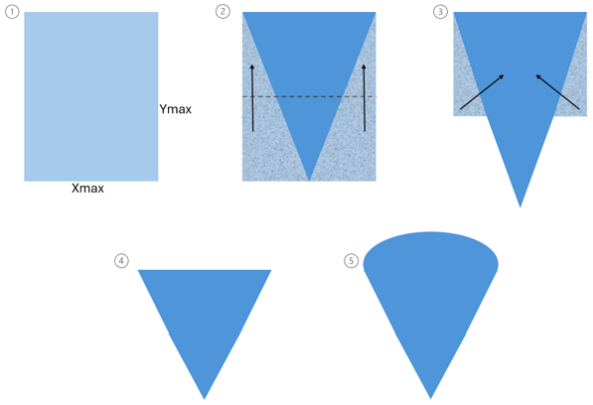
Reinforcement Learning to Catch a Ball

Chris LeBlanc
Northeastern University
leblanc.ch@northeastern.edu

Abstract

The goal of the project was to use reinforcement learning to teach a robotic agent to catch a ball in flight. Policy gradient methods were used to output the continuously valued actions of the agent, expressed as the bend angles of the individual actuators of the robotic arm. Experiments were done on using different means of exploration of the agent's environment, and on passing different features for the model to train on. The trajectory of the ball is considered to have drag due to air resistance, and the ball is considered "caught" whenever the euclidean distance between the ball and the end effector of the robot is small enough. The euclidean distance from the ball's position to the end effector of the robotic agent's position was used as the metric for model performance.

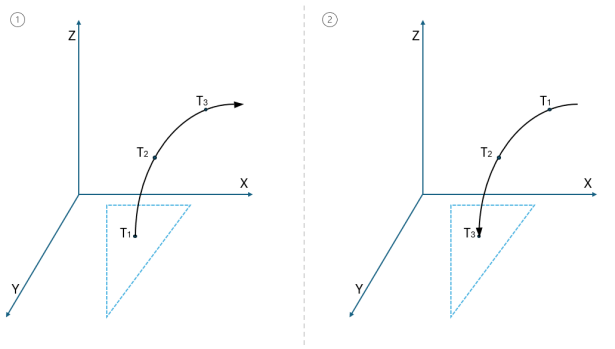


Dataset Generation

To get an agent to catch a ball, I first had to generate data for the model to be trained on. Points are first randomly generated within a 2D rectangular space given within the bounds of (0, Xmax) in the x-axis and (0, Ymax) in the y-axis. The problem exists that there are points within these bounds that are unreachable by the robotic agent, but we can move these points that are outside the working space into the working space. The points in the light gray area, below the mid-line of the triangle's height, are reflected across that line. Next, these points are each reflected across the lines given by the left and right sides of the triangle represented by the 2D working range of the robotic arm. After the triangular are of the working space is generated, a semi-circular area is added onto the end of the triangle.

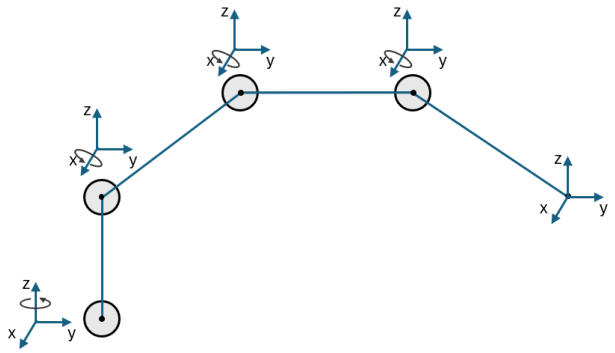
Given a random (x,y) position within the working space, we use that as a starting point to throw the ball from. We generate a random starting velocity between 2 and 7 m/s, as well as three random angles: psi, theta, and phi. The three angles are in relation to the x, y, and z axes, respectively. With those parameters, the ball is thrown from that point at the initial velocity, which is decomposed into its dimensional components, using the angles psi, theta, and phi.

