PokAImon: Competitive Multi-Agent RL

Cole Sohn
Department of Computer Science
Stanford University
csohn@stanford.edu

Rita Tlemcani
Department of Computer Science
Stanford University
ritabt@stanford.edu

1. Introduction

Reinforcement Learning (RL) for competitive tasks is interesting because agents trained with relatively simple reward functions can exhibit complex emergent behavior that would otherwise require detailed environments or meticulously crafted reward functions. Emergent complex behavior is essential for many real-world RL applications such as autonomous vehicles, robotic manipulation, and enemy AI in games. Competitive RL agents have been shown to exhibit emergent behavior without the need for these hand-tuned steps.

For this project, we will experiment with competitive AI by training autonomous agents to compete in a sumo wrestling game to push the other off of a platform, with the goal of achieving good performance as a video game enemy AI. To demonstrate the ability of these competitive agents to perform in real-world scenarios, we will use simulated visual sensors as input features. Through competitive RL for a sumo task, agents will learn emergent sub-tasks such as locomotion, posture and stance, and strategy.

Our implementation can be found at `https://github.com/ritabt/PokAImon`.

2. Problem Statement

Physics-based 3D agents will be trained to compete in a competitive sumo wresting game. Agents can make observations about their environment as input data. Observations include position, rotation, velocity, and angular velocities of each limb, acceleration and angular acceleration of the body, and global position of the body. We will experiment and compare performance with various features to locate the agent's rival such as relative position, ray-casts, and visual imaging. Agents will learn policies that output target limb rotations in order to maximize reward by being the last agent on a platform. Agents are trained with self-play, meaning they compete against agents with previous policy iterations to simulate practice with varying skill levels and ensure no agent's opponent dominates.

We will quantitatively evaluate our agent's performance

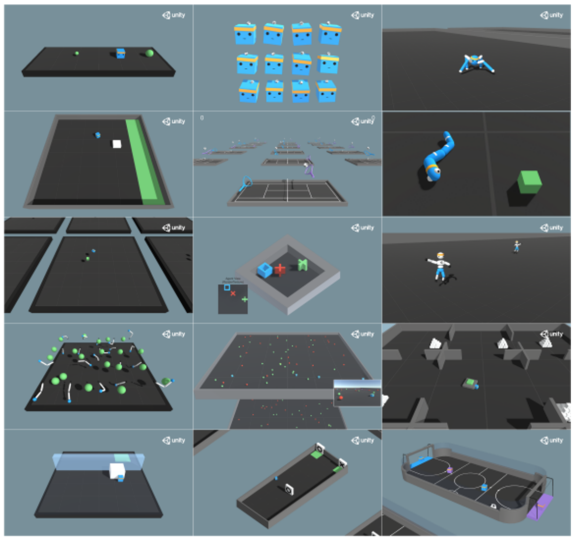
Figure 1. Example of environments available in Unity for Deep RL training - Image by Juliani et al. [7]

through reward-per-episode. We expect this to increase initially as agents learn dense rewards such as locomotion and remaining upright - followed by oscillations around the midpoint between winning and losing rewards as agents learn new strategies against previous policies. We will also evaluate performance of our agents following training through competition with previous policies of the same agent - expecting higher win rates on earlier policy iterations. We can also compare agents of different visual input features, hyper-parameters, and morphologies through direct competition and win-rates.

We will evaluate our agents qualitatively by inspecting emergent behaviors such as locomotion, stance, and strategies (such as trickery and building up velocity). We will also evaluate our agents qualitatively and inspect emergent behavior by piloting user controlled agents and observing trained agent reactions to user input.

2.1. Literature Review

We are using the Unity ML-Agents Toolkit developed and introduced by Juliani et al.[7] as an environment for our project. This Unity package offers a user-friendly API for simulated deep RL training. The API allows us define different types of neural networks architecture, different simulated environment and agents, as well as different training methods. The paper by Juliani et al.[7] introduced several example environments, see Fig.1, as well as a detailed study comparing different training and design strategies. Juliani et al. found that a PPO[8] trainer paired with an LSTM[6] network architecture performed best for training in 11 out of the 15 studied environments.

While the work by Juliani et al. gave us the tools and the background to work on our project, most of the strategies we are using are inspired by the work of Bansal et al.[2] from OpenAI. Bansal at al. did an extensive study of deep RL training of different agents in different simulated environments. They used the MuJoCo[9] simulator, which is a popular simulator for AI research but because of the limited visual rendering capabilities of the engine and the lack of complex lighting, textures, and shaders[7], we decided to use Unity as our simulator. Moreover, MuJoCo models are compiled which makes it impossible for us to set up a game-like environment where a human can fight against the AI.

