Report Machine Learning
AY 2023/24
Table Tennis Contest

Team 19

1st July 2024

Università degli Studi di Salerno

Name	**Surname**	**Serial**	**E-mail**
Antonio	Sessa	0622702305	a.sessa108@studenti.unisa.it
Angelo	Molinario	0622702311	a.molinario3@studenti.unisa.it
Massimiliano	Ranauro	0622702373	m.ranauro2@studenti.unisa.it
Pietro	Martano	0622702402	p.martano@studenti.unisa.it

Contents

1 Approach to the problem

The main idea was to split the problem into three different sub-problems:

- Inverse Kinematics
- Positioning
- Responding

1.1 Inverse kinematics

The problem involves the ability to move a paddle attached to a robotic arm. Moving the arm was not straightforward. To accomplish the task effectively, we needed to understand how to set the pitch and rotation of 3 joints and a slider, so that the paddle could reach a specific coordinate. Formally the problem that we intend to solve is the following: Given a desired end position (x, y, z) for the paddle centre position, find the joint pitch angles $\theta_1, \theta_2, \theta_3$ and the slider position s_1, s_2 that position the centre of the paddle at the desired coordinate

1.1.1 Inverse Kinematics Model

In order to develop an accurate inverse kinematics model that takes a desired x-y-z coordinate as input and outputs the rotation and pitch of the joints, as well as the movement of the slider, we have decided to use a Multi-Layer Perceptron network with supervised learning.
We decided to limit ourselves to only move the pitches of the three joints, ignoring their rotations.
Supervised learning requires a dataset, so the first task to accomplish was to create a dataset of joints pitches and slider movements and the centre coordinate of the paddle. The joint pitches and slider movement will be used as the target for the network, the x-y-z coordinate will be used as the input for the network.
To obtain the dataset, we picked eleven different stances that we

deemed could be useful in a game, and scanned the whole table with each stances. These produced a dataset of 116,000 samples. We ignored the paddle unit vector, as each stance was designed with an appropriate versor for the specific shot it was intended to face.
We trained the model on this dataset, leaving 10% out for validation.

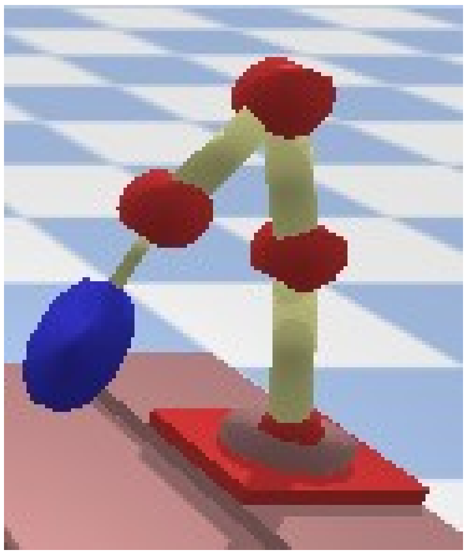


Figure 1: One of the stances used to create the dataset

1.1.2 Model architecture, training and validation

The model architecture is a feed forward neural network with one hidden layer with 256 neurons, the activation function used is the Relu, and the output uses the linear activation function, the optimizer used was the Adam Optimizer, with a learning rate of 0.001 and the MSE as a loss function. The loss after 100 epoch of training is nearly 0%, on both training set and validation, but the true validation of the model's performance was its deployment in the PyBullet environment. The model was tested to ensure that it could accurately move the paddle to any reasonable position of the table (for example we have not tested the ability of the model to reach position that are not physically reachable, or position that would never be likely useful during a game), to test this, considering an inverse kinematics problem as more than one solution, and we are not interested on how the joints are positioned but only on how accurately it can reach a certain point, we sampled 500 x-y-z coordinate from the dataset to use as desired positions to reach, added some light noise (a value uniformly sampled between -0.03 and 0.03), and checked the distance from the desired position and the actual position reached by the robot. We achieved a mean distance of 0.0198, that we retain acceptable. Also the model would still be validated in the following steps, as positioning and responding require the use of the inverse kinematics model.

