Reinforcement Learning: Solving Two Player Games

Pratim Chowdhary
Dartmouth College
pratim.chowdhary.25@dartmouth.edu

Hardik Gupta
Thayer School of Engineering
hardik.gupta.th@dartmouth.edu

Sarah Nam
Dartmouth College
sarah.j.nam.th@dartmouth.edu

Abstract

This paper investigates the application of reinforcement learning (RL) techniques in solving two-person zero-sum games, a class of adversarial scenarios with profound implications across various domains. The study focuses on the inherent challenges posed by the dynamic and strategic nature of such games, where the success of one player is directly tied to the losses of the other. Our work delves into the exploration-exploitation tradeoffs, sample inefficiency, and convergence concerns inherent in RL algorithms within the context of these competitive interactions. The core of this paper involves the implementation and evaluation of various RL algorithms in diverse zero-sum game settings within the scope of a course project.

1 Background

Reinforcement learning (RL) has gained significant attention and application in solving non-zero-sum games due to its ability to address complex and dynamic decision-making problems. Non-zero-sum games involve scenarios where the gains and losses of the involved players are not equal, and the success of one player does not necessarily result in an equivalent loss for another. This type of game is prevalent in various real-world situations, such as economic competition, resource allocation, and strategic interactions.

The conventional approach to solving non-zero-sum games often relies on game theory, where Nash equilibrium solutions are sought. However, in dynamic and uncertain environments, finding optimal strategies becomes a challenging task. This is where reinforcement learning comes into play.

By combining principles from game theory, such as Nash equilibrium solutions, with the adaptability and learning capabilities of reinforcement learning, we aim to develop effective strategies for navigating and solving non-zero-sum games in a wide range of applications. This approach has been applied to areas such as multi-agent systems, economics, and autonomous decision-making in dynamic environments.

2 Literature Review

We first began this project by delving into the realm of non-zero-sum game output regulation within a continuous-time framework, specifically targeting a class of multi-player linear uncertain systems.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

We found that the primary objective for most of the previous work was to devise feedback control strategies for individual players, ensuring that the system output closely follows a predetermined reference signal, withstands disturbances, and concurrently enhances each player's distinct performance objective[1, 2, 3, 4]. There have been many attempts to tackle this challenge through various methods such as a data-driven adaptive optimum control technique [3]. This approach integrates reinforcement learning (RL), differential game theory, and output regulation methodologies. Notably, the Nash equilibrium solution, comprising both feedback and feed-forward control gains tailored to each player, is approximated in real-time through the utilization of empirical data.

We found this novel approach to be quite appealing and decided to model our own attempt of using reinforcement learning after this.

2.1 Reinforcement Learning

Reinforcement learning is a machine learning paradigm inspired by behavioral psychology, focusing on how agents can learn to make decisions by interacting with an environment. RL provides a flexible framework for modeling and solving non-zero-sum games by allowing agents to learn and adapt their strategies over time based on feedback from the environment. Here are some key reasons why RL is well-suited for addressing non-zero-sum games:

2.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture designed to overcome the limitations of traditional RNNs in capturing and remembering long-range dependencies in sequential data. Developed to address the vanishing gradient problem, which hinders the training of RNNs on sequences with long-term dependencies, LSTMs have proven effective in various applications involving sequential data, such as natural language processing, speech recognition, and time-series analysis.

2.3 Non-Linear Programming

Nonlinear programming (NLP) is a mathematical optimization technique used to solve problems where the objective function or the constraints are nonlinear. In optimization, the goal is to find the values of decision variables that minimize or maximize an objective function while satisfying a set of constraints. It provides a powerful framework for optimizing complex systems where the relationships between variables are nonlinear, allowing for a more realistic representation of real-world scenarios compared to linear programming.

3 Implementation

3.1 Agent v. Linear Solver

We started by configuring the game in OpenAI Gym, a tool for creating and comparing machine learning algorithms. Consider our game to be a strategic playground where players make decisions using mixed strategies—basically, a mix of different moves with some randomness.

A linear model, a simplified representation of a player's decision-making process, was one of our contenders. Based on the payoffs it received during the game, it learned and adjusted its strategies. On the other hand, we had the simplex algorithm, a well-known linear programming tool. This algorithm seeks the best strategy that produces the highest or lowest results while taking into account specific rules.

