Embedded Machine Learning

Project Magic Wanding

Florian Mayer, BSc.

05.01.2023

Content

1 Project Contents and Problem Definition 4
1.1 Introduction to the Project 4
1.2 Problem Definition 4
1.3 Data Acquisition and Preprocessing 4
1.4 Model Development and Training 4
1.5 Challenges and Considerations 4
2 The Hardware Arduino 33 BLE 5
2.1 Key Features and Specifications 5
2.2 Advantages in Gesture Recognition 5
2.3 Onboard LSM9DS1 in Arduino Nano 33 BLE 5
3 IMU Features considered for the generation of the model 6
3.1 Noise filtering 6
3.2 Gyroscope Processing, get the change in degree 7
3.3 Calculation of the Sensor Position 8
3.4 Data Normalization 9
4 Generation of the Model 10
4.1 Processing the CSV Data 10
4.2 Setting the structure of the Neural Network 10
4.3 Convert calculated Model to TensorFlowLite and evaluation 11
4.4 Generate and plot the confusion Matrix 12

List of illustration

Figure 1: Orientation of the Accelerometer, Gyroscope and Magnetometer of the LSM9DS1 6
Figure 2: Code Snipped of the used lowpass-filter to reduce the accelerometer noise 6
Figure 3: Complementary filter reducing the drift effect of the gyro 7
Figure 4: Fast atan2 function 7
Figure 5: Error prone estimation of the elongated position in x and y 8
Figure 6: The GUI recorder (in the ZIP Package), used to record the movement data 9
Figure 7: Function to Normalize the data to be later used as input fort he neural network 9
Figure 8: Sandglass structure of a neural network 11
Figure 9: Confusion Matrix of the trained and evaluated dataset 12

1 Project Contents and Problem Definition

1.1 Introduction to the Project

The project focuses on developing a gesture recognition system utilizing a neural network model. This system interprets data from an Inertial Measurement Unit (IMU), specifically from an Arduino Nano 33 BLE Sense board, to identify distinct gestures. The primary goal is to classify these gestures accurately into predefined categories, enabling applications in areas like human-computer interaction, gaming interfaces, and assistive technology.

1.2 Problem Definition

Gesture recognition involves the identification and interpretation of human gestures via computational algorithms and sensors. The complexity arises from the need to process multi-dimensional IMU data (comprising accelerometer and gyroscope readings) and accurately classify it into distinct gesture patterns. This project's challenge lies in effectively capturing the nuances of each gesture and ensuring the model's robustness against variations in gesture execution, such as speed, intensity, and user-specific characteristics.

1.3 Data Acquisition and Preprocessing

Data acquisition is a critical phase where IMU readings are collected as the user performs various gestures. Each gesture results in a unique pattern of movement and orientation, captured through accelerometer and gyroscope sensors. Preprocessing these readings involves normalizing the data to a consistent range and format, making it suitable for feeding into the neural network.

1.4 Model Development and Training

The project employs a deep learning approach, specifically a feedforward neural network, to classify gestures. The model's architecture, including the number and structure of layers, is designed to handle the intricacies of time-series IMU data. Training the model involves feeding it with labeled gesture data, allowing it to learn the distinguishing features of each gesture.

1.5 Challenges and Considerations

Key challenges in this project include dealing with the variability in gesture execution and mitigating overfitting in the model to ensure it generalizes well to new, unseen data. Techniques like data augmentation, dropout, and regularization are employed to enhance the model's performance and reliability.

2 The Hardware Arduino 33 BLE

The Arduino Nano 33 BLE is a compact and versatile board based on the nRF52840 microcontroller. It stands out for its low power consumption and BLE (Bluetooth Low Energy) capabilities, making it ideal for IoT applications. The board is equipped with a rich set of features, including digital and analog I/O pins, USB interface, and onboard sensors.