On each time step, the drag force acting on the ball is calculated in each of the 3 dimensions. This is used to update the acceleration on the ball in the 3 dimensions, which is then used to update the 3 velocity components. From the velocity, we update the ball's (x,y,z) position. The position of the ball is tracked until it either hits the ground (z=0) or it travels outside of the 3D range of the arm. Once that happens, the simulation stops. From there, the time step of the trajectory is reversed, i.e., the ball is considered to be thrown from outside the working space, enters into the working space, and then eventually lands on the ground, within the working space.



The reason that this is done is to generate trajectories of a ball that the robotic agent has a reasonable chance to catch, with the assumption being that if the ball lands somewhere within the 2D working space of the arm, the ball can be caught at some point before it hits the ground.

Agent State



The agent has 4 degrees of freedom, all rotational. The first frame is located at the base, where it rotates around the z-axis. The second, third, and fourth frames all rotate about the x-axis. The final frame is located at the end of the last joint, and this is the point from which the euclidean distance to the ball will be measured. The reference frame from which all distances are measured in relation to is located at the base of the robotic agent, which is considered to be the point (0,0,0). On one time step, the actor chooses 4 thetas, which allows it to move to a different position in space, with the selection of the thetas constituting one action. The agent is allowed to act until the ball is caught or hits the ground. If either of those events occurs, the agent is set back to its initial position and a new episode begins.

Displacement Function

The displacement of the end effector is found using the homogeneous transformation matrix (H_5^1) from frame 1 (base) to frame 5 (last frame).

Knowing the bend angles of the actuators ($\theta_1, \theta_2, \theta_3, \theta_4$), we can find the displacement from the first frame to the second frame, the second frame to the third frame, and so on. With those HT matrices, we can multiply those to get the HT matrix from frame 1 to frame 5. This tells us the orientation and displacement of the end effector from the base of the agent. The actual position of the agent is represented as a function of the result of the homogeneous transformation from the first frame to the last frame, which is determined by the particular expression of the 4 thetas.

$$\text{HTM} = \begin{pmatrix} \text{Orientation,} & & \text{Displacement} \\ 0, & 0, & 0, \quad 1 \end{pmatrix}$$

$$\text{We can find: } \left(H_2^1,\ H_3^2,\ H_4^3,\ H_5^4\right)$$

$$H_5^1 = \left(H_2^1 \cdot H_3^2 \cdot H_4^3 \cdot H_5^4\right)$$

displacement(theta1, theta2, theta3, theta4) : {

H_2^1 = change in orientation and displacement from frame 1 to frame 2 due to theta1

H_3^2 = change in orientation and displacement from frame 2 to frame 3 due to theta2

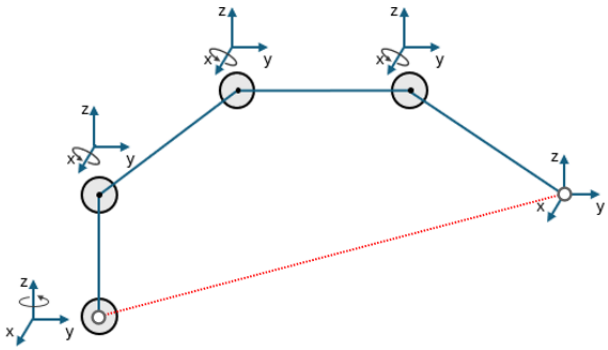
H_4^3 = change in orientation and displacement from frame 3 to frame 4 due to theta3

H_5^4 = change in orientation and displacement from frame 4 to frame 5 due to theta4

