Selected Topics Communications and Mobile Computing – Embedded Machine Learning

Robust Magic Wand

David Mihola
12211951

January 10th, 2023

Contents

1 Introduction

The purpose of this project was to create robust machine learning empowered recognition of magic spells from the Harry Potter movies performed with an Arduino Nano 33 BLE Sense mounted facing upwards on a top of a 30 cm stick. The on board inertial measurement unit (IMU) with sampling rate 119 Hz was used to obtain the motion of the Arduino and therefore the magic wand when performing spells. The capture of a spell starts when the absolute acceleration across all axis surpasses 2.0 G and lasts for 1 s. The machine learning models used for recognition were trained off device using the TensorFlow[1] library. The trained models were converted afterwards to binary and run on device using the TensorFlow Lite[2] library.
The text in the following sections refers to the code available at `https://github.com/xmihol00/robust_magic_wand`.

2 Data

The data were collected for 5 spells shown on figure 1. The collection was done using the program `data_collection.ino` for Arduino while running the script `data_collection.py` on a computer. This setup made it possible to collect only samples reasonably representing the spells. The collected samples can be also plotted with the `data_set_plot.py` script.

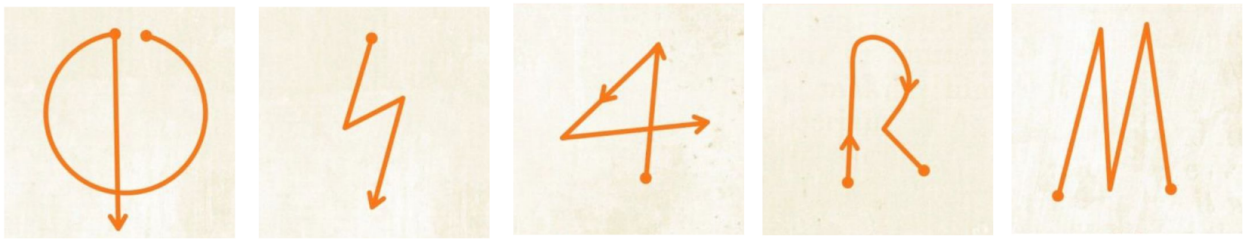

Figure 1: Collected spell[3] strokes

2.1 Data processing

Each measurement of a spell consists of 119 samples. Each sample contains values given by the accelerometer, gyroscope and magnetometer for the X, Y and Z axis relative to the board. In total one measurement has 714 floating point values. This implementation takes into account only the values given by the accelerometer and gyroscope for the Y and Z axis relative to the board, later on only referred to as X and Y axis relative to the plane of a spell stroke. These four vectors are

[1] `https://www.tensorflow.org`
[2] `https://github.com/tensorflow/tflite-micro-arduino-examples`
[3] primary source: `https://studylib.net/flashcards/set/a-beginner-s-guide-to-wand-motions_188857`, secondary source: `https://tc.tugraz.at/main/mod/resource/view.php?id=325033`

processed in to two final vectors holding the X and Y coordinates of the spell stroke over time, i.e. 238 floating point values.
The processing of the accelerometer and gyroscope data is performed in the following steps:

1. average acceleration is calculated for each axis,
2. angle is calculated by numerically integrating the angular velocity for each axis,
3. average angle is calculated for each axis,
4. normalized acceleration is calculated by dividing the average acceleration with the magnitude of the average acceleration vector across axis,
5. normalized angle is calculated by subtracting the average angle from each value of the angle vectors with respect to the axis,
6. spell stroke is calculated by rotating the normalized acceleration by the normalized angle with respect to the axis,
7. minimum stroke value is subtracted from each value of the stroke with respect to the axis,
8. the stroke is normalized by dividing it with the difference of the maximum an minimum stroke values with respect to the axis.

Furthermore, the spell stroke can be rasterized into a gray scale image of an arbitrary size, by using the X and Y stroke values as coordinates of non-zero pixels. The intensity of the stroke pixels might be increased over time to visualise the progression of the stroke, as shown on figure 2 also with the raw data and processed stroke. Similar comparisons between 2 randomly drawn samples of the same spell can be plotted using the data_analysis.py script.

