Lebanese University, Faculty of Science - V

Project Report for I3308

Table Tennis Shots Detection Using Machine Learning

Ali Koteich

Under the supervision of:
Dr. Mohamad Chaitou

October 2, 2022

Abstract

This project aims to use machine learning skills to take the game of table tennis one step forward by giving the players a self-analytic model which detects the type of shot they perform during matches or training sessions, including basic statistics around their gameplay. This whole process consists of several steps, from data collection, which requires a lot of work since the dataset is not already preprepared so it has to be collected manually with the available tools. Data preprocessing, to convert this raw data collected into a clean and meaningful dataset which will include some data analysis to track how our data is behaving. And finally, creating the machine learning model where we will use the Artificial Neural Network to manipulate our data and come out with useful predictions which in our case is detecting the type of shot the player is performing.

Contents

List of Figures

List of Tables

Introduction

Over the last decade, the field of artificial intelligence has evolved exponentially due to the demand for its applications and due to the growth of fast and high-quality computers. It invaded every aspect of our lives and we see it in almost all fields such as medicine, economy, marketing, sports, social media, etc.

Talking about sports, we can notice how artificial intelligence made significant changes to this domain. For example, in football, sensors can be used to collect information about other features of a football match – the physical fitness of players can be assessed by collecting medical data about them in real-time. While in tennis, AI is helping coaches as well by understanding metrics such as spin, speed, placement, and the position of players, coaches can provide insight into improved decision-making. AI can deliver clips of every player's winning tactics or visual analysis to help coaches develop counter-strategies. And the list goes on for all other sports.

In table tennis (Figure 1), a good application of AI is a model that detects the type of shot a player is performing, whether it is a forehand topspin, backhand topspin, forehand slice, backhand slice, or just a bad form shot which is none of the recent shots, and makes a statistics about the gameplay so the player can analyze his performance.

Figure 1: Table Tennis game

Chapter 1

State of the Art

Few works were done on this topic, so we will cover these approaches and also similar work done using gyroscope and accelerometer to record data. The only same approach was a model that detects how well a shot is performed [1], and similar work using the same sensors in our project but to detect if a person is walking or running [2].

In [3] the author used the Arduino Nano 33 BLE Sense to get the benefits of the gyroscope and accelerometer to record the shots performed. This approach was a little bit different, without going through any coding aspect. The tiny motion trainer website [4] was used which has the feature to connect the Arduino board with Bluetooth, collect the data and label it, and finally train the model, all done without writing any code, just using the user interface. When recording the shots, a capture threshold of 0.40 and 20 samples of data were recorded for each shot and a delay time of 0.2 sec between shots. While in our project, a 0.30 capture threshold, 10 samples of data recorded, and 0.6 sec delay time between shots is used. After collecting the dataset and training the model, they claimed a training accuracy of 99% and training loss of 0.0133%, while in my case, training accuracy reached 0.988% and training loss 0.006%, which seems to have similar results. For the setup part, a 3D printed case fits the Arduino board with a 5v/1A booster and a small 3.7v/400mAh LiPo battery to power the board(Figure 1.1).

Figure 1.1: 3D printed case fitting the setup tools

The second approach has a completely different goal [5] but the technique is the same, the author used the gyroscope and accelerometer sensors of the phone to detect whether he is running or walking, there is no capture threshold since the person walking/ running is in continuous motion. The number of samples used is 12 samples of data to record each step of movement. The accuracy of the model achieved was 99% too, the number of hidden layers is 2 with 15 neurons each, and the activation function used was ReLU, while in my case, the number of hidden layers is 1 with 30 neurons and ReLU activation function. The author trained the model and then converted it to coreML which is a framework developed by Apple used to apply a machine learning algorithm to a set of training data to create a model to run on Apple devices.

Chapter 2

Model Overview

The first impression when trying to build the architecture of the model is to specify a logical methodology for the work. We should ask ourselves a few questions before moving on to the implementation part of the project. For example, Is my data supervised or unsupervised? Should I implement a classification or regression model? After answering these questions we can start collecting and doing the preprocessing side of the project.

2.1 Supervised Learning

Supervised learning is when we train the machine using data that is well labeled. Which means some data is already tagged with the correct answer. After that, the machine is provided with a new set of data so that the supervised learning algorithm analyses the training data and produces a correct outcome from labeled data.

In our case, we can visualize the structure of our data since we will record the shots played using the IMU sensors, and expect an output of one of the types of shots. Thus, our data is supervised since the features and labels are well defined. Moreover, we can consider that we are working with a multi-label classification model.

2.2 Multi-Label Classification

If we are using a multi-label classification to implement our model, we have to specify the features and labels associated with our dataset. For the features, the model will detect the different types of shots using the gyroscope and accelerometer coordinates (gX, gY, gZ, aX, aY, aZ), these 6 coordinates will be the features for the model and the labels will be as follows:

- forehand topspin
- forehand slice
- backhand topspin
- backhand slice
- bad form shot

2.2.1 Forehand Topspin

Forehand topspin is performed by starting with your hand holding the table tennis racket towards the outside of your body, start from a low hand position close to the knee, and as

the ball approaches, move your hands all the way up while brushing the ball to generate spin till your hand is close to your head.

2.2.2 Forehand slice

Forehand slice also known as forehand backspin, is performed by starting your hand at a high position at the level of the head, and when the ball approaches, move your hand down while brushing the ball to generate a spin opposite to the spin generated in the forehand topspin.

2.2.3 Backhand topspin

Same motion as the forehand topspin but instead of the hand starting from outside the body position, it starts from inside (Right-handed players position the racket near their left hand towards the inside of their body) and move your hand all the way up while brushing the ball to generate spin.

2.2.4 Backhand slice

Same motion as the forehand slice but instead of the hand starting from outside the body position, it starts from inside the body position.

2.2.5 Bad form shot

This label represents any shot that is not classified as one of the above 4 types of shots, maybe a push shot, a smash, random motion of the racket, or any unknown shot.

