RL & Sequential Decision Making, CS 5180, Spring 2024
Team: Ramez Mubarak, Kruthika Gangaraju
Project Report: Simulating and Controlling a Ball-Balancing Table

GitHub Link: https://github.com/RMubarak/RL_Project_Ball_Balancing_Table.git

Problem Statement:

In autonomous field robotics, one of the main challenges is handling unknown scenarios; a truly autonomous system can react to new stimuli and act accordingly. One situation we are interested in is balancing unknown loads using sensor data. We expect that through reinforcement learning, a robot can balance a previously unknown load properly and keep it upright and within its bounds.

For our project, we will investigate the use of RL algorithms on a simulated ball-balancing table. The main goal is to investigate which classical RL algorithm can produce the most successful ball balancing policy.

System Model:

The system in question is a ball-balancing table. It consists of a glass surface with integrated pressure sensors, a solid metal ball, and 2 servomotors attached to neighboring ends of the table. The desired result is to keep the ball centered and balanced when the system is disturbed by some external force to the ball or the table for a certain amount of time. Balancing an unconstrained item on a free surface can have many real-life applications, including designing mobile robots that carry and move unknown loads from one location to another.

The first step for any control system is to obtain the system model. Therefore, the first step was illustrating the free-body diagram of a sphere on a tilted table. To keep this model linear, it will be assumed that there is no slip between the ball and the table, the ball stays in contact with the surface at all times, and there is no air drag. The ball is modelled as a solid sphere and the table is tilted at angle θ.

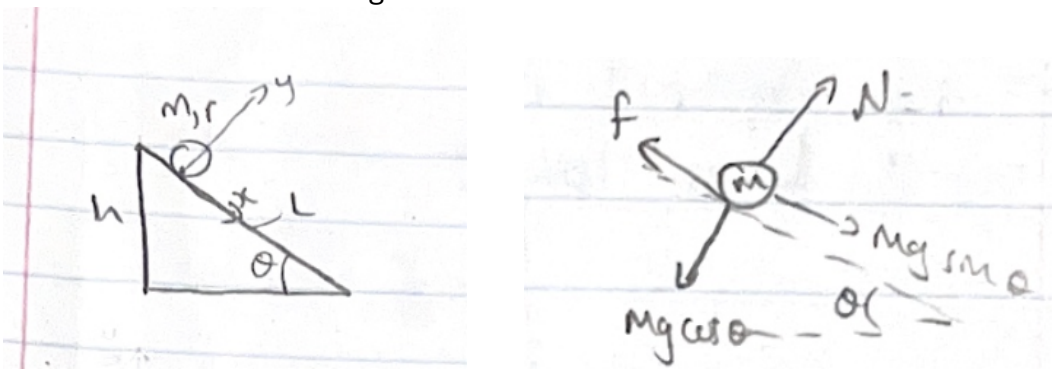


Figure 1: FBDs of Table and Ball

Summing the forces in the X and Y axes, we can say $\Sigma f_y = 0$ since the ball is always in contact with surface $\rightarrow$ $N = mg\cos\theta$, where N is the normal force exerted by the table on the ball.

Around the X axis, we derive: $\rightarrow f = \mu N = \mu_s mg\cos\theta$.

Also, $r*\alpha = ax. \rightarrow \alpha = \frac{ax}{r}$

Where $ax.$ = acceleration in the x direction, r is the radius of the ball, and α is the angular acceleration of the ball. Combining the previous equations,

$$\rightarrow \mu s\, \alpha\, g\cos\theta = \frac{1}{2} m\, ax \rightarrow \mu s = \frac{ax}{2g\cos\theta}$$

$$\rightarrow \Sigma Fx = m\, ax \rightarrow Mg\sin\theta - \mu s\, Mg\cos(\theta) = m\, ax$$

$$\rightarrow g(\sin\theta - \mu s \cos(\theta)) = ax$$
$$\Rightarrow ax = \frac{2}{3} g\sin(\theta)$$

For our table, we set the X and Y axes of our space to lie on the table, with the Z axis perpendicular to the surface of the table. Therefore, the table will be able to rotate about both the X and Y axes. Therefore, we assign Θx and Θy to induce movement in the X and Y directions, respectively. Giving us the following 2 equations that model the ball accelaration according to the table angles.

$$\boldsymbol{ax = \frac{2}{3} g \sin(\theta_x)}$$
$$\boldsymbol{ay = \frac{2}{3} g \sin(\theta_x)}$$

To derive the equations of motion that relate the servomotors to the table, the following FBD was considered.

