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 on a top of a 30 cm stick and facing upwards. The on board inertial measurement (IMU) unit 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 the 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 representing of the spells. The collected samples can be also plotted with the `data_set_plot.py` script.

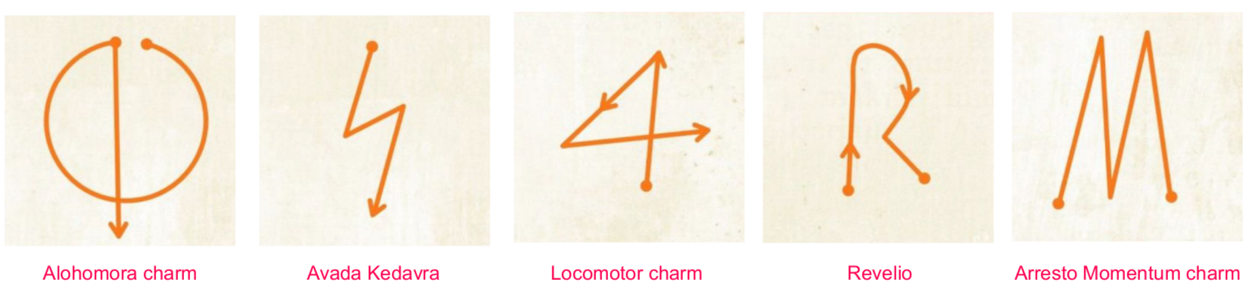

Figure 1: Collected spell[3] strokes

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 (or 220, see subsection 2.3) 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 by 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 the respect to the axis,
6. spell stroke is calculated by rotating the normalized acceleration by the normalized angle with the respect to the axis,
7. minimum stroke value is subtracted from each value of the stroke with the respect to the axis,
8. the stroke is normalized by dividing it with the maximum stroke value with the 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 the 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.

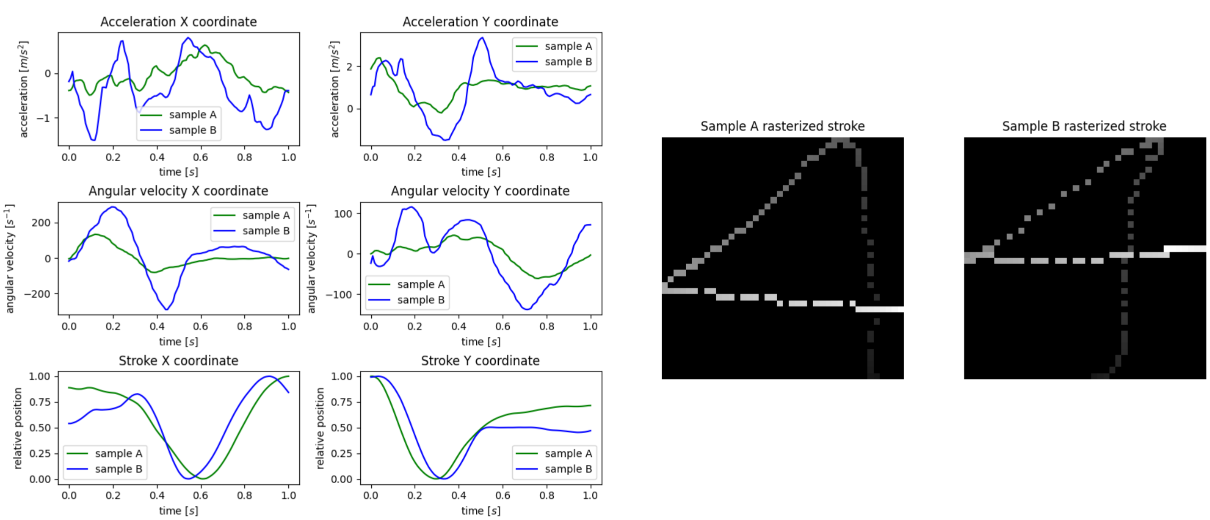

Figure 2: Raw and processed data of two Locomotor spells

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 6.