Chapter 3

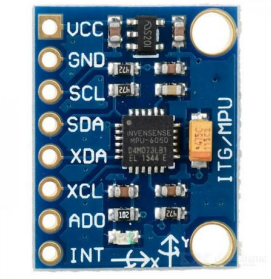
Data Collection

Data collection is considered the foundation of the Machine Learning model building. Without data, the concept of building a Machine Learning model is pointless. This covers the methods used to collect and manipulate this raw data to get the best value out of our model.

3.1 Tools Used for Collecting Data

In this project, we want to collect the needed data to fit it in our model and detect the types of shots, and since there is not any dataset online related to our topic, we have to manually collect the dataset.

To record a shot there are multiple ways, first, we can use a camera to record the motion of the hand while performing the shot, but this method does not seem efficient since sometimes the player performs a shot from under the table or far away from it, or even from a different angle where the player's body blocks the view of the camera. In addition to that, using a camera consumes high power, and even training the model and dealing with image processing is heavy for the machine. Another method is by using a 6-axis IMU sensor which is a gyroscope sensor (Figure 3.1 (a)) to track the motion and rotation of the racket, and an accelerometer sensor (Figure 3.1 (b)) to track the acceleration of the racket while performing a shot.



(a) Gyroscope sensor

(b) Accelerometer sensor

Figure 3.1: Sensors used to collect data

If we look deeper into the technical aspect, to record a shot, the sensors must be attached to the racket, and each sensor is connected through its pins to a microcontroller (for example an Arduino board) or a computer (a raspberry pi 4, raspberry pi zero w or

any other computer) to store the data and manipulate it later. This whole process is not practical, instead, we can use the Arduino Nano 33 BLE Sense [6] which supports both sensors embedded on the board in addition to multiple other sensors as shown in (figure 3.2).

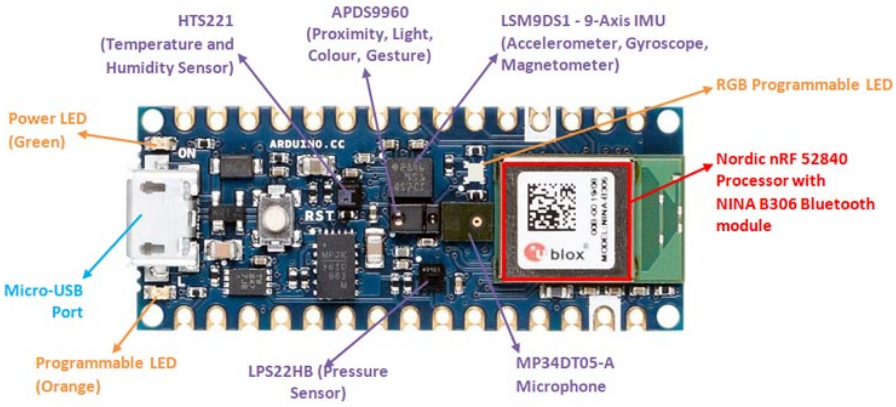


Figure 3.2: Arduino Nano 33 BLE Sense

This Arduino board weighs 5g only, which we can attach it to the bottom of the racket with a small BMS and LiPo Battery to power it on, then we use the builtin BLE to send the data recorded to the computer (Figure 3.3). But in my current situation, I didn't go through making the full setup since I don't have a LiPo Battery. I only connected the Arduino board to the computer via a long USB cable and started collecting the data.

Figure 3.3: The racket's setup used to record the shots

3.2 Data Preprocessing

After setting up the Arduino board and the sensors using the Arduino IDE Platform, the sensors start recording data (aX, aY, aZ, gX, gY, gZ) in an infinite loop. After

performing a few shots with my racket, and saving the output in a CSV file using an Arduino external library called ArduSpreadsheet, this CSV file is loaded in the Jupyter Notebook and stored in a dataframe using the pandas python library [7] which is used to import data from files, manipulate and analyze it, then plotted it on a graph using the matplotlib python library [8] which is used for data visualization.

3.2.1 Filtering the Shots

If we take a close look in the graph (Figure 3.4 (a)), we notice that the shots are not clearly readable, there is a lot of noise mixed between the shots which makes it hard to deal with such data. We have to understand first where the noise is coming from so we can filter it. To solve this issue, we have to realize that a table tennis player performs various shots in a single match. In fact, these shots are not synchronized and there is no specific delay between each shot, also the player might move his racket randomly between points (for example, using the racket to point at a certain position, using it as a fan, blowing away a fly, etc.). All this motion is recorded and that's the reason behind all this noise.