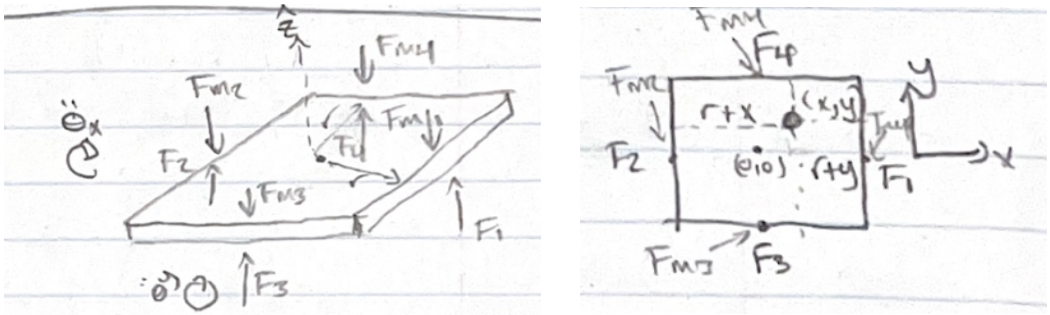


Figure 2: FBDs of Table and Forces

Forces F2 and F3 act on neighboring sides of the table to represent the servomotors. Opposing forces F1 and F4 are used to model the physical system's resistance/damping to the servomotors. Since we do not have a reference point to estimate the table's rotational damping or general response to the servomotors, the opposing forces keep the model steady.

The ball also applies weight on the table, and using force distribution equations for static loads, we can model the effect of the ball's mass on each point in the system as follows:

$$\text{Fm1} = \frac{(r+x)mbg}{2r}, \text{Fm2} = \frac{(r-x)mbg}{2r}, \text{Fm3} = \frac{(r-y)mbg}{2r}, \text{Fm4} = \frac{(r+y)mbg}{2r}$$

Where x and y represent the location of the ball on the table, with the origin (0,0) at the center of the table. Fmi is the force exerted by the weight of the ball on the locations of Fi in the opposite direction. mb is the ball mass. r is the distance from the center of the table to any of its edges (table dimensions are 2r x 2r), and g is 9.81.

Applying Newton's laws of motion, we get the following equations.

$$Ix * \ddot{\Theta}x = r(\text{F2-F1+Fm1-Fm2})$$
$$Iy * \ddot{\Theta}y = r(\text{F3-F4+Fm4-Fm3})$$
$$Icm = \frac{1}{12} Mt\ (4r^2)$$
$$Ix = Icm + Mt * hx^2$$
$$Iy = Icm + Mt * hy^2$$

Substituting the values of Fm1, Fm2, Fm3 and Fm4, and the moments of Inertia Ix and Iy,

$$Ix * \ddot{\Theta}x = r(\text{F2 - F1}) + Mb * g * x$$

$$Iy * \ddot{\Theta}y = r(F3 - F4) + Mb * g * y$$

The 4 equations to represent our system are as follows:

1. $\ddot{x} = \frac{2}{3} g \sin(\Theta x)$
2. $\ddot{y} = \frac{2}{3} g \sin(\Theta y)$
3. $Ix * \ddot{\Theta}x = r(\text{F2-F1}) + Mb * g * x$
4. $Iy * \ddot{\Theta}y = r(F3 - F4) + Mb * g * y$

Simulation Setup: Please refer to env.py.

The simulation was created in Python and adapted to the gymnasium environment for ease of use. The environment uses the 4 equations shown above to drive the game forward.

The game is advanced through the step method, which takes in an action and uses the actions_to_changes helper method to update the forces in the servomotors, update the angle based off the new forces, then update the ball position. If the ball goes out of bounds (falls off), that is considered a failure with a large negative reward. If the ball remains on the table it gets +1 reward per timestep. If the time elapsed without failure exceeds a certain threshold, the game has been won and a high reward is given. The game is then reset with random ball velocities.

In terms of interacting with the agent, the simulation can provide a simulated sensor measurement to the agent relaying ball position only. The simulation can also add gaussian noise to simulate the real glass pressure sensor if the init parameter *sensor_noise* is set to true at initialization.