2.1 Key Features and Specifications

Microcontroller: The heart of the Nano 33 BLE is the nRF52840, which integrates a 32-bit ARM Cortex-M4 CPU, running at 64 MHz. This powerful microcontroller is capable of handling complex computations required for processing sensor data.
Connectivity: The board includes BLE functionality, allowing for wireless communication with other BLE-enabled devices, such as smartphones and tablets.
Inertial Measurement Unit (IMU): A key component for gesture recognition projects is the onboard IMU. The IMU in the Nano 33 BLE consists of a 3-axis accelerometer and a 3-axis gyroscope, providing detailed information about motion and orientation.
Digital & Analog Pins: The board offers a variety of pins for interfacing with external sensors and devices, adding to its versatility in project implementations.
USB Interface: The board can be easily programmed via a micro USB cable, which also powers the device.

2.2 Advantages in Gesture Recognition

The Arduino Nano 33 BLE's IMU is crucial for gesture recognition. The accelerometer measures linear acceleration, while the gyroscope provides information on angular velocity. Together, these sensors capture the dynamics of hand movements, enabling the precise detection and classification of gestures. The BLE functionality adds the possibility of sending gesture data wirelessly to other devices for further processing or real-time applications.

2.3 Onboard LSM9DS1 in Arduino Nano 33 BLE

The Arduino Nano 33 BLE is equipped with the LSM9DS1 module, a state-of-the-art motion sensor that includes a 3D digital linear acceleration sensor, a 3D digital angular rate sensor (gyroscope), and a 3D digital magnetic sensor (magnetometer). This integrated module provides a comprehensive solution for motion detection, which is crucial for various applications including gesture recognition.

- **3D Accelerometer:** Measures linear acceleration along the X, Y, and Z axes. Essential for detecting movement and orientation changes.
- **3D Gyroscope:** Provides angular rate measurements, crucial for understanding rotational movements.
- **3D Magnetometer:** Detects magnetic fields, enabling orientation with respect to the Earth's magnetic north.
- **High Precision and Sensitivity:** The LSM9DS1 offers high resolution and sensitivity, capturing subtle movements and rotations, which is vital for accurate gesture recognition.

The combination of accelerometer, gyroscope, and magnetometer data allows for a thorough understanding of hand gestures. The accelerometer detects changes in speed and direction, while the gyroscope tracks rotational movement, and the magnetometer can offer additional orientation information. In the context of gesture recognition:

- **Complex Gestures:** The LSM9DS1 allows for the detection of complex gestures that involve various types of movement, including linear motion and rotation.
- **Orientation Awareness:** By combining data from all three sensors, the system can understand the gesture's orientation, enhancing recognition accuracy.
- **Real-time Feedback:** The high sensitivity and fast data output of the LSM9DS1 enable real-time gesture tracking and feedback, crucial for interactive applications.

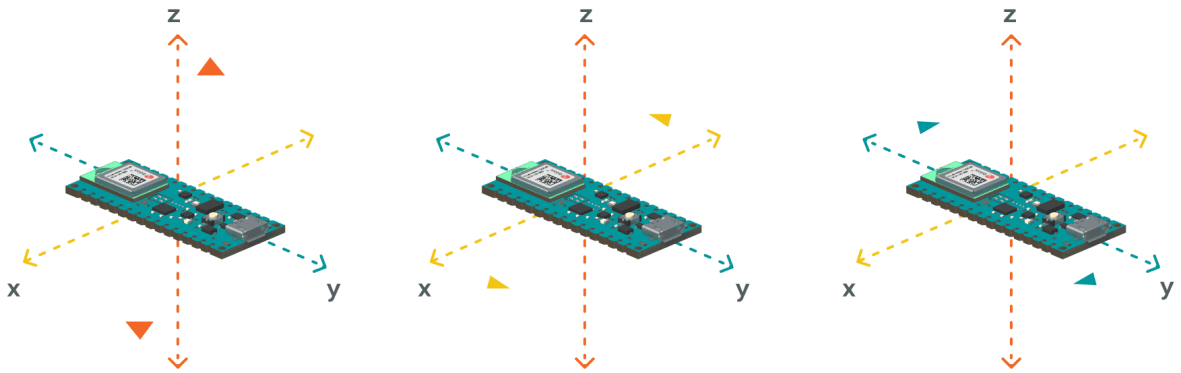

Figure 1: Orientation of the Accelerometer, Gyroscope and Magnetometer of the LSM9DS1

