# PokAImon: Multi-Agent Competitive RL with Vision

Cole Sohn
Stanford University
csohn@stanford.edu

## Abstract

*We study multi-agent competitive Reinforcement Learning trained with vision inputs. We train Pokemon agents in a Unity Environment for the Sumo task. We experiment with various network architectures and training strategies. We show that a transfer learning approach utilizing Resnet50 as a preprocessing step outperforms other vision strategies and performs comparably to a detailed feature input.*

## 1. Introduction

Traditionally, behavior of video game non-player characters (NPCs) is hand-coded through heuristics, behavior trees, and decision graphs. Agents trained with reinforcement learning (RL) provides an interesting opportunity to develop complex emergent enemy NPC behavior and provides unique challenges to players. Moreover, developing RL agents with complex emergent behavior in simulated environments can be applied in real-world scenarios such as autonomous vehicles and robotic manipulation. RL agents trained through competition can develop this complex emergent behavior that would otherwise require detailed environments or meticulously crafted reward functions in traditional RL scenarios.

Computer Vision (CV) can be applied to multi-agent competitive RL scenarios by placing a simulated camera at the front of the agent to capture images at a set frame rate. This technique allows us to avoid the need to hand-pick input features and potentially generalize to real-world scenarios with camera sensors.

In order to explore the space of multi-agent competitive RL with CV, we implement PokAImon, a collection of agents and environments built on top of the UnityML agents toolkit to train agents to play a sumo wrestling game where the goal is to push the opponent off a platform. PokAImon has the benefit of being a real-time physics-driven simulation where a user can pilot a competing agent. In addition to providing a fun challenge, the PokAImon framework can be used to examine the emergent properties of the learned policies as a reaction to specific user input.

Figure 1. Magnemite Sumo Environment in Unity with assets from [1].

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

Our Demo Game can be found at https://colesohn.itch.io/pokaimon.

## 2. Related Work

We are using the Unity ML-Agents Toolkit developed and introduced by Juliani et al. [6] as an environment for our project. This Unity package offers a user-friendly API for simulated deep RL training. The API allows us to define different types of neural network architectures, different simulated environments and agents, as well as different training methods. The paper by Juliani et al. [6] introduced several example environments, as well as a detailed study comparing different training and design strategies. Juliani et al. found that a PPO [9] trainer paired with an LSTM [5] 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 et al. did an extensive study of deep RL training of different agents in different simulated environments. They used the MuJoCo [11] 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 [6], 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 [5] paired with PPO [9] 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 $\alpha$. 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.

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

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. This pipeline also allows us to compare an older version of the network to the most recent one which is necessary to evaluate the training with the ELO score [3].

We also investigate Proximal Policy Optimization (PPO), introduced by Schulman et al. [9], PPO addresses the challenges of optimizing policies in reinforcement learning by striking a balance between stability and sample efficiency. PPO uses a trust region approach, ensuring that policy updates do not deviate significantly from the previous policy, which helps maintain stability during training. Moreover, PPO utilizes a clipped surrogate objective that constrains policy updates to a specified neighborhood around the current policy. This technique mitigates the issues associated with large policy updates and allows for better sample efficiency. The work by Bansal et al. [2] and Juliani et al. [6] both utilize PPO and show promising results in the context of competitive RL.

For vision input, we investigate various image processing approaches. Transfer Learning [7] is a popular strategy in deep learning that involves leveraging pre-trained models on large-scale datasets to solve new tasks with limited labeled data. One widely used architecture for transfer learning is ResNet. Proposed by He et al. [4], ResNet introduced residual connections that allow the network to effectively learn from deeper architectures without suffering from the degradation problem. This network architecture enables the training of significantly deeper networks, leading to improved feature representation. When applied to transfer learning, ResNet has proven to be particularly effective in extracting meaningful features from input images, enabling better generalization and reducing the need for large amounts of labeled data. Transfer Learning with ResNet can be used in a RL context with visual inputs to extract features from the image and use the output vector to train the RL model.

