POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Smart Tennis Racket with Shot Recognition Capability

Authors: **Francesco Re**
Federico Cattaneo

Group Number: **16**

Academic Year: 2022-2023

Contents

1 | Introduction

This technical report presents an in-depth analysis of the design, development, and implementation of a Smart Tennis Racket. By leveraging a deep learning model this racket can accurately detect and categorize different types of tennis shots, including forehand, backhand, and serve. The system's ability to automatically recognize the number and nature of shots brings a better performance analysis capability to tennis players.

The repository with the code of this entire project can be found at SmartTennisRacket[1]

[1] https://github.com/Fexcatta/SmartTennisRacket

2 | Prototype

Firstly, we created a prototype of the racket. In order to do so we had to mount the Arduino® on the racket. This can be done in a plenty of different ways, we analyzed advantages and disadvantages of some of them.

- **BOTTOM OF THE HANDLE**

  Advantages Good for **minimizing the inertia** when swinging the racket, thus reducing the effort of the tennis player. Additionally, the tangential acceleration is also minimized which is useful to have values compatible with the $\pm 16\, g$ full scale range of the accelerometer.

  Disadvantages It's hard to have it well secured to the racket.

- **INSIDE THE HANDLE**

  Advantages It is very well fixed to the racket.

  Disadvantages It's **difficult to implement** due to the size of the 9v battery.

- **IN THE MIDDLE OF THE RACKET**

  Advantages Pretty good from the inertia point of view, well fixed to the racket and pretty **easy to implement**.

  Disadvantages Difficult to deploy in a real-world scenario.

We decided to go with the latter (see Figure 2.1)

Figure 2.1: Arduino mounted on the tennis racket

3 | Hardware and Firmware

3.1. Hardware

We based our first prototype on the **Arduino® Nano 33 BLE sense rev.2 board** which is a compact and feature rich development board. It comes with a **Nordic® Semiconductor microcontroller**, the **nRF52840** that is **BLE enabled**. The board also features different sensors, in particular the **Bosch BMI270 IMU**, extensively used in our project.

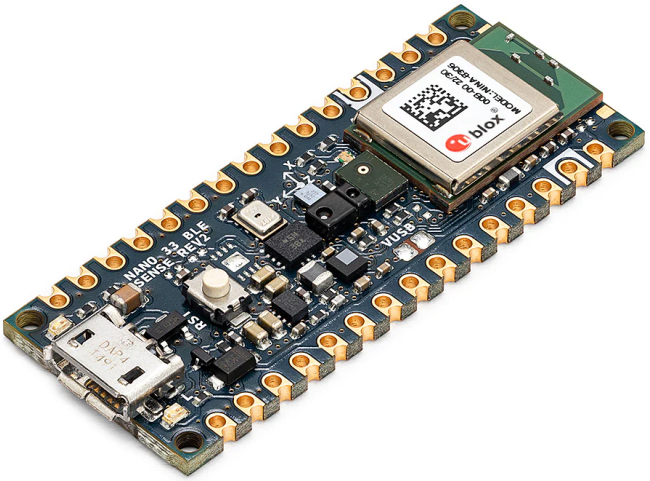

Figure 3.1: Arduino Nano 33 BLE sense rev.2 board

3.2. Firmware

3.2.1. Data acquisition

On the programming side, we developed two different firmwares, one designed to acquire data and the other to perform inference on the final prototype. Both of them shared similar goals: collect data from the accelerometer and gyroscope and use the BLE module to send data.

With no prior knowledge of the intricacies of a tennis swing we set conservative goals for the data acquisition window (up to **10 seconds**) and the data rate (down to $\mathbf{3}\,\boldsymbol{ms}$).

The development started with the full use of the premade Arduino® libraries using the Arduino® IDE. We soon realized that the simple plug-and-play code wasn't sufficient for our performance targets.