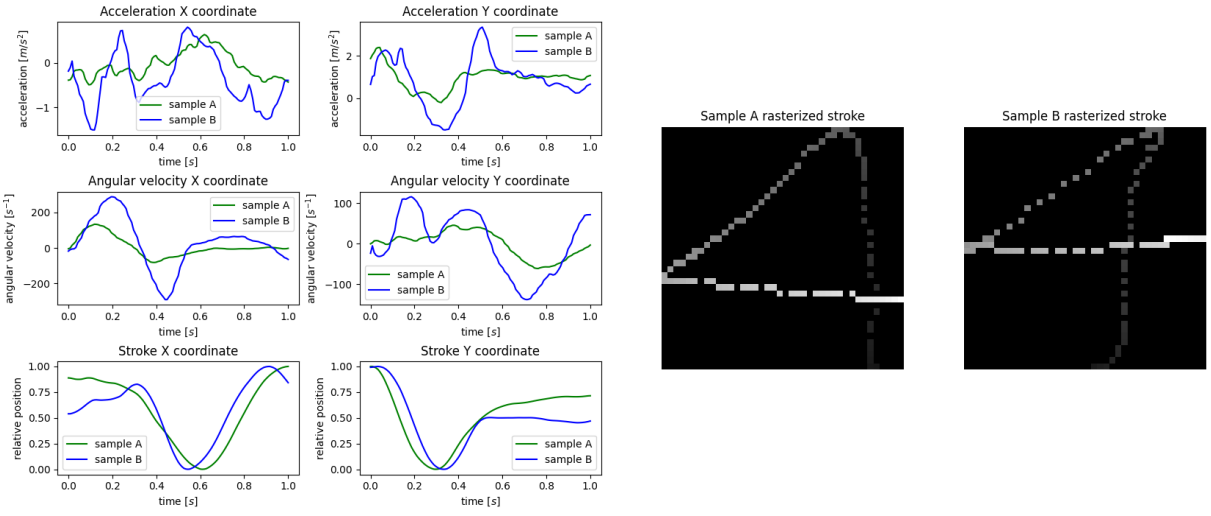

Figure 2: Raw and processed data of two Locomotor spells

The data processing described above as well as the rasterization were inspired by the Magic Wand[4] example supplied with the TensorFlow Lite library.

2.2 Data augmentation

All the data augmentations were done during data collection. The samples were collected with various absolute acceleration thresholds combined across all axis ranging from 1.5 to 3.0 G. The spells were casted with movements originating from the whole arm, from the elbow or by twisting the wrist. Additionally, different speeds of spell casting and different grips of the magic wand as well as slight rotations of it were used to ensure variability of the collected samples. Great attention was also put on making the collected data imperfect, e.g. by including not fully casted spells. The various augmentations are visualised on figures 3, 4 and 5.

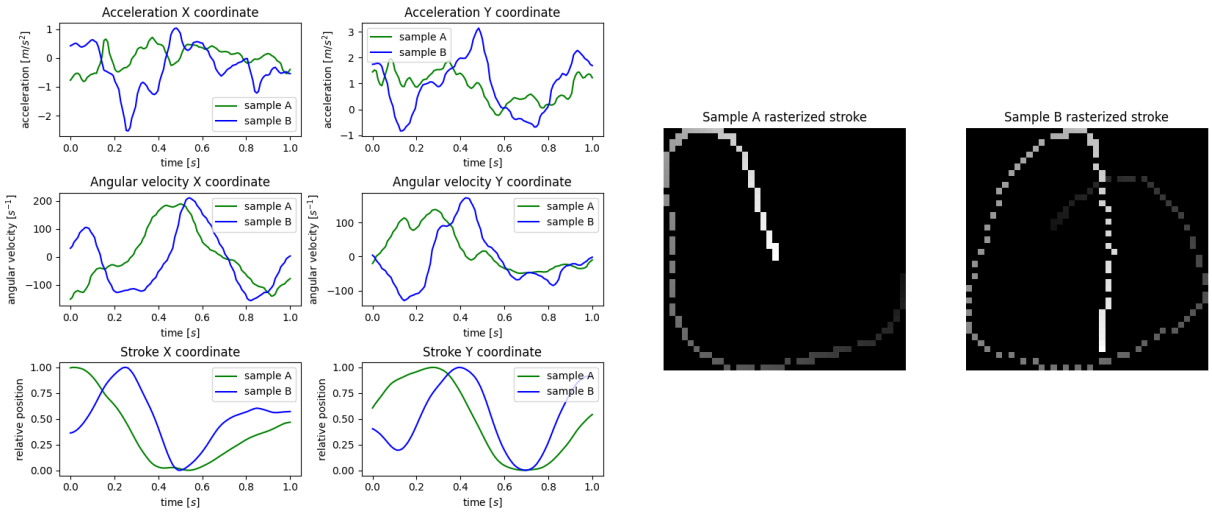

Figure 3: Comparison of slowly casted Alohomora spell with high acceleration threshold and almost perfectly casted Alohomora spell

2.3 Data sets

The collected data are separated into 5 CSV files each containing exactly 100 samples in the data directory. The same files with additional 100 samples per file, i.e. 200 samples per file in total, are available in the data_large directory. Increasing the number of captured spells was mainly motivated by the fact, that the biggest factor in model performances described in section 3 was the chosen seed, with which the data were split into train and test data sets. The larger data set quite significantly improved this issue.