## 3. Methods

We implement the PokAImon agents and environments on top of MLAgents [6], a framework developed by Unity to interface Pytorch [8] with the Unity Game Engine to train RL agents. We set up the Sumo Environment which consists of a platform and two symmetric agents in competition. All agents use the same sparse reward function to update their policy.

To develop our various agent types, we use pre-rigged models from The Models Resource [1], an open-source database of 3D Pokemon models. To allow our agents to interact with the Unity physics engine we assign colliders and physics rigidbody components to various parts of our agents. In the case of jointed agents like Caterpie, we assign joint constraints which define axes and degrees of rotation and connected components. We do further work to visually develop our agents by designing a custom keyframed idle animation and script simple logic to change agent expressions.

### 3.1. Reward

All agent types use the same sparse reward function defined in Eq. 2. If an agent falls off the platform within a time limit, they are the losing agent and receive a reward of $-1$ for the episode. The winning opponent left on the platform receives a score of $+1$ for the episode. If there is no losing agent by the end of the time limit, both agents receive a score of $-1$ to encourage risk-taking actions.

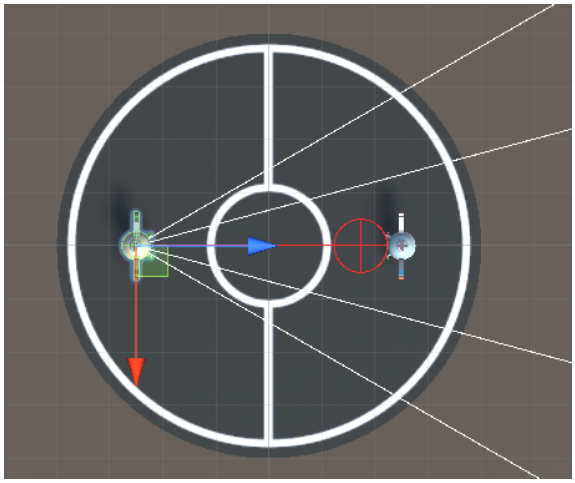

Figure 2. Visualization of Ray Cast Input Features

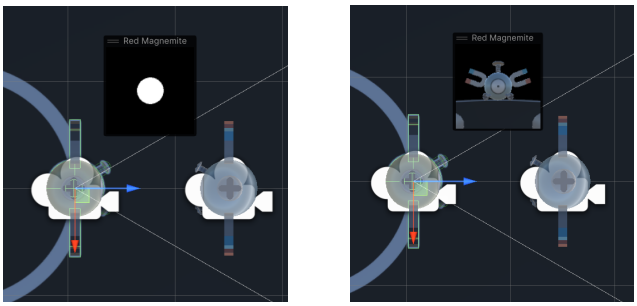


Figure 3. Visualization of Agent's Simplified View (left) and Agent's Full View (right)

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

### 3.2. Agent Types

Agents observe different input features and can take different actions depending on the agent type. We implement a rolling ball agent – aka Voltorb, a hovering and rotating agent – aka Magnemite, and a worm agent composing of 7 physics rigibodies connected by 6 joints – aka Caterpie. Each agent was used to explore different RL strategies. For vision tasks, we use the Magnemite agent as it can rotate to capture visual inputs from different directions. We developed multiple Magnemite agents with different input features to test different strategies.

### 3.3. Features/Observations

All Magnemite agents receive a scalar representing time remaining in the game in seconds, and a vector representing the closest edge to the agent normalized by the radius of the platform. Agents receive different features for different experiments to learn the relative position of the opponent. The baseline Magnemite Agent receives the vector from the agent's position to the opponent's position normalized by platform diameter. Raycast Magnemite receives the collision distance of 5 ray casts radiating out from the agent's forward direction spanning 60 degrees shown in Fig. 2. Reduced Vision Magnemite receives a 32x32 binary image with 60 degree FOV in the agent's forward direction where a white circle represents the opponent to remove visual noise shown in Fig. 3. Full Vision Magnemite receives a 64x64 grayscale image also with a 60 degree FOV of the full scene shown in Fig. 3

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

$$S = \begin{bmatrix} t_{\text{remaining}} \\ D_{\text{self2edge}} \\ D_{\text{self2opponent}} \\ V_{\text{self}} \\ V_{\text{opponent}} \end{bmatrix} \quad (3)$$

where $t_{\text{remaining}}$ is the time remaining in the episode (max is $20s$), $D_{\text{self2edge}}$ is a 2D vector representing the radial displacement between the agent and the edge, $D_{\text{self2opponent}}$ is a 2D vector representing the displacement of the opponent from the agent, $V_{\text{self}}$ is the 2D vector of velocities of the agent, and $V_{\text{opponent}}$ is the 2D vector of velocities of the opponent.