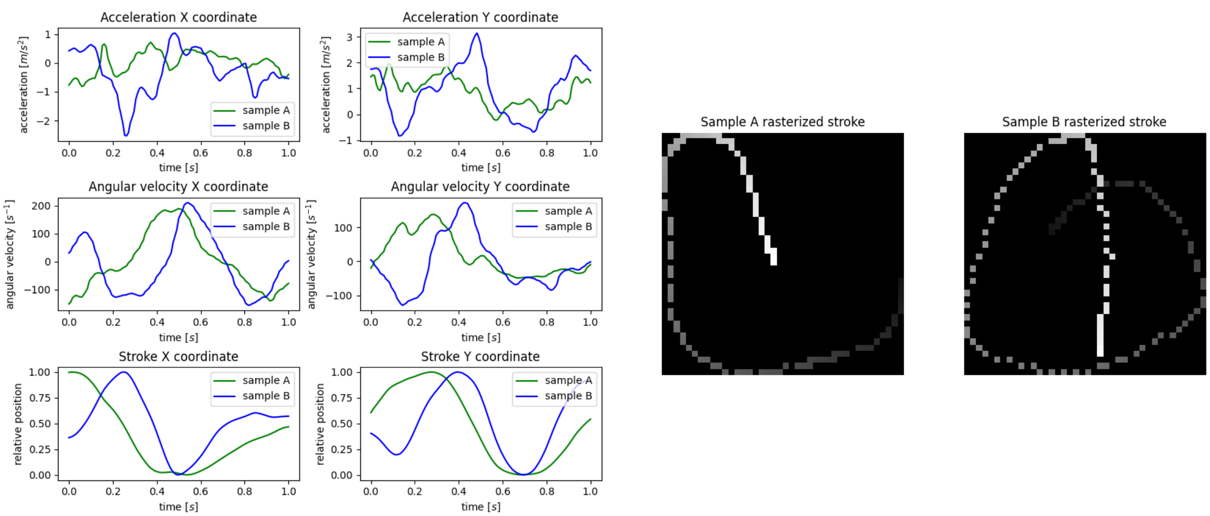

Figure 3: Comparison of slowly casted Alohomora spell with high acceleration threshold and almost perfectly casted Alohomora spell

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.
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

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

test data sets. The size of images was chosen to 20 by 20 pixels, as smaller sizes started to loose performance.
Additionally, motivated by the findings in figures 3, 4 and 6, the data sets can be processed with only 110 samples for each axis of the stroke. In this case first 2 and last 7 samples are removed. This lead to a small improvement of both the accuracy and model size in case of the fully connected models.

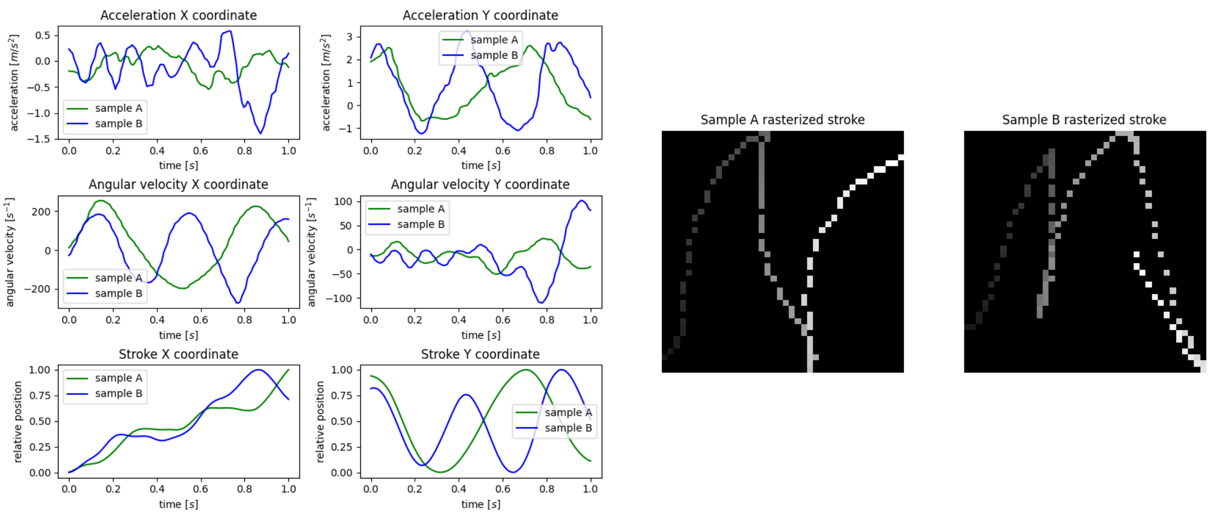

Figure 4: Comparison of two Aresto Momentum charms, the first performed too slowly and the second too quickly

