Machine Learning Presentation
Table Tennis Contest
Group 19

Members:

Antonio Sessa 0622702305
Angelo Molinario 0622702311
Massimiliano Ranauro 0622702373
Pietro Martano 0622702402

Prof. Mario Vento
Prof. Pasquale Foggia
Prof. Diego Gragnaniello

The main tasks

Inverse kinematics

- Create an inverse kinematics dataset
- Train a model with supervised learning on the dataset
- Validate the model

Positioning

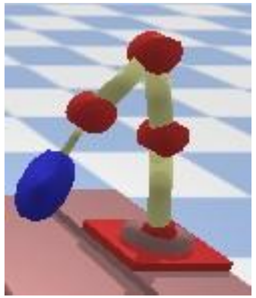
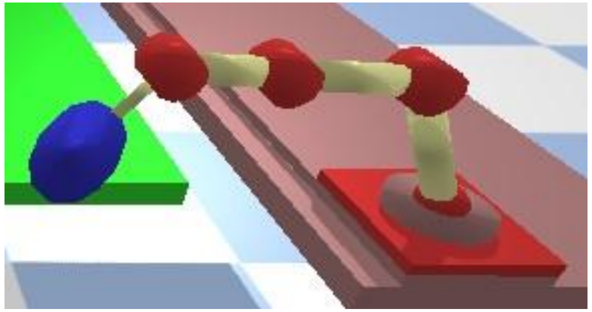
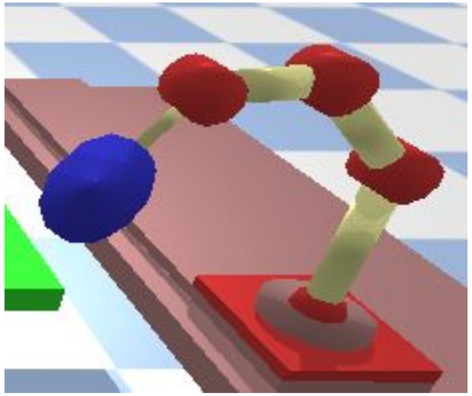
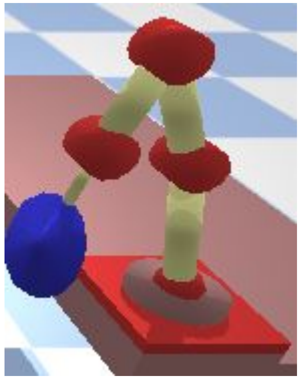
- Decide how and when the robot should move
- Write the trajectories functions

Responding

- Decide the algorithm, DQN or DDPG
- Decide the inputs and action space
- Reward functions
- Train the model

To create the dataset...

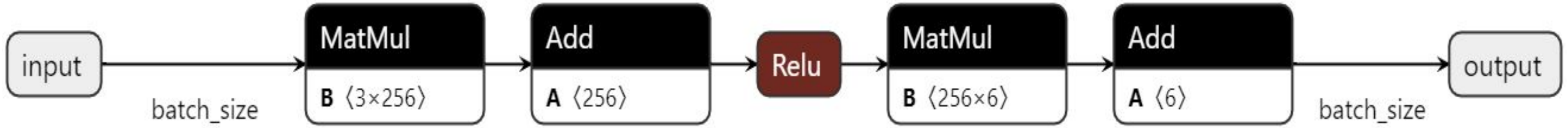
We scanned the table with some stances that we deemed could be useful in a game. We started with 11 stances and got 116.000 samples. Later we added another 97.000 samples.