1.2 Positioning

With the inverse kinematics model we have now the ability to move the paddle centre to any "reasonable point" of the table. Now we have to decide when to move it and where to move it. We had two possibility, try to hit the ball before it hit our side of the table (as the rule allows it) or hit the ball after it bounces on our side of the table, just as a normal match of table tennis.
We went for the second approach as a deeper ball allows us to hit more interesting shot. In order to hit the ball we need to predict where the ball will be so we can position the paddle. The environment gives us the ball positions and speed, so we can predict the ball trajectory. Our approach is the following: while the ball has not hit the table shadow its movement on the x-axis, when the ball has hit the table, use the ball state to predict where it will be when its height will be 0.2, if the predicted y is too far back chose an increased height for the prediction. When the prediction has been made, pass the predicted point to the inverse kinematic model and receive the joint and slider values. Send these values to the environment. If the trajectory of the ball is going out of the table, avoid the ball predicted position on the x-axis. This process will be repeated many times until the ball is within range to be struck by the paddle.

The reason to repeat the process is that the trajectory prediction does not take in consideration ball acceleration. There are three reasons for the avoidance of the acceleration. The first reason is to make easier the prediction of the ball position; more specifically use the acceleration meaning either calculate it externally through two consecutive states or pass the two consecutive states to the function that calculates the prediction. The second reason is to make the prediction faster in order to take the shorter possible time. The third reason is, since the prediction is made more times until the distance between ball and paddle is within a certain range, the error made by the avoidance of the acceleration decreases more and more. With this approach we could reliably position the paddle in the right spot to be able to hit the ball.

Algorithm 1 Positioning Algorithm

```
 1: b_pos, b_vel, b_touched_table ← read_environment()
 2: x, y, z ← predict_ball_pos(b_pos, b_vel, height=0)
 3: if b_going_out(x, y) then
 4:     joints ← avoid_stance(x)
 5: else
 6:     if b_touched_table then
 7:         height ← 0.2
 8:         x, y, z ← predict_ball_pos(b_pos, b_vel, height)
 9:         while too_deep(height,y) do
10:             height ← height + 0.1
11:             x, y, z ← predict_ball_pos(b_pos, b_vel, height)
12:         end while
13:         joints ← inverse_kinematics_model(x, y, z)
14:     else
15:         x ← b_pos.x
16:         joints ← default_stance(x)
17:     end if
18: end if
19: send_joints(joints)
```

1.2.1 Trajectory functions

The projectile motion model is used to calculate the trajectory. Even with respect to the further axis the motion is considered uniform rectilinear. What we decided to do is to calculate the time needed for the object to be at a certain height, and we used the time obtained to calculate the values of x and y. If the value under the square root is negative then the object can never be at that height

$$\begin{cases} x(t) = vel_x \cdot t + x_0 \\ y(t) = vel_y \cdot t + y_0 \\ z(t) = -\frac{1}{2} \cdot g \cdot t^2 + vel_z \cdot t + z_0 \end{cases} \tag{1}$$

$$z(t) = height \rightarrow time = \frac{-vel_z \pm \sqrt{vel_z^2 - 4(-\frac{1}{2} \cdot g)(z_0 - height)}}{2 \cdot (-\frac{1}{2} \cdot g)} \tag{2}$$

$$(x(time), y(time), height) \rightarrow (pred_x, pred_y, pred_z) \tag{3}$$

1.3 Responding

Now that we can position the paddle, we need to decide how to respond to a ball. To tackle this task, many reinforcement learning approaches were tried, but they all shared the idea that the action to take would be the adjustment of the paddle's pitch (we briefly explored paddle rotation also, but decided not to, to speed up training) and this adjustment should be made when the ball is within a range of 0.3 units from the centre of the paddle and should be a single, clean strike, performed in one movement of the joints. Also in the replay buffer only meaningful transitions should be added: add transitions in the replay buffer only when the action of the robotic arm had an effect on the ball trajectories. What changed between different approaches were the reward function, the inputs to the model its architecture and the learning algorithm.

1.3.1 Q-learning with discrete action

The first approach that had some success was the use of a deep Q-learning algorithm to train the model to pick between a limited set of pitch adjustments, ranging from 0.0 to 2.2 with a step of 0.2. The input to the model were the ball positions, and the ball velocity. The reward was equal to 1, if the robot managed to hit the ball so that it reached the other side of the table, and -1 if it was not able. During training the other robot was programmed to avoid the ball so that it would never respond, so every transaction was final, meaning there was no need for a q-target network to calculate the loss. The network architecture is the following:

- Input layer: 6 inputs, the ball velocity and position.
- Hidden layers: Three Fully connected layer with 256 neurons, followed by Layer Normalization and ReLU activation.
- Output layer: Tanh activation function, each output represent the expected reward for the eight possible actions.