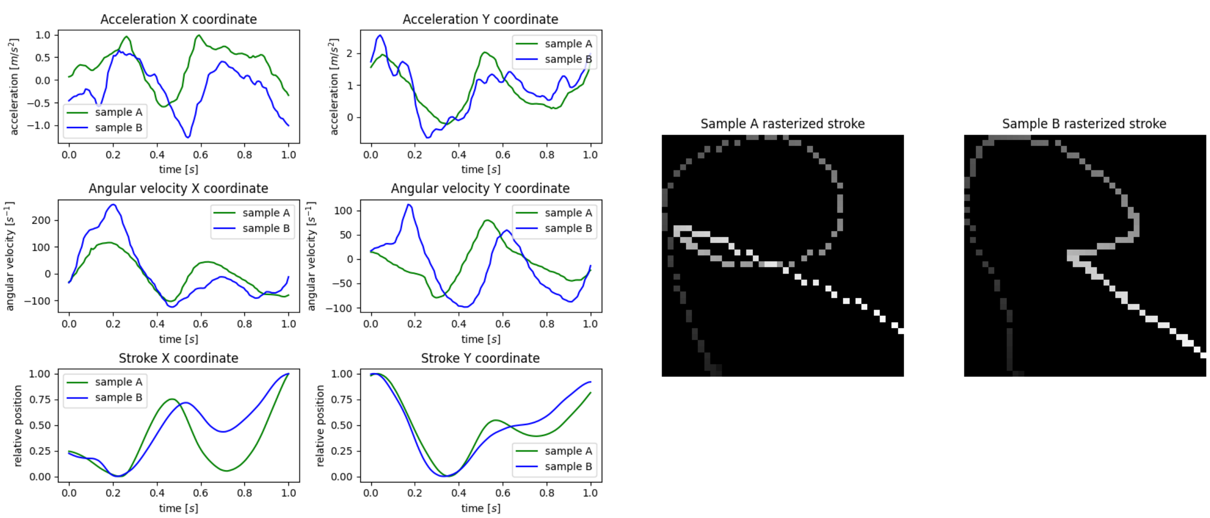

Figure 5: Comparison of two Revelio spells performed with a twist of the wrist and a movement of the whole arm

3 Model selection and hyperparameter tuning

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

- Only_DENS – composed by 2 or 3 fully connected layers,
- Only_CONV – composed by a convolutional layer with 5x5 kernel size followed by 3 times repeated 2x2 kernel max pooling and 3x3 kernel convolutional layers followed by 2 convolutional layers with 1x1 kernel sizes,
- 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 a form of dropout (DO model name suffix) and batch normalization (BN model name suffix).

	Total parameters	Trainable parameters	Non trainable parameters	Size	Optimized size	Training time GPU	Epochs	FLOPS	Full model accuracy	Optimized model accuracy	Full model inference time	Optimized model inference time
baseline_linear	1195	1195	0	6028	2656	0.85 s	10	2410	81.00 %	80.00 %	—	—
Only_DENS_S	24405	24405	0	99436	26888	1.21 s	6	48730	86.00 %	86.00 %	—	—
Only_DENS_L	39705	39705	0	161168	43168	1.28 s	5	79230	91.00 %	91.00 %	—	—
CONV_DENS_S	10181	10181	0	45524	17432	1.95 s	7	537886	90.00 %	91.00 %	—	—
CONV_DENS_L	40069	40069	0	165104	48944	3.40 s	11	2013726	96.00 %	95.00 %	—	—
Only_CONV_S	23877	23877	0	100484	32632	2.23 s	11	584670	93.00 %	93.00 %	—	—
Only_CONV_L	93829	93829	0	380340	105648	5.04 s	11	1906590	98.00 %	96.00 %	—	—
Only_DENS_DO_S	24405	24405	0	99460	26912	1.12 s	4	48730	85.00 %	88.00 %	—	—
Only_DENS_DO_L	39705	39705	0	161200	43192	1.25 s	6	79230	93.00 %	92.00 %	—	—
CONV_DENS_DO_S	10181	10181	0	45596	17488	2.96 s	15	537886	95.00 %	95.00 %	—	—
CONV_DENS_DO_L	40069	40069	0	165184	48984	5.02 s	13	2013726	97.00 %	96.00 %	—	—
Only_CONV_BN_S	24325	24101	224	103292	35168	0.94 s	1	595966	44.00 %	46.00 %	—	—
Only_CONV_BN_L	94725	94277	448	384012	108384	1.01 s	1	1929182	22.00 %	22.00 %	—	—
CONV_DENS_BN_DO_S	10405	10293	112	47928	19888	1.61 s	4	549422	31.00 %	31.00 %	—	—
CONV_DENS_BN_DO_L	40517	40293	224	167928	51496	1.65 s	1	2036798	57.00 %	48.00 %	—	—

Table 1: Summary of analyzed models trained on the smaller data set

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. The epoch number, when the best validation accuracy is reached, is then used to train the model again on the whole train data set. The Adam optimizer with its default settings is used, i.e. learning rate of 0.01. The time necessary for training and the number of training epochs is summarized for each model in table 1. As already mentioned above, the selected seed for random weight initialization as well as data splitting has large impact on the final results. Seed 42 was selected for training on Google Colab[5][6], which has representative results, based on the quite low performance of the linear model[7].