The model architecture

input = x-y-z

output = joint pitches



Training and validation

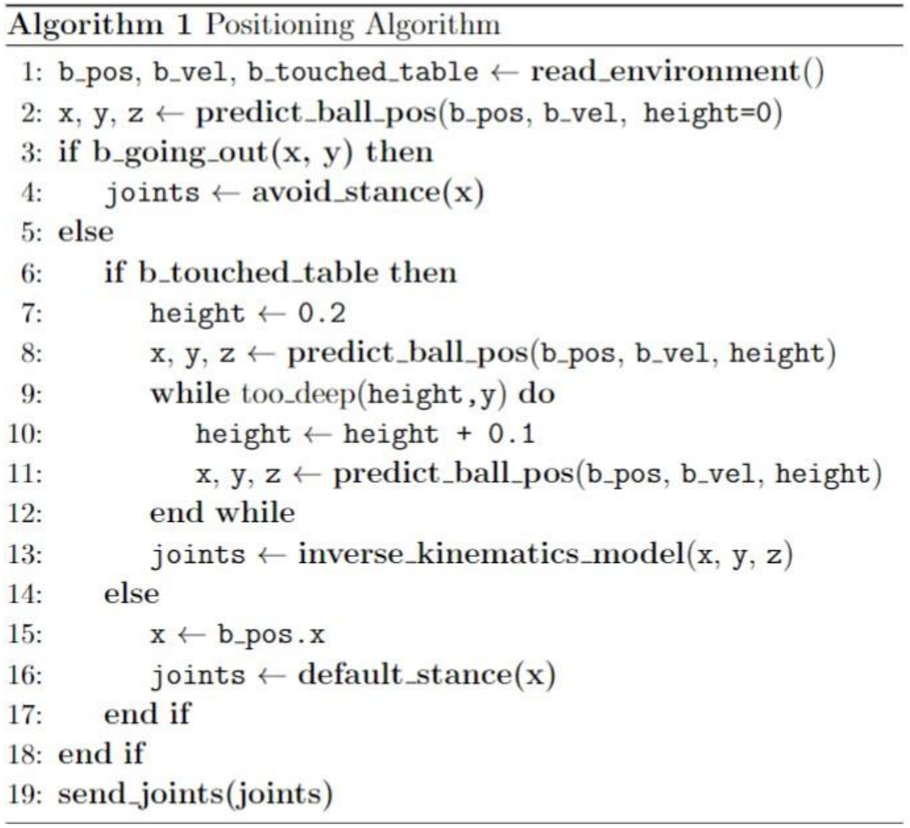
- 100 epochs, we reached a loss of nearly 0% on training and validation set
- Environment test: we added noise to the x-y-z coordinate and checked the distance between the desired x-y-z and the actual distant, getting a mean distance of 0.0198
- The model kept being validated during the following steps (bringing the augmentation of the dataset)

Positioning algorithm

We decided to play the ball after the bounce on our side of the table.

We try to hit the ball when its height is at 0.2 if it is reasonable.

While the ball has not hit our table, we shadow it’s movement on the x-axis or avoid it based on if it is going out or not.



Algorithm 1 Positioning Algorithm

```
b_pos, b_vel, b_touched_table ← read_environment()
x, y, z ← predict_ball_pos(b_pos, b_vel, height=0)
if b_going_out(x, y) then
    joints ← avoid_stance(x)
else
    if b_touched_table then
        height ← 0.2
        x, y, z ← predict_ball_pos(b_pos, b_vel, height)
        while too_deep(height,y) do
            height ← height + 0.1
            x, y, z ← predict_ball_pos(b_pos, b_vel, height)
        end while
        joints ← inverse_kinematics_model(x, y, z)
    else
        x ← b_pos.x
        joints ← default_stance(x)
    end if
end if
send_joints(joints)
```

The trajectory function

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

The projectile motion is used for the trajectory

The flight time for the desired height is calculated

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

The max time obtained is used to calculate x and y, if the value under root is not negative

$$(x(time), y(time), height) \rightarrow (pred_x, pred_y, pred_z)$$

Main ideas on how to learn to respond

- Use a reinforcement learning approach
- Try to hit ball when it’s within a range of 0.3, after it bounced on our side of the table
- Trying to hit the balls means to adjust the pitch of the paddle, we briefly explored the idea to also adjust the rotation with the reinforcement learning model, but to speed up training we have chosen to limit ourselves to pitches and hard code slight rotation to the paddle when responding.
- Meaningful transitions: add transitions in the replay buffer only when the action of the robotic arm had an effect on the ball trajectory.