This is why we wrote from scratch a new library that fully exploits the hardware timers of the Nordic Semiconductor microcontroller used on the Arduino® board. With this library we could precisely time the **3 *ms*** rate using hardware interrupts in order to acquire data from the IMU sensor.
Another challenge came from the default Arduino® BMI270 library which is easy to use but lacks any form of configuration, since it misses functions to modify the behavior of the sensor. We went through the library and changed how the sensor is configured, setting the output data rate to **800 Hz** for both the accelerometer and gyroscope and increasing the full scale range to **±16 *g*** and **±2000 *dps*** respectively. We tested the amount of time it took for a single acquisition of the sensor data through the built in function and found out that it would take up to **10 *ms*** per sensor read. An outcome which is completely out of the requirements. As a result we wrote custom functions that use the **I^2C interface** to directly talk to the sensor reducing the acquisition time to just **500 *µs***.
The last adjustment we made involved the BLE library. We discovered that every couple of milliseconds the Bluetooth® module would take over and delay the sensor data acquisition by up to **6 *ms*** which was too much to meet the **3 *ms*** data rate. In order to solve this issue we had to disable a timer call in the ArduinoBLE library. This change could limit the capabilities of the module but it didn't affect our use case while instead solving the occasional lag problems during data acquisition.

3.2.2. Data transfer

Once the data acquisition part of the firmware was solved we needed to find a way to send data to the PC for storing and training the AI model. We chose the BLE module instead of a wired solution because it was way more practical to not have a long cable dangling from the tennis racket which could also affect the outcome of the sensor readings. Since the Bluetooth® Low Energy protocol isn't made for long byte transfers, we had to come up with the most optimized way of sending a data packet. We went through different iterations starting from a formatted string containing x, y, z values of the accelerometer and gyroscope which would take up to **60 seconds** for a **3 *s*** window of data. The final solution we found was to group together **10 sets of floats** (more would exceed the BLE data packet limit), and with that we could lower the transmission time to few seconds.

3.2.3. Final firmware

For what concerns the final prototype firmware we imported the TensorFlow Lite model and its accompanying library. As a starting point we used a data window of **3 seconds**, but soon we realized that a significant shorter time span was sufficient. Therefore we lowered the window down to **900 *ms*** decreasing the RAM usage and time between inferences substantially. The entire final program occupied around **60%** of the total **1 MB** of flash memory. RAM usage was in the order of **35%** to **40%** including the full model and the memory necessary to store the accelerometer and gyroscope live data.

Max RAM usage	42% (110.4 kB)
Flash memory used	61% (608.32 kB)
IMU data rate	3 ms
Inference time	$\approx 25ms$
Time between acquisitions	$\approx 1s$
9V battery life	10 h+

Table 3.1: Firmware performance metrics

4 | Data acquisition

4.1. Data acquisition software

The data acquisition phase was crucial for the success of the project, for this reason we decided to develop a **custom acquisition software** that accelerated the process considerably. The software (shown in Figure 4.1) is constantly connected via Bluetooth® to the Arduino® board, which continuously listen for spikes in accelerometer data, when one is detected a new sample is taken and sent to the acquisition software. After a few seconds, the time needed for the transmission, the sample is displayed on screen and **immediately labelled**. Once the correct label is selected, the sample is saved to a CSV file in the dataset folder.