We observed how well these contenders performed by putting them to the test in the OpenAI Gym environment. We were curious about how quickly they could develop good strategies, how accurate those strategies were, and how well they adapted to different game situations.

Algorithm 1 OnePayoffMatrix Environment

Initialization:
  Input: Payoff matrix $payoff\_matrix$
  $action\_space \leftarrow Discrete(payoff\_matrix.shape[0])$
  $observation\_space \leftarrow Discrete(1)$
  $bounds \leftarrow [(0,1)] \times payoff\_matrix.shape[1]$
  $linear\_constraint \leftarrow LinearConstraint(\mathbf{1}, [1], [1])$
  Call: RESET
procedure RESET
  return GETSTATE()
end procedure
procedure GETSTATE
  return [0]
end procedure
procedure CALCULATEREWARD($agent\_distribution, agent\_actions$)
  $other\_agent\_action\_space \leftarrow payoff\_matrix.shape[1]$
  procedure EXPECTEDREWARD($action\_dist, opponent\_distribution, sign$)
    $exp \leftarrow 0$
    for i **in range** $action\_space.n$ **do**
      $exp \mathrel{+}= opponent\_distribution[i] \times \sum_{j=1}^{other\_agent\_action\_space} action\_dist[j] \times sign \times payoff\_matrix[i][j]$
    end for
    return $-exp$
  end procedure
  $res \leftarrow$ MINIMIZE $\left(\text{EXPECTEDREWARD}, \frac{1}{other\_agent\_action\_space}, args = (agent\_distribution, -1), bounds = bounds\right.$
  $opt\_actions \leftarrow$ RANDOMCHOICE $(payoff\_matrix.shape[1], len(agent\_actions), p = res.x)$
  $reward \leftarrow 0$
  for i **in range** $len(agent\_actions)$ **do**
    $reward \mathrel{+}= payoff\_matrix[agent\_actions[i]][opt\_actions[i]]$
  end for
  return $reward$
end procedure
procedure STEP($agent\_distribution, agent\_actions$)
  $rewards \leftarrow$ CALCULATEREWARD($agent\_distribution, agent\_actions$)
  return GETSTATE(), $rewards$, **True**, [0]
end procedure

Figure 1: Environment Setup

Data: $agent, env, target\_dist, num\_episodes = 100000, log\_interval = 1000, tolerance = 1e-3, round\_size$
Result: Converged agent distribution
for $episode \leftarrow 1$ **to** $num\_episodes$ **do**
  $state \leftarrow env.reset()$;
  $done \leftarrow False$;
  while $\neg done$ **do**
    $agent\_distribution \leftarrow agent.get\_agent\_distribution(state)$;
    $agent\_actions \leftarrow$
     $np.random.choice(env.payoff\_matrix.shape[0], round\_size, p = agent\_distribution)$;
    $states \leftarrow [env.get\_state()] \times round\_size$;
    $\_, reward, done, \_ \leftarrow env.step(agent\_distribution, agent\_actions)$;
    $\_ \leftarrow agent.update\_model(states, agent\_actions, reward)$;
  end
  $distance \leftarrow$
   $np.linalg.norm(target\_dist - np.array(agent\_distribution))$;
  if $distance \leq 0.001$ **then**
    break;
  end
  if $episodelog\_interval = 0$ **then**
    $print("Episode", episode)$;
    $print('Agentdistribution :', agent\_distribution)$;
    $print("Distance :", distance)$;
  end
end

Algorithm 1: Training Algorithm

Figure 2: Training Algorithm

3.2 Agent v. Agent

The goal was for two agents to engage in a strategic duel, with each attempting to discover the most effective mixed strategies in the context of a two-person game.

We used the OpenAI Gym environment to lay the groundwork. This enabled us to create a dynamic and interactive environment in which our agents could learn and adapt their strategies. The core of our approach was to build an LSTM brain model for each agent. LSTMs are a type of recurrent neural network that excels at detecting sequential patterns. In our case, they were critical in learning and predicting the best mixed strategies throughout the game.

A unique twist was incorporated into the training process. As a critical component, we introduced the negative log of the loss multiplied by the reward. This novel approach sought to align the model's learning objectives with the ultimate goal of maximizing cumulative rewards. The combination of negative log loss and reward provided a complex mechanism for steering agents toward optimal strategies.