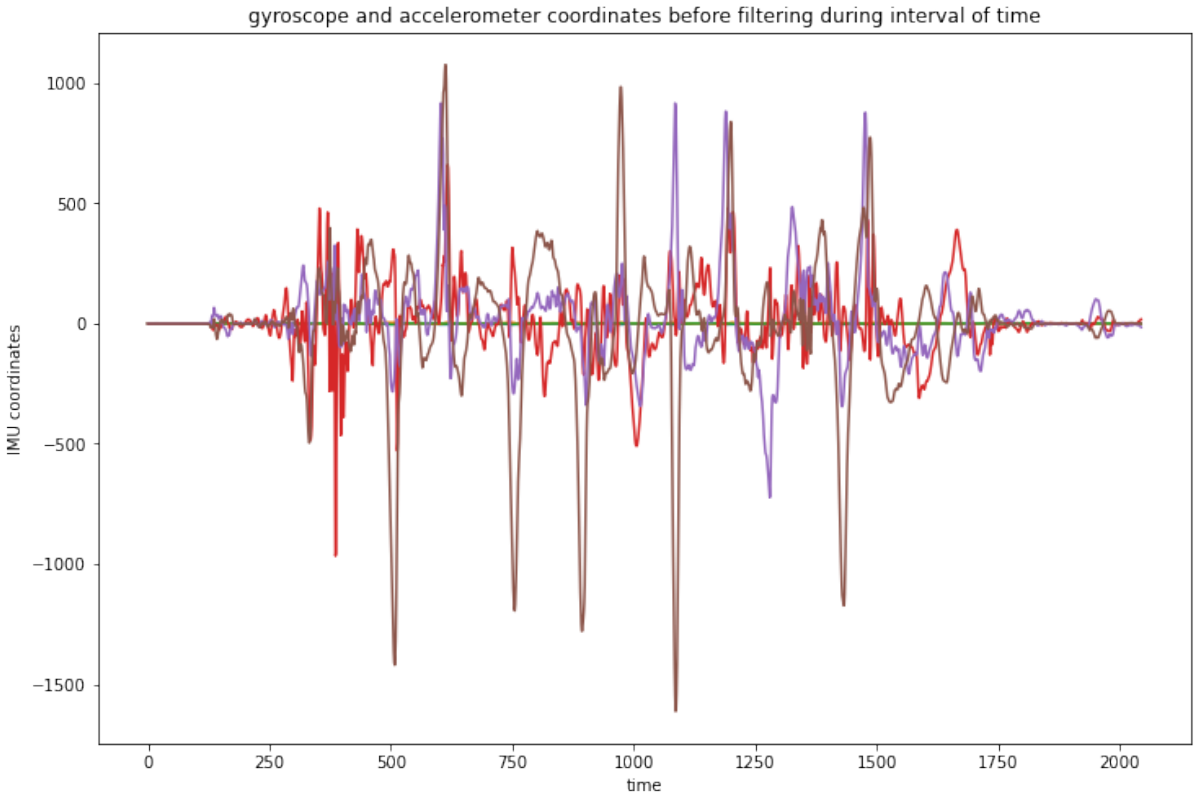
After studying the factors that cause the noise in our dataset, one way to fix it is by keeping the sensors in a standby mode (reading data from the sensors but without recording any output data) until a shot is detected. This requires us to set a certain threshold for the sensors and a trigger that triggers the board to start recording data when the threshold is reached (Figure 3.4 (b)).

$$Threshold = \frac{|aX + aY + aZ|}{4} + \frac{|gX + gY + gZ|}{2000} \Rightarrow Threshold = \frac{Threshold}{6} \tag{3.1}$$

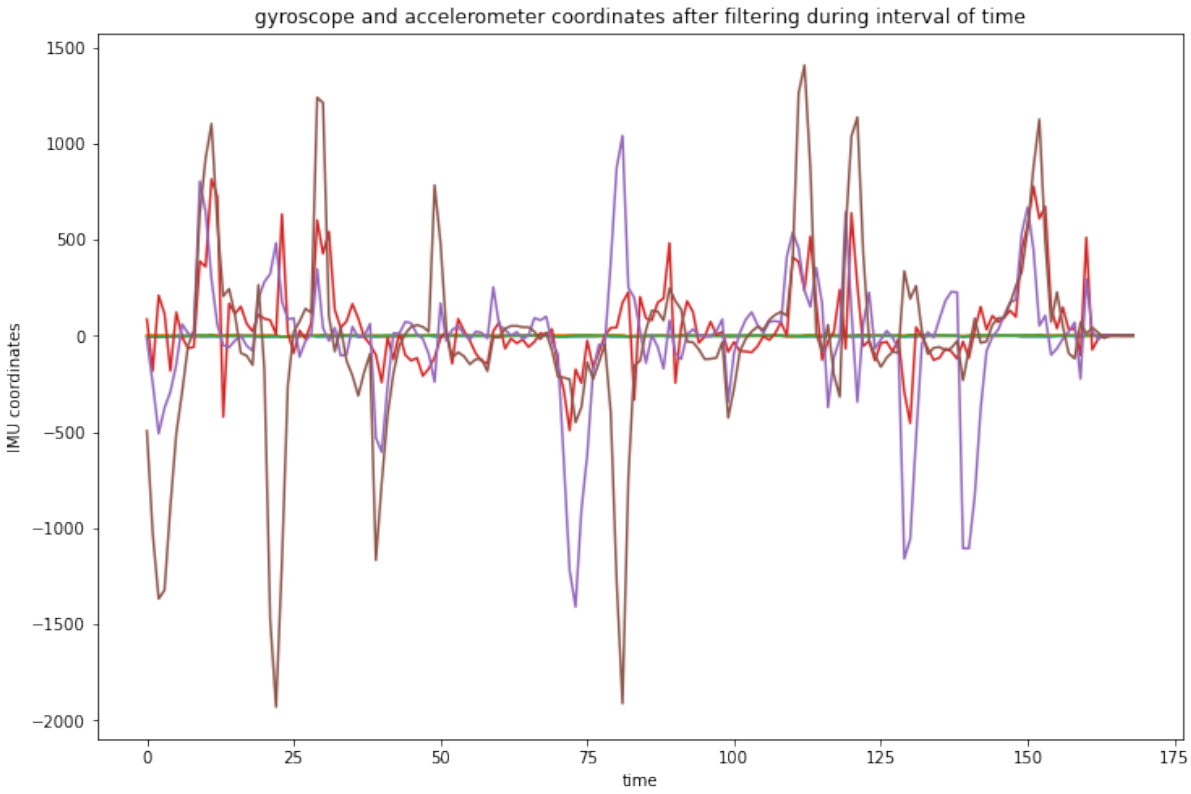
A shot cannot be described as an instant of time, or a point. Recording 1 sample of each coordinate of the 2 sensors doesn't represent a shot. In fact, it is represented as an interval of time (from the start till the end of the racket's motion), so we need to set the boundaries of each shot, when it starts and when it terminates. After several testings and experiments, we found out a shot lasts about 300 ms, so we can capture 10 samples of coordinates, with 30 ms between each sample. We also added 600 ms delay between shots to avoid recording unnecessary motion between them. The algorithm 1 will work as the following:

Algorithm 1: Recording complete clean shots

```
while True do
    IMU sensors read data;
    Define Threshold equation;
    Set Trigger to 0.30;
    if Threshold > Trigger then
        foreach 10 samples do
            Record gyroscope and accelerometer coordinates;
            delay 30 ms;
        end
        delay 600 ms;
    end
end
```



(a) IMU sensors coordinates recorded before filtering



(b) IMU sensors coordinates recorded after filtering

Figure 3.4: Sensors used to collect data

The final form of a single shot after storing it in a dataframe is a 10x6 matrix (10 samples of data recorded for the 6 coordinates of the gyroscope and accelerometer sensors) (Figure 3.5).

	aX	aY	aZ	gX	gY	gZ
0	-2.51	4.00	0.83	0.49	37.11	-187.44
1	-4.00	4.00	-0.28	42.85	80.14	-386.84
2	-4.00	4.00	-1.69	-2.26	50.35	-734.07
3	-4.00	4.00	-1.58	57.80	-136.78	-1219.06
4	-4.00	2.51	-0.55	31.98	-83.74	-1232.12
5	-4.00	0.32	0.16	-62.01	-115.91	-995.24
6	-4.00	-1.47	1.44	-146.12	-123.66	-653.20
7	-3.37	-1.53	0.52	-82.34	-13.24	-327.39
8	-2.28	-1.38	1.04	-94.60	-42.60	-125.12
9	-1.40	-0.73	1.63	-84.72	-49.93	27.10

Figure 3.5: A shot recorded after filtering

3.3 Data Processing and Feature Engineering

Now after we succeeded in recording filtered shots and setting their boundaries, we have the shot as a form of a 10 x 6 matrix, and as we know feeding data into a model must be a column matrix and not an nxm matrix. So in our case, we have to combine all 6 coordinates in a single learning iteration which means that we need 6 separate models for each coordinate, and that is not a practical solution.

Instead of dealing with 6 models, we can apply an ensemble technique which combines the predictions from the 6 models. In order to apply this technique we have to transform our 10x6 matrix into a 1x60 matrix as shown in (Figure 3.6).

Figure 3.6: Structure of a shot as 1x60 matrix

After transforming a shot into a 1x60 matrix, the general form of input data for the model is getting clear and simple, with a single row representing a single shot. But before feeding our model the data, we have to label the shots in this dataset. If we look at our shots we find out that we have 5 different types of shots and since Forehand is a perfect

predictor of backhand, and topspin is also a perfect predictor of a slice, and to avoid multicollinearity we use 2 labels only; forehand and topspin instead of 4 labels and set it as 1 or 0, in addition to a third label which is the bad form column (Figure 3.7).

	0	1	2	3	4	5	6	7	8	9	...	53	54	55	56	57	58	59	forehand	topspin	bad form
0	-2.51	-4	-4	-4	-4	-4	-4	-3.37	-2.28	-1.4	...	-1219.06	-1232.12	-995.24	-653.2	-327.39	-125.12	27.1	1	1	0

1 rows × 63 columns

Figure 3.7: Example of a forehand topspin shot dataframe

3.4 Dataset Deep Insight

A dataset is a collection of input features related to each other that has a specific label that describes it. In our case, we specified the features and labels of our data, and did all the needed processes to filter, sort, and transform our raw data into clean shots in order to be fed to the model to produce meaningful and precise predictions.

In general, datasets in any Machine Learning model are collected in several ways, using online APIs, open source datasets, data collected at certain companies, or even using sensors. Datasets can range from a few megabytes to hundreds of gigabytes, and the size of the dataset plays a big role in the Machine Learning model's performance. Since we are dealing with only 5 labels of shots. and since each shot has its specific range of motion, we don't actually need a large amount of data recorded to cover all forms of shots, after several experiments I found out that multiple hundreds of samples of each type of shot are enough for the model to make accurate predictions.

After I collected more than 4000 shots manually (performing all these shots by my own with all forms possible) and after several experiments, the final form of our dataset consists of 1951 input data, where each input represents a single shot with its label. The shots were distributed as follows:

- 400 forehand topspin samples
- 400 backhand topspin samples
- 401 backhand slice samples
- 400 forehand slice samples
- 350 bad form samples

The overall size of our dataset was only 682 kilobytes, which is a really small dataset, yet it still gets the job done. We can add more samples from different players too in order to make sure that the results cover all the possible odds out there.

Chapter 4

Model Architecture

Building a Machine Learning model is not as hard as we might think, choosing the correct algorithm is a critical point in getting accurate predictions. Since we are dealing with a classification model, multiple algorithms can be used. One issue we might face when working with our dataset is that it has 60 features for each input with unclear form (the 60 features are just frames representing the coordinates of the sensors), as a result, our data is complex and nonlinear. In this case, an artificial neural network seems to work very well with such type of data.

4.1 Splitting and Normalizing the Dataset

Splitting the dataset into a train set and a test set is important to evaluate how well our machine learning model performs. The train set is used to fit the model, while the test set is used for predictions. Splitting the dataset prevents overfitting, which occurs when our machine learning model tries to cover all the data points or more than the required data points present in the given dataset. Because of this, the model starts learning noise and inaccurate values present in the dataset, and all these factors reduce the efficiency and accuracy of the model.

In order to split our data we need to specify first our feature data X and labels y in our dataset, then we will use the train_test_split() function from the scikit-learn library [9] to split the dataset into 70% training data, and 30% testing data (Figure 4.1). The shape of the training data becomes (1365, 60) and the testing data (586, 60).