The environment hyperparameters can be initialized as follows:
Time_limit: A time limit to win if you keep the ball on the table.
sensor_noise: A boolean to determine whether to add sensor noise
sensor_std: Standard deviation of the normal distribution to represent sensor noise
sensor_sensitivity: The decimal places representing the sensitivity of the table's pressure sensor (readings are in m)
ball_mass: the mass of the metal ball (Kg) placed on the table (please note that the radius of the ball is unneeded as it cancels out --> check the system dynamics section of the project)
table_mass: mass of the glass surface/table (Kg) to determine the inertia of the table movement
table_length: length of one side of the square table (m)
force_step: The magnitude of force each servomotor can apply at a given timestep, this can be changed along with dt to represent the actual servomotors' performance if the real system specifications are provided
dt: the time step to be used in this environment
angle_limit: The maximum angle the table can produce (models a physical limitation)
force_limit: The maximum force the servomotors can produce

The following 5 actions are permitted in the game where 'X_X' means:-

'Action for Motor 1 along x axis_ Action for Motor 2 along y axis'

N --> Nothing

U --> Up

D --> Down

N_N = 0 N_U = 1 N_D = 2 U_N = 3 D_N = 4

Algorithms: Please refer to algorithms.py and policy.py

The following algorithms will be used to train policies:

- On-Policy MC Control
- Expected SARSA
- Q-Learning

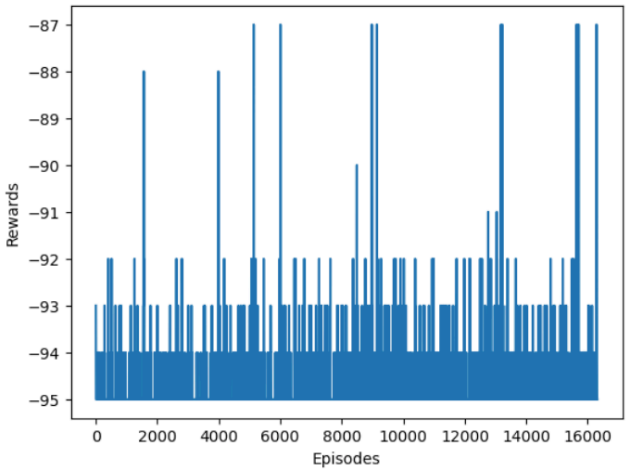
Note: Please refer to trail_run.py for all 3 trial runs.

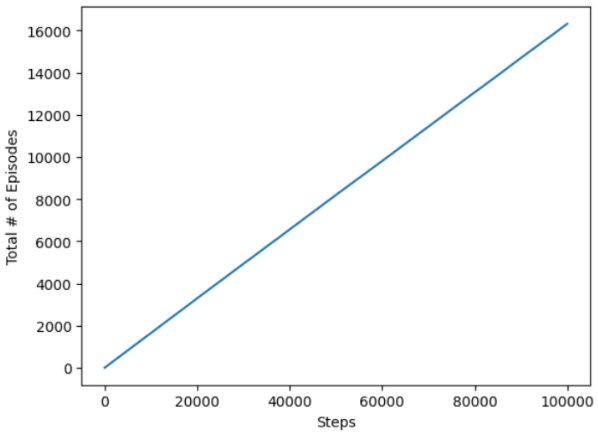
First Trial Run:

The first test run was used to check if the agent learns or not. Therefore, only the Q-learning algorithm we developed in class was used with the following system hyperparameters. This was run for 100,000 steps.

Time_limit: 5
force_step:100
force_limit: 600
win_reward = 1000
loss_reward = 100
standard_reward_per_timestep = 1
epsilon = 0.1
step_size = 0.5
discount_factor = 0.99

The results did not show any improvement in policy and the following results were graphed:





The policies do not learn over time. The rewards are negative and non-increasing over time. Moreover, there were no successes in any of the 16,000 episodes played.

One possible issue with the current setup is that the agent only knows ball position. In this game, the ball speed is very important to decide which actions to take. Therefore, a new approach should be taken for our agents where they keep track of a few previous states and estimate the ball velocity from that.

Another possible issue is that the model itself is not accurate or playable. That could mean that the model hyperparameters are not tuned properly. Or that the model is too complex. This trial run started with controlling the table servomotors and dynamics, which add a lot to the back-end calculations and could prove too complex for classical RL algorithms. Therefore, we decided to take a step back and simplify the model.