First approach: DQN

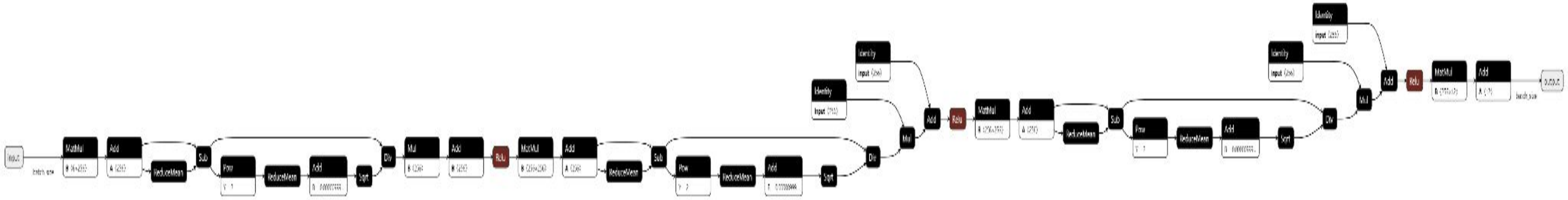
With these action space values = [0.0,0.2,0.4,...,2.2]

With this reward function = 1 if scored, -1 if not.

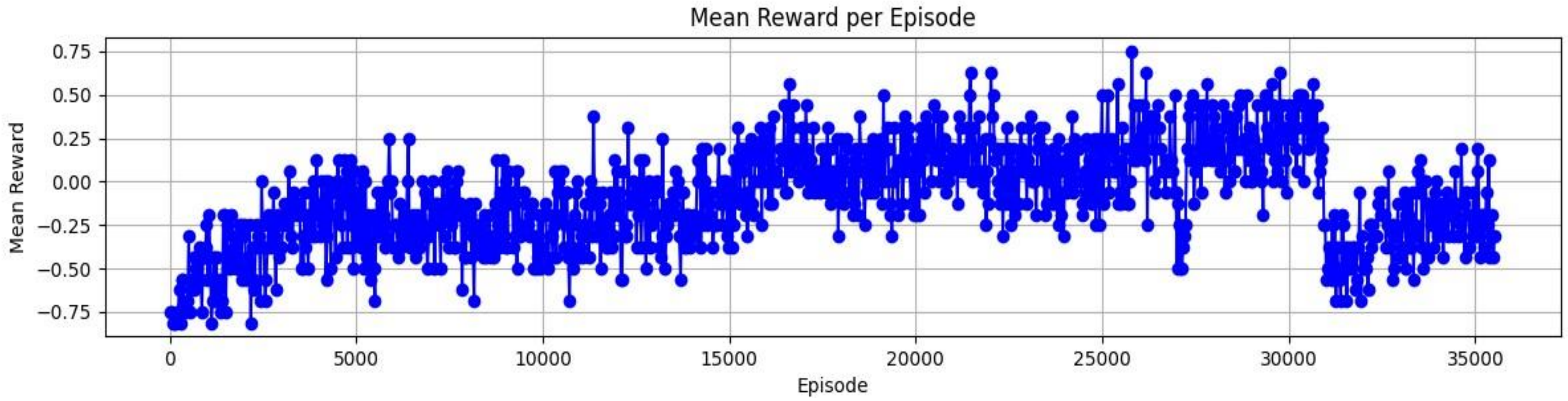
With this input values = ball position & velocity

With this architecture = 3 layer norm+relu, 256 neuron per layer

All transactions are final, so no need for a q-target network.



