Unity Soccer AI

[redacted] Samuel Dahlberg

1. INTRODUCTION

This project uses a Unity learning environment called MLAgents which can be accessed via a Python API. The Unity toolkit contains multiple environments that can be used for training agents using reinforcement learning. The API then interacts with and can manipulate the chosen environment. For our project, we decided to train a soccer playing agent, implementing our own algorithm in Python, to be able to chase a soccer ball and score a goal in a simulated game. This paper addresses our approach to train an agent that is able to navigate the soccer field appropriately by moving towards the soccer ball, pushing it in the direction of the goal, and being able to score a goal.

Training an agent to score a goal on its own is challenging for a few reasons. For one, we had to train an agent in a continuous state space, which required choosing an algorithm that could select the best discrete actions to take in a large state space. Second, we had to decide which features to include in our observation vectors, which we implemented on our own. The approach we took was using a semi gradient SARSA algorithm for learning. A variety of observation vectors were tried and tuned according to the speed of learning. Features that were tried and included in observations include the x and y position of the player and ball, the distance from the ball to the goal, the distance from the player to the ball, and the distance of the player to the net. Values of rewards, such as the reward for the player hitting the ball, the reward for the player scoring in their own goal, and the reward for the ball ending up in the other team's goal, were also changed to optimize performance.

We implemented our own feature vectors and tested how well the agent was able to consistently hit the ball and score goals. We tested five main feature vectors with different observations from the environment. Overall, the vectors that performed best were those in which a tile coding representation of the field was used. One of the top performing vectors only included the tile of the player, the ball, and the action value. The other vector contained these features but also added on the velocity of the ball as well as Manhattan distances between the player, ball, and net. Adding in distances and velocities did not significantly improve the performance, and therefore it was determined that changing the representation of our state space had the largest impact on our results.

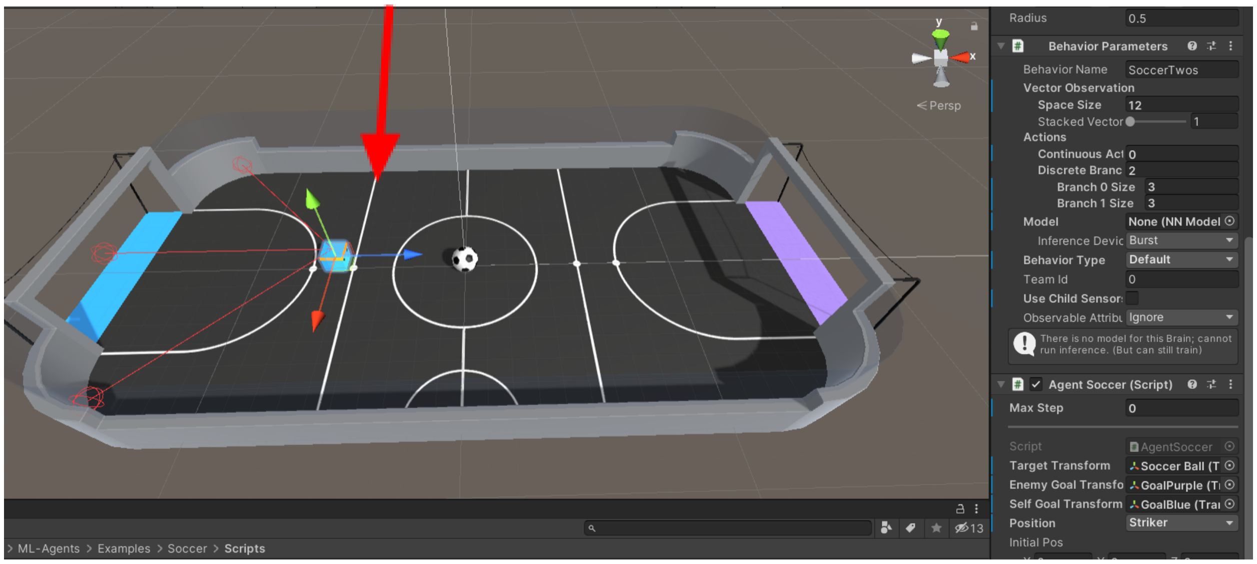
2. RELATED WORK

An example of an approach to a similar problem was documented by Peter Stone, Richard Sutton and Gregory Kuhlmann in their paper, “Reinforcement Learning for Robocup Soccer Keepaway”. In this paper, the authors trained between 5 and 9 agents to compete against each other in a game of soccer keepaway, where the goal of a team was to keep the ball away from the other team for as long as possible. Their problem was similar to ours in that they were dealing with training agents to make real time decisions in a large, continuous state space (a field) [1]. They also used the SARSA algorithm for training their agents, and function approximation for observations. This paper discusses similar challenges that we ran into, such as navigating a large, continuous state space. As a part of their solution, they implemented linear tile coding in order to track where the agent was on the field and incorporate into their function approximation [1]. We ended up following a similar approach.