### 3.4. Actions

Each agent has a different action space. For this paper we focus on the Magnemite vision agent. Magnemite can perform three continous actions in the range -1 to 1. Two actions represent a two-dimensional force in a specific direction. These actions allow the agent to translate accross the play space. The third action represents an applied torque to rotate the agent around its center of mass. See Eq. 4 for the action space. This allows the agent to capture different views of the scene as well as perform the unexpected behavior of hitting its opponent with its arms by spinning to push it off the play space.

$$\mathcal{A} = \begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} \quad (4)$$

where $F_x, F_y, \tau_z \in [-1, 1]$

### 3.5. Model Architectures

All models contain a fully-connected MLP stage that takes previously described features as inputs and outputs previously described continous actions. This MLP has two hidden layers with 512 hidden units each. Model weights

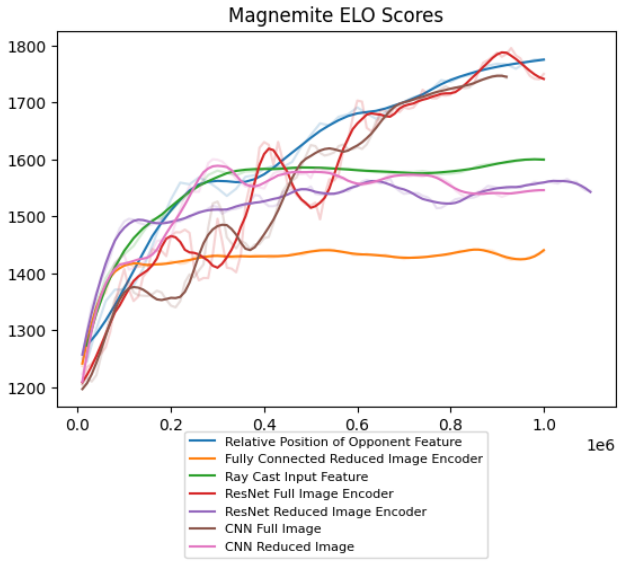


Figure 4. ELO Score per Iteration for Magnemite Agents

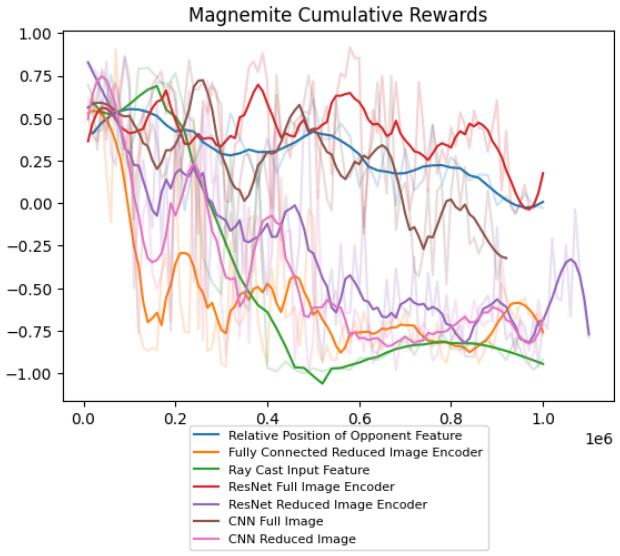


Figure 5. Cumulative Reward per Iteration for Magnemite Agents

are updated with PPO which has been shown to work better for competitive RL agents than other comparable optimization techniques [2]. Baseline Magnemite and Ray Cast Magnemite both follow this setup. For Reduced Vision Magnemite we experiment with three different image embedding layers on top of the default FC MLP including a single fully connected embedding layer that simply flattens the input image into a feature vector, a simple 2-layer convolutional network (CNN), and a pretrained residual network ResNet [4] encoder with a fine-tuned final layer for the specific task. For Full Vision Magnemite we experiment with the simple CNN and ResNet architectures previously described.