Result with DQN, sameserve



- 0.75 -> 87% success rate, -0.75 -> 12% success rate
- Possible cause for the drop: relatively small replay buffer (1000)

Is DQN good enough?

We have been using a discrete action space to handle a continuous task.

The paddle might hit the ball too weakly or too strongly, lacking the capability to determine the optimal shot that may fall in between two actions.

The discrete nature of the action space does not allow for fine-tuning and may result in less effective control over the ball.

There are infinite possible actions between 0.0 and 0.2, so we decided to try a different approach.

DDPG

In order to have a continuous action space we use the ddpg algorithm.

The Actor and Critic networks have two different architectures, the actor network has 2 hidden layer with 128 and 256 neurons while the critic network has 3 hidden layer with 128, 256 and 256 neurons. Both networks use normalization layers and a ReLU activation function for the hidden layers.

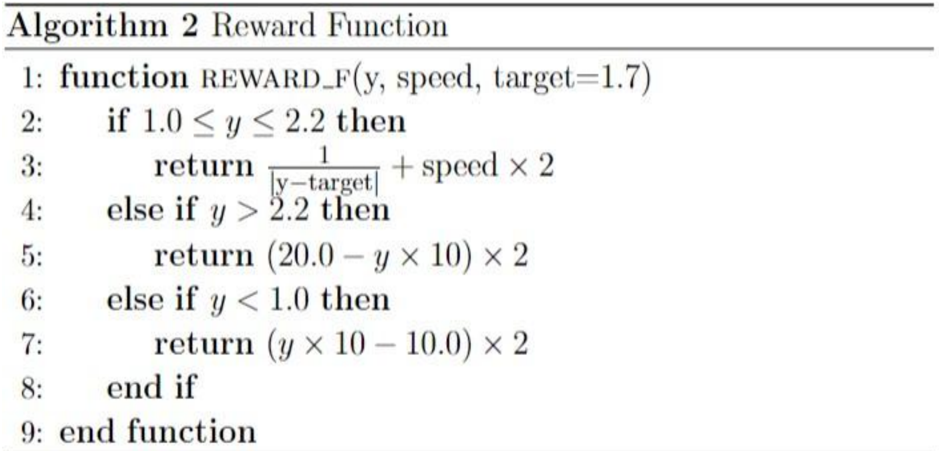
The Actor network gives us the action to perform, action which is between 0 and 1 due to the activation function of the output layer being a sigmoid.

Unlike discrete actions which had a maximum value of 2.2 after some simulation we observe that actions greater than 1 hit the ball to hard so we directly use the output of the sigmoid function.

Reward functions

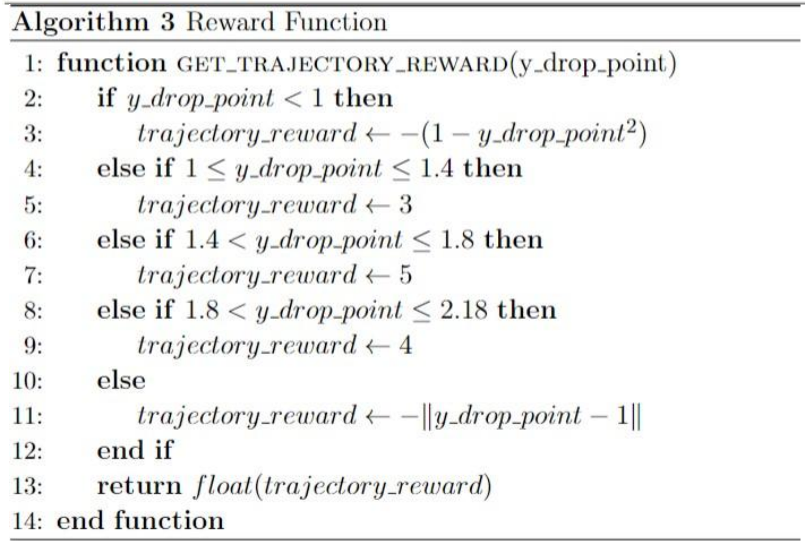
We tested different reward functions:

- The first had a target depth and gave reward based on the distance from it and speed of the ball also
- The second divided the table in different zones and gave reward based on those
- Both gave penalty based on how far the shot missed



Algorithm 2 Reward Function

```
function REWARD_F(y, speed, target=1.7)
    if $1.0 \le y \le 2.2$ then
        return $\frac{1}{|y-target|}$ + speed $\times$ 2
    else if $y > 2.2$ then
        return $(20.0 - y \times 10) \times 2$
    else if $y < 1.0$ then
        return $(y \times 10 - 10.0) \times 2$
    end if
end function
```

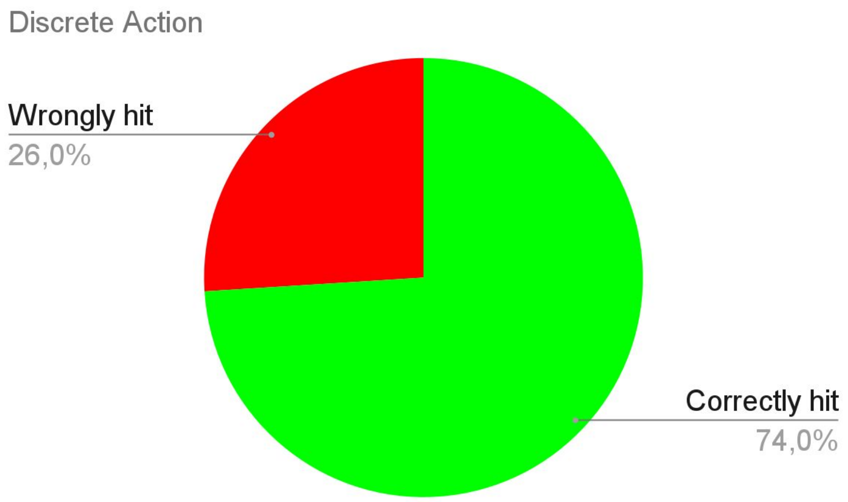


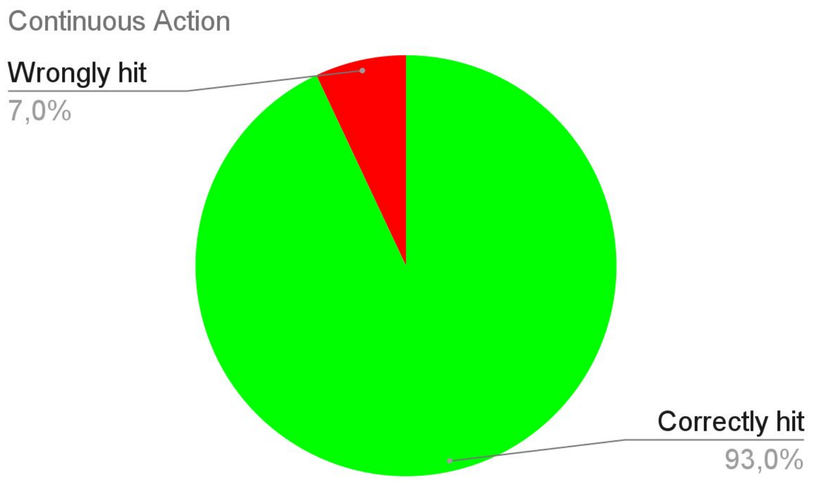
Algorithm 3 Reward Function

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

DDPG for sameserve mode