$H_5^1 = H_2^1 \cdot H_3^2 \cdot H_4^3 \cdot H_5^4$

return the displacement found by H_5^1

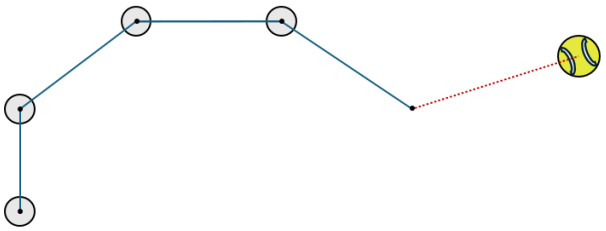
}



Reward Function

The agent receives a reward from the environment to tell it how good or bad its actions are. The reward function is the negative euclidean distance from the center of the ball to the position of the end effector. In this way, when the agent tries to maximize its reward, it does so by trying to minimize the

distance between its end effector and the current position of the ball. Because all rewards are negative, the best reward it can hope for is a reward of 0. Whenever the euclidean distance is within a small enough range, the ball is considered to be caught, and once caught, the agent receives a reward of 0 for that time step, the agent is reset to its starting position (represented by the pre-determined starting values of θ_1, θ_2, θ_3, and θ_4) the next episode immediately begins.



```
reward_function(ball_position, arm_position) : {
        If the euclidean distance between ball position and
        arm position is less than some threshold, then return 0

        Else, return −(euclidean distance)
}
```

Model and Training

The agent uses an actor-critic architecture, with the actor and the critic being represented by their own deep neural networks. Both the actor and critics have a target network, which is used to generate the targets that are passed into their respective actor and critic networks. The target actor and target critics are updated every 200 episodes, with an episode being one throw of the ball.

A trajectory of a ball is highly temporal, meaning the current state is dependent on the previous state, which violates the independence assumption of the data for training deep neural networks. Therefore, an experience replay buffer is used to train the agent's model, randomly sampling from its experiences, allowing the model to learn from general trends of a ball's trajectory as opposed to the most recent ball trajectory.

```
Training_loop() {
        take an action based on the current state

        determine the reward from that action and if the ball
        was caught

        determine what the next state will be

        add the experience to the replay buffer

        train the agent on some random samples from the
        experience replay buffer

        update the state, so the next state becomes the current
        state

        if the ball was caught or the ball hits the ground, end the episode
        else continue on with the loop
}
```

```
Agent_train_critic() {
        sample current state, action, next state, and done flag from the
        experience replay buffer

        target_action = action that the target actor network would choose
                        for the next state

        target_Q_next_value = Q value that the target critic network predicts
                              for the next state and target_action

        target_Q_value = reward + (λ * target_Q_next_value)

        Q_value_predict = Q value the critic network would predict for the
                          current state and action

        The critic then uses gradient descent to partially update its
        parameters to in the direction that would minimize the MSE loss
        between target_Q_value and Q_value_predict
}
```

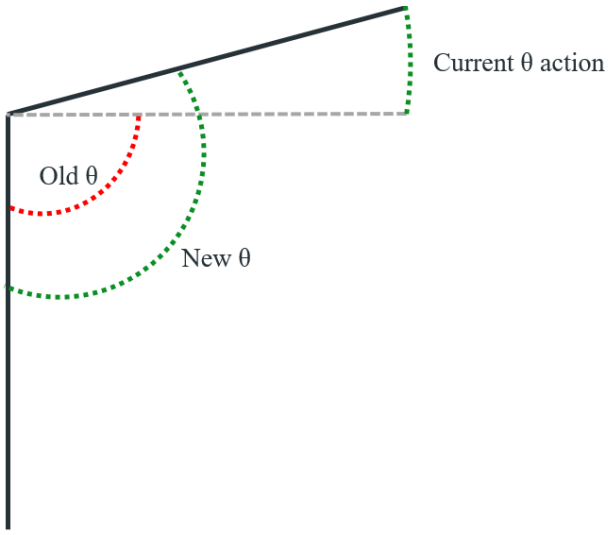
```
Agent_train_actor() {
        sample current state, action, next state, and done flag from the
        experience replay buffer

        Action_predict = action that the actor network would choose for
                         current state

        Q_value_predict = Q value the critic network would predict for the
                          current state and Action_predict

        The actor then uses gradient ascent to partially update its
        parameters to in the direction that would maximize the expected
        return, based on the value it sees for Q_value_predict
}
```