Both the processing into strokes and images, as described above, is performed from the raw data at execution time, when the samples are also labeled, merged, shuffled and separated into train and test

[4] https://github.com/tensorflow/tflite-micro-arduino-examples/tree/main/examples/magic_wand

data sets. The size of images was chosen to be 20 by 20 pixels, as smaller sizes started to loose performance.

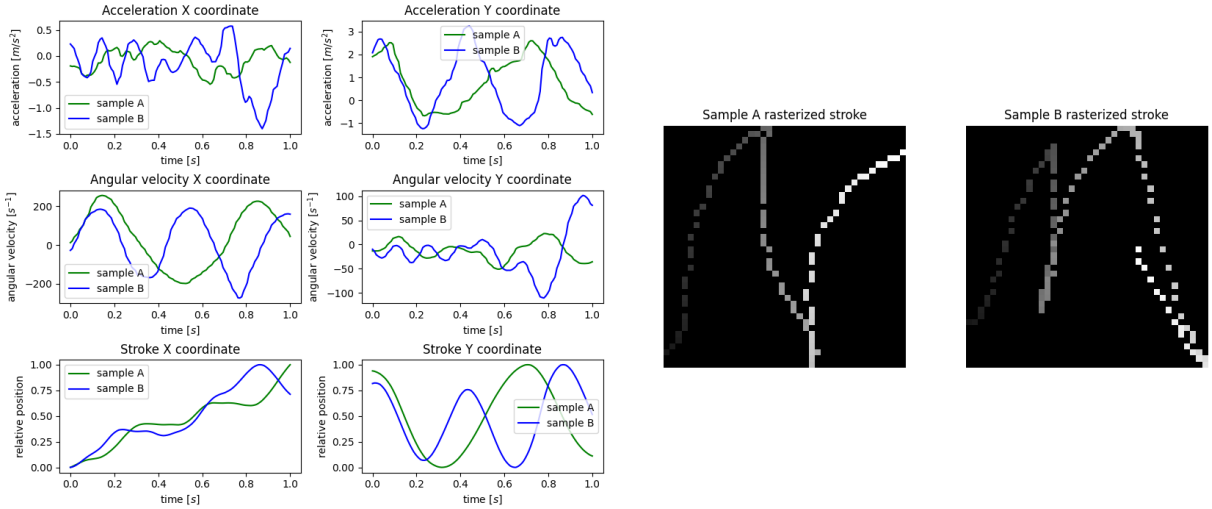

Figure 4: Comparison of two Aresto Momentum charms, the first performed too slowly and the second too quickly