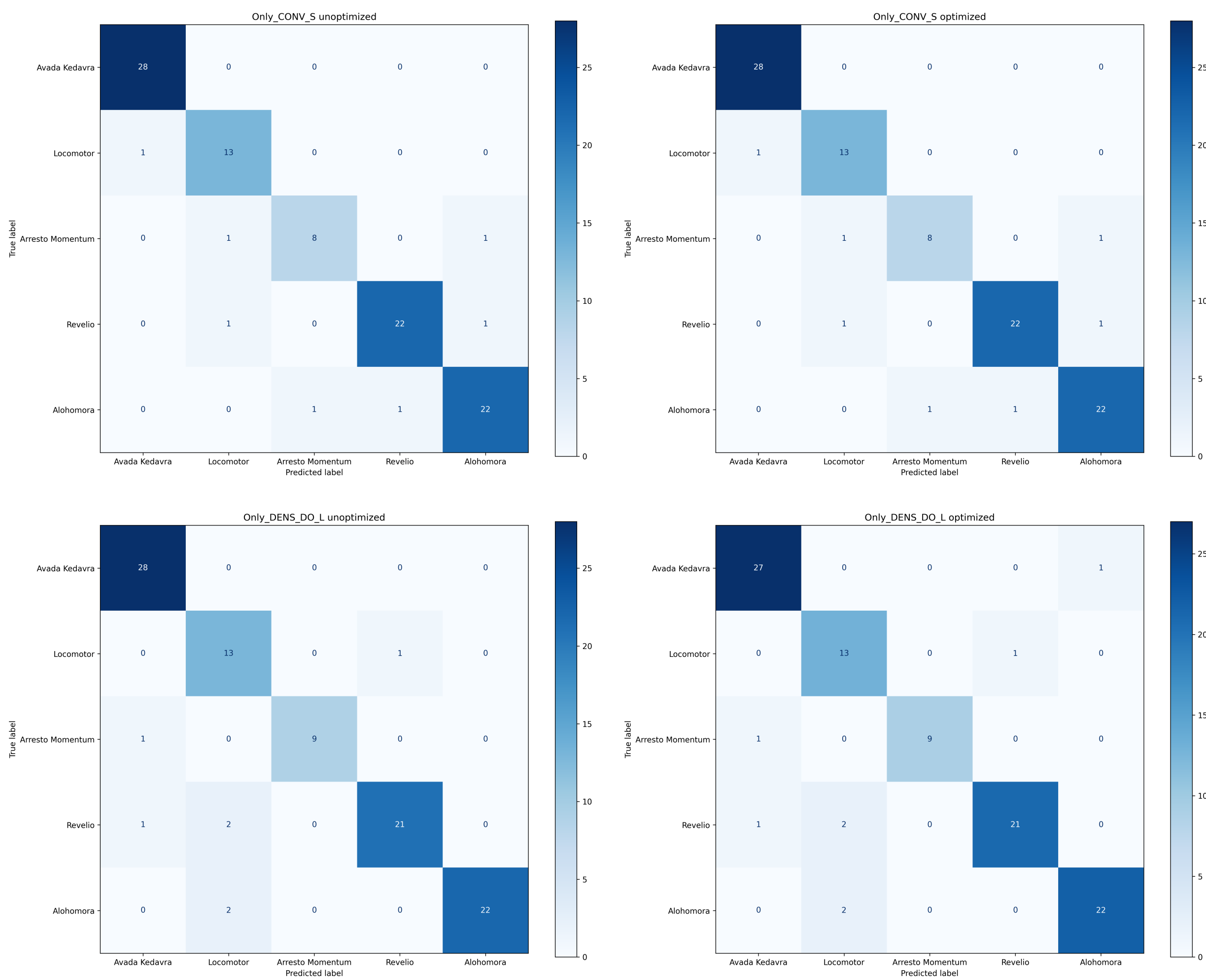

Figure 6: Confusion matrices of chosen models trained on the smaller data set

3.2 Evaluation and results

As can be seen in the table 1, all the not regularized models are performing well, even the simplest possible linear model. The regularization in the form of dropout is effective and in some cases

[5] https://colab.research.google.com
[6] The same seed may produce different results on a local machine.
[7] Try seed 0, for which even the linear model performs with 94% accuracy.

further improves the result. On the other hand, batch normalization makes the models in most cases unusable, as they are not able to train. This is very likely given by the train data set being too small.

4 Inference

The performance of the final solution was captured in a video. The set up displayed in the video can be reproduced by uploading the dense_recognition.ino or image_recognition.ino programs to the Arduino with the `PRETTY_OUTPUT` define set to 1 and running 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 used model, in this case `Only_DENS_DO_L`, is able to correctly classify most of the casted spells with first misclassifications appearing only later in the video.

5 Summary

The lack of large data set makes it harder 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 in to images can further improve the robustness, while still maintaining good performance, as convolution layers are not as computationally expensive as fully connected layers.