The main difference between their paper and our project is that they attempted to implement a team component, where two or more agents work together as a team. In our project, we only focused on a single agent attempting to score a goal. Therefore, their observation vectors focused more on the distances and angles between teammates, opposing players, and the ball, rather than the relative positions and distances of the ball to players and to the goal state. They also focused more on tracking the amount of time that a team was able to keep a ball in their possession, rather than on goal scoring, to measure the success of their model. They used a different variant of SARSA, an Episodic Semi Markov Decision Process (SMDP) SARSA.

We also used the MLAgents Python API for reference. In our code, we had to call their methods and attributes, such as behavior_specs and obs, and use their objects, such as ActionTuple, to output the actions we generated so that our outputs were compatible with the Unity environment.This required reading their documentation carefully to understand how they use the API to train their own default agents. Their API documentation outlines the various types of entities and methods necessary to run a successful simulation. The API focuses on their Agent objects, which have various attributes and functions, and are uniquely identified by their agent_id. The environment is initialized at the beginning of the code, and from the environment all the other objects can be called. Other functions that directly affect the environment object are reset, close, and step. Step() moves the simulation forward by a single step, reset() resets the simulation, and close() closes it [3]. A Behavior is an object that “defines specific attributes of the agent such as the number of actions that agent can take” [5]. Steps are categorized into DecisionStep objects and TerminalStep objects, which can be called via the method get_steps, which returns a tuple of “the steps of the agents that requested a step in the simulation” [3]. A decision step is an object containing data the agent acquired since the last step of the simulation, such as an observation vector and reward. A terminal step returns the data collected at the end of an episode. Actions are accessed via objects called ActionTuples, which contain an attribute called discrete (in the case of discrete action spaces), a numpy array of actions [4].

3. APPROACH



```
Radius                    0.5
Behavior Parameters
  Behavior Name           SoccerTwos
  Vector Observation
    Space Size            12
    Stacked Vector        1
  Actions
    Continuous Act        0
    Discrete Branc        2
      Branch 0 Size       3
      Branch 1 Size       3
  Model                   None (NN Model
    Inference Devic       Burst
  Behavior Type           Default
  Team Id                 0
  Use Child Sensor
  Observable Attribu      Ignore
  There is no model for this Brain; cannot
  run inference. (But can still train)
Agent Soccer (Script)
Max Step                  0
Script                    AgentSoccer
Target Transform          Soccer Ball (T
Enemy Goal Transfo        GoalPurple (T
Self Goal Transform       GoalBlue (Tra
Position                  Striker
Initial Pos
```

Figure 1

3.1 Environment

The Unity MLAgents toolkit is made up of four main components. At the base of the project is the learning environment. An environment represents the selected scene, of which there are many to choose from, as well as the characters and objects of that scene. The Unity environment provided a place where we could control various aspects of the game settings. The environment we used was called "SoccerTwos", which is a 2v2 soccer game. We decided to limit our environment to just 1 agent to start, focusing on chasing the ball and scoring goals. With added agents we felt the learning process may become too complex. **Figure 1** shows an example of how the Unity environment is represented in the Unity application. C# code was provided to control the agents and the environment around it. Through the provided code, and additions made by ourselves, we were able to control all aspects of the environment that we needed, from rewards and observations to ball placement on reset. Having a premade environment like Unity that provides very easy ways to control the scene was essential for us, as it took away the large burden of the actual game development and allowed us to focus on the algorithm. In the environment, we have the player spawn in a random location anywhere behind the line in **Figure 1** indicated by the red arrow. The ball is spawned in randomly as well, and is placed within the two midfield lines (the line indicated by the red arrow, and its counterpart on the other side of the field).

The Python API is outside of Unity and communicates with the environment via a communicator which is built into the environment. The Unity MLAgents package has its own algorithm to train soccer agents, which is stored in the Python trainer component of the package and communicates directly with the API. This algorithm produces a neural net that the agent uses to navigate the field and play competitively. While none of these built in algorithms were used, it was a useful tool to see an example of a working agent, and additionally provide an “enemy” we could test against if we decided to implement 1v1 soccer. **Figure 2** showcases the relationships between the components that allow the application to run.