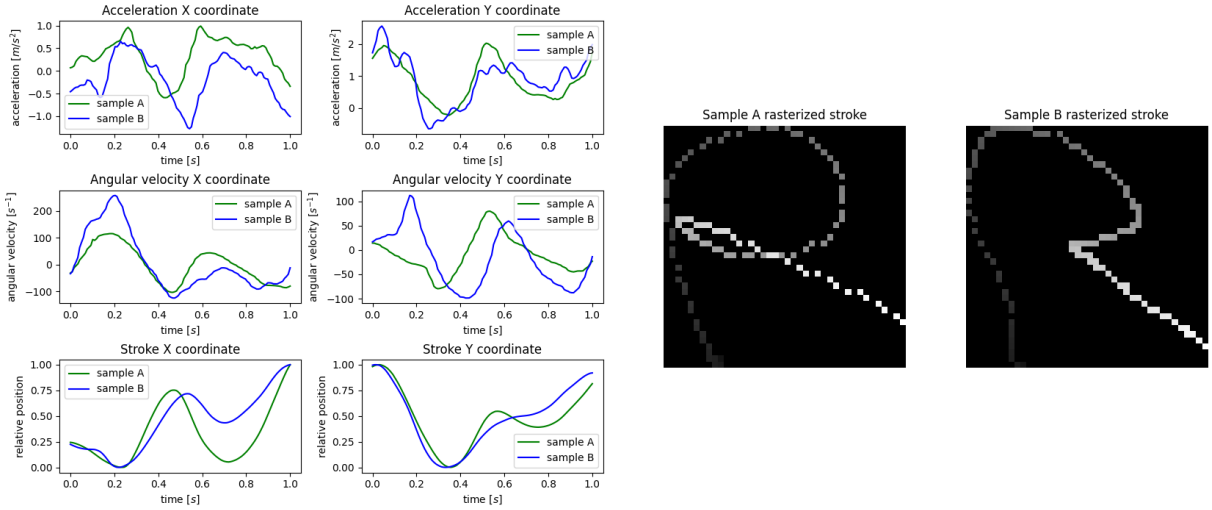

Figure 5: Comparison of two Revelio spells performed with a twist of the wrist and a movement of the whole arm

3 Model selection and hyperparameter tuning

Model selection and hyperparameter tuning was carried out using the jupyter notebook model_search.ipynb. Initially, 3 different model architectures were chosen. The named and later on shown model architectures in tables 1 and 2 are:

- `Only_DENS` – composed by 2 or 3 fully connected layers,
- `Only_CONV` – composed by a convolutional layer with 5x5 kernel size followed by 2 times repeated 2x2 kernel max pooling and 3x3 kernel convolutional layers followed by a compressing convolutional layer with 1x1 kernel size,
- `CONV_DENS` – composed by 3 times repeated 3x3 kernel convolutional 2x2 kernel and max pooling layers followed by 2 fully connected layers.

Additionally, these architectures were repeatedly created with different numbers of kernels and units per layer. Furthermore, regularization was added into some architectures in the form of dropout (DO model name suffix) and batch normalization (BN model name suffix).

3.1 Training

The training is performed for each model in a 2 step manner. Firstly, the model is trained using early stopping by setting aside 20 % of the train data set into a validation data set. The early stopping is set up to maximize accuracy on the validation data set with patience for 3 epochs. Secondly, the weights are reseted and the model is trained again on the whole train data set using the number of epochs, when the best validation accuracy was reached. The Adam optimizer with its default settings is used, i.e. learning rate of 0.01. The time necessary for train and the number of training epochs on both training data sets is summarized for each model in tables 1 and 2 respectively.
The training is replicable with the jupyter notebook mentioned above on Google Colab[5][6] using CPU.

3.2 Evaluation

As can be seen in the table 1, all the not regularized models are performing well, even the simplest possible linear model. The performance on the smaller data set, see table 2, seems to follow the same pattern when not considering outliers, such as the `Only_DENS_DO_M` model. The regularization in the form of dropout can in some cases further improves the results. On the other hand, batch normalization prevents the models from training properly. This is very likely caused by the train data set being too small.

3.3 Final model

Given the fact, that even the linear model performs very well on the larger data set, there is no need to select an overly complicated model. Moreover, the size of a model and mainly the inference time are usually the leading factors when selecting a relevant model to run on an embedded device, as need for larger memory can non-negligibly increase the price when buying in large quantities and the inference time can have a drastic impact on battery life.

[5] https://colab.research.google.com
[6] Seed 42 was selected, but the same seed may produce different results on a local machine.

	Total parameters	Trainable parameters	Non trainable parameters	Full Size [B]	Optimized size [B]	Training time Colab CPU [s]	Number of epochs	FLOPS	Full model accuracy [%]	Optimized model accuracy [%]	Full model inference time [ms]	Optimized model inference time [ms]
baseline_linear	1195	1195	0	6072	2704	1.82	9	2410	86.00	88.00	0.55	0.44
Only_DENS_S	13355	13355	0	55840	16512	1.40	6	26655	93.00	92.00	2.79	1.68
Only_DENS_M	24405	24405	0	99484	26936	2.17	6	48730	93.50	92.50	4.74	2.60
Only_DENS_L	29205	29205	0	119240	32584	1.84	15	58280	96.00	95.50	5.63	3.13
CONV_DENS_S	10181	10181	0	45724	17632	5.56	15	537886	97.50	97.50	104.81	46.52
CONV_DENS_L	40069	40069	0	165276	49128	4.42	6	2013726	96.00	96.00	312.33	121.37
Only_CONV_S	23877	23877	0	100664	32808	4.69	10	584670	96.50	96.00	161.52	50.11
Only_CONV_L	93829	93829	0	380472	105784	6.61	8	1906590	96.50	96.00	423.07	119.60
Only_DENS_DO_S	13355	13355	0	55840	16512	1.78	14	26655	94.50	93.00	2.80	1.68
Only_DENS_DO_M	24405	24405	0	99484	26936	1.16	10	48730	94.50	93.50	4.72	2.60
Only_DENS_DO_L	29205	29205	0	119240	32584	1.01	10	58280	93.00	93.50	5.64	3.14
CONV_DENS_DO_S	10181	10181	0	45724	17632	4.80	6	537886	92.00	92.00	104.80	46.54
CONV_DENS_DO_L	40069	40069	0	165276	49128	7.81	14	2013726	99.00	99.00	312.29	121.41
Only_CONV_BN_S	24325	24101	224	103408	35312	2.02	1	595966	42.50	41.00	—	—
Only_CONV_BN_L	94725	94277	448	384136	108544	1.57	1	1929182	18.50	18.50	—	—
CONV_DENS_BN_DO_S	10405	10293	112	48056	20080	2.05	1	549422	50.50	50.50	—	—
CONV_DENS_BN_DO_L	40517	40293	224	168056	51688	1.44	1	2036798	23.00	22.50	—	—