3 IMU Features considered for the generation of the model

The project involves collecting raw data from the Arduino Nano 33 BLE's onboard LSM9DS1 IMU (Inertial Measurement Unit). This unit provides accelerometer and gyroscope readings which are crucial for gesture recognition. The raw data, comprised of acceleration (ax, ay, az) and gyroscope (gx, gy, gz) readings, needs to be pre-processed before it can be effectively used for training a machine learning model. Pre-processing steps include filtering noise from the data, normalizing it to a consistent scale, and converting it into a suitable format for the neural network.

3.1 Noise filtering

To enhance the quality of the IMU data, a low-pass filter is applied, which helps in reducing noise and smoothing out the data. This is particularly important for removing short-term fluctuations that can adversely affect the model's performance. Following this, the data is normalized, which involves scaling the various parameters to a uniform range. Normalization is critical as it ensures that all input features contribute equally to the model's training process, preventing any one feature from disproportionately influencing the model's learning.

```
180  float lowPassFilter(float currentVal, float previousVal, float alpha)
181  {
182      if (fabs(currentVal) < ACC_Treshold)
183      {
184        currentVal = 0;
185      }
186      return alpha * currentVal + (1 - alpha) * previousVal;
187  }
```

Figure 2: Code Snipped of the used lowpass-filter to reduce the accelerometer noise

3.2 Gyroscope Processing, get the change in degree

In order to reduce the negative drift effect of the gyroscope, a complementary filter was used. Here, in a simple way, the change in degree is additionally calculated using the two other accelerometer axis. The result is weighted with integrated gyro values in order compensate the drift and to avoid an introduced noise to the gyroscope values.

```
307     float rollAcc = fastAtan2(data[1]*GRAVITY,data[2]*GRAVITY) * (180 / M_PI);
308     data[6] = (data[3] * 0.8f) * deltaTime + rollAcc * 0.2f;
309     // Complementary filter roll
310     float pitchAcc = fastAtan2(data[0]*GRAVITY,data[2]*GRAVITY) * (180 / M_PI);
311     data[7] = (data[4] * 0.8f) * deltaTime + pitchAcc * 0.2f;
312
313     if (fabs(data[5]) < GYRO_THRESHOLD)
314     {
315         data[8] = 0;
316     }
317     else
318     {
319         data[8] = data[5] * deltaTime;
320     }
```

Figure 3: Complementary filter reducing the drift effect of the gyro

Figure 3 shows, that the value of gz is just “thresholded”, that is due to the missing reference point around the z-axis. As we do not gather the magnetometer data, to provide a consistent dataset over all participants. Therefore “GYRO_TRESHOLD” was introduced!

In order to calculate the angle using the accelerometer values, the function “atan2” was quantized and the function “fastAtan2” was used. Here we just speed up the code while reducing the precision of the result.

```
337 float fastAtan2(float y, float x)
338 {
339     const float n1 = 0.97239411f;
340     const float n2 = -0.19194795f;
341     float result = 0.0f;
342     if (x != 0.0f) {
343         const float atan = (float)M_PI / 4.0f * y / x;
344         const float z = atan * atan;
345         result = atan * ((n1 + n2 * z) / (1.0f + (n1 + n2) * z));
346         if (x < 0.0f) {
347             if (y < 0.0f) {
348                 return result - (float)M_PI;
349             }
350             return result + (float)M_PI;
351         }
352     } else if (y > 0.0f) {
353         result = (float)M_PI / 2.0f;
354     } else if (y < 0.0f) {
355         result = -(float)M_PI / 2.0f;
356     }
357     return result;
358 }
```