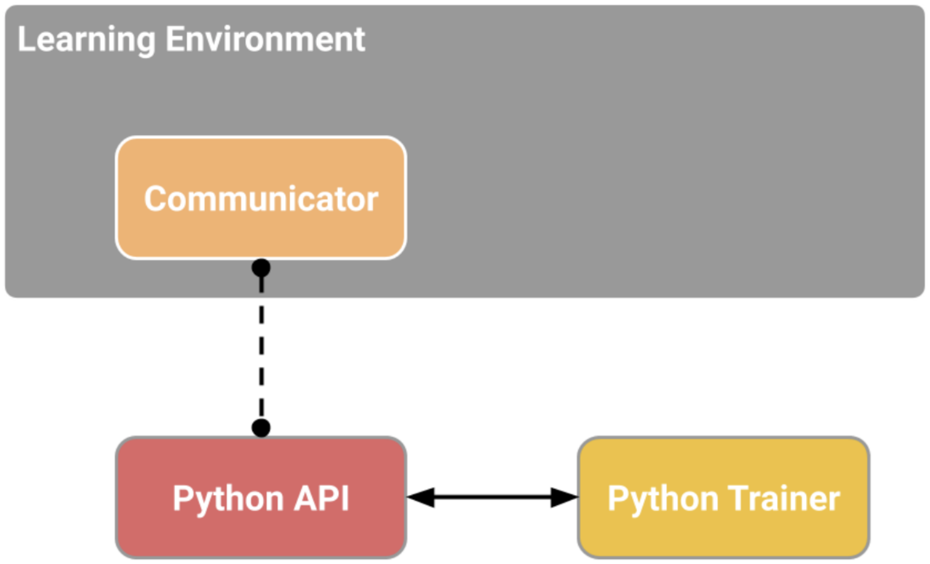


Figure 2

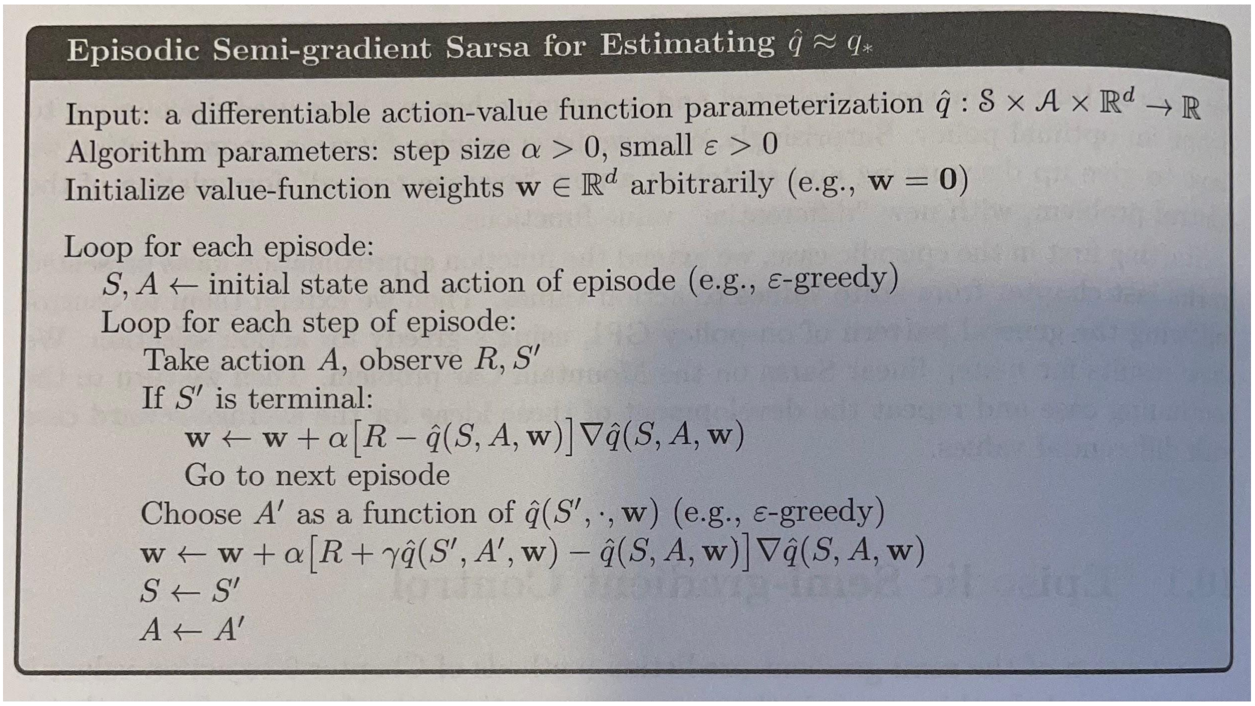
3.2 Problem

A Unity learning environment is built off two main components: Agents and Behaviors. An Agent, which is linked to a Unity object and represents a character in a simulation, “handles generating its observations, performing the actions it receives and assigning a reward (positive/negative) when appropriate” [5]. Each agent is linked to a Behavior, which “defines specific attributes of the Agent such as the number of actions that agent can take” [5]. It is essentially a function that takes in observations and rewards from the Agent and outputs actions. We aimed to figure out what the best way to train a soccer agent to chase a ball and score goals was using our own custom algorithm.