Bansal et al.[2] also found that an LSTM[6] paired with PPO[8] worked best for many of the tested environments. One of the novel ideas they introduced is the concept of curriculum training. The idea is that it is very difficult to train a complex agent, such as a humanoid, to play a game like ”You shall not pass” where one agent tries to block the other agent from reaching a goal. In curriculum training, the learning is done in steps, first the agent learns how to walk and stand up, then the agent learns how to compete with another agent. This is done by using the same network in different setups as well as using a reward function that combines dense and sparse rewards. Dense rewards are calculated at every timestep and sparse rewards are calculated at the end of the episode. While dense rewards offer more detailed feedback to the agent and speed up the training, it is computationally expensive. Combining the two is achieved using a linear annealing factor α. So, at time-step t, if the exploration reward is s_t, the competition reward is R and T is the termination time-step, then the reward [2] is defined in Eq.1 below.

$$r_t = \alpha_t s_t + (1 - \alpha_t)\mathbb{1}[t == T]R \tag{1}$$

Another novel and interesting idea introduced by Bansal et al.[2] is the concept of distributed large scale self-play RL training. The idea here is to initialize many instances of one environment where each instance has two agents of the same type competing against each other. All of these instances are used to update the same network. This approach dramatically increases the learning speed by creating a batch of rollouts that can be used to update the network. Another main idea introduced with this concept is opponent sampling. With opponent sampling, the latest version of the trained network will randomly compete against an older version of the network. This prevents one agent type/role to overpower the competitor and can unstick training in some scenarios. Also comparing an older version of the network to the most recent one is useful to evaluate the training.

Another influential paper in our work is the paper by Hester et al.[4] from DeepMind where they introduced Deep Q-Learning from demonstrations. The idea is that if we provide a RL training agent a human-made demonstration of what a ”good” performance is, this can speed up the training by a lot and could even allow us to skip the curriculum training approach by Bansal et al.[2]. Hester et al. found that training a Deep Q-Network (DQN) from a demonstration first speeds up the training by at least 83 million steps (number of steps it took a regular DQN to catch up). Although we do not plan on using a DQN, we would like to experiment with this strategy in the PPO+LSTM approach. Juliani et al.[7] explored a very similar concept in their work and called it imitation learning support for self-play. Although Juliani et al. didn't extensively study the impact of incorporating human demonstrations in training, the tools they provide can be customized for this purpose.

2.2. Dataset

As this is a reinforcement learning problem, data is generated from interactions with the simulated environment. More specifically, as the competing agents take actions in the simulated world, this results in states and we use the action-state pairs as training inputs for our model. See problem statement above for a full list of input features.

3. Technical Approach

3.1. Baseline

First, we train a toy example with a roller-ball agent which can apply force in two directions to self-compete in a sumo environment. A visual of the setup can be seen in Fig.2. We sample previous policy iterations to be opponents following the work of Bansal et al.[2] who showed how one agent will overpower the other if always using the latest policy when training both. Following their methods, we update agent policy with PPO [8], and a simple sparse reward function with win, lose, and stalemate states defined in Eq.2 below.

$$R(s) = \begin{cases} 1\text{ , if } s = \text{won} \\ -1\text{, if } s = \text{lost} \\ -1\text{, if } s = \text{stalemate} \end{cases} \tag{2}$$

The observation/state vector used for training is defined in Eq.3 below.

$$S = \begin{bmatrix} t_{\text{left}} \\ d_{\text{self2edge}} \\ d_{\text{self2opponent}} \\ Vx_{\text{self}} \\ Vy_{\text{self}} \\ Vx_{\text{opponent}} \\ Vy_{\text{opponent}} \end{bmatrix} \tag{3}$$

where t_{left} is the time left in the episode (max is $20s$), $d_{\text{self2edge}}$ is the radial distance between the agent and the edge, $d_{\text{self2opponent}}$ is the relative distance of the opponent to the agent, Vx_{self} and Vy_{self} are the x and y axes velocities of the agent (no movement along z-axis), and Vx_{opponent} and Vy_{opponent} are the x and y axes velocities of the opponent.

3.2. Next Steps

To extend the baseline [2], we will train with more complex agents such as crawlers to compete against agents with the same morphology. An example of a crawler agent from the Unit-ML Toolkit [7] can be seen in Fig.3

We will use the more complex dense reward function defined by Bansal et al. shown above in Eq.1. This allows our agent to learn locomotion and remain standing. We will also experiment with a 2-layer feed-forward architecture as well as an LSTM[6] model architecture. We will evaluate these methods by training agents in parallel with both model architectures and plotting win rate per epoch.

We will investigate potential imitation learning approaches such as GAIL [5] and DQL from demonstration[4] to improve performance against human-piloted agents.

As an additional stretch extension, we will generalize our agents to multiple opponent types through random opponent type sampling during training. We will also evaluate performance when competing with a human piloted agent and note any interesting emergent behavior.

3.3. Novelty

To clarify the novelty parts of our project, we itemized our unique contributions below:

- Use Nintendo 3DS Pokemon X/Y models[1] for training in a custom Unity environment, see Fig2. These are open-source rigged 3D models with different morphologies and body types. These agents are more complex and more interesting than the models used by our reference papers [7][2].