The loss function used was the SmoothL1Loss.
For exploration we used an epsilon greedy policy, with an exponential decay, so it would start with an high value of epsilon of 0.9 and decrease exponentially until an epsilon value of 0.05. The epsilon decay chosen was 1000. The size of the replay buffer was 1000.

```
% eps_threshold formula from PyTorch DQN documentation
eps_threshold = EPS_END + (EPS_START - EPS_END) *
        math.exp(-1. * steps_done / EPS_DECAY)
```

The training was done in the sameserve mode, so that our robot would first have to learn to respond to the service done by the server. The other player was a dummy client that would avoid to hit the ball. In Figure 2 we show the result of our training. The best performances were achieved around 25000 and 30000 episodes and after that there was a steep drop in performances, from which it seems to start to recover, but we preferred to interrupt the training. The steep drop may have been caused by the relatively small size of the replay buffer. Having saved the model when it achieved it's best performance, we could have tried to restart the train using a larger buffer, but we preferred to try different approaches, as a dqn approach would always be limited on the possible action it could take. Also this first model was useful to validate our inverse kinematic model, as we noted that our first model many times positioned the paddle with the versor slightly towards down, and this made hitting a good shot rather difficult. So we augmented the inverse kinematic dataset with more samples (97.000) where the paddle is more inclined upwards and trained a new inverse kinematics model on it. Briefly it was explored the idea to create a dataset of 'good shoots', from which to train a supervised learning model to learn to respond, that would not be limited by discrete actions.

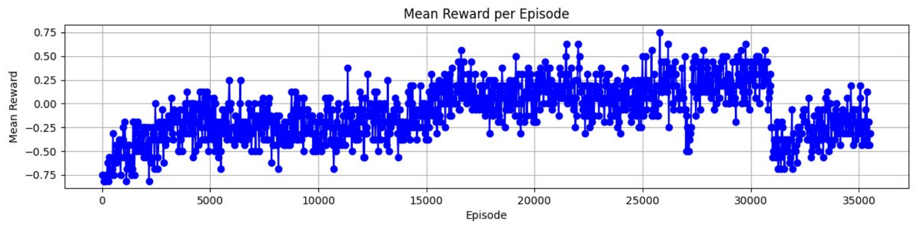


Figure 2: Mean reward every 32 episodes. A mean reward of 0.75 means that about 87% of ball went in, while a reward of -0.75 means that about 12% of ball went in

1.3.2 DDPG

After observing that the objective pursued with the first approach had some results, we set ourselves the problem of understanding where and why the Q-Learning algorithm was failing. After carrying out some simulation, we realized that the main problem was that every time the network made a mistake, it was because the optimal pitch value to be given was between two of the discrete actions defined. For this reason, we switched to an approach that allowed us to have pitch values chosen from a continuous range using the deep deterministic policy gradient algorithm.

1.3.3 Architecture

The code used for the implementation of the ddpg algorithm is the one proposed during the class exercises, with some modifications made to the actor and critic networks used.
The two networks used by the ddpg have 5 values ad input:

- The angle between the paddle unit vector and the ball speed vector.
- The ball speed magnitude.
- The ball's y and z position.
- The Euclidean distance between the ball and the paddle's center.

The critic network has also an additional input value that is the action given for that state.
The two networks have a different architecture, the actor network has:

- Two hidden layers of 128 and 256 neurons with a Relu activation function.
- A normalization layer before each hidden layer.
- An output layer with a sigmoid activation function.

The critic network has:

- Three hidden layers of 128, 256 and 256 Neurons with a Relu activation function.
- A normalization layer before each hidden layer

While in the previous approach the maximum range for the pitch value was from 0 to 2.2 in this case we decided to give a pitch value between 0 and 1 because we notice that values greater than 1 cause the ball to be hit too hard. For the exploration we followed an approach based on Ornstein Uhlenbeck Noise which add noise to the network output with a sigma value of 0.01. The replay buffer used has a size of 8000 and the batch size used for training is 32.