Tuning the model:

To avoid a rabbit hole, we took a step back with our dynamical model and assumed that since the agent can control table servomotors, which in turn control the table angles, we can directly have our agent increment table angles for the simulation. In real life, that would be akin to figuring out how much your servomotors increase or decrease table angles per step input and use that as your control method.

Therefore, we added a Boolean field to the environment that allows agents to bypass table dynamics and directly control the angles. The actions were modified accordingly.

Another change was our reward structure. Since our win condition is to keep the ball balanced for a certain time-period, and the standard reward for keeping the ball on the table was always 1, the agent would not learn to keep the ball as close to the center as possible, which should be rewarded positively.

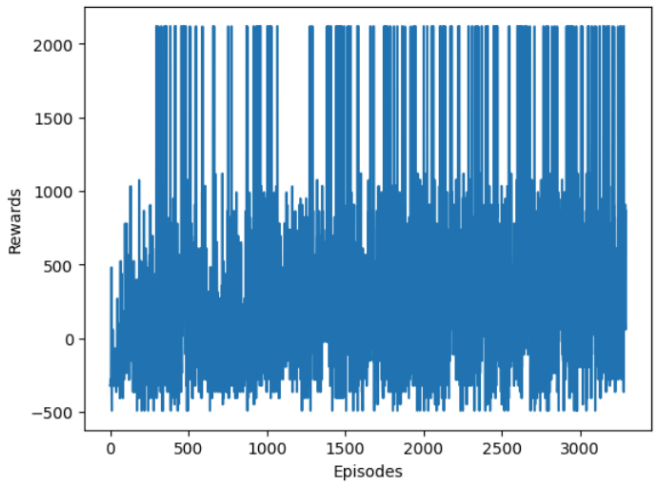
Therefore, we changed the standard reward for each time step to be inversely proportional to how far away from the center you are and removed the win reward. The loss reward was changed to -1000 to identify failed episodes easily.

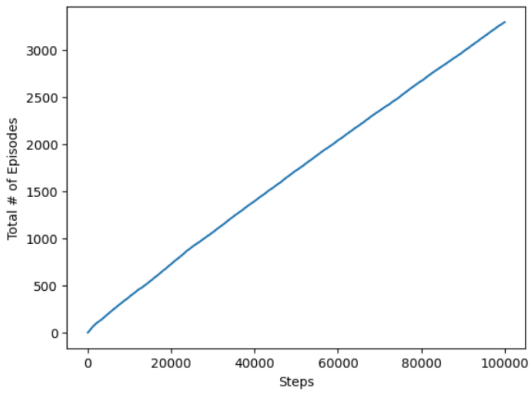
Tuning the agents:

A crucial aspect of this game is ball velocity, it is a component that will affect your decision making at any time as a player. Therefore, we decided to feed the agent the ball velocity as well as part of the state. While this is not very realistic in a real-world setting as the table only has pressure sensors, the ball velocity can be estimated by the agent based on previous steps and for our simulation, it will act as an indicator to whether the agents can learn with velocity added to the states.

Second Trial Run:

The second run used the same Q-learning algorithm and number of steps as trial run 1. However, the agent controls the table angles, states include velocity data, and the reward structure is different.



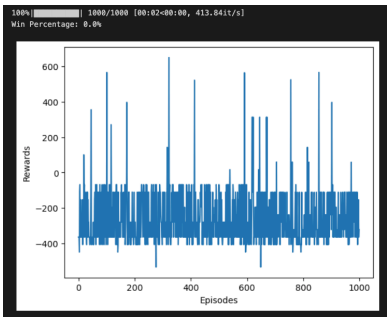


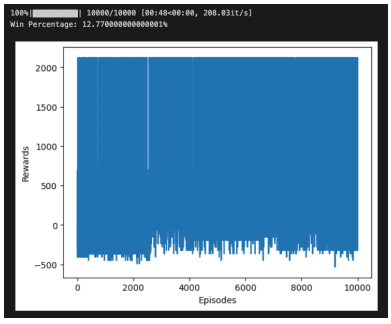
These results are much more promising. There are a lot of positive rewards indicating successes. While this is a short run that cannot determine training trends, it seems that the rewards are increasing per episode. One other indicator of success is that the total # of episodes over 100,000 steps drastically decreased from ~16,000 to ~3,300, indicating longer playing time on the table. It is important to note there are still more failures than we would like, and the policy is far from optimal, however; this is a short trial run, and further studies will be conducted to find an acceptable policy.