As the agents played, their LSTM brain models evolved through a series of interactions, refining their understanding of game dynamics and adapting their mixed strategies as needed. The negative log loss multiplied by the reward functioned as a compass, directing the models toward strategies that not only minimized loss but also maximized overall reward.

Algorithm 1: TwoPayOffMatrix Class

Data: payoff_matrix, round_size=10
Result: TwoPayOffMatrix instance

TwoPayOffMatrix **Data:** payoff_matrix, action_space, observation_space, round_size
Input: `payoff_matrix, round_size=10`

`// Constructor`
$_{-i}init_{payoff\_matrix,round\_size=10}self.payoff\_matrix \leftarrow$ payoff_matrix;
self.action_space ← None;
self.observation_space ← Discrete(1);
self.round_size ← round_size;
self.reset();

`// Get action space for a given agent`
get_action_spaceagent_idx **return** shape of self.payoff_matrix at index agent_idx;

`// Reset the environment`
reset **return** get_state();

`// Get the initial state`
get_state **return** [0];

`// Calculate rewards based on agent actions`
calculate_rewardagent_1_actions, agent_2_actions rewards ← [0, 0];
foreach *action_1, action_2* ***in*** *zip(agent_1_actions, agent_2_actions)* **do**
    rewards[0] += self.payoff_matrix[action_1][action_2];
    rewards[1] += -self.payoff_matrix[action_1][action_2];
rewards[0] /= length of agent_1_actions;
rewards[1] /= length of agent_1_actions;
return rewards;

`// Take a step in the environment`
stepagent_actions rewards ← calculate_reward(agent_actions[0], agent_actions[1]);
return get_state(), rewards, True, [0];

Figure 3: Environment Setup

Algorithm 1: Training Algorithm

```
Data: agent1, agent2, env, target_dist1, target_dist2,
      num_episodes=1000000, log_interval=1000, round_size=25
for episode ← 1 to num_episodes do
    state ← env.reset();
    done ← False;
    while not done do
        agent_1_distribution ← agent1.get_agent_distribution(state);
        agent_2_distribution ← agent2.get_agent_distribution(state);
        agent_1_actions ← np.random.choice(env.get_action_space(0),
         round_size, p=agent_1_distribution);
        agent_2_actions ← np.random.choice(env.get_action_space(1),
         round_size, p=agent_2_distribution);
        _, reward, done, _ ← env.step([agent_1_actions, agent_2_actions]);
        states ← [env.get_state()] × round_size;
        agent1_loss ← agent1.update_model(states, agent_1_actions,
         reward[0]);
        agent2_loss ← agent2.update_model(states, agent_2_actions,
         reward[1]);
        distance1 ← np.linalg.norm(target_dist1 -
         np.array(agent_1_distribution));
        distance2 ← np.linalg.norm(target_dist2 -
         np.array(agent_2_distribution));
    end
    if episode % log_interval == 0 then
        print("Episode", episode);
        print('Agent1 Dist:', agent_1_distribution, agent1_loss,
         reward[0]);
        print('Agent2 Dist:', agent_2_distribution, agent2_loss,
         reward[1]);
    end
end
plt.show();
```

Figure 4: Training Algorithm

4 Results

4.1 Rock, Paper, Scissors

The reinforcement learning agents exhibited a capacity to converge to a stable Nash equilibrium in the Rock, Paper, Scissors game. Over multiple training iterations, the agents learned to balance exploration and exploitation, leading to strategies that approximated optimal responses to opponent actions.