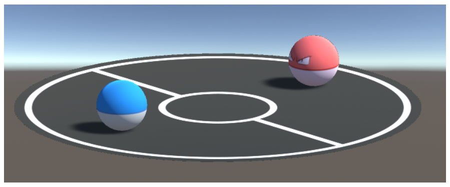
Figure 2. Illustration of Sumo Ball Agents

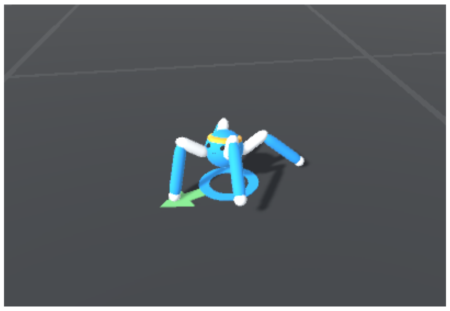
Figure 3. Example of 4-limbed crawler agent from Unity ML toolkit [7]

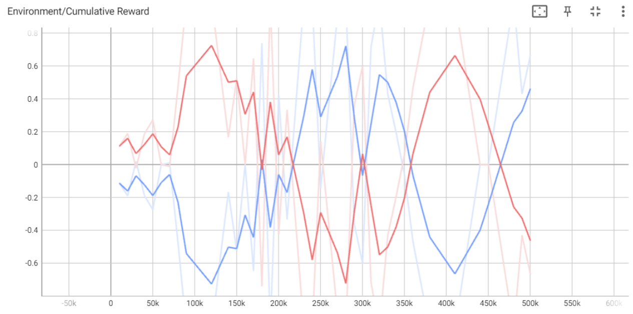

Figure 4. Reward per iteration for sumo ball agents

- Combine PPO training, LSTM network architecture, and learning from demonstration for competitive RL: Bansal et al.[2] found that PPO+LSTM worked best in many training scenarios and Hester et al.[4] found that human demonstrations (similar to imitation learning set up) can dramatically speed up RL training. We haven't seen any paper study this combination in competitive RL. Juliani et al[7] mentioned this concept in the context of reaching a goal (very simple environment and goal) but no quantitative study has been done. Bui et al.[3] studied a similar combination in MuJoCo but without the use of LSTM layers.

4. Results and Evaluation

We train a competitive roller-ball agent with the sumo training environment seen in Fig.2. We plot the average reward per episode in Fig.4. Interestingly, we see a periodic trend in our reward function. This roughly corresponds to one agent learning a new strategy to beat the opponent's

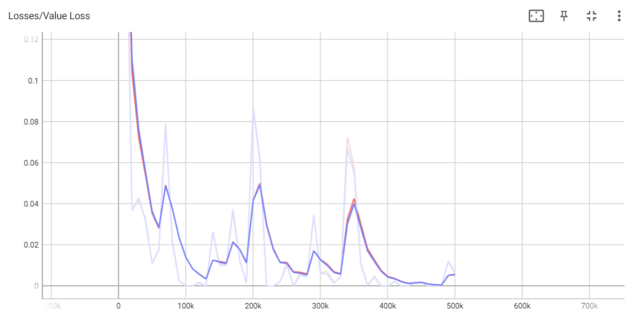

Figure 5. Value loss for sumo ball example

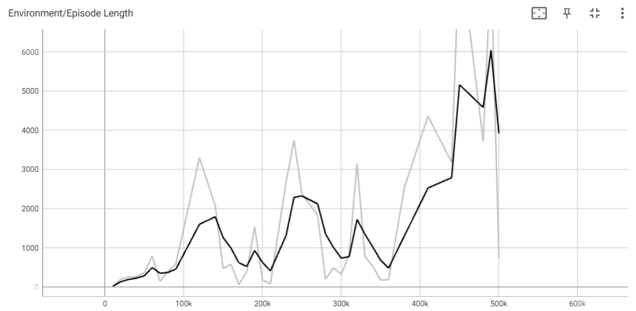

Figure 6. Episode length for sumo ball training example

policy. As the other agent catches up, the reward for the first agent falls. Episode lengths also increase as agents learn more complex behaviors. See Fig.6.

We observed many interesting emergent behaviors for such a simple agent and reward function. The first apparent strategy seen was for agents to ram into each other after building up momentum to push the other off the platform with more force. This resulted in agents spiraling around each other to be the first to push the other off the platform with more force. Another interesting behavior that occured was one agent tricking the other. The agent would stay still at the edge of the platform, and quickly move out of the way as the other agent attempted to collide.

5. Contributions

Cole and Rita worked on all parts of this project together at the same time.

References

[1] The models resource - pokemon x/y. Available at `https://www.models-resource.com/3ds/pokemonxy/`.

[2] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition, 2018.

[3] T. V. Bui, T. Mai, and T. H. Nguyen. Imitating opponent to win: Adversarial policy imitation learning in two-player competitive games, 2022.

[4] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys. Deep q-learning from demonstrations, 2017.

[5] J. Ho and S. Ermon. Generative adversarial imitation learning. *CoRR*, abs/1606.03476, 2016.

[6] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange. Unity: A general platform for intelligent agents, 2020.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017.

[9] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.