In order to use our own custom algorithm, we had to configure our Python algorithm to take in and output the proper attributes and objects. Unfortunately, as comprehensive as the MLAgents extension is, it lacks clear documentation when it comes to interfacing with Unity through Python. The extension was mostly built to run the built in algorithms, so creating your own is less documented. We spent more time than anticipated on this interfacing stage, spending many hours debugging the code, and correctly labelling the inputs and outputs.

3.3 Semi gradient SARSA algorithm

The algorithm we used to train our agent was semi gradient SARSA, which uses function approximation to select the best actions to take. This is an on policy algorithm, which means it estimates the optimal action subject to the current policy being executed by the agent. The agent then continually updates the policy according to changing estimates of Q(s,a). **Figure 3** shows pseudocode for how the algorithm works.



```
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        If $S'$ is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$
```

Figure 3

This algorithm was chosen due to the nature of our discrete action and continuous state spaces. When comparing Monte Carlo with Q-Learning/TD Learning/SARSA we choose to go the SARSA route. This is because Monte Carlo is an offline policy and has to finish an episode in order to learn. However there is a high chance it never makes it to the ball in 1 episode and thus we thought Monte Carlo wouldn’t perform well. SARSA, being an online policy, made it more appropriate for this problem.

3.4 Observations

The default Unity soccer agent uses ray perception sensors, an object which is implemented as a multi-dimensional numpy array. An agent has sensors in the front and back and the observations are recorded in the array as read by the sensors. There wasn't enough documentation for us to fully understand what type of information the default observation vector was returning, therefore we determined that using the ray perception sensors as part of our observations for our agent was too complex. We implemented our own observation tools to make our own observation vectors. At first, we used the following observations to create our vectors:

Features 1:

- x, y, z positions of the player
- x, y, z positions of the ball
- Manhattan distance between the player and ball
- Manhattan distance between player and net
- Manhattan distance between the ball and net

In order to make it a state-action value, we had to do a one-hot encoding of the actions at the end of the feature vector.

After observing the results of the initial vector, we tried to include the velocity of the ball in its X, Y, Z. This was an attempt in order to get the agent to chase the ball for a second time.

Features 2:

- x, y, z positions of the player
- x, y, z positions of the ball
- Manhattan distance between the player and ball
- Manhattan distance between player and net
- Manhattan distance between the ball and net
- x velocity, y velocity, z velocity

The third approach we tried was including not only the Manhattan distances between player, ball, and net, but also the relative distances between the player, ball and net's x, y and z coordinates. Our thinking here was that in separating the dimensions, the agent may learn which direction it generally needed to go to chase it.

Features 3:

- Distance X between player and ball
- Distance Y between player and ball
- Distance Z between player and ball

- Manhattan distance between the player and ball
- Manhattan distance between player and net
- Manhattan distance between the ball and net
- x, y, z velocities

After seeing the poor performance of the third feature vector approach, we decided to attempt to use a different representation of the field. Having one weight for each position is a very generalized feature that is not precise enough to properly train. So, rather than using X, Y and Z positions to determine the optimal action to take, we implemented a tile coding representation. This means splitting the board up into small sections, or tiles, to input as features. This is a solution that was found in "Reinforcement Learning for Robocup Soccer Keepaway" by Peter Stone, Richard Sutton, and Gregory Kuhlmann. According to Stone, Sutton and Kuhlmann, "An advantage of tile coding is that it allows us ultimately to learn weights associated with discrete, binary features, thus eliminating issues of scaling among features of different types" [1].

We divided the board into 465, single unit sized grids. The feature vector became a one-hot tile encoding of the player, ball, and action space. This feature vector had a size of 936.

Features 4:

- Tile of the player
- Tile of the ball
- Value of action

After seeing some increase in performance, but still room for improvement, we added in some of our original features, such as distances and velocities. This set of features was the final agent we trained.