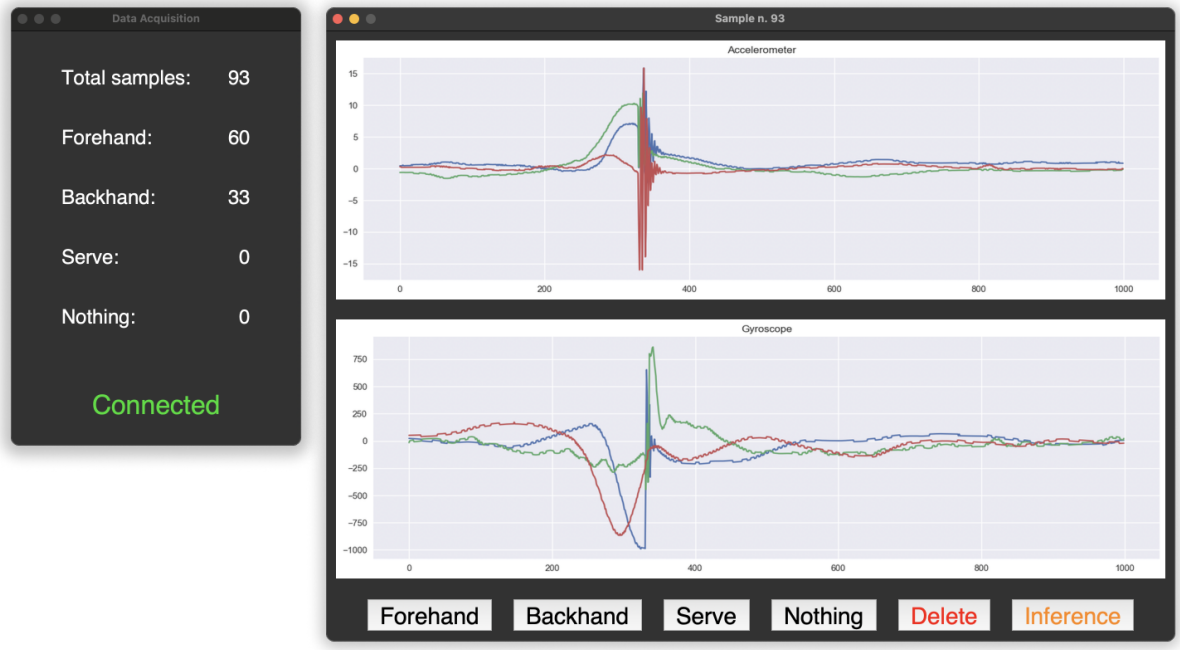

Figure 4.1: Data acquisition software

4.2. Dataset

4.2.1. Dataset structure

The output of the previously described software is the dataset folder which is structured in 4 directories: **forehand**, **backhand**, **serve** and **nothing** (see Figure 4.2). The first 3 classes are self explanatory since

they represent directly the 3 different types of shot, the **nothing** class, instead, is used as a sink for all the movements that can be done by the player during a match that are not shots. For example: stopping the ball, dropping the racket on the ground, making the ball bounce etc...

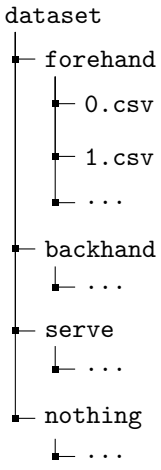

Figure 4.2: Dataset structure

In each of these directories there are several **CSV** files. Each file represents a shot, containing 1000 measurements (one for each row) acquired every *3 ms*. The structure is the following:

```
accX,accY,accZ,gyrX,gyrY,gyrZ
-0.065429,-0.778808,0.684570,-56.945800,-41.198730,5.249023
-0.067871,-0.781738,0.581054,-57.434082,-43.640136,5.126953
-0.066894,-0.783691,0.586425,-58.288574,-46.142578,4.882812
...
```

4.2.2. Dataset composition

Thanks to the custom data acquisition software, we managed to acquire a total of **544 shots** (details in Table 4.1) from 3 different people, all right-handed. As a result of this, the model will not perform well for left-handed players, this is an issue that must be addressed in future developments (see section 6.3). During this process we also paid attention to acquire the same amount of shots for both the **orientations of the racket**.

Class	Person 1	Person 2	Person 3	Total
Forehand	61	60	30	151
Backhand	61	60	30	151
Serve	61	60	30	151
Nothing	61	0	30	91
Total	**244**	**180**	**120**	**544**

Table 4.1: Dataset composition

5 | Model

The core of the project is the Convolutional Neural Network used for classification, which is deeply explored in the following paragraphs.

5.1. Pre-processing

After some training, we discovered that we could discard the majority of the 1000 measurements by retaining only 300 measurements centered in the accelerometer spike. This corresponds roughly to a time window of 1 second which is enough to obtain a great accuracy. A sample extracted from the dataset can be seen in Figure 5.1.