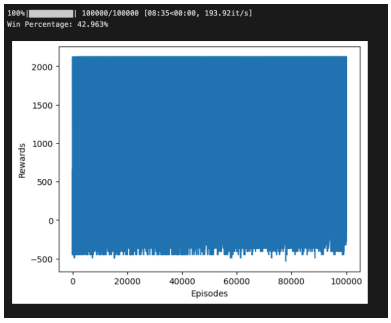
The next steps were to investigate how well the algorithms learn over time. A few modifications were needed, such that instead of running the agent for a certain number of steps, we use number of episodes to play so we can things consistent across other algorithms. The learning agent was also modified to keep track of wins and losses so we can compare policies by win percentages as our main indicator of success.

Third Trial Run:

To make sure our agents are learning with each episode, the same Q-learning algorithm was trained 3 times, once with 1,000 episodes, then 10,000 episodes, and finally 100,000 episodes. The reward graphs and win percentages were as such:







# of Episodes	1,000	10,000	100,000
Win Percentage	0.0%	12.7%	43.0%

It is clear from the win percentages that the agent is learning with each episode. The graphs are not showing a clear trend due to the failed episodes in between successful ones, and the large difference in win/loss rewards makes it hard to see a trend in the graph other than the filled in space.

It is important to note that with 100,000 episodes, we only get a total of 43% success rate which contributes to the graph's unclear trends. Moreover, the win percentage includes all trials even when training. Therefore, to properly compare algorithms, we will need to test the policies separate from training them to get a clear picture of success rates.

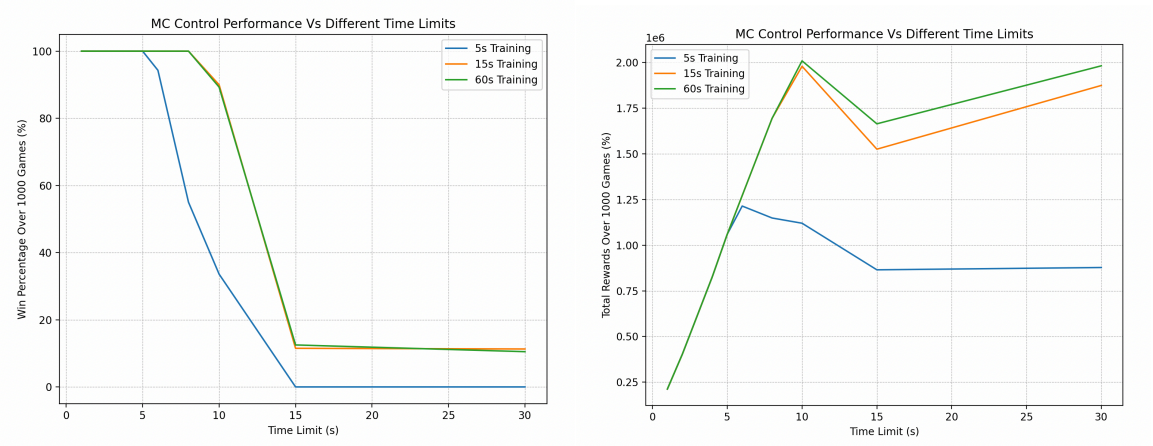
Policy Training & Evaluation:

Now that we have a system for training an algorithm properly, the next steps are to create near-optimal policies using different algorithms and test their success rates. We trained each of our algorithms (On-Policy MC Control, Expected SARSA, Q-Learning) thrice for up to 1,000,000 episodes and recorded the state-action values (Q dictionaries). Each time, the policy was trained on a game with a different time limit (5s, 15s, and 60s). Overall, 9 policies were trained for 1,000,000 episodes each (Expect for 2 MC policies that were trained for less). Please refer to train_policies.py for policy generation. For the state-action value dictionaries, please refer to the following google drive (Files were too large for GitHub): Policies_Folder.

Then, the values were used to create a greedy policy to play the game for 10,000 episodes. The total reward over all episodes and overall win rate were recorded as the success factors for each policy. Instead of plotting the rewards per episode, the 10,000 episodes were divided into 10 games with different success conditions. In other words, a single policy would be evaluated on 10 different games, each with a different time limit (game success is defined as keeping the ball on the table for the set time limit), and play each game 1,000 times to provide trends showing win percentage vs time limit and total rewards vs time limit. Please refer to play_game.py. To be able to play the game properly, please add a 'Policies' folder in the same directory and download trained policies from Google Drive (link above).