Actions

Actions are selected based on the current state, and these features are passed into the actor, which outputs 4 actions: $\theta_{1_{action}}, \theta_{2_{action}}, \theta_{3_{action}}, \theta_{4_{action}}$. The $\theta_{1_{action}}$ is added onto θ_1 to give us a new θ_1, the $\theta_{2_{action}}$ is added onto θ_2 to give us a new θ_2, the $\theta_{3_{action}}$ is added onto θ_3 to give us a new θ_3, and the $\theta_{4_{action}}$ is added onto θ_4 to give us a new θ_4.The 4 new theta values are passed into the displacement function, which gives us the new position (X,Y,Z) position of the end effector. Adding on the actions of the theta values represents moving the agent from the old position to its new position.

New theta = Old theta + current theta action

Displacement = displacement_function(new theta values)



This displacement gives us the new arm position, which we use to determine the reward based on the current position of the arm and the current position of the ball. After the reward is determined, the current state becomes the next state, and the next state is used to determine the current actions and so on.

Baseline

The control model was trained using an epsilon greedy action selection, meaning that a random probability was generated, and if this random probability was less than epsilon, the agent would choose 4 random theta actions, and would take those actions. The epsilon decays as the episode count increases, allowing the model to gradually shift from exploration to exploitation, with the initial random action selection allowing the agent to learn the value of its actions and how that would affect the reward that it received from having performed those actions.

The state features that the model trained on were the current ball position and the current arm position. The reward function is the negative euclidean distance from the ball to the arm, unless the euclidean distance is less then or equal to 0.032 meters (the radius of a tennis ball), in which case it receives a reward of 0.

State : (Ball Position, Arm Position)

Actions : (θ_1 choice , θ_2 choice , θ_3 choice , θ_4 choice)

Reward : (-Euclidean Distance)

The model was tested on 1000 episodes of a ball being thrown, specifically on ball trajectories generated using a different random seed than was used to generate the training data. The agent is allowed to choose actions at each of the discrete time steps in the episode, and the euclidean distance is measured between the arm and the ball at each time step. If the arm reaches close enough to catch the ball, it receives a reward of 0,the agent is reset to its initial position, and the next episode starts. Otherwise, the arm keeps trying to catch the ball until the ball hits the ground. Once the ball hits the ground, the agent is reset to its initial position and the next episode starts. The euclidean distance between the arm and the ball is summed up over all the discrete time steps in the 1000 episodes, with the sum of euclidean distances representing the performance of the agent.

A smaller euclidean distance sum represents an agent reaching the ball closer and sooner than other agents, therefore a smaller score represents a better agent.

From the visual representation of the model testing, the baseline model tends to follow some ball trajectory, but not the trajectory of the current ball. This is what inspired the next experiment, seeing that the action selection of the baseline model wasn't random, just wrong.

Prediction Error

I added an extra feature to the state of the model, a sort of "predictive error", representing the difference between the ball position and the arm position, i.e, where the ball is and where the agent thinks it should be. This allowed the model to more closely follow the trajectory of the ball, now that it understood more directly where to go to actually reach the ball. I believe the problem the baseline model suffered in comparison to this model is that the euclidean distance doesn't actually encode any information in where to move the agent to reach the ball. The baseline model understands how far it is from the ball, but can't actually determine where to go knowing that information.

State : (Ball Position, Arm Position, Ball Position – Arm Position)

Actions : (θ_1 choice , θ_2 choice , θ_3 choice , θ_4 choice)

Reward : (-Euclidean Distance)

With the prediction error, the agent actually has more positional information, and can maximize its reward by minimizing the prediction error on each time step. Given that these are the first features that show significant improvement in the model, all other experiments include the prediction error feature.