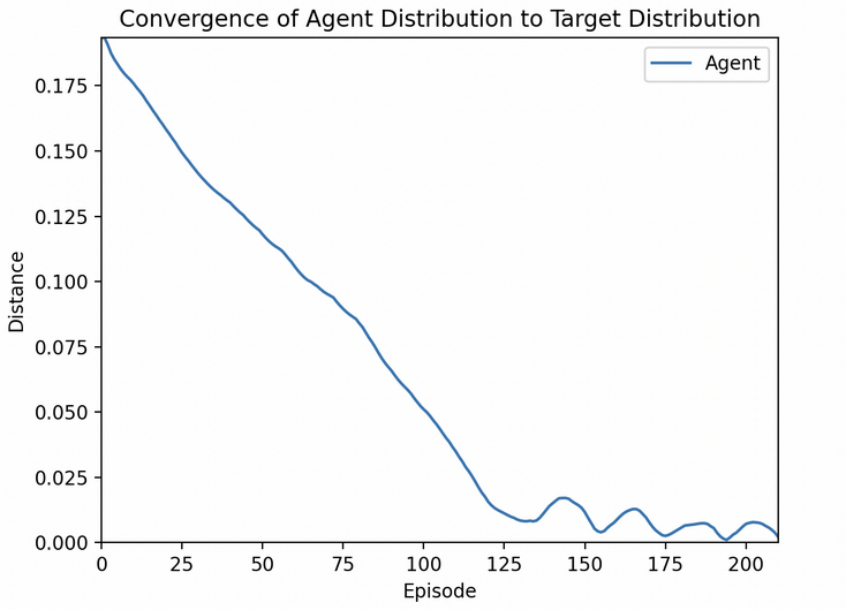

Figure 5: Agent v. Linear Solver

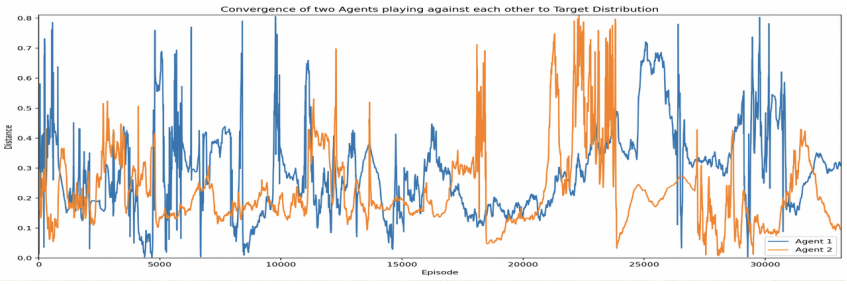

Figure 6: Agent v. Agent

4.2 Colonel Blotto

In the context of the Colonel Blotto scenario, reinforcement learning agents were able to successfully learn optimal resource allocation strategies across multiple battlefields. The agents exhibited a capacity to balance offensive and defensive priorities, converging to solutions that approximated Nash equilibria for the given game.

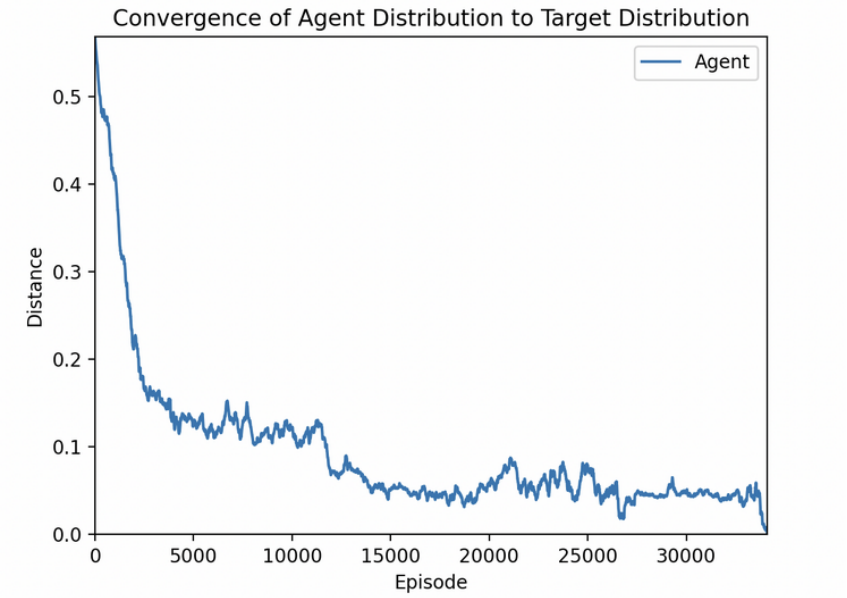

Figure 7: Agent v. Linear Solver

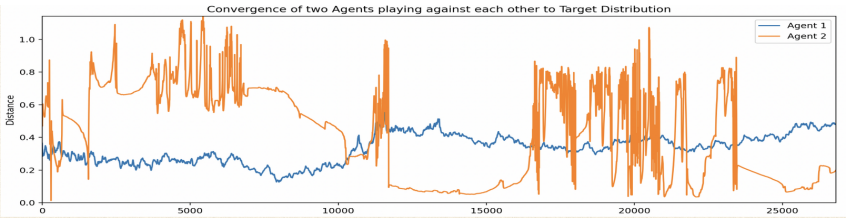

Figure 8: Agent v. Agent

4.3 Random Payoff Matrix

We found that the model is also able to converge when given a random payoff matrix. We are able to verify the convergence by also providing a sample optimal strategy to the model.