Features 5:

- Tile of the player
- Tile of the ball
- Action value
- Manhattan distance between the player and ball
- Manhattan distance between player and net
- Manhattan distance between the ball and net
- X, Y, and Z velocities

3.5 Rewards

Rewards were obtained via the rewards attribute that is callable on a steps object. Rewards were pre-set in the Unity backend code which sends the results of the game to the Python API [5]. There were three types of rewards involved in this simulation: a negative living reward, a reward for touching the ball, and a reward for goal scoring. A negative living reward is a small negative penalty which brings the overall reward score down for every step in the simulation where the player has not “died” (in other words, has not scored). This allows for longer episodes where the agent did not score or touch the ball to have more negative scores, and helps the agent understand that the quicker a goal is scored, the better. Originally, the default reward values for touching the ball and scoring a goal were extremely low (0.2 for hitting the ball, 1.0 for goal scoring). We believe that these values were so low because the original environment was set up for a 2v2 game. If the reward for goal scoring was much higher, the agents would probably not focus as much on learning to pass or defend. However, since we were only focused on training a single agent to chase a ball and score, we increased the reward for touching a ball to 10.0 and the reward for scoring a goal to 100.0.

4. RESULTS

4.1 Evaluation

The performance of agents was evaluated by plotting the episode against the steps and rewards using Matplotlib, a data visualization library in Python. Better performance was indicated by less steps, as less steps were indicative that the episode ended sooner due to a goal being scored. Additionally, higher reward values meant that the agent was hitting the ball consistently. Looking at **Figure 3** and **4** for example, in **Figure 3**, each episode is plotted along the x axis, with the number of steps to indicate the general performance. If the step count is at the max (100), then the agent did not score in that episode. The lower the step count, the quicker it scored in the specific episode. In **Figure 4**, again the episodes are plotted along the x axis, with reward as the performance measurement. The higher the reward, the more ball touches the agent had.

4.2 Performance

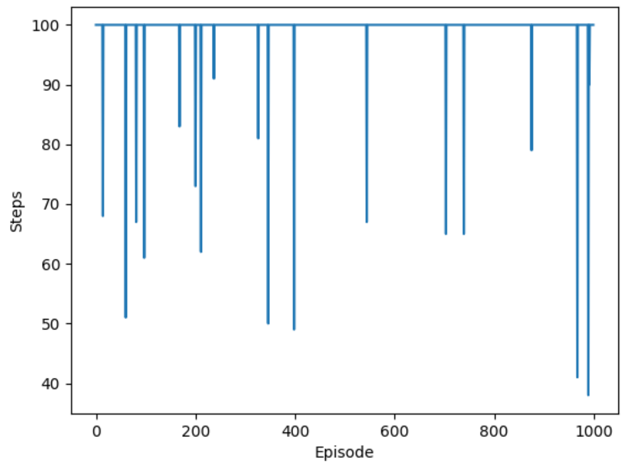


Figure 4

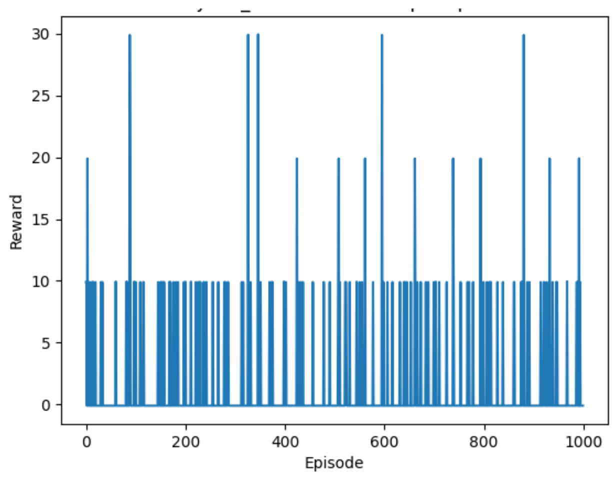


Figure 5

Figures 4 and 5 use our first feature vector, *Features 1.* As seen, this agent performed quite poorly. It only scored on rare occasions, and it was not as consistent with hitting the soccer ball as other feature vectors.