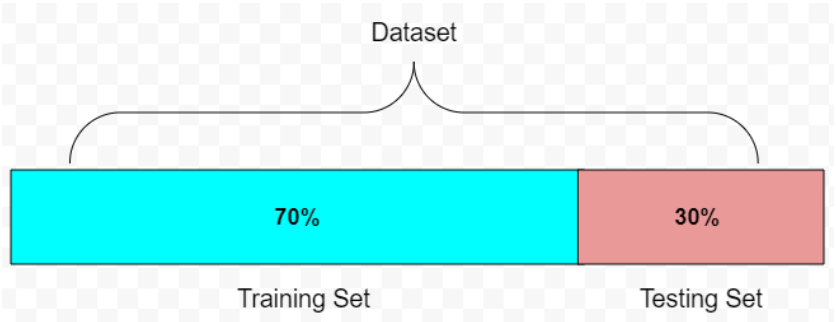


Figure 4.1: Training and Testing Sets

Our dataset is collected using 2 sensors, a gyroscope, and an accelerometer. The

records of a gyroscope range between -2000 and 2000, while the records of an accelerometer range between -4 and 4. So we have to normalize the data by scaling it between 0 and 1 using the scikit-learn function MinMaxScaler() to make it easier for the model to learn.

4.2 Artificial Neural Network Structure

Neural Network is a simulation of the human brain's behavior, it allows the machine to solve sophisticated problems and recognize patterns, even the human brain might fail to recognize. Neural Network comprises one input layer, one or more hidden layers, and an output layer (Figure 4.2). Each layer consists of a set of neurons. The number of neurons in the input layer depends on the number of features in our dataset, while the output layer depends on the number of labels we have, and the hidden layers depend on the type or complexity of the data since these layers perform nonlinear transformations to the inputs entered to the network.

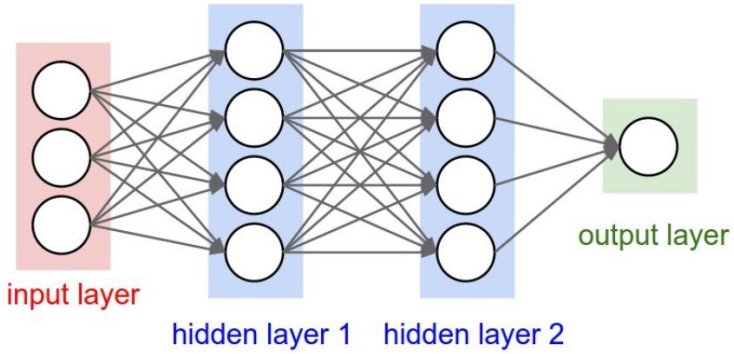


Figure 4.2: Layers of a Neural Network

In order to build our neural network, we have to specify the number of layers and neurons. For the input layer, we will use 60 neurons since we have 60 features for each input data. For the hidden layers, we will start with 1 hidden layer consisting of 30 neurons (30 is an arbitrary number, we can adjust it after multiple experiments) and the output layer consists of 3 neurons since we have 3 labels in our case.

Before we dive deeper into building the model, we have to understand why the hidden layers are important in training a model. If we look closer into a single neuron in the Input Layer, we can see that it holds a certain number that represents a specific coordinate of the 2 sensors (e.g a neuron that carries the value of coordinate aX[6]), this number is called activation. The activations in the input layer determine the activations of the next layer, the same for the activations of the second layer determine the activations of the third layer, and so on. In addition to the activation, each neuron has its own weight, which determines the strength of the connection between itself and the neuron it is picking up on. Using activations and weights for neurons will have very limited movement to look up for solutions so we need a bias to make our model more flexible.

$$\widehat{Y} = w_1 \cdot a_1 + w_2 \cdot a_2 + w_3 \cdot a_3 + \cdots + w_n \cdot a_n + b \tag{4.1}$$

- $\widehat{Y}$: predicted value (label)

- w : weight
- b : bias

Here comes the importance of hidden layers, since they apply an activation function to the model and break its linearity to avoid underfitting, which means that your model makes accurate, but initially incorrect predictions.

4.2.1 Activation Function

An activation function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations. There are multiple activation functions used.

- Tanh Function is a zero-centered function having a range between -1 to 1, however, the tanh function has a limitation , it cannot solve the vanishing gradient problem.

$$\sigma(Z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{4.2}$$

- Softmax Function is mainly used in multi-class models where it returns probabilities of each class, with the target class having the highest probability.

$$\sigma(\overrightarrow{Z}) = \frac{e^{zi}}{\sum_{j=1}^{K} e^{zj}} \tag{4.3}$$

- Rectified Linear Unit (ReLU) Function is a fast-learning activation function, compared to other activation functions like the sigmoid and tanh functions, the ReLU function offers much better performance and generalization in deep learning.

$$R(z) = max(0, z) \tag{4.4}$$

- Sigmoid Function is a non-linear AF used primarily in feedforward neural networks. It appears in the output layer of the deep learning models and is used for predicting probability-based outputs.

$$\sigma(Z) = \frac{1}{1 + e^{-z}} \tag{4.5}$$

To make sure we are using the right activation function for each layer, we decided to compare ReLU's function performance with softmax and sigmoid functions as input and hidden layers (Table 4.1), and compare the same activation functions' performance but on the output layer (Table 4.2).

In order to compare different hyperparameters we will use an open source platform called mlflow [10] for managing the end-to-end machine learning life-cycle. It allows us to track our experiments to record and compare parameters and results.

It is obvious that the Sigmoid Function (Figure 4.3 (a)) gave the best performance for the output layer, and the ReLU Function (Figure 4.3 (b)) for the input and hidden layers, So we will apply them to our neural network.

