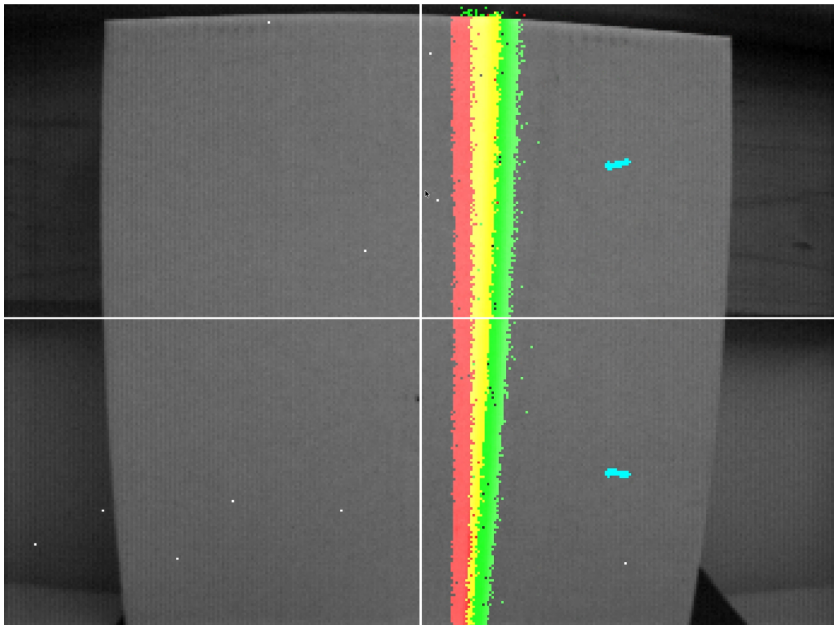

Figure 4.2: Sample output of the previously described system.

4.1.2 Data Preprocessing and AI Model

Both the second and third phases of the experiment follow the approaches previously described in Sections 3.2 and 3.3, respectively.

4.2 Results

The model was executed multiple times, consistently yielding astonishing results, with a success rate ranging from 95% to 100%. After running the model ten times, the average accuracy rate achieved was 99%. It was observed that the model frequently reached 100% accuracy, which can be attributed to the following 2 reasons: (1) the task is very simple and (2) the brain-inspired approach is very effective. Figure 4.3 depicts the output obtained from training and testing the model on the data. Moreover, the model required only a brief training period. As demonstrated in Figure 4.3, the training process was completed within a very small fraction of a second, further highlighting the efficiency of the novel approach.

Figure 4.3: Result from training and testing the model a single time.

Chapter 5

CONCLUSIONS AND FUTURE WORK

Despite the unusual approach taken in this work (using a non-traditional sensor with a novel learning algorithm), our results turned out to be quite promising: as with the EMG data used in [12], we obtained a testing success rate between 90 and 100%. Based on this success, it would make sense to continue this work. Specifically, I think it could be a promising approach to automatic visual gesture recognition, where the current state-of-the-art focuses on the recognition of static gestures rather than dynamic motions [3].

The main drawback of my approach is that I am not fully exploiting the advantages of the DVS. Specifically, by grouping the events into static image frames and applying the motion function to those frames, I am discarding much of the dynamic information available (timestamps) in the raw event data. This approach allowed me to reduce the enormous bandwidth of the sensor (which can reach 1.2 x 10^7 events per second [9]), but it obscures what part of my results are due to the sensor itself. For example, it is possible that I could have obtained the same results with an ordinary web camera combined with an image-processing library like OpenCV. In order to exploit the power of the DVS to the fullest extent, I would need an approach that incorporated event-time information into the input to the learning algorithm. Before undertaking that work, I would first want to compare my current approach to (1) the web-cam / OpenCV approach; and (2) the use of a convolutional neural network with backpropagation to get a better sense of how much my results differ from those more traditional methods.

BIBLIOGRAPHY

[1] Ron Cole, Yeshwant Muthusamy, and Mark Fanty. The isolet spoken letter database, 1990.

[2] Francis Crick. The recent excitement about neural networks. *Nature*, 337:129–132, 1989.

[3] Prangon Das, Tanvir Ahmed, and Md. Firoj Ali. Static hand gesture recognition for american sign language using deep convolutional neural network. In *2020 IEEE Region 10 Symposium (TENSYMP)*, pages 1762–1765, 2020.

[4] Shijin Duan, Yejia Liu, Shaolei Ren, and Xiaolin Xu. Lehdc: Learning-based hyperdimensional computing classifier. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1111–1116, 2022.

[5] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J Davison, Jörg Conradt, Kostas Daniilidis, et al. Event-based vision: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(1):154–180, 2020.

[6] Mike Heddes, Igor Nunes, Pere Vergés, Dheyay Desai, Tony Givargis, and Alexandru Nicolau. Torchhd: An open-source python library to support hyperdimensional computing research. *arXiv preprint arXiv:2205.09208*, 2022.

[7] Alejandro Hernández-Cano, Namiko Matsumoto, Eric Ping, and Mohsen Imani. Onlinehd: Robust, efficient, and single-pass online learning using hyperdimensional system. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 56–61, 2021.

[8] Michael Hersche, Edoardo Mello Rella, Alfio Di Mauro, Luca Benini, and Rahimi Abbas. Integrating event-based dynamic vision sensors with sparse hyperdimensional computing: A low-power accelerator with online learning capability. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Kathy Hylton, Parker Mitchell, Blake Van Hoy, and Thomas Karnowski. Experiments and analysis for measuring mechanical motion with event cameras. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2021.

[10] Pentti Kanerva. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation*, 1:139–159, 2009.

[11] Elias Mueggler, Basil Huber, and Davide Scaramuzza. Event-based, 6-dof pose tracking for high-speed maneuvers. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2761–2768. IEEE, 2014. Event camera animation: https://youtu.be/LauQ6LWTkxM?t=25.

[12] Abbas Rahimi, Simone Benatti, Pentti Kanerva, Luca Benini, and Jan M Rabaey. Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2016.

[13] Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. A 128 times 128 1.5% contrast sensitivity 0.9% fpn 3 μs latency 4 mw asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers. *IEEE Journal of Solid-State Circuits*, 48(3):827–838, 2013.

[14] Zhuowen Zou, Haleh Alimohamadi, Yeseong Kim, M. Hassan Najafi, Narayan Srinivasa, and Mohsen Imani. Eventhd: Robust and efficient hyperdimensional learning with neuromorphic sensor. *Frontiers in Neuroscience*, 16, 2022.