1.3.4 Reward functions

An important issue in the Reinforcement learning is the definition of the reward function, we used different type of reward function during our tests but all of them have the same base concept, we should give a positive reward if the ball falls in the opponent field after out agent hit it, and penalize the agent if the ball doesn't hit the opponent half field.
The reward function described in Algorithm 2 gives a positive rewards based on the distance between a target depth and speed of the ball, and a negative reward if the ball ends up in our side or the field or the ball goes beyond the opponent's field.

Algorithm 2 Reward Function

```
1: function REWARD_F(y, speed, target=1.7)
2:     if $1.0 \leq y \leq 2.2$ then
3:         return $\frac{1}{|\text{y}-\text{target}|} + \text{speed} \times 2$
4:     else if $y > 2.2$ then
5:         return $(20.0 - y \times 10) \times 2$
6:     else if $y < 1.0$ then
7:         return $(y \times 10 - 10.0) \times 2$
8:     end if
9: end function
```

The reward function described in Algorithm 3 gives a different positive reward for each strip the ball hits in the opponent's field, a negative reward equal to the distance between the drop point and the net position if the ball goes beyond the opponent's field, and a negative reward if the ball falls in my half field, this reward is much more negative if the ball falls far from the net and less negative, close to 0, if the ball falls in my half field but is near the net. Using this last reward function we trained out robot for about 8000 episodes saving the model each time the mean reward for 32 episodes improved.

Algorithm 3 Reward Function

```
function GET_TRAJECTORY_REWARD(y_drop_point)
    if $y\_drop\_point < 1$ then
        $trajectory\_reward \leftarrow -(1 - y\_drop\_point^2)$
    else if $1 \leq y\_drop\_point \leq 1.4$ then
        $trajectory\_reward \leftarrow 3$
    else if $1.4 < y\_drop\_point \leq 1.8$ then
        $trajectory\_reward \leftarrow 5$
    else if $1.8 < y\_drop\_point \leq 2.18$ then
        $trajectory\_reward \leftarrow 4$
    else
        $trajectory\_reward \leftarrow -\|y\_drop\_point - 1\|$
    end if
    return $float(trajectory\_reward)$
end function
```

1.3.5 Training the service

Like before with DQN, we trained our robot first in the same-serve mode with a dummy player as its opponent so we can compare the different performance between the discrete approach and the continuous approach. During the training we monitored the mean reward each 32 episodes with the value of the policy loss and value loss after each training. To speed up the training process we decided to train the networks 32 times each 32 episodes,this approach may cause some problems during the first training because there are too few sample in the replay buffer so all the samples that we choose may be highly correlated but this problem is solved by repeat the training for a larger number of episodes.
After 8000 episodes we loaded the saved model with the best mean

reward in evaluation mode and we counted how many balls the robot manages to hit correctly in the service phase. To get a realistic value we observed about 800 episodes at the end of which our robot hit correctly the 92% of the ball, where the remaining 8% were hit by out robot but the ball fell in our half field.

The ddpg algorithm also make use of two target networks, one for the critic network and the other for the actor network, so the target value used in the critic loss function is:

$$target = r_{t+1} + \gamma \cdot Q'(s_{t+1}, \pi'(s_{t+1}))$$

however in the scenario where we train our robot with the dummy opponent we have that each state for which the agent chooses an action is always a final state therefore by definition

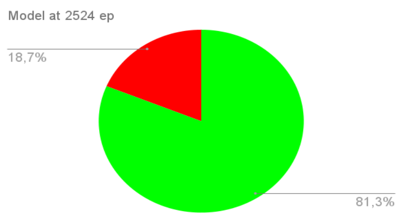
$$Q'(s_{t+1}, \pi'(s_{t+1}))$$

is equal to 0 because we do not have a next state after the final so even if there are two target network we don't really use them but since the replay buffer requires a next state in order to add a transition we set the next state like a tensor of all 0. Since the target networks are never used in this scenario we could have omitted them but we choose to maintain them so we can use the same code to train our agent in scenarios in which we don't have only final states, such as scenarios in which an exchange of blows occurs between the two players.