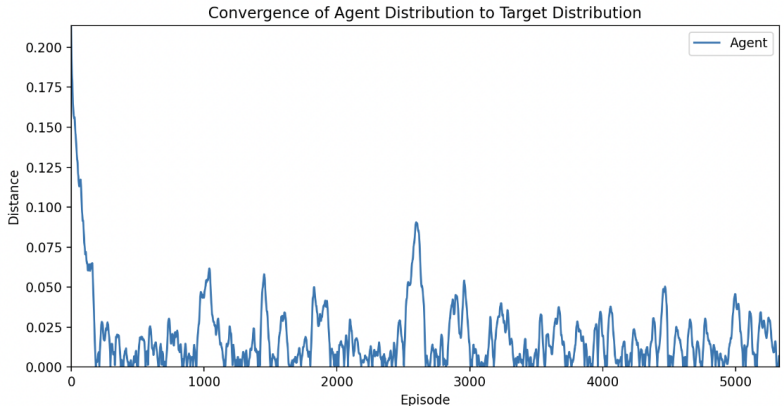

Figure 9: Convergence

5 Challenges and Limitations

5.1 Non-Convexity

RL problems often involve non-convex optimization, and this challenge is exacerbated in two-person zero-sum games. The non-convexity arises from the complex interaction between the agents, making it difficult to find a global optimum. Training algorithms might converge to local minima, affecting the stability and performance of the learned policies.

5.2 Dynamic Environments

Two-person zero-sum games often occur in dynamic and evolving environments where the optimal strategy can change over time. Adapting to these changes and ensuring that the learned policies remain effective requires continuous exploration and updating. Handling dynamic environments adds complexity to the training process, as the agents must adjust their strategies in response to the opponent's actions.

5.3 Computational Complexity

Training RL agents in two-person zero-sum games can be computationally demanding. The agents need to explore a large state-action space to learn effective strategies, and the optimization process can be computationally expensive. As the complexity of the game increases, training times and resource requirements grow, making it challenging to scale RL approaches for large and complex games.

5.4 Stateless System

Striking the right balance between exploration and exploitation is crucial for effective learning. In stateless games where the only changes are due to the opponent's strategy, exploration is limited, and the learning process becomes less effective. Especially as the opponent's strategy is typically unknown, agents must explore various actions to discover the best responses. However, too much exploration hinders the learning process. Balancing exploration and exploitation to find optimal policies is a delicate challenge.

References

[1] Bian, T. & Jiang, Z.P. (2018) Stochastic and adaptive optimal control of uncertain interconnected systems: A data-driven approach, *Systems & Control Letters* **115**: 48-54

[2] Jiane, Y., Fan, J., Gao, W., Chai, T. & Lewis, F. (2020) Cooperative adaptive optimal output regulation of nonlinear discrete-time multi-agent systems, *Automatica* **121**: 109149

[3] Odekunle, A., Gao, W., Davari, M., & Jiang, Z.P. (2020) Reinforcement learning and non-zero-sum game output regulation for multi-player linear uncertain systems, *Automatica* **112**: 108672

[4] Yaghmaie, F.A., Gunnarsson, S. & Lewis, F. (2019) Output regulation of unknown linear systems using average cost reinforcement learning, *Automatica* **110**: 108549