4.2.2 Dropout Layers

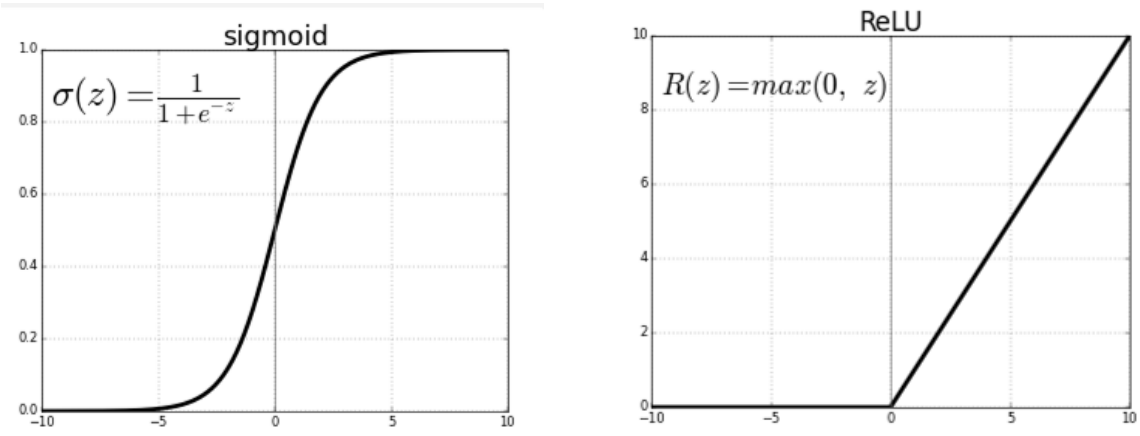
Neural Networks are likely to overfit a training dataset when trying too hard to learn different features, especially in our case we have 60 different features. They sometimes

Activation Function		Sigmoid	Softmax	ReLU
Metrics	Training accuracy	0.788	0.711	0.988
	Training loss	0.164	0.036	0.006

Table 4.1: Comparison between different AF's for the input and hidden layers

Activation Function		Sigmoid	Softmax	ReLU
Metrics	Training accuracy	0.988	0.73	0.696
	Training loss	0.006	0.161	0.355

Table 4.2: Comparison between different AF's for the output layer



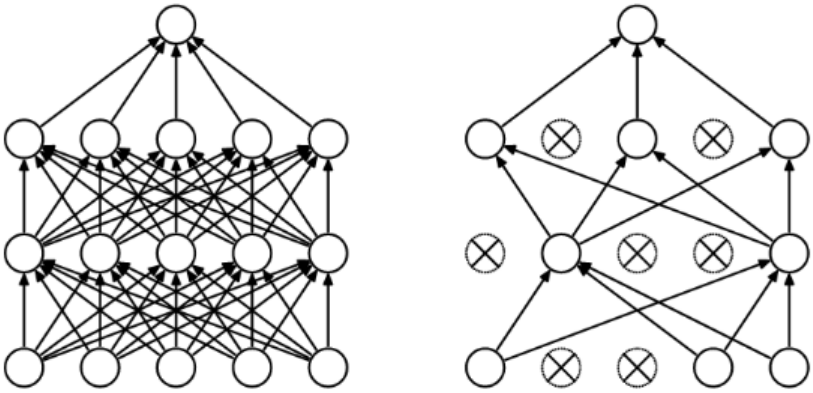
(a) Sigmoid Function

(b) ReLU Function

Figure 4.3: Activation functions used to train the model

even learn noise in the dataset which leads to overfitting. To avoid this issue, we modify the structure of the neural network after every iteration, so our model will not memorize the data or noise. Actually it does not seem a practical idea, since we need to construct multiple models with different structures. One simple way is to use Dropout Layers.

Dropouts work by dropping nodes from the input and hidden layers. So all the forward and backward connections with a dropped node are temporarily removed. which creates a new neural network structure. The nodes are dropped by a specified dropout probability (Figure 4.4).



(a) Standard Neural Net (b) After applying dropout.

Figure 4.4: Structure of a Neural Net after applying dropouts

In our case, we used dropout layers and set the dropout probability to 0.1, in both input and hidden layers.

4.2.3 Loss Function

The loss Function is a method to evaluate how well our algorithm is modeling our dataset. The lower its value the better our model is performing. It measures how far a predicted value is from its true value.

There are several types of loss functions, every function works with specific types of models. It depends if we are dealing with a classification model or regression model.

For Regression models, we have 2 important loss functions:

- Mean Squared Error Function
- Mean Absolute Error Function

For Classification models, we have 2 important loss functions:

- Binary Cross Entropy Function
- Categorical Cross Entropy Function

For the first impression, we might use one of the loss functions associated with the classification models. If we look closer at our model, it falls into the category of multi-label classification, but not necessarily in the same way as classification models work. It is not a multi-class classification since we don't expect the output of a shot to be only a forehand or only a slice. Our shots are a combination of 3 labels (For example, forehand topspin, backhand slice, forehand bad form, etc..).

The model seems confusing a little bit. To understand it furthermore, we will train our model using all 4 loss functions and try to come up with a conclusion (Table 4.3).

Loss Function		MSE	MAE	Binary Cross Entropy	Categorical Cross Entropy
Metrics	Training accuracy	0.988	0.73	0.696	0.5670
	Training loss	0.006	0.161	0.355	746598.937

Table 4.3: Comparison between different Loss Functions' performance

The results shown in the table are surprising. The regression loss functions did a great job, unlike the classification loss functions. The only conclusion we can come up with is that our model's structure looks like a classification model, while technically it appears to follow regression rules. In this case, we will choose MSE or MAE loss functions since they both gave similar results concerning the training loss and accuracy.

4.2.4 Optimizers and Other Hyperparameters

We can't talk about machine learning and neural networks without covering the concept of Gradient descent. Gradient descent is an optimization algorithm that is commonly used to train machine learning models and neural networks.

In order to find the local maximum of a certain function, we use the positive gradient of this function, and whenever we move towards a negative gradient or away from the gradient of the function at the current point, we will get the local minimum of that function (Figure 4.5). The Gradient descent is simply the negative gradient of a function. The main objective of it is to minimize the cost function using iterations.

Gradient descent multiplies the gradient by a scalar called Learning Rate, which is the step size at each iteration while moving towards the minimum of a loss function. Choosing the learning rate has a huge impact on training the model. A low learning rate may not reach the minimum or take a long time to reach it, while a high learning rate leads to overshooting the minimum (Figure 4.6). So we have to adjust it in a way it reaches the minimum of the loss function without taking forever or even overshooting this minimum.