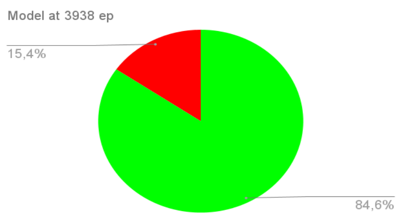
1.3.6 Training against itself

We have a model that can respond to the opponent's serve in an acceptable way, but it doesn't know how to respond to other kind of shots. So using the same code and the same architectures, we took the model trained to respond to the service as a baseline and we continued to train that model in the same-serve mode, but this time it is our robot that performs the service and the opponent is the robot trained to respond to the service loaded in test mode. In this

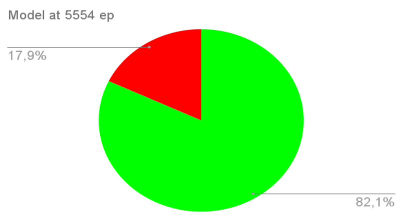
scenario our robot learns to respond to balls hit by the opponent with the aim to score a point.
In this case we could appreciate the contribution of the target networks because each state is not always final, this is because when we hit correctly an opponent's ball it isn't certain that with that action we score a point but our opponent could get the ball and resend it to us, so in a single episode we have more than one transition to put in the replay buffer. One problem that we faced at this time is given a state the agent choose an action and get a reward but what is the next state that we should consider? Our initial approach was that the agent should act only when the ball is 0.3 units away from the paddle and has already bounced on the table, so we limit the possible state that out agent can view for this reasons. We cannot choose any next state after hitting the ball because if we get a state where the ball is more that 0.3 units away from the paddle than that state isn't a valid state for out agent. To resolve this issue we consider as the next state the next time that the ball is in the same condition in which our robot can act that is 0.3 units away from the paddle and has already bounced on the table, condition that are achieved when our opponent hit the ball and our agent need to respond. Using this method a state is final when the corresponding action causes a change in the score.
For training the agent to play against an opponent, an additional reward has been defined in addition to the one defined by algorithm 3, this reward function described in the algorithm 4 has the purpose of assigning an additional reward to a final state of an episode, -10 if the final state score a point for the opponent, +10 otherwise. In the training, to make our model see different shots, every 2000 episodes we changed the opponent with the best model trained in the past 2000 episode. At the end of the training, another 8000 episodes, we took the four saved models that obtained the best performance in terms of average reward obtained every 32 episodes and we performed a validation of the models using auto as the opponent and the percentage of point scored over 200 episodes as the performance index, obtaining the graphs in the figure. So we picked as our final shooting model the one trained after 3938 episodes. 3.



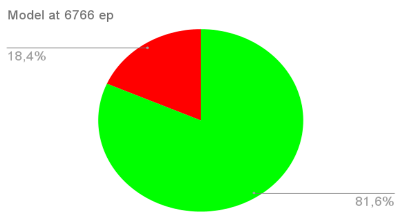
(a) 2524 episodes



(b) 3938 episodes



(c) 5554 episodes



(d) 6766 episodes

Figure 3: Validation of final models

Algorithm 4 Function GET_SCORE_REWARD(score)

```
1: function GET_SCORE_REWARD($score$)
2:     $score\_reward \leftarrow 0$
3:     if $score.old.my < score.new.my$ then
4:         $score\_reward \leftarrow 10$
5:     end if
6:     if $score.old.opponent < score.new.opponent$ then
7:         $score\_reward \leftarrow -10$
8:     end if
9:     return $score\_reward$
10: end function
```

1.4 Limits in our approach

The main limits in our approach comes from the positioning strategy that we have chosen and the trajectories functions. Having chosen to respond to a ball only after it bounced on the table and calculating only then the trajectory for the ball, makes for us impossible to respond to some kind of shots, in particular the very deep shots that hit the table near it's back edge. The two main problem that come from this shots are: not enough time to position the paddle, and the awkward position that are reached if reached, as the inverse kinematics model has only position where the paddle is held pointing down, and this shots requires a pointing up position. Also training our reinforcement learning model is a relatively long process as to train a model for 8000 episodes took nearly 11 hours (5 second for episode when there is not an exchange of blows) and when we trained on the laboratory environment and then loaded the model trained on our local environment the performances were not the same. The problem seems the discrepancy on the loop time, as on the laboratory environment it took always a time of around 0.02 second while on our local environment the loop time tended to spike when more calculations had to be done. As a metric for this, the standard deviation for the loop on the laboratory environment that we measured is 0.003 while on our local environment we measured 0.005477. The main effect of this is that the input the model sees on the the two environments are different, for example, on laboratory environment the ball distance is always measured between 0.24 and 0.3, while on our local environment this metric as a range from 0.1 to 0.3. This can be one of the explanation for the difference in performances.