# 4. Experiments and Results

## 4.1. Metrics

### 4.1.1 ELO Score

The main metric we used for evaluating model performance is ELO score [3], which was developed for providing a skill metric for zero-sum games. Different checkpoints of our model are saved and assigned an initial ELO score of 1200. Each checkpoint is treated as an independent agent. If an agent wins an episode, its ELO score increases as a function of the opponents score - winning against an opponent with a higher score will result in a larger increase. Inversely, the loser's score drops as a function of its opponent - losing against an opponent with a lower score results in a larger drop. In the case of a draw, the agent with a higher score will lose points and the agent with a lower score will lose points.

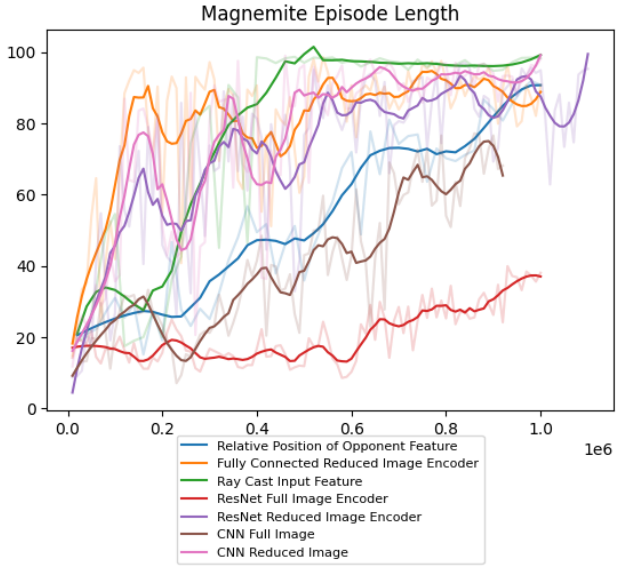


Figure 6. Episode Length per Iteration for Magnemite Agents

We evaluate the ELO [3] score during training using the self-play [10] component. Assuming we have two players, $A$ and $B$, with initial scores of 1200, named $R_A$ and $R_B$, the the score for player $A$ is updated as in Eq. 5 and Eq. 6. The score for player $B$ is updated similarly just by swapping the $A$ and $B$ in the formula.

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \tag{5}$$

$$R_A := R_A + K_A(S_A - E_A) \tag{6}$$

where $K$ is a mastery factor whereas the most recent version of the model (higher skill agent) receives a lower factor and $S$ is the score of the agent from the episode (1 for winning and $-1$ otherwise).

#### 4.1.2 Episode Length

We also keep track of Episode length, which can give us some insight into early agent behavior (i.e. if the agents have learned to attempt to stay on the platform). Stagnant long episode lengths can also tell us if agents are drawing frequently and not taking enough risk.

#### 4.1.3 Cumulative Reward

We also plot cumulative reward. Cumulative reward is not well suited for zero-sum games as the agent's environment becomes more difficult as its opponent improves. Cumulative reward, however, can give us insights about agent behavior. A reward that has plateaued to -1 tells us that opponents are drawing frequently. This will likely correspond with a stagnant ELO score. A reward that oscillates around -1/3 means that agents are winning, losing, and drawing at roughly the same frequency.

### 4.2. Experiments

#### 4.2.1 Baselines

First, we perform baseline experiments with no vision and the enemy relative position as a 2d feature vector as described in section 3.3. We hypothesize that this agent will have the best performance due to reduced noise and reduced input features. Additionally, agent orientation is not important to judge opponent distance for this experiment. The ELO score is represented by the blue line plot in Fig. 4.