The downside of gradient descent is that it is slow on huge data, and to fix this issue, Stochastic gradient descent is used in such cases. Stochastic gradient descent or SGD is a type of gradient descent that runs one training example per iteration. It updates each training example's parameters one at a time. We can divide our dataset into small batches and then perform the updates on those batches separately, this is known as MiniBatch Gradient Descent which is a combination of gradient descent and stochastic gradient descent.

Now we covered all the concepts needed to understand the optimizers and how they work. In our model, we will use the "Adam" optimizer which is an extension of stochastic gradient descent used to update network weights iteratively based on training data.

The default learning rate in Adam's algorithm is set to 0.001, which works well in our model, in addition to other parameters we kept them as their default values.

The last hyperparameter we will cover is the number of epochs and batch size used to train the model. First of all, an epoch is the number of times the learning algorithm will work through the entire training dataset. While batch size is the number of training examples utilized in one iteration. We used a batch size of 64 examples and the number of epochs equals 1000.

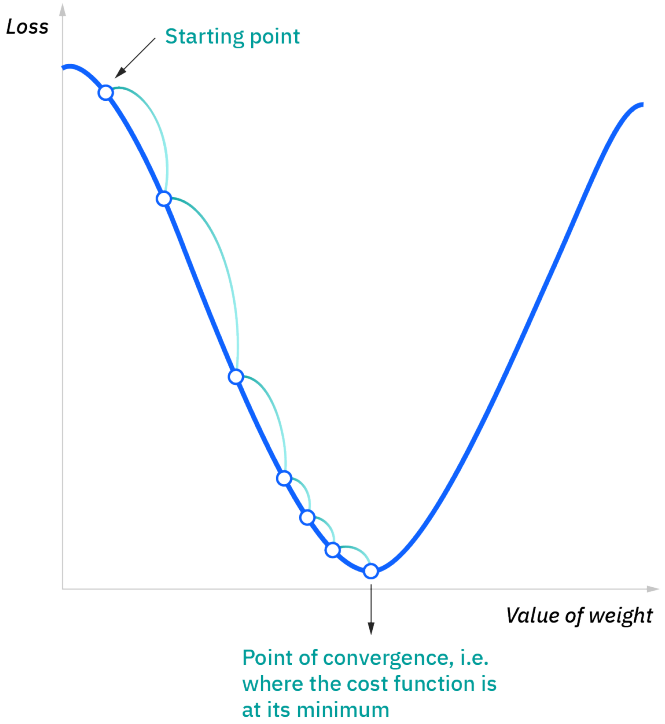


Figure 4.5: Method of Gradient descent of a loss function

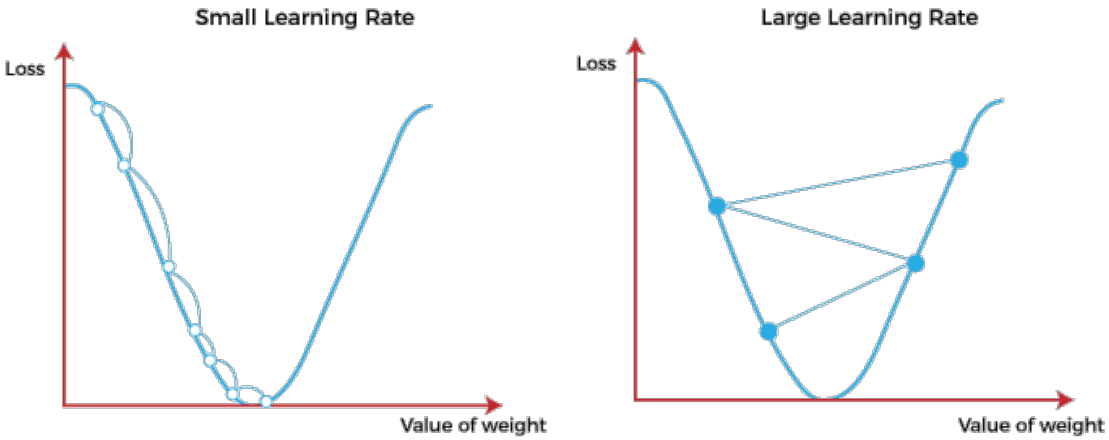


Figure 4.6: Learning Rate of a loss function

4.3 Training the Neural Network

A Machine Learning model's performance depends on the hyperparameters chosen. In our case, we tried to specify the hyperparameters that give the best value out of our model. Training the model is affected by the size of the dataset too, some datasets range from 100 megabytes to gigabytes of data, which requires careful selection of hyperparameters, especially the number of epochs. If we don't want to confuse ourselves with the optimal number of epochs, we can use the early stopping function supported by the Keras library which is a high-level neural network library that runs on top of TensorFlow [11] which we used in building our neural network as well.

We mentioned before that our dataset is considered too small compared with other datasets (less than 1 megabyte) so basically the training time takes less than 1 minute in the average case, so choosing a high number of epochs like in our case 1000 epochs doesn't make any problem concerning the training time. What matters about increasing the number of epochs is falling into the trap of overfitting, but we have to feel comfortable doing that since we optimized every hyperparameter to avoid overfitting from the neural network overall structure, dropout layers, loss function, and optimizer used.

To find out if using early stopping gives better results than training the network using 1000 epochs. We used mlflow to track our metrics as shown in the following table (Table 4.4):

		Early Stopping	Without Early Stopping
Metrics	Training accuracy	0.965	0.993
	Training loss	0.016	0.005

Table 4.4: Comparison between training with and without early stopping

If we look at the results we notice that there isn't much difference between training with early stopping or without it. But early stopping still knocks out around 3% of our training accuracy, same for the training loss where training with 1000 epochs gave a value 3 times lower than using early stopping. So we will stick to not using it to get the best value possible from training the model (Figure 4.7).