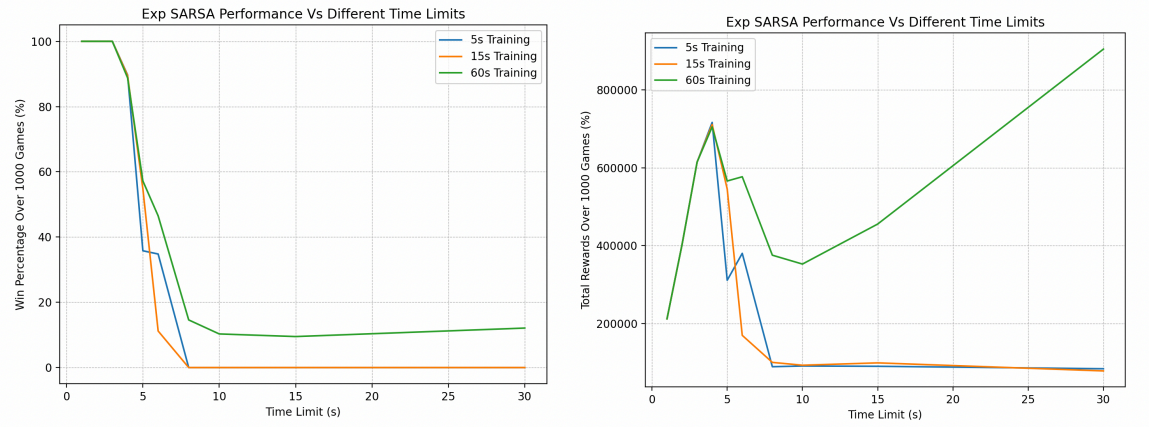
Preliminary Results:

On-Policy Monte Carlo Control: (15s and 60s policy trained for 300,000 episodes only due to computational and loading/unloading issues)



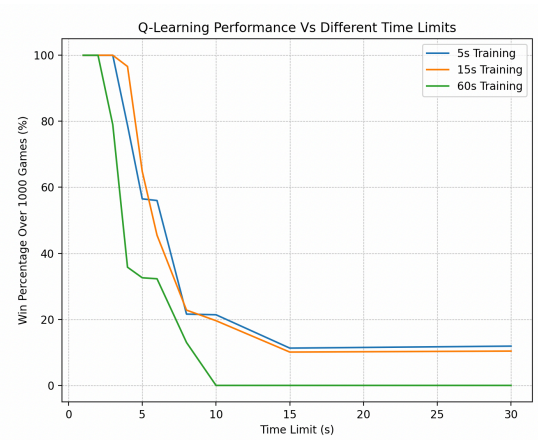
The 5s policy performs the worst for MC control even though it was trained for ~3x longer, and the 15s and 60s policies perform similarly in terms of win percentage. However, due to the total reward being higher for the 60s policy, it is considered the most optimal MC control policy.

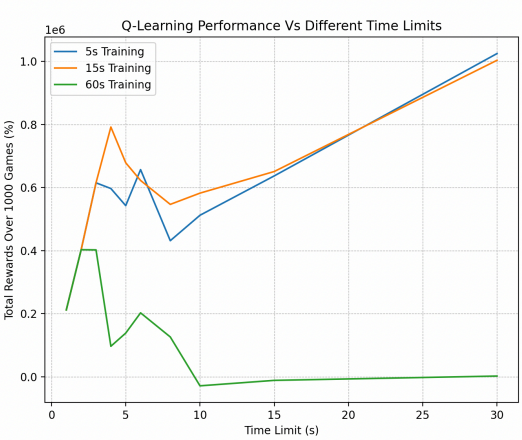
Expected SARSA Policies:



For expected SARSA, the 60s policy outperforms the 5s and 15s policy in all success metrics.