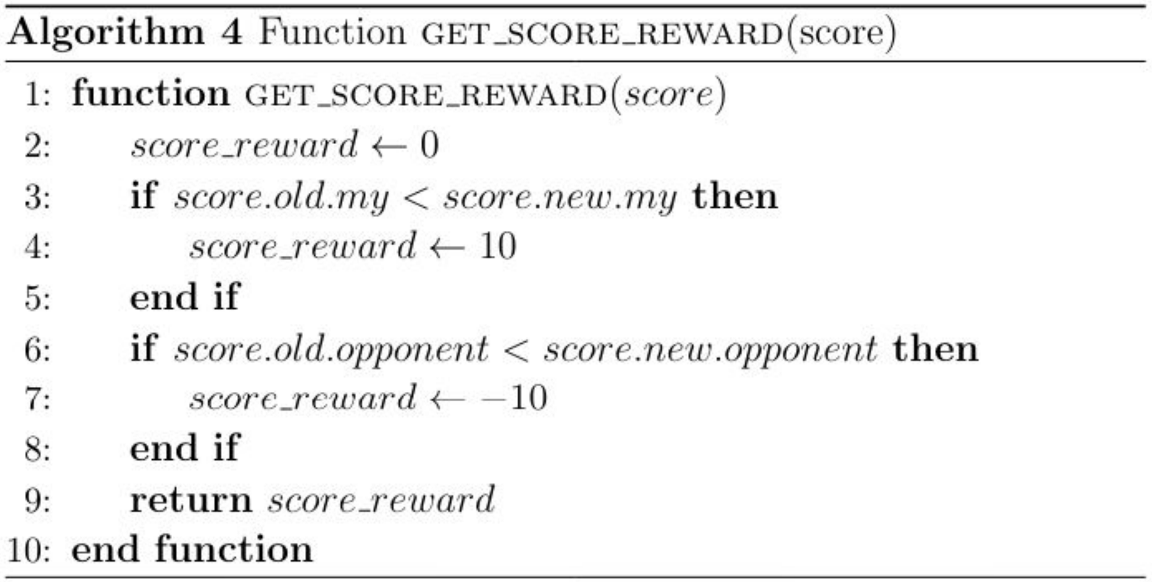
We initially trained our agent to respond to service only using a dummy player to see if there was an improvement in the performance wrt the dqn approach.





An additional reward

We have defined a further reward function which is responsible for assigning a positive reward to the agent when he scores a point and a negative one when he doesn’t.



Algorithm 4 Function GET_SCORE_REWARD(score)

```
function GET_SCORE_REWARD(score)
    $score\_reward \leftarrow 0$
    if $score.old.my < score.new.my$ then
        $score\_reward \leftarrow 10$
    end if
    if $score.old.opponent < score.new.opponent$ then
        $score\_reward \leftarrow -10$
    end if
    return $score\_reward$
end function
```

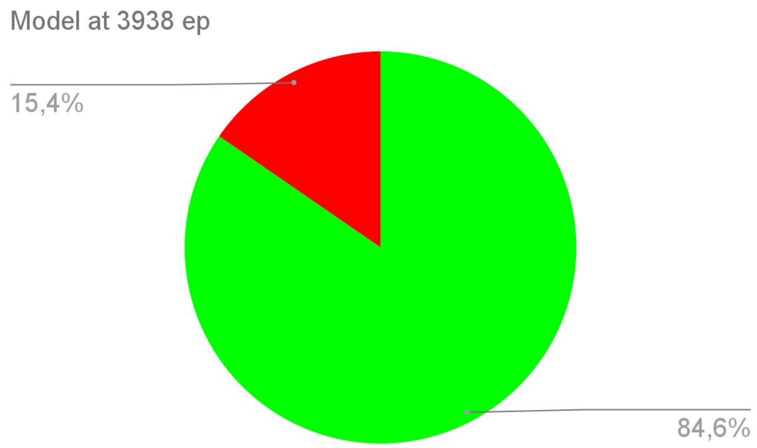
Next step ...