Table 1: Summary of analyzed models trained on the larger data set with full input size

	Training time Colab CPU [s]	Number of epochs	Full model accuracy [%]	Optimized model accuracy [%]
baseline_linear	1.13	10	81.00	80.00
Only_DENS_S	0.72	6	89.00	88.00
Only_DENS_M	0.74	7	91.00	88.00
Only_DENS_L	0.76	6	93.00	94.00
CONV_DENS_S	4.95	11	90.00	90.00
CONV_DENS_L	4.92	12	95.00	95.00
Only_CONV_S	2.49	9	93.00	95.00
Only_CONV_L	4.89	12	94.00	95.00
Only_DENS_DO_S	1.38	6	91.00	91.00
Only_DENS_DO_M	0.83	6	86.00	84.00
Only_DENS_DO_L	1.25	10	94.00	94.00
CONV_DENS_DO_S	2.97	13	94.00	93.00
CONV_DENS_DO_L	4.93	13	96.00	96.00

Table 2: Summary of relevant analyzed models trained on the smaller data set with full input size

The optimized `Only_DENS_DO_S` model can deliver good performance with relatively small number of parameters and very fast inference time, therefore it was chosen as the final model. Comparison between the best performing models and the final model can be seen of figures 6 and 7. The training and conversion of this model can be done using the final_training.ipynb jupyter notebook.