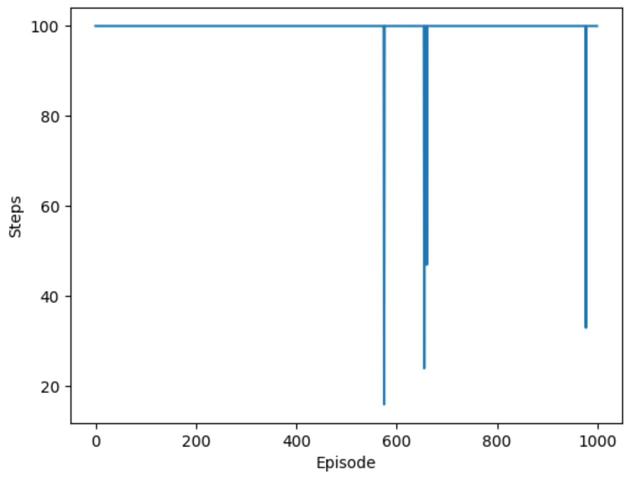


Figure 6: Steps per episode, epsilon .6 to .1

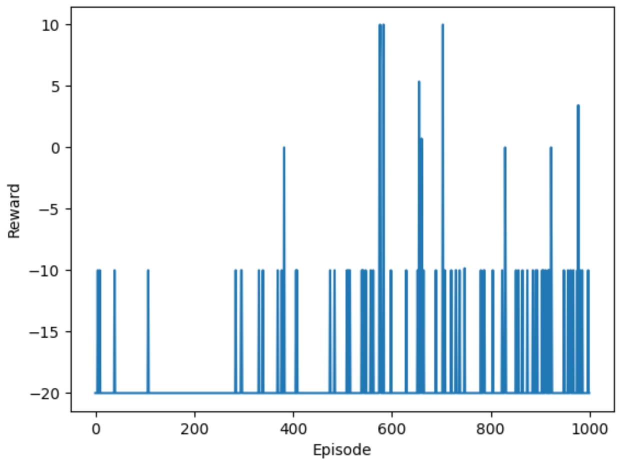


Figure 7: Rewards per episode, epsilon .6 to .1

Figures 6 and 7 use our second feature vector, *Features 2.* In these figures we demonstrate changing our epsilon from large to small, to slowly limiting the amount of random actions taken. This process is used in all of our algorithms, but only shown in some of our figures. Towards the end of the training session the results are very similar to *Features 1*.

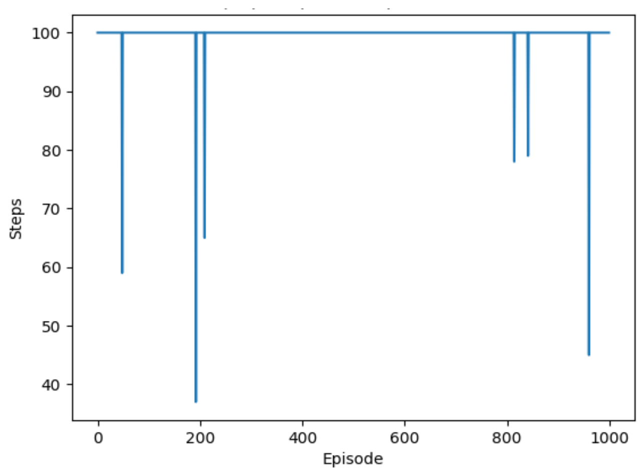


Figure 8: Steps per episode, epsilon .6 to .1

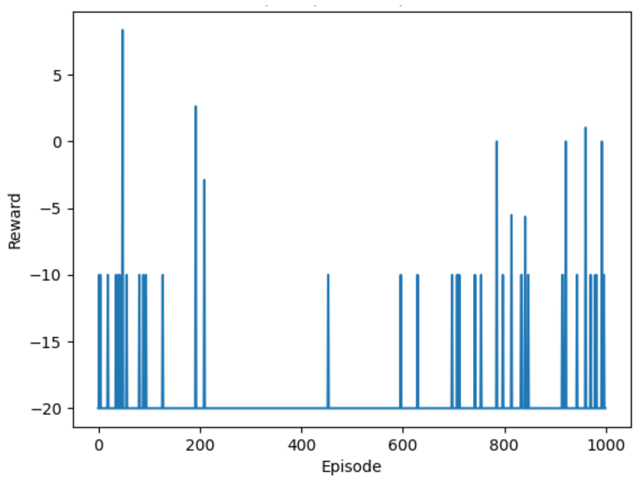


Figure 9: Rewards per episode, epsilon .6 to .1

Figures 8 and 9 use our third feature vector, *Features 3.* It still failed to follow the ball, and was performing even worse than *Features 2,* as it was even unable to hit the ball the first time.