6 Appendix

6.1 Rock, Paper, Scissors

6.1.1 Single Agent

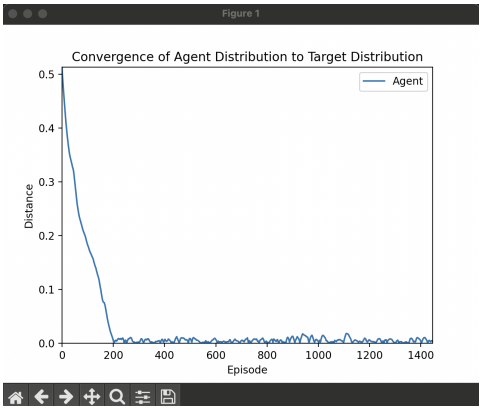

Figure 10: Learning Rate = 0.01

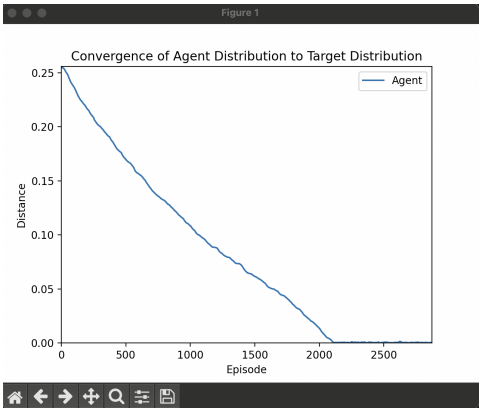

Figure 11: Learning Rate = 0.001

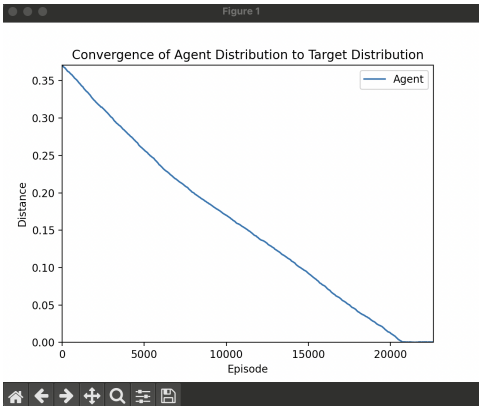

Figure 12: Learning Rate = 0.0001

6.1.2 Using LSTM

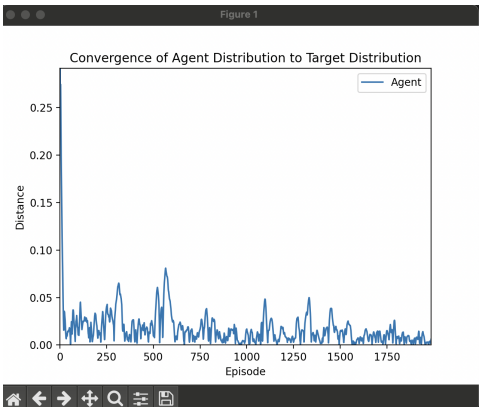

Figure 13: Learning Rate = 0.01

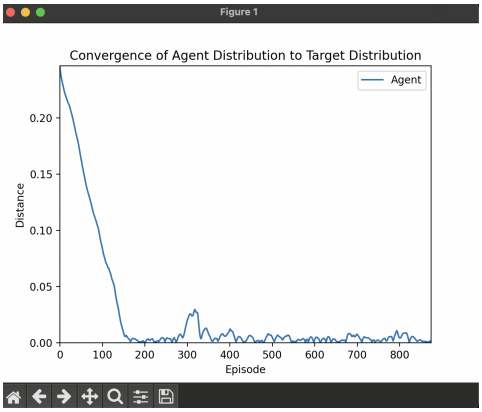

Figure 14: Learning Rate = 0.001

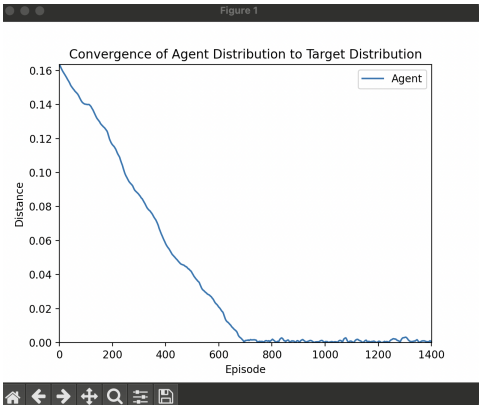

Figure 15: Learning Rate = 0.0001