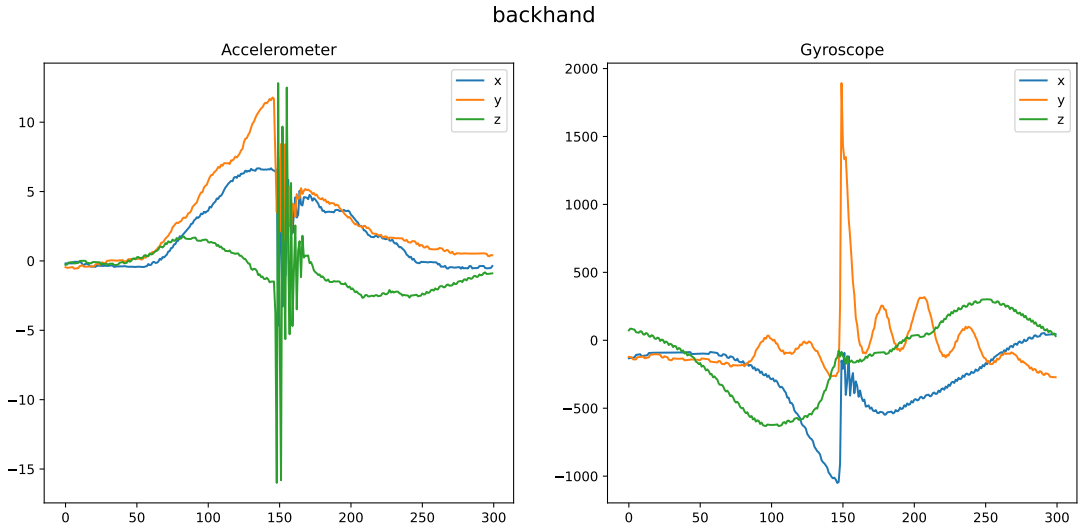

Figure 5.1: Dataset sample

The input to the network is a tensor with shape **(B, 300, 6)** where **B** is the batch size, **300** is the number of measurements and **6** is the number of channels per measurement (3 for the accelerometer and 3 for the gyroscope).
The input tensor is **normalized** by removing the mean and dividing by the standard deviation computed among each measurement of the training dataset.

```
# dataset is a list of tensors with shape (300, 6)
means = np.stack(dataset).mean(axis=(0,2), keepdims=True)
stds = np.stack(dataset).std(axis=(0,2), keepdims=True)
```

After computing the means and the standard deviations, we can exploit the broadcasting property of numpy and compute the normalization in this way:

```
normalized = (tensor - means) / stds
```

Means and standard deviations are computed once in the training phase and then are directly embedded in the final model.
For convenience, a custom Normalization layer was defined.

```
class Normalize(tf.keras.layers.Layer):

    def __init__(self, means, stds):
        super(Normalize, self).__init__()
        self.means = means
        self.stds = stds

    def call(self, inputs):
        return (inputs - self.means) / self.stds
```

5.2. Architecture

We began with an initial architecture inspired by a CNN employed in a Human Activity Recognition project. This architecture consisted of **38,314 parameters** and an estimated **352,920 FLOPS**.
Through a series of iterations of pruning, using a trial and error approach, we arrived at the final model, which now comprises **1,456 parameters** and **37,200 FLOPS**. This represents a reduction of approximately **96.2%** in the number of parameters and a corresponding **89.5%** decrease in FLOPS count with the **same loss and accuracy**.

	Initial	Final	**Reduction**
Parameters	38,314	1,456	**-96.2 %**
Model size (w/o quantization)	153 KB	5.8 KB	**-96.2 %**
FLOPS count	352,920	37,200	**-89.5 %**

Table 5.1: Pruning details

The model consists of **2 Conv1D layers** interleaved by a **ReLU** and **Dropout** to reduce overfitting. Afterwards, a **MaxPool1D** with strides 2 is applied in order to reduce the dimensionality of the input to the **Dense** layer whose output is sent through a **Softmax** obtaining probabilities.