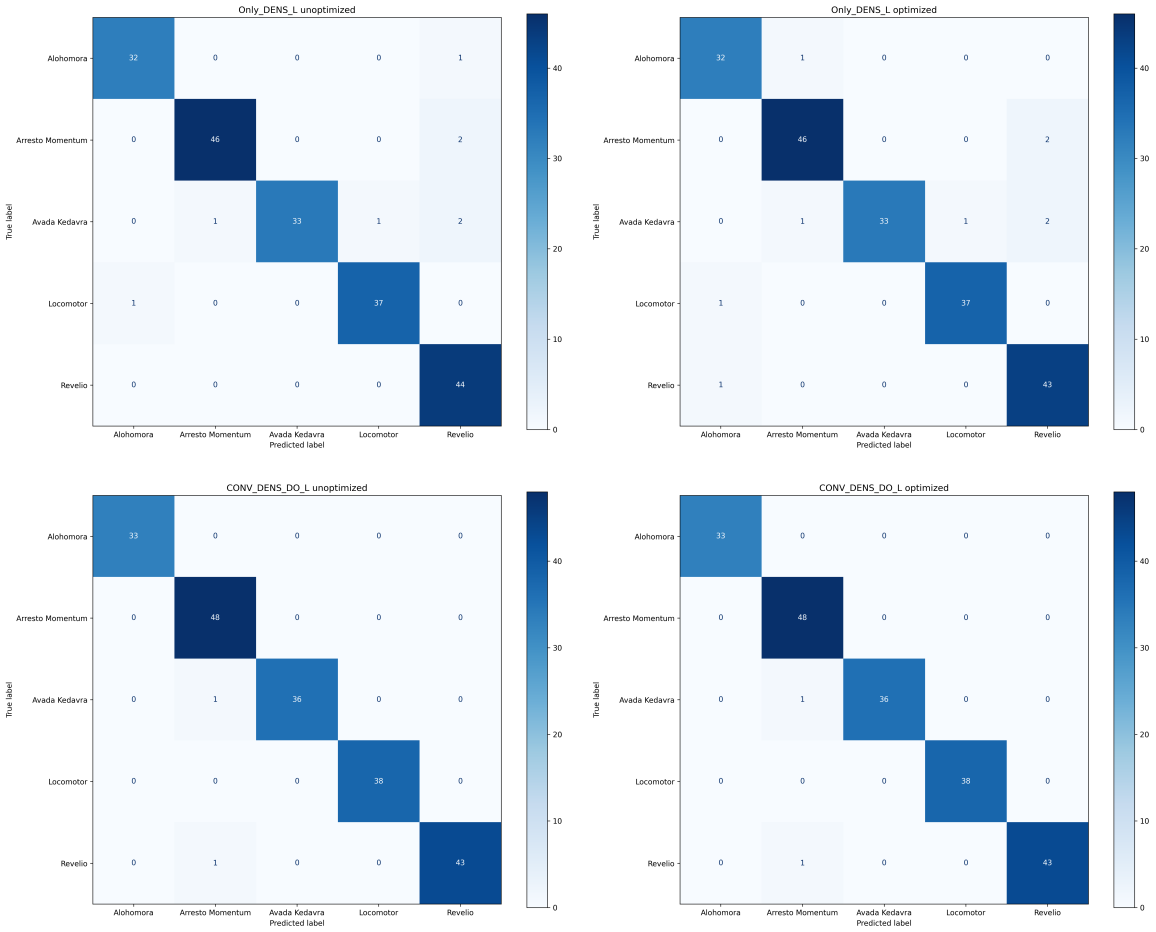

Figure 6: Confusion matrices of the best performing models for dense and image classification

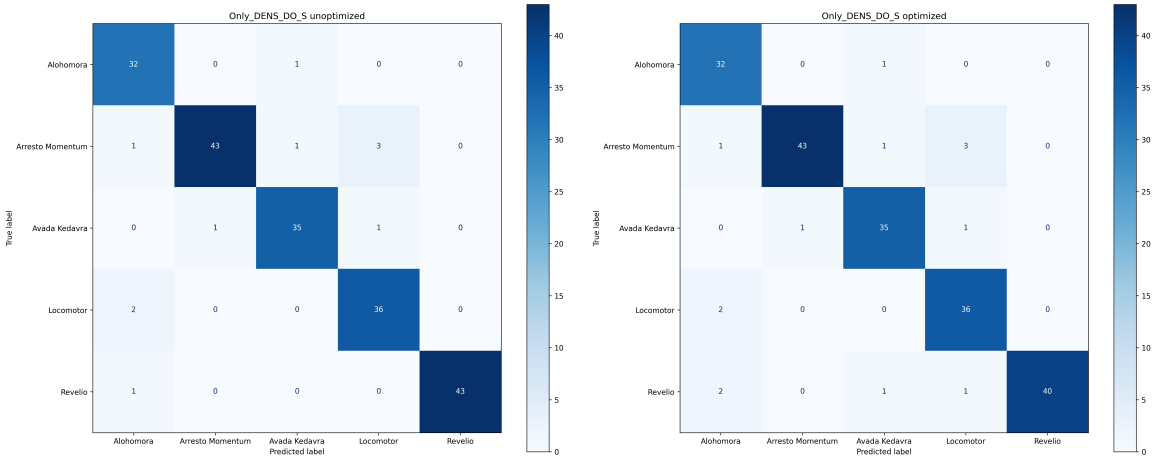

Figure 7: Confusion matrices of the final model

4 Inference

The performance of the final solution was captured in a video. The set up displayed in the video can be reproduced with the dense_recognition.ino or image_recognition.ino programs when setting the `PRETTY_OUTPUT` define to 1 and inserting correct models in their respective `model.h` header files. After uploading one of these programs to the Arduino use the pretty_serial_echo.py script while performing spells.
The spells in the video are performed in the following order: Alohomora, Avada Kedavra, Locomotor, Revelio and Aresto Momentum. They are casted in rounds simulating different magicians, which are progressively becoming more extreme. Nevertheless, the final model is able to correctly classify most of the casted spells. First misclassifications appear only later in the video.

5 Summary

The lack of a large data set makes it harder to precisely recognize the best performing model, but overall the following conclusion can be made. The processing of the sensor values into the stroke vectors makes it possible to design very efficient yet robust machine learning models. The rasterization into images can further improve accuracy and possibly robustness, with the cost of more needed compute power and consequently longer inference times.