6.1.3 2 Agents - Linear

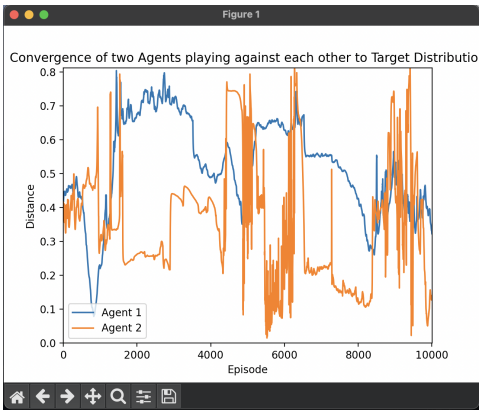

Figure 16: Learning Rate = 0.01

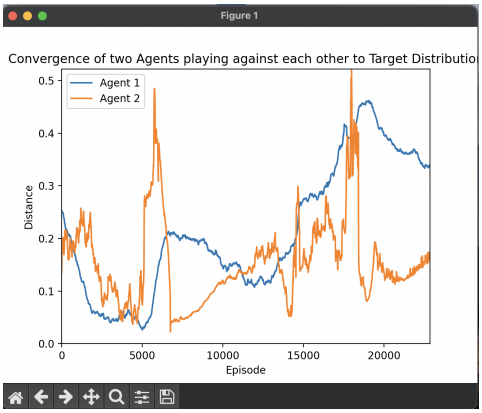

Figure 17: Learning Rate = 0.001

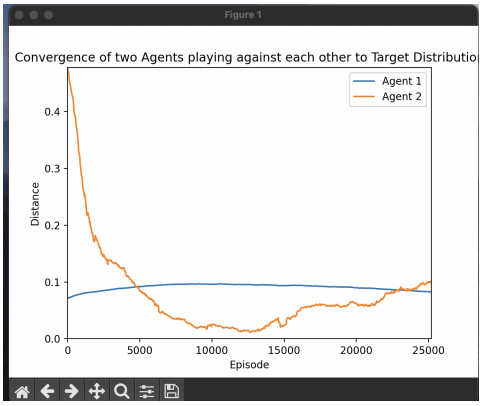

Figure 18: Learning Rate = 0.0001

6.2 Colonel Blotto

6.2.1 Single Agent

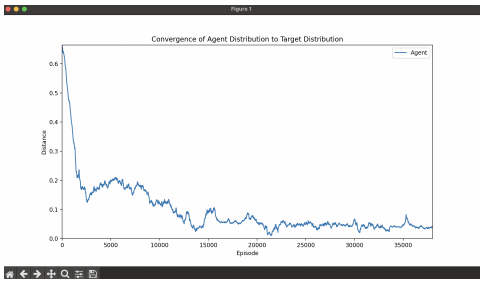

Figure 19: Learning Rate = 0.01

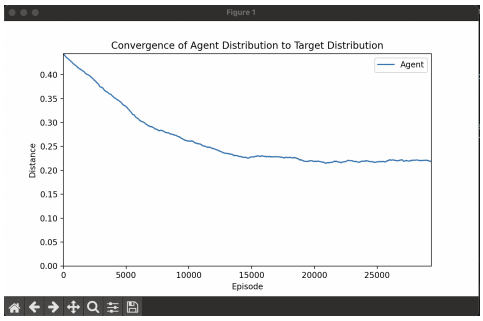

Figure 20: Learning Rate = 0.001

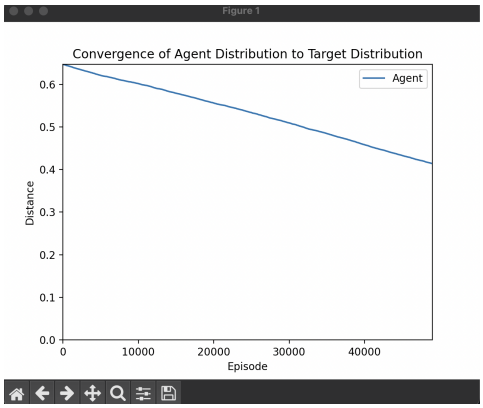