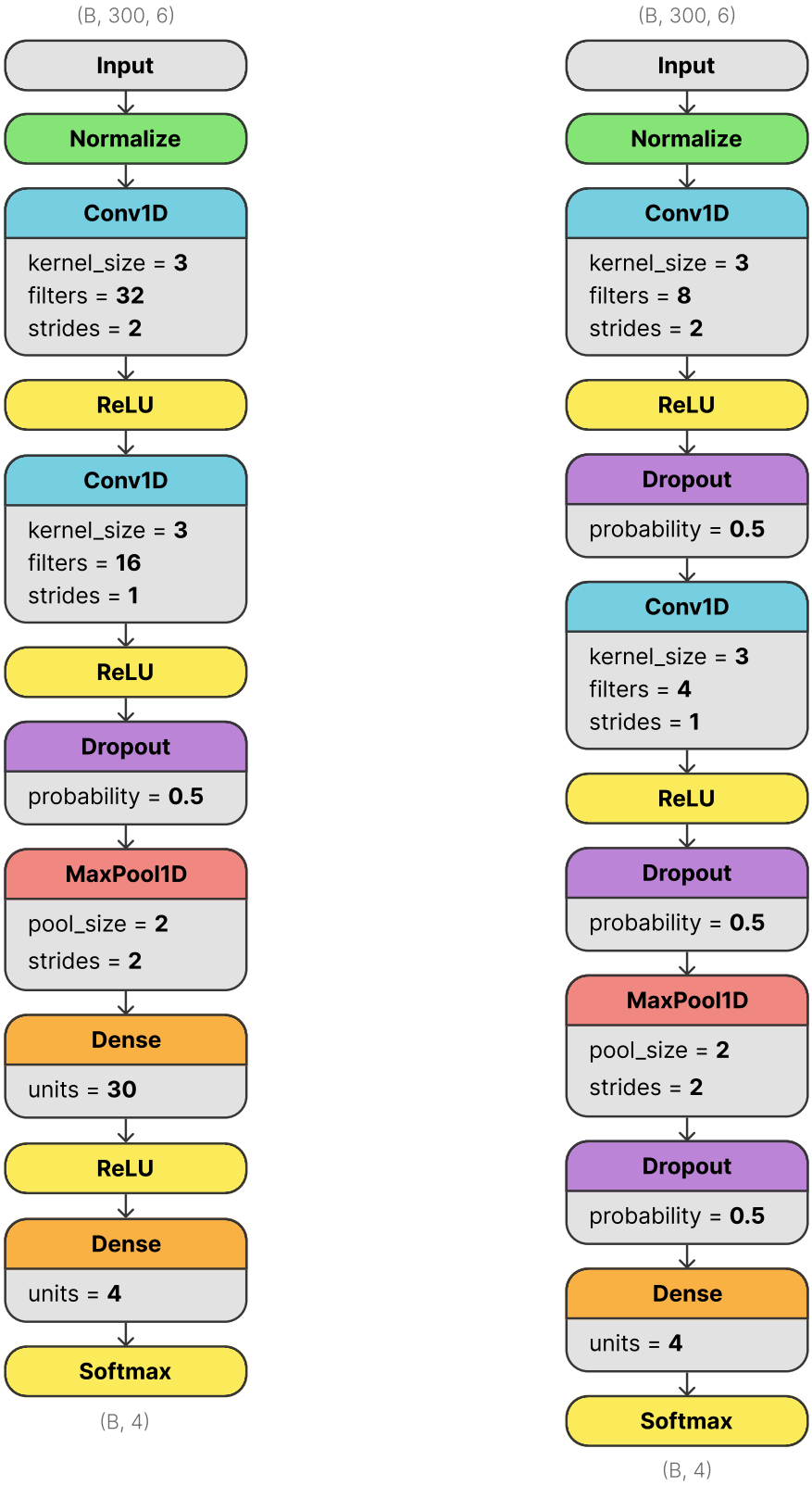

Figure 5.2: Initial architecture (left) and final architecture (right)

The model is defined as a subclass of the *keras.Model* class in this way:

```
class Model(tf.keras.Model):

    def __init__(self, means, stds):
        super(Model, self).__init__()

        self.norm = Normalize(means=means, stds=stds)
        self.conv1 = l.Conv1D(filters=8, kernel_size=3, strides=2, padding='same',
        ↪  activation='relu')
        self.dropout1 = l.Dropout(0.5)
        self.conv2 = l.Conv1D(filters=4, kernel_size=3, strides=1, padding='same',
        ↪  activation='relu')
        self.dropout2 = l.Dropout(0.5)
        self.max_pool = l.MaxPool1D(pool_size=2, strides=2)
        self.flatten = l.Flatten()
        self.dropout3 = l.Dropout(0.5)
        self.dense = l.Dense(units=4, activation='softmax')

    def call(self, inputs):
        x = self.norm(inputs)
        x = self.conv1(x)
        x = self.dropout1(x)
        x = self.conv2(x)
        x = self.dropout2(x)
        x = self.max_pool(x)
        x = self.flatten(x)
        x = self.dropout3(x)
        x = self.dense(x)
        return x
```

5.3. Training

The model was trained for **800 epochs** with a batch size of **32** using **Adam** optimizer. The train/validation split was 80%, which corresponds to a total of 435 samples for training and 109 for validation. The results can be seen in Table 5.2 and the plots can be seen in Figure 5.3.