Positional Relations

From there, I tried to add even more features, specifically about the position of the ball in one axis in comparison to the position of the arm in one axis. For example, the product of (ball_x * arm_x), with ball_x representing the ball's current location on the x axis, and arm_x representing the arm's current position on the x axis. From this, the intention was to show the interaction of the ball's x position on the arm's x position. This was done for every combination of choosing one ball axis and one arm axis.

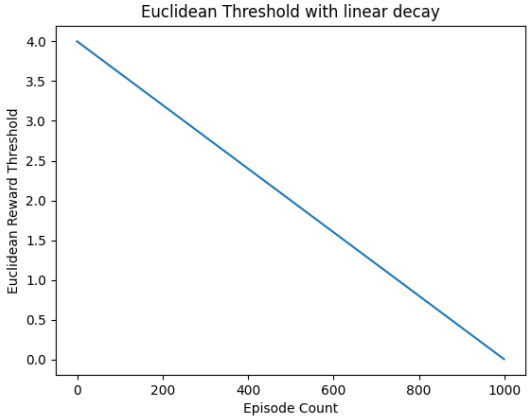
Positional relations = {
Ball_x * Arm_x,
Ball_x * Arm_y,
Ball_x * Arm_z,
Ball_y * Arm_x,
Ball_y * Arm_y,
Ball_y * Arm_z,
Ball_z * Arm_x,
Ball_z * Arm_y,
Ball_z * Arm_z,
}

These positional relationships were added to the model's state.

State : (Ball Position, Arm Position, Prediction Error, Positional relations)
Actions : (θ_1 choice , θ_2 choice , θ_3 choice , θ_4 choice)
Reward : (-Euclidean Distance)

Easier reward function

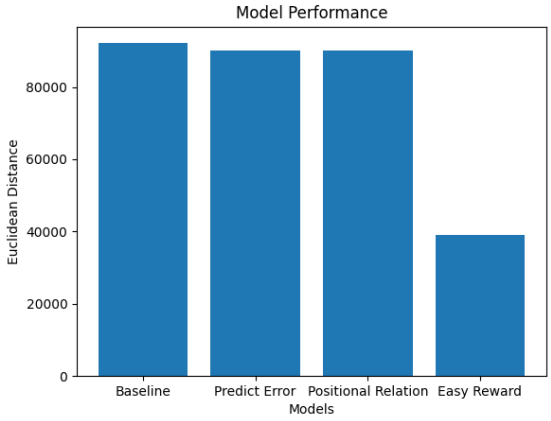
From preliminary testing, it seemed that the initial euclidean distance threshold used in training was too difficult given that it seemed that none of the models were actually catching the ball, even during training. To try to alleviate that, I changed the reward function to be easier. It starts with an extremely high initial threshold (in this case 4 meters), and if the arm is within 4 meters of the ball, it receives the reward. As the episode count increases, the euclidean threshold to receive the reward linearly decays, making it harder to receive that same reward. The goal was to gradually incentivize the agent to move closer and closer to the ball as time went on.



reward_easy(ball_position, arm_position, current_episode, max_episode) : {
euclidean threshold = 4 − (current_episode/max_episode)*4
If euclidean distance is less than some threshold
return 0
Else
return −(euclidean distance)
The threshold decreases with the episode count, making it harder to receive a reward over time
}

State : (Ball Position, Arm Position, Prediction Error)
Actions : (θ_1 choice , θ_2 choice , θ_3 choice , θ_4 choice)
Reward : (-Euclidean Distance, with a threshold that linearly decays)

Results



Of all the models, the model with the easy reward function performed best with a summed euclidean distance of 38936. Next best was the model with just the prediction error with a score of 90031. In third was the positional relation model with 90204, and in last was the baseline model with 92120.

For the model with the positional relations, I believe it performed worse compared to the prediction error model due to it possibly overfitting to the data, with all the additional features. The model with the easier reward seemed to have performed best due to the gradual increase in difficulty in receiving the reward, compared to all the other models starting out with the difficult to receive reward.

For further model improvement, I expect a reward threshold that decays even more slowly to produce even better results. This can be done by simply training the model for more episodes.

Click here for the code