Figure 4: Fast atan2 function

3.3 Calculation of the Sensor Position

Again, the process begins with raw data obtained from the onboard IMU sensor, which includes accelerometer and gyroscope readings. The accelerometer provides data on linear acceleration along three axes, while the gyroscope offers angular velocity information. To determine the position, the algorithm first applies a low-pass filter to the accelerometer data to mitigate noise and smooth out the readings. This step is crucial for reducing errors caused by short-term fluctuations. The filtered acceleration data are then integrated over time to obtain velocity, and a second integration yields the displacement or position. (Figure 5)

```
280     ax = lowPassFilter(data[0], lastAx, 0.5);
281     ay = lowPassFilter(data[1], lastAy, 0.5);
282     az = lowPassFilter(data[2], lastAz, 0.5);
283
284     data[0] = ax;
285     data[1] = ay;
286     data[2] = az;
287
288     lastAx = ax;
289     lastAy = ay;
290     lastAz = az;
291
292     // Integrate twice and calculate alongation
293     vx = ax * GRAVITY * deltaTime;
294     vy = ay * GRAVITY * deltaTime;
295     vz = az * GRAVITY * deltaTime;
296
297     x += vx * deltaTime;
298     y += vy * deltaTime;
299     z += vz * deltaTime;
300
301     // allongation in m
302     data[9] = x;
303     data[10] = y;
304     data[11] = z;
```

Figure 5: Error prone estimation of the elongated position in x and y

This process is performed for each axis separately, resulting in a three-dimensional representation of the sensor's movement. By doing so, the algorithm effectively translates raw IMU data into meaningful positional information, laying the groundwork for subsequent gesture recognition and classification. The positional data are then normalized to ensure consistency and compatibility with the neural network model used for gesture recognition, making this step a cornerstone of the project's data processing pipeline.

While collecting the data with the created python GUI, the path of the wand (IMU) was drawn additionally to give the user and idea of the reproducibility of his / her movement.