	Train	**Validation**
Loss	0.0025	0.0105
Accuracy	100%	100%

Table 5.2: Training accuracy and loss

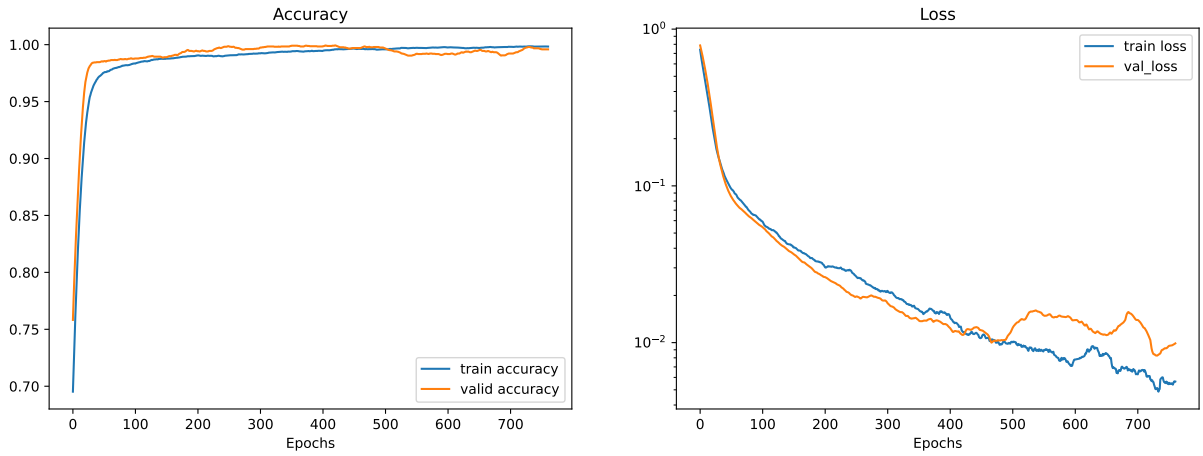

Figure 5.3: Training accuracy and loss plot

5.4. Quantization

In order to reduce furthermore the size of the model we applied **INT8 Post Training Quantization**. The quantization process is done automatically by providing a representative set of the input data. The weights and the activations are quantized while the input and the output tensors are kept in **FP32**. This simplifies the use and the debugging of the model in the firmware by avoiding conversions before and after inference without introducing significant performance drops.
Since this process generates a quantized model with an incredibly high accuracy and a loss value (computed on validation dataset) that is almost comparable to the non-quantized model, we decided to not use Quantization Aware Training.
In conclusion, the final quantized model has an accuracy of **99.1%** (computed on validation set), weights **8600 bytes** and takes approximately **25ms** to run inference on the Arduino.

5.5. Final test results

We did a final test of the complete system doing inference directly on the board with a completely new and unseen dataset. As a result, 73 out of 76 shots were correctly categorized *by the model*, giving an accuracy of **96%**, while 70 out of the 76 shots were correctly classified *by the complete system*, giving a **92%** accuracy. The reason behind this are:

- The **accelerometer threshold** being a little bit too high ($13\,g$) which lead to missing some shots
- The **model threshold being set to 80%** which discarded all the predictions below that confidence level.