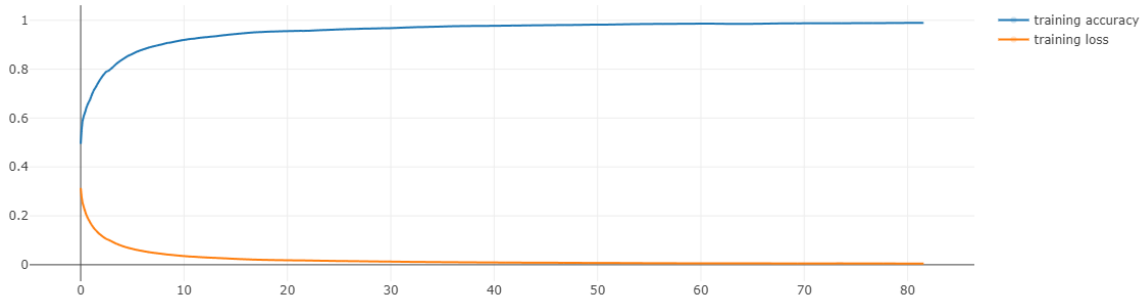


Figure 4.7: Training accuracy vs Training loss of the final model

4.4 Testing the Model

Before testing our model, a small interface that lets us visualize our predictions would be cool. To do that we have to make a small webpage or website. Luckily, there is a python library that does the work for us which is Gradio [12]. Gradio is the fastest way to demo our machine learning model with a friendly web interface. To test the model, we have to input the shots we want to detect their type, these shots are recorded and stored in a CSV file. The Gradio interface has a textbox that accepts the name of the file and passes it to the model to make predictions. The output is shown in 2 ways, a bar plot that plots the number of each shot detected. and a List that contains the types of shots recorded sequentially.

To test if our accuracy of 99% is correct, I played 100 shots as follows:

- 20 forehand topspin shots
- 20 backhand topspin shots
- 20 forehand slice shots
- 20 backhand slice shots
- 20 bad form shots

I tried to cover all motions of each shot to check if the model will detect them. The results were as follows (Figure 4.8):

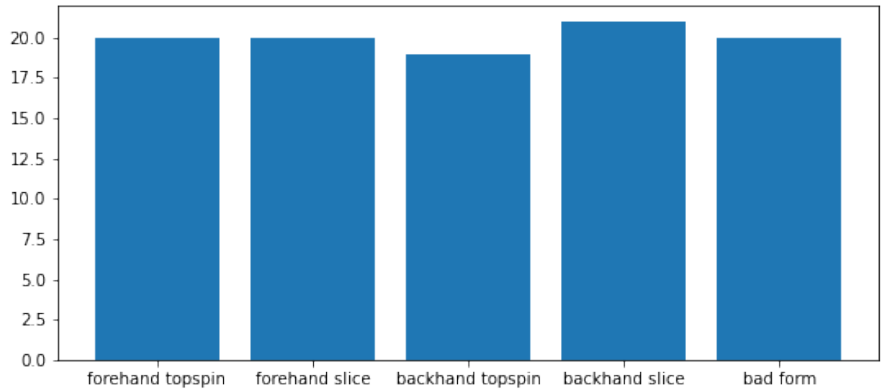


Figure 4.8: 100 types of shots detected by the model

As we can see, the model's predictions were:

- 20 forehand topspin shots
- 20 forehand slice shots
- 19 backhand topspin shots
- 21 backhand slice shots
- 20 bad form shots

The model misclassified only one shot which is a backhand topspin, it classified it as a backhand slice. This results in detecting 99 correct shots and 1 incorrect shot, which means our model had 99% detection of the types of shots.

Chapter 5

Conclusions and Perspective

This project can be considered a first step into the artificial intelligence world in table tennis. This model allows us to record our gameplay in a match or a training session, and have brief statistics so we can analyze and improve our performance.

Based on this model, we can build a model that detects the quality of the shots. we can bring in some top-tier players in our country and record their shots, then bring in players ranked between 20 and 50 and record their shots, same process for players ranked above 100, then players who train and play the game very well but not at a professional level, and finally record shots for beginner players, and train our model on this dataset. This model can detect the level I am playing at and recommends a training program associated with my skill level to improve and rank up to the next level.

Many more ideas can be implemented using Machine Learning in the field of table tennis or even sports in general. And that's what is happening, as we can see in every sport we found applications of AI where it improves the game overall and delivers the best experience for the players and the audience at the same time.

5.1 Perspective and Future Work

The model we created detects the types of shots a player plays, but not any player. Since I am a left-handed player, the dataset I collected is associated with left-handed players only. We can let right-handed players collect another dataset associated with right-handed players, train the same model and prompt the user to choose which hand he plays with.

Another possible improvement is to find a board that is even smaller than the one I used where it supports the gyroscope and accelerometer sensors in addition to the BLE module, 3d print a case that fits it with a small Lipo battery, and stick it at the bottom of the racket to avoid any unnecessary contact with the player's hand while playing.

The Arduino Nano 33 BLE Sense board supports Machine Learning capabilities, where we can convert our TensorFlow model into TensorFlowLite and deploy it to the Arduino core. TensorFlowLite is a lite version of the TensorFlow library that works on mobiles, edge devices, and microcontrollers with Machine Learning capabilities like the one we used in our project.

Bibliography

[1] https://www.instructables.com/Table-Tennis-Bat-With-Machine-Learning-AI-Arduino-/.

[2] https://towardsdatascience.com/run-or-walk-detecting-user-activity-with-machine-learning-and-core-ml-part-1-9658c0dcdd90.

[3] https://www.instructables.com/Table-Tennis-Bat-With-Machine-Learning-AI-Arduino-/.

[4] https://github.com/googlecreativelab/tiny-motion-trainer.

[5] https://towardsdatascience.com/run-or-walk-detecting-user-activity-with-machine-learning-and-core-ml-part-1-9658c0dcdd90.

[6] https://docs.arduino.cc/hardware/nano-33-ble-sense.

[7] https://pandas.pydata.org/docs/index.html.

[8] https://matplotlib.org/stable/index.html.

[9] https://scikit-learn.org/stable/.

[10] https://www.mlflow.org/docs/latest/index.html.

[11] https://keras.io/api/.

[12] https://gradio.app/docs/.