Q-Learning Policies:

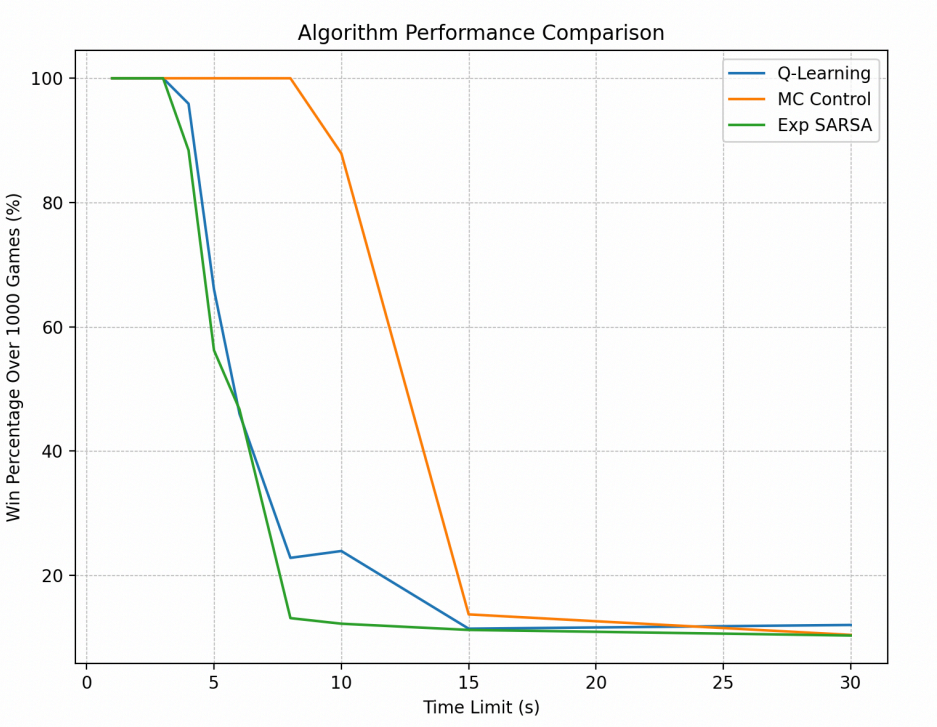


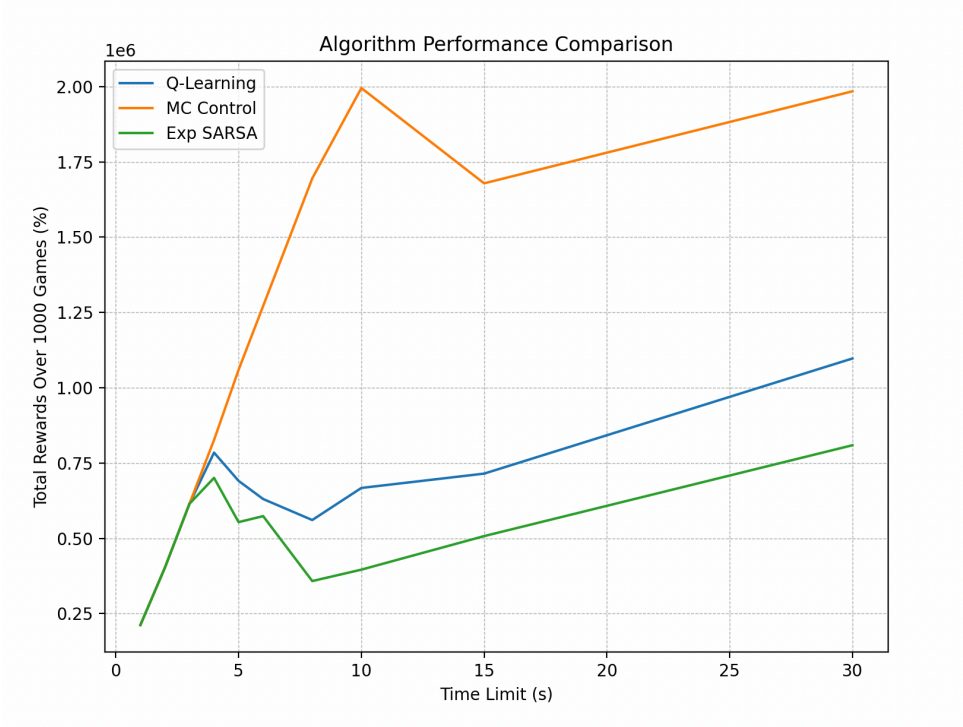


For the Q-learning policies, the policy trained on games with a 15s time limit performs the best (slightly better than the 5s policy but much better than the 60s policy).