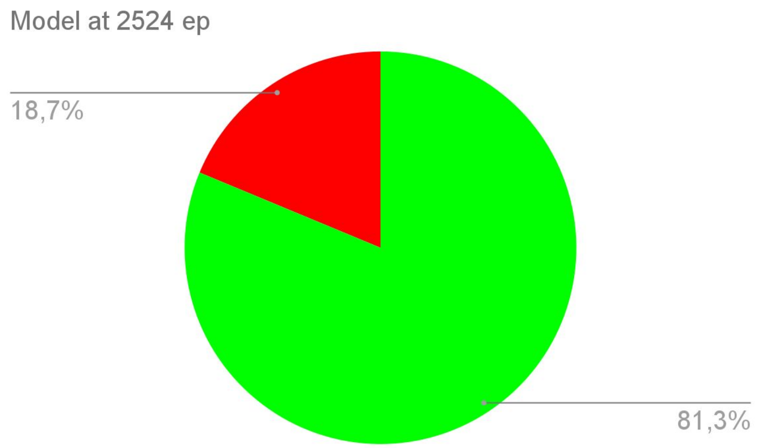
After training our agent to return the serve in an acceptable manner, we moved on to training the agent to respond not only to the serve but also to the opponent's shots. To do this we loaded the trained model in evaluation mode and use it as our opponent.

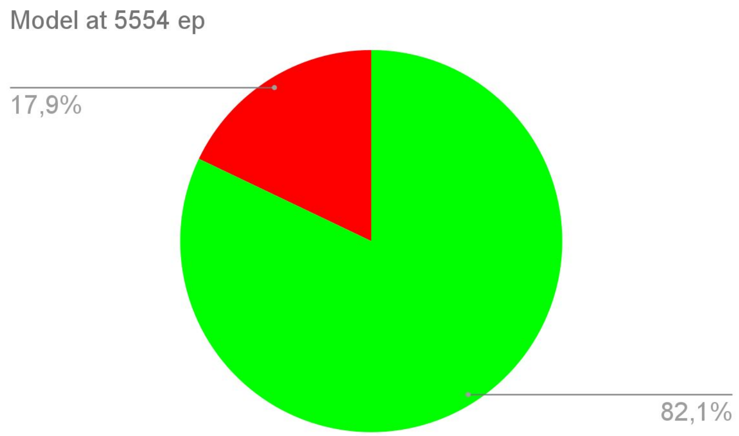
After a certain number of episodes we replaced our opponent with the last trained model that now knows how to return the serves and the opponent’s shots, so that the model in training can have increasingly longer and more varied exchanges of shots available.

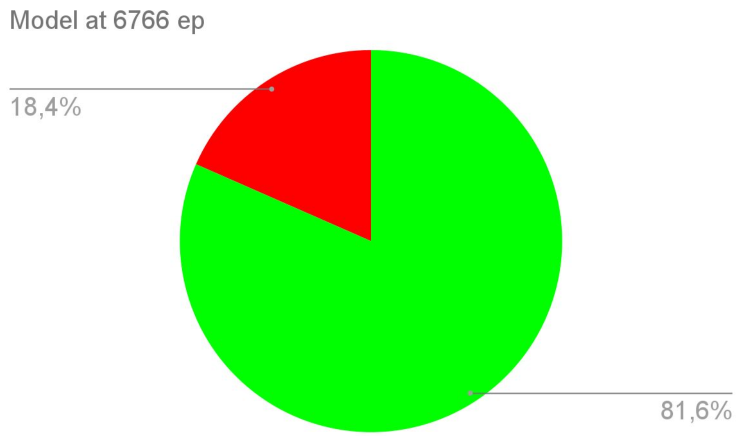
Final result

After training our agent for about 8000 episodes we took the models that obtained the greatest mean reward and tested each of them against auto and finally choose the final model based on the performance shown on the winning rate against auto.


Model at 3938 ep
15,4%
84,6%


Model at 2524 ep
18,7%
81,3%


Model at 5554 ep
17,9%
82,1%


Model at 6766 ep
18,4%
81,6%