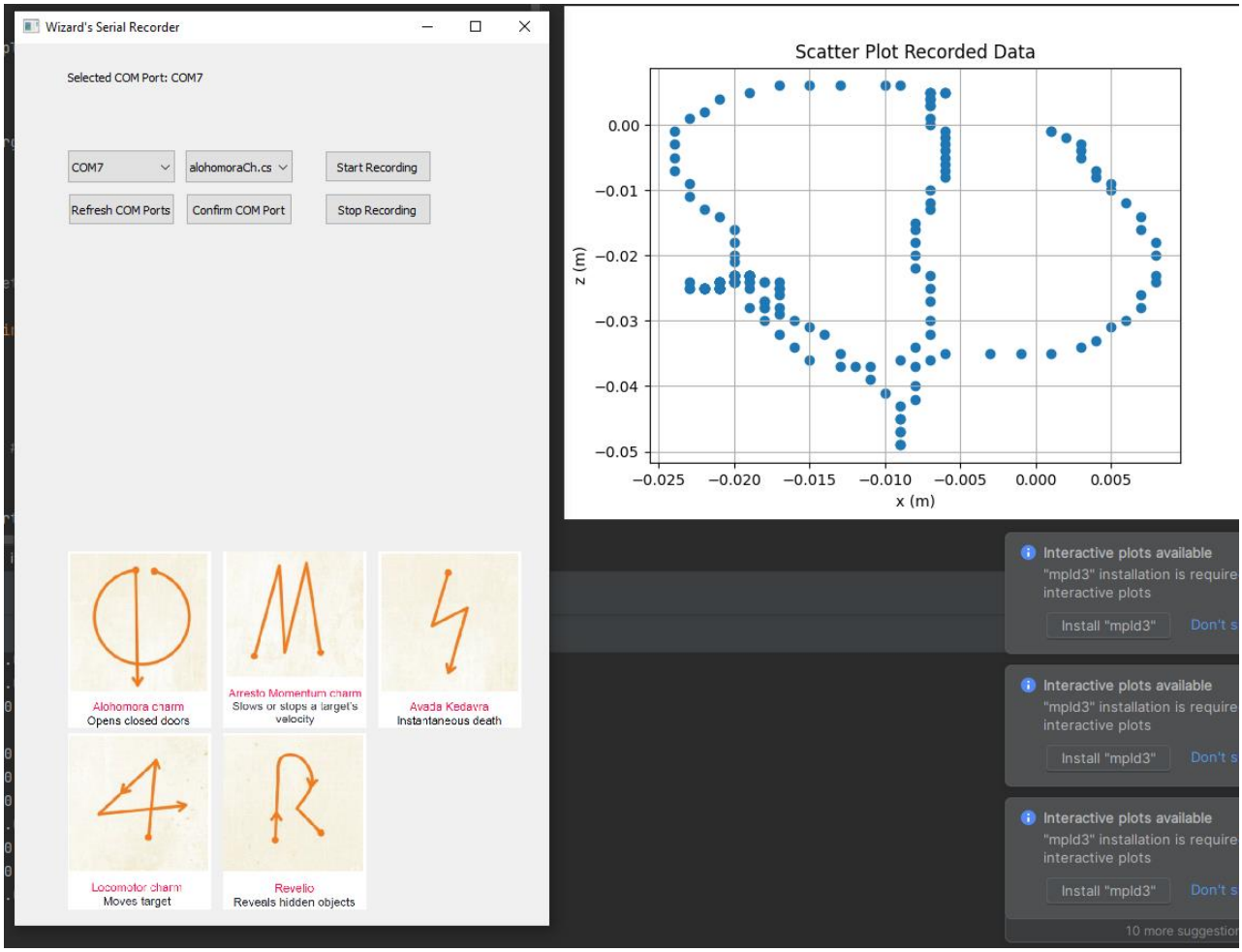

Figure 6: The GUI recorder (in the ZIP Package), used to record the movement data

3.4 Data Normalization

After filtering and drift correction, the gyroscope data is normalized. This involves scaling the data within a specified range, ensuring consistency and preventing any axis from disproportionately influencing the gesture recognition process. Normalization is essential for achieving balance and uniformity in the data fed into the neural network.

```
417  float normalize_range(float val, float minR, float maxR)
418  {
419
420      return (val - minR) / (maxR - minR);
421
422  }
```

Figure 7: Function to Normalize the data to be later used as input fort he neural network

4 Generation of the Model

4.1 Processing the CSV Data

This code segment processes and normalizes data from an IMU (Inertial Measurement Unit) for gesture recognition in machine learning. It involves smoothing accelerometer readings with a low-pass filter, calculating velocities and positions, and correcting gyro drift. The data is then normalized across different metrics (acceleration, gyro, degrees, and distance) for consistency. Optionally, noise can be added to simulate real-world conditions. Finally, the processed data is reshaped and split into training and testing sets, making it ready for use in training a neural network model for gesture classification.

- Read IMU Data: Extracts accelerometer and gyroscope readings from the IMU sensor.
- Apply Low-Pass Filter: Smoothing of the accelerometer data to reduce noise and fluctuations.
- Calculate Velocity and Position: Integrating accelerometer data to derive velocity and then position information.
- Correct Gyro Drift: Adjusting the gyroscope data to account for any drift over time.
- Normalize Data: Scaling of various types of data (acceleration, gyro, degrees, and position) to a consistent range for model input.
- Optional Noise Addition: Random noise is introduced to the data to enhance the robustness of the model against real-world measurement noise.
- Reshape and Store Processed Data: Convert the processed data into a suitable format for machine learning, typically involving reshaping the data into arrays.
- Split Data into Training and Testing Sets: The dataset is divided into training and testing subsets to enable model training and evaluation.
- Prepare Data for Machine Learning Model: Ensures that the data is in the correct format and ready for feeding into a machine learning model for gesture recognition training.

4.2 Setting the structure of the Neural Network

- Activation Function: A LeakyReLU activation function with a 0.1 negative slope is employed. This function helps to address the issue of "dying neurons" in the network. "Dying neurons" in neural networks using ReLU activation functions occur when neurons stop learning and only output zeros, becoming inactive due to consistently negative input weights leading to zero gradients during backpropagation.
- Model Architecture: The model is a sequential stack of layers, specifically designed to prevent overfitting and enhance learning.