Algorithm Comparison:

For each RL algorithm, the best policy was used to compare between the 3 algorithms and the following graphs were produced.

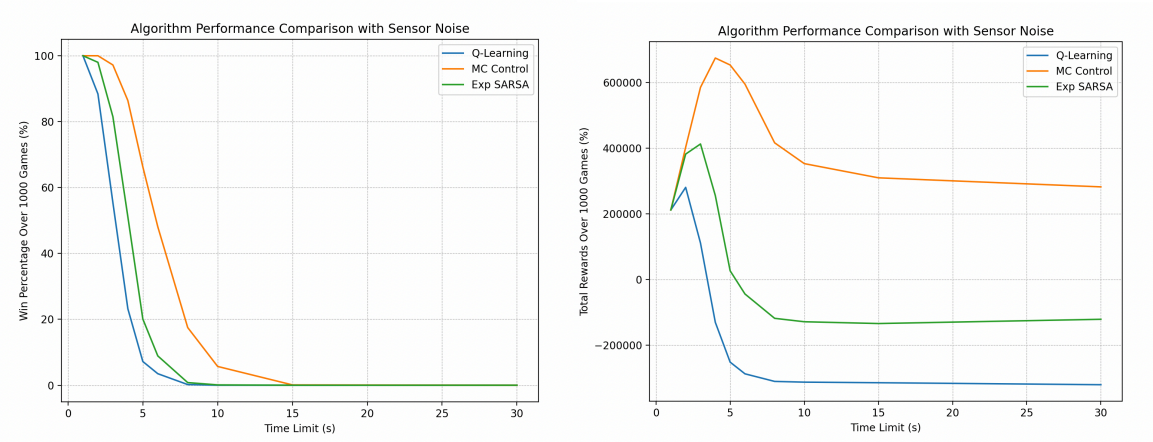




The results show a clear advantage for the MC Control policy. In terms of win percentages, it 100% succeeds to control the ball every time for up to 8 seconds compared to <4 seconds for the other policies. While all trends show a large and similar linear dip in success rates at different points, they all converge to ~10% success rate once the win condition is 15 seconds or longer. That could be due to the model itself as it might require more hyperparameter tuning to be able to play the game longer. In terms of total reward, MC control outshines Q-Learning and Expected SARSA.

From our class experiences, our initial expectations were that MC Control would perform the worst, followed by Q-Learning then Expected SARSA outperforming the rest. However, the results were the exact opposite. One possible reason MC control is performing best is that it learns from full episodes, and since our model has a short time limit, it is able to quickly end episodes and learn from them. However, the other algorithms are updating state-action values every time step. Another reason could be that actions affect states several time steps ahead. That means that deciding an action early in the game greatly affects the rest of it, which would make sense in our environment given a large input forces the table to rotate to high angles, making it harder to correct in future steps. MC control can learn from those failed episodes better than Q-learning and Exp SARSA.

Sensor Noise Addition:
Since we want to emulate the real world, we also tested the best policies on a game with a noisy sensor.



Unfortunately but expectedly, our policies perform much worse with noise, even failing to 100% keep the ball balanced for 5s every time. However, an important observation is that policies continue to converge in terms of win rates around 15s, further suggesting that the model parameters need to be finely tuned for policies to succeed after 15s. While MC continues to outperform the rest, Exp SARSA performs better than Q-Learning. While these results are preliminary and more studies should be run with different noise parameters, overall; On-Policy Monte Carlo Control is our most trusted algorithm so far!

Learned Lessons and Next Steps:
Lessons Learned:

- Test the model as much as you can! One of the challenges faced in this project was tuning the hyperparameters of the model to make it playable. We did not have a manual controller to play it ourselves, which would have been very helpful.
- MC Control can work better than more advanced classical RL methods. The results definitely took us by surprise here!
- Start simple! While the professor did warn us about this, we started with the full dynamical model of the system including the table and servomotors. That proved to be impossible to tune properly and play. Once we simplified the model while keeping it realistic, the project started to progress much faster.

Next Steps in the Project:

- Get an optimal policy that can balance the ball forever! (Easier said than done)
- Run parametric studies on how different model and agent hyperparameters affect the results to find an optimal policy.
- Try to train policies on a game with sensor noise. Our policies were trained with a noiseless environment then tested on noisy ones. Maybe policies would perform differently if they were trained with noise.
- Try to develop policies using deep Q-learning models or use more advanced RL algorithms.