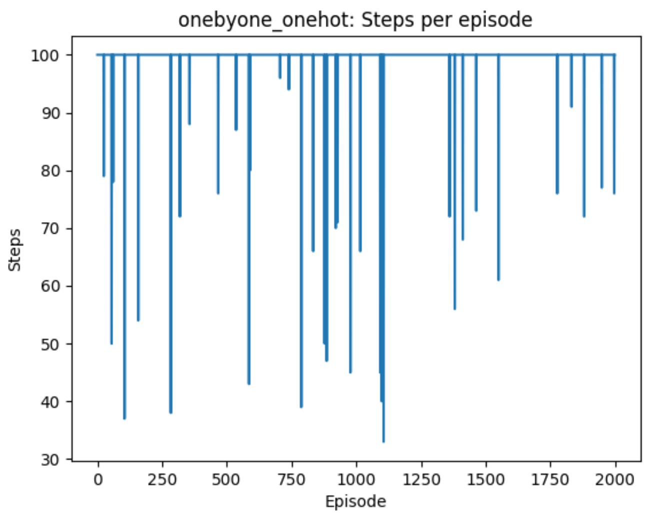


Figure 10

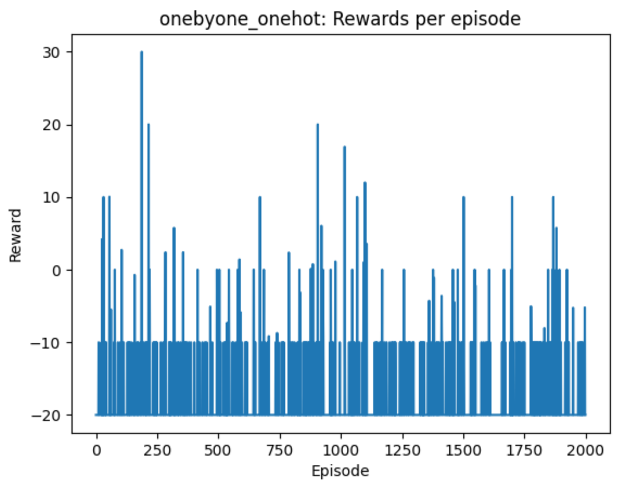


Figure 11

Figures 10 and 11 use our fourth feature vector, *Features 4*. This is our first implementation of the tileable solution. As seen, there was significant improvement when it came to hitting the ball the first time, however, the agent was still not as successful at learning to hit the ball a second time. Overall, the tileable solution provided much better learning.

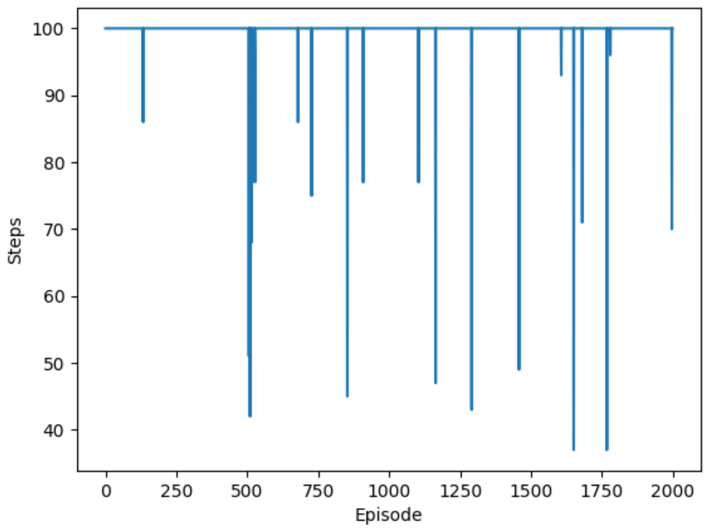


Figure 12

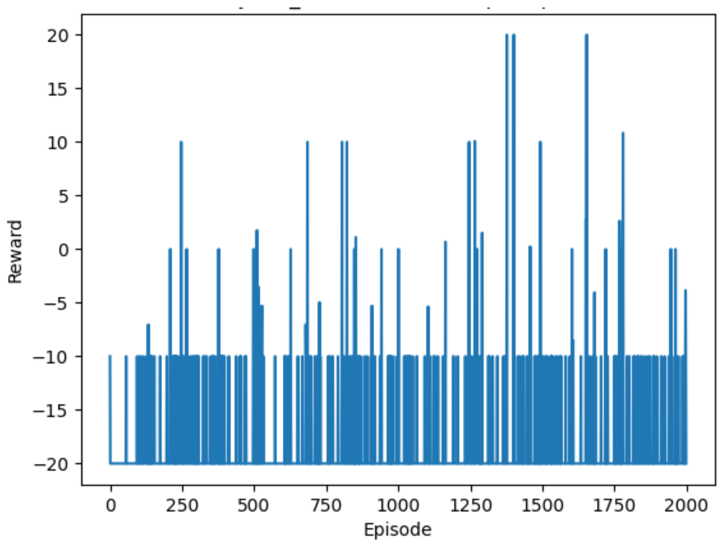


Figure 13

Figures 12 and 13 use our fifth features vector, *Features 5.* The results are similar to *Features 4*, with a very consistent first hit on the ball, but poor follow up on multiple hits. Goal scoring seemed to be slightly less than *Features 4*.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusions

Our goal for this project was to train an agent with our algorithm that could chase a soccer ball and successfully score it into a goal. Our end result is an agent that can successfully hit the ball. However, repeat hits of the ball are not as consistent. The agent has developed the knowledge to hit the ball from behind when going for multiple hits, but goal scoring still has room for improvement. Based on our testing and our research, which came mainly from "Reinforcement Learning for Robocup Soccer Keepaway" we believe our implementation of a tileable feature vector with Semi-Gradient SARSA is an optimal solution to this problem, as this performed significantly better than using X, Y, and Z positions as observations for a feature vector.

We were limited by the amount of time it took to train as well as interfacing with Unity. As we progressed along with the project, certain information we needed to access from the Unity agents were difficult to gain access to, and required guess and check work to see if we had accessed the proper observations. One example of this was the ray perception sensors. While we discarded the idea of using them very early into our process, the information they provided could have shaped and changed how we went about this problem. Secondly, when testing many different feature vectors, the time limitation was a concern.

5.2 Future Work

In the future, we would focus on improving the performance of our agent so that it is able to compete against another agent, possibly starting by comparing it against the Unity default agent and eventually testing it against a human player. One improvement could be made by doing overlapping tile coding of the player and ball, as was done in the Robocup Reinforcement Learning article. We divided up the field into discrete subsections, which the article described as the most straightforward way to go about breaking up the state space. They stressed, however, that a limitation of this was that "doing so can lead to over-generalization based on the fact that points in the same bin are required to have the same value and under-generalization due to the fact that points in different bins, no matter how close, have unrelated values. By overlaying multiple tilings it is possible to achieve quick generalization while maintaining the ability to learn fine distinctions" [1]. If we made the feature vector have a bin for every possible player and ball combination, we would have a feature vector of a size over 200,000. To avoid this, we could begin by making tiles a 2x2 unit size rather than our current 1x1, giving us a feature vector of just over 4,000.

Another possible solution is to use neural networks, which is what the Unity MLAgents package uses for their agents. Function approximation solves the problem in a linear representation, and it appears that this problem doesn't seem to be easily solved when done linearly. Using a neural network is likely more optimal.

Once we have trained a more successful single agent, we could then expand to training another agent on the same team, possibly referencing the keepaway strategy described in Peter Stone's paper to incorporate cooperation between agents.

ACKNOWLEDGEMENTS

The project contributors are [redacted] Samuel Dahlberg and [redacted]

[redacted]

- Researched API and communicated documentation to team members
- Wrote "proof of concept" code that interacted with the API and created a sample environment
- Set up the structure and outline of the final report, discussing notes the other members had taken on their findings, and incorporating these notes into a completed report

[redacted]

- Implemented the algorithm and trained the agent
- Created data visualizations that showcased the performance of the agent throughout episodes

Samuel Dahlberg

- Researched and navigated the Unity MLAgents package
- Worked with the Unity environment to create proper environments for our testing, writing custom observations for each of our feature vectors

The authors of this paper would also like to give credit to the software developers at Unity which created the Machine Learning Agents Toolkit and Python API. Without their software and documentation we would not have been able to set up this environment and train our agent.

REFERENCES

1. Stone, Peter, et al. “Reinforcement Learning for Robocup Soccer Keepaway.” *Adaptive Behavior*, vol. 13, no. 3, 2005, pp. 165–188., https://doi.org/10.1177/105971230501300301.
2. O'Neill, Michael. “Reinforcement Learning Tutorial: Semi-Gradient n-Step Sarsa and Sarsa Theory and Implementation.” *Michael ONeills ML Blog*, 14 Mar. 2018, https://michaeloneill.github.io/RL-tutorial.html.
3. Unity Technologies. “Python API Documentation.” *GitHub*, 13 Oct. 2021, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Python-API-Documentation.md#mlagents_envs.base_env.BaseEnv.step.
4. Miguel Alonso Jr. (2021) mlagents_envs/base_env.py [Source code]. https://github.com/Unity-Technologies/ml-agents/blob/5a25d782e9788166a3588e1c72632999a9f5b307/ml-agents-envs/mlagents_envs/base_env.py#L478
5. Unity Technologies. “ML-Agents Toolkit Overview.” *Github*, 15 Apr. 2021, https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md.

APPENDICES

Attached is a zip file containing the Python code for the algorithm and its feature vectors. If you would like to run an example of our algorithm, we can provide links to our github, along with instructions on how to properly set it up.