Figure 21: Learning Rate = 0.0001

6.2.2 Using LSTM

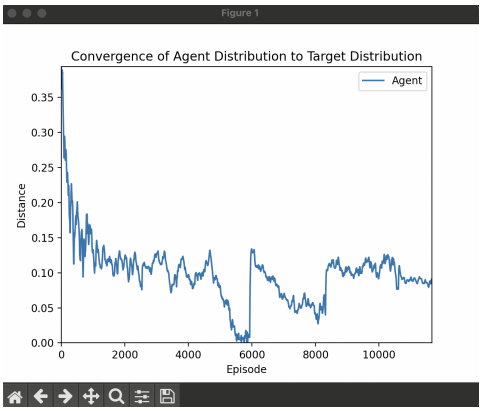

Figure 22: Learning Rate = 0.01

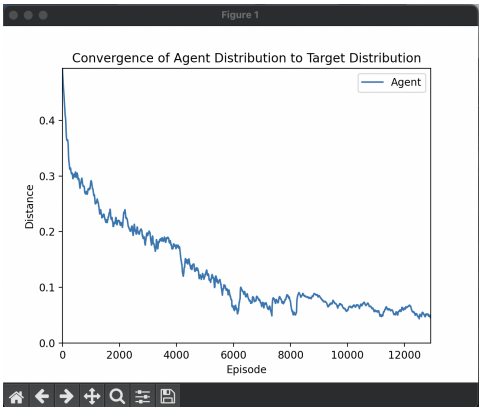

Figure 23: Learning Rate = 0.001

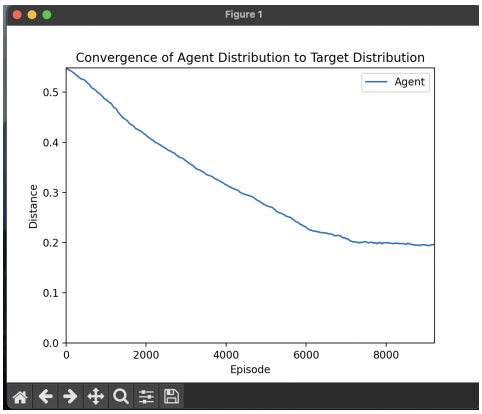

Figure 24: Learning Rate = 0.0001

6.2.3 2 Agents - Linear

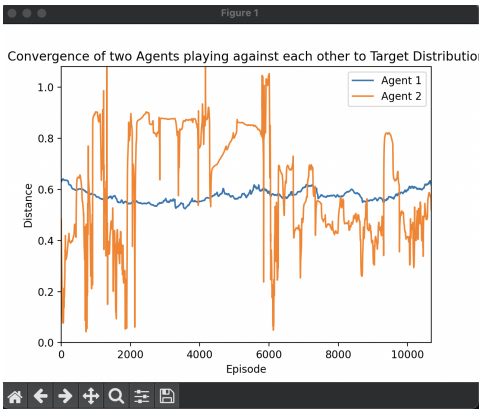

Figure 25: Learning Rate = 0.01

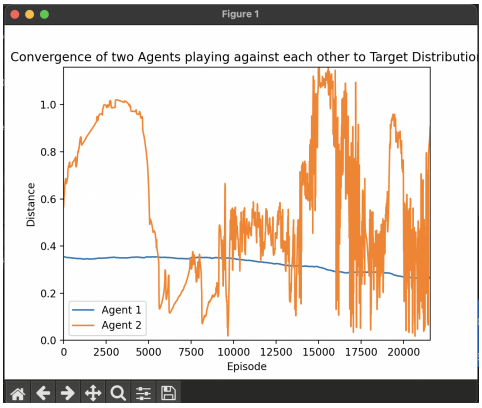

Figure 26: Learning Rate = 0.001

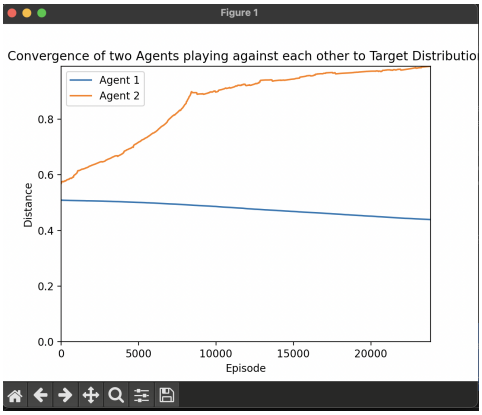

Figure 27: Learning Rate = 0.0001