The confusion matrix of this final test can be seen in Figure 5.4

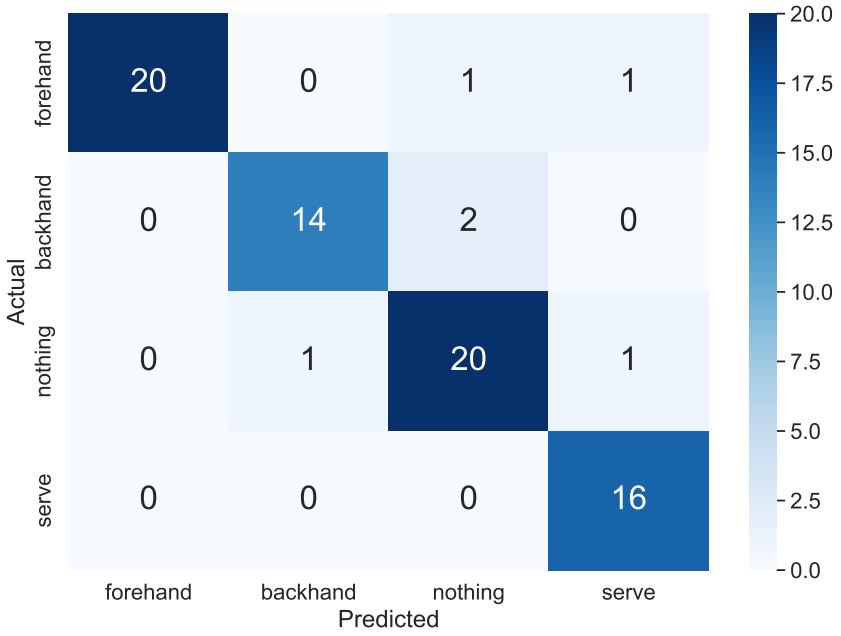

Figure 5.4: Test set confusion matrix

6 Future developments

6.1. Firmware improvements

For what concerns further improvements to the firmware, there is still room for minor optimizations and feature additions. One of the main contributions to program size is the TensorFlow library and in order to limit its memory footprint we could tweak the files included in the firmware.

A feature that could be essential for the final product is a **battery saving mode**. While not in active use, the firmware still records data at a 3ms rate. To prevent too much power draw we could implement a "triggering mechanism" where the board acquires data at a much lower frequency, for example every second and when an acceleration or rotation threshold has been reached enter in the usual active mode. This technique would help preserve battery life and improve the user experience: it would be possible to leave the tracker on and have it available instantly without having to switch it on and off continuously.

Another useful feature could be the addition of **user custom settings**. Through the BLE connection we could let the user change selected parameters in order to cater better to each specific person use cases. As an example, the user could change the **inference result certainty** in order to commit to a specific swing. Each player could also adjust the **trigger threshold** needed to record a shot if there are too many false positives or too little acquisitions.

6.2. Hardware improvements

The most notable future development to the prototype undoubtedly must be the hardware.

For this project we used an Arduino® board that allowed us to build our ideas quickly and efficiently, however it cannot be the solution for a final product due to cost and features. The development board comes with many different sensors which we don't use, and therefore it exhibits an inefficient use of space.

We need to design a custom PCB with only the essential features: a microcontroller, a BLE module, a IMU and a power delivery system.
For what concerns the brain of the device, we found a good tradeoff between performance, features and power utilization in the **STM32-L4-96 family**. These microcontrollers are part of the Ultra Low Power line of STM® and come with **320 kB** of RAM and a **80 MHz** system clock which is a notable upgrade over the nRF52840 used on the Arduino® board.
An additional highlight on the hardware improvements is the substitution of the 9V battery used for the prototype with a **rechargeable Li-ion** one. This change allows to reduce the size and weight of the device by removing the bulky alkaline battery, and to improve the user experience by allowing users to simply recharge the product without having to change every so often a battery.

6.3. Software improvements

During the development of this project we focused specifically on shot recognition even though there are other useful metrics that could be estimated, including **swing speed**, **ball speed**, **shot power** and **ball impact location**. Acquiring a good dataset that can enable these features is quite challenging and it's beyond the scope of this project, but it would certainly be a future development for a production device. Even more important, in order to sell this product, is the implementation of an **app that interacts with the device** and let the user keep track of all the sessions recorded.
Lastly, it is necessary to address the issue of left-handed players, either by acquiring the same amount of data from lefty people and training the model also on them or by creating a new model and let the user specify its preference from the app.

List of Figures