The "sandglass" structure of the model, characterized by a particular pattern of layer units (71-28-71), suggests a focus on feature extraction and re-expansion. This architecture initially compresses or condenses the input data (narrowing down to 28 units), which can help in extracting essential features. Subsequently, it re-expands these features (back to 71 units), potentially allowing the network to reconstruct or refine these features before making a final classification with the output layer. This design can be effective in capturing complex patterns in data while maintaining a balance between model complexity and generalization.

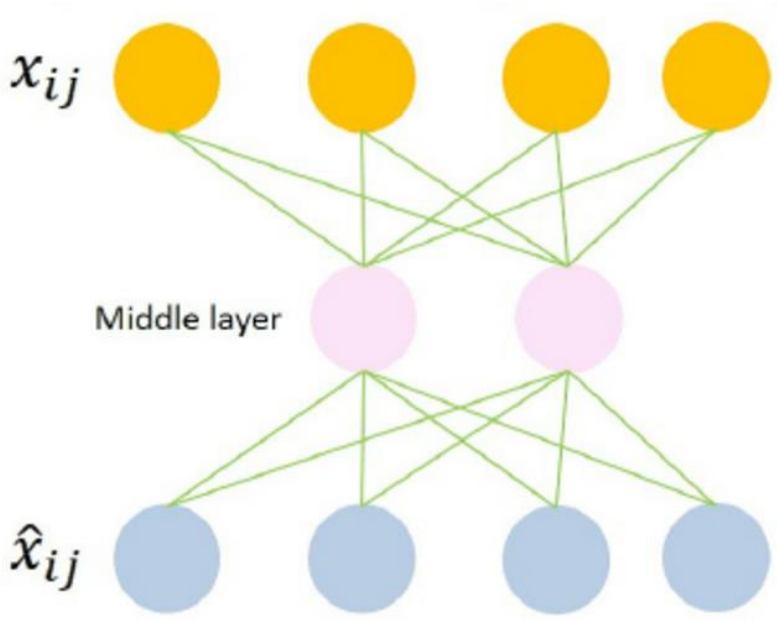

Figure 8: Sandglass structure of a neural network

4.3 Convert calculated Model to TensorFlowLite and evaluation

This code should efficiently convert a TensorFlow Keras model into TensorFlow Lite (TFLite) format, offering both standard and optimized versions. The conversion employs TFLiteConverter, adapting the model for lightweight deployment. The optimized version incorporates quantization and INT8 operations, significantly reducing the model size and potentially boosting inference speed, crucial for resource-constrained environments like mobile or embedded systems. The process includes generating a representative dataset for quantization accuracy. Post-conversion, the models are saved, and their sizes are reported, highlighting the effectiveness of optimization in shrinking the model footprint while maintaining functionality, a key benefit for deployment in limited-resource scenarios.

A comprehensive evaluation of both a TensorFlow model and its optimized TensorFlow Lite (TFLite) counterpart. The evaluate_model function first assesses the standard model's accuracy on test data, comparing predicted and actual labels. It then evaluates the optimized TFLite model, handling quantized inputs and outputs to accurately gauge its performance. The process generates accuracies for both models, providing a clear comparison of their effectiveness. This evaluation is particularly useful in balancing the trade-offs between model size and accuracy, a critical consideration in resource-constrained deployment environments like edge computing or mobile devices. By executing and printing these results, the code aids in decision-making for deploying the most efficient model without compromising on performance.

4.4 Generate and plot the confusion Matrix

Specialized tool for visualizing the performance of an optimized machine learning model, specifically focusing on classification accuracy. It works by first ensuring compatibility with one-hot encoded labels, converting them to class labels if necessary. The core of the function involves generating a confusion matrix that contrasts the model's predictions against actual labels, providing a clear, quantified view of its performance. This matrix is then visualized in an easily interpretable format, using a blue color scheme for clarity. The resulting plot, saved as a high-resolution image, serves as a valuable tool for analyzing the model's strengths and weaknesses in classification tasks, thereby aiding in refining and enhancing the model's accuracy.