Next, we perform another baseline experiment with ray cast features described in section 3.3 resulting in the green line plot in Fig. 4. We saw what we expected, that there was a decrease in performance due to reduced field of view. The agent would only know the distance and rough direction of opponents in the field of view of the ray casts. A real-world analogy to the ray cast agent could be a robotic agent with a Lidar sensor.

As expected, agent quantitative behavior changed drastically. The agent with relative position applied a consistent torque to its body to rapidly spin around and increase the chances of whacking its opponent off the platform. With ray casts, we observed agents balancing rotation to push opponents off the platform with keeping the opponent within its field of view. This behavior was present in all subsequent vision agents. We predicted agents rotating to keep opponents in view but did not predict this behavior of agents using rotation to knock opponents off the platform.

#### 4.2.2 Vision

Next, we experimented with reduced/simplified vision inputs. As described in section 3.3, we only allowed agents to see a white sphere corresponding with the agents position to make it easier for the agent to learn relative opponent distance and location. (More pixels with high values represent a closer opponent, white pixels on the left side of the image represents an opponent to the left, etc.). The goal of these experiments was to reduce noise and number of input features, with the real-world analogy of an image processing step.

We tested this approach with three visual embedding layers as described previously in section 3.5. As expected, we observed significantly worse ELO scores with a fully-connected embedder (orange line in Fig. 4) and similar scores to the ray cast agents (with more variance) for the simple 2-layer CNN embedder and the ResNet embedder. These are represented by the purple and pink line plots in Fig. 4.

Next, we tested an input image of a higher resolution without any simplification. We expected slightly decreased ELO scores as opposed to the simplified image experiments. Interestingly, we achieved better ELO scores with this approach for both the simple CNN and ResNet architectures. These can be seen in the brown and green line plots in Fig. 4. This can be possibly be explained by the fact that earlier iterations of the model performed worse allowing the ELO score to climb faster. Additionally, more visual information could help the agent reinforce knowledge about position on the platform. Additionally, the simpler input examples may have been too low resolution for the agent to produce an accurate estimate of enemy position.

## 5. Conclusion

From our experiments in this project, we conclude that transfer learning with ResNet can be a great generalizing strategy in the Competitive RL task. Using the image features extracted from ResNet, we train a relatively small neural network to learn how to extract the information needed from the features to take the appropriate action. When taking vision inputs, there is no need to meticulously craft a feature vector to train the RL model. This is also applicable to real-world use cases where taking a visual input from a front camera is very feasible making the simulation to world transfer more straightforward.

## 6. Future Work

We would like to experiment with dense rewards to create a curriculum training approach as in the work done by Bansal et al. [2]. This can lead to better and faster training for more complex agents such as the Caterpie crawler.

We would also like to add more agents to this setup by creating more custom rigs from the open-source 3D models database [1].

## 7. Joint Final Project and Contributions

We share portions of this project's codebase with our final project from CS230. For both projects we set up the PokAImon sumo environment. Everything having to do with vision including the Magnemite agent, ray casts, different visual inputs, and experimenting with different CNN model architectures is all for the CS231N final project. The CS230 project was developed by Cole Sohn and Rita Tlemcani and focused on a different topic: Imitation Learning, and used a different set of agents. Our final project report for CS230 can be found here: https://github.com/ritabt/PokAImon/blob/main/Report%20Paper.pdf. The CS230 project was worked on in collaboration with Rita Tlemcani (ritabt@stanford.edu), which includes code used for the sumo environment used in this project. We used code from the UnityML agents toolkit found here: https://github.com/Unity-Technologies/ml-agents. We used 3D models from The Models Resource found here: https://www.models-resource.com/.

## References


[1] The models resource - pokemon x/y. Available at https://www.models-resource.com/3ds/pokemonxy/. 1, 2, 6

[2] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition, 2018. 1, 2, 4, 5

[3] Arpad E. Elo. The proposed uscf rating system, its development, theory, and applications. *Chess Life*, XXII(8):242–247, August 1967. 2, 4

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 2, 4

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 1, 2

[6] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents, 2020. 1, 2

[7] Sinno Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010. 2

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 2

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 1, 2

[10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. 4

[11] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. 1