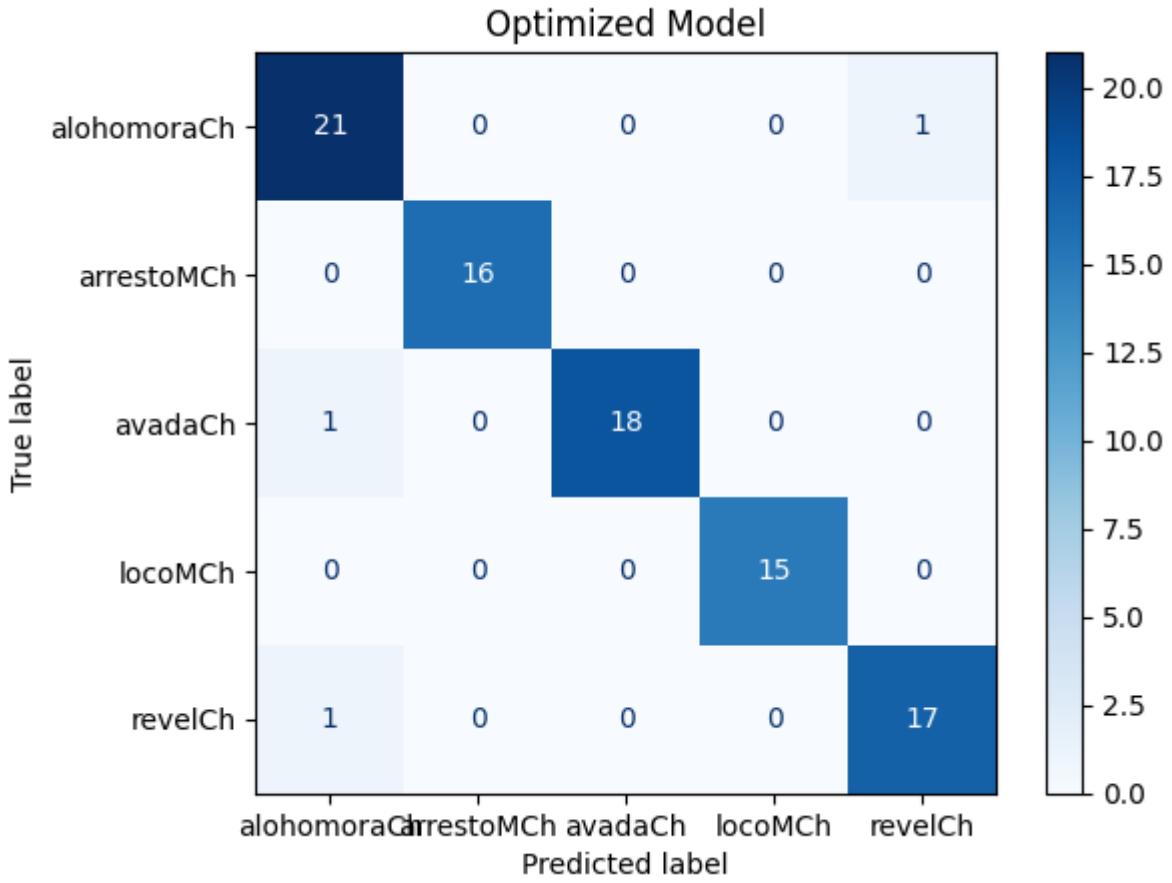

Figure 9: Confusion Matrix of the trained and evaluated dataset

The confusion matrix indicates high accuracy for a five-class optimized model, with most predictions correctly falling on the diagonal. Minor misclassifications are rare, showing the model's robust performance in classifying gestures.

The displayed confusion matrix for an optimized model demonstrates excellent predictive performance for gesture recognition, with the majority of gestures such as "Alohomora" and "Revelio" being accurately identified. There is a slight confusion between "Alohomora" and "Locomotor," indicating a small area for improvement. Overall, the model shows strong classification capabilities with minimal errors, here.