EMERGENT AUTONOMOUS RACING VIA MULTI-AGENT PROXIMAL POLICY OPTIMIZATION

Ryan Sander

Massachusetts Institute of Technology
77 Massachusetts Avenue, Cambridge MA 02139
rmsander@mit.edu

ABSTRACT

Multi-Agent Reinforcement Learning, or MARL, is a crucial framework for learning emergent cooperative and competitive behaviors in the presence of other intelligent agents. In this paper, we apply MARL techniques, namely Multi-Agent Proximal Policy Optimization (PPO), to a novel multi-agent OpenAI Gym car racing environment developed by (Schwarting et al., 2020). Using an Elo Rating tournament evaluation framework, we empirically observe that agents trained in competitive, multi-car environments perform better in racing competitions than agents trained in single-car environments.

1 INTRODUCTION

1.1 MOTIVATION: LEARNING BEHAVIORS IN MULTI-AGENT SETTINGS

Intelligent agents rarely act in a vacuum - in applications ranging from autonomous vehicles to negotiation and compromise, agents must learn to act optimally not only within their environment, but also in the presence of other adaptive agents in their environment. Learning in these non-stationary environments can lead to learning emergent behaviors that are intrinsic to an environment, but not explicitly a consequence of the environment's reward structure - i.e. the reward need not be shaped (Wiewiora, 2010). In this paper, we study emergent racing behaviors of autonomous Proximal Policy Optimization (PPO) agents.

Proximal Policy Optimization (PPO) algorithms have proven themselves to be viable actor-critic algorithms across a variety of single-agent reinforcement learning tasks. This paper aims to examine the multi-agent performance of agents trained using a variant of Proximal Policy Optimization that leverages a Kullback-Leibler (KL) policy divergence penalty. This algorithm, in addition to our environment implementation, is discussed below.

1.2 OVERVIEW OF APPROACH

This work introduces a novel[1] OpenAI Gym car racing environment designed for Multi-Agent Reinforcement Learning (MARL) tasks as well as policy gradient and visual predictive control models for solving these tasks, such as PPO. PPO is a class of policy gradient actor-critic algorithms designed for robust policy updates. In non-stationary MARL tasks such as this multi-agent car racing environment, this robustness may prove especially beneficial, as optimal actions and estimated values of trajectories can change as other agents in an environment update their policies and value functions. This work also empirically studies different effects of reinforcement learning training paradigms, and quantifies the respective performance of each training paradigm through a competitive Elo Rating (Balduzzi et al., 2018) system. We evaluate three different methods of multi-agent reinforcement learning: single-agent (one car, one agent), multi-agent (two cars, two agents), and self-play (two cars, one agent). The car-to-agent correspondences for these different training paradigms can be seen visually below:

[1]The GitHub repository for this code base can be found here: https://github.com/mit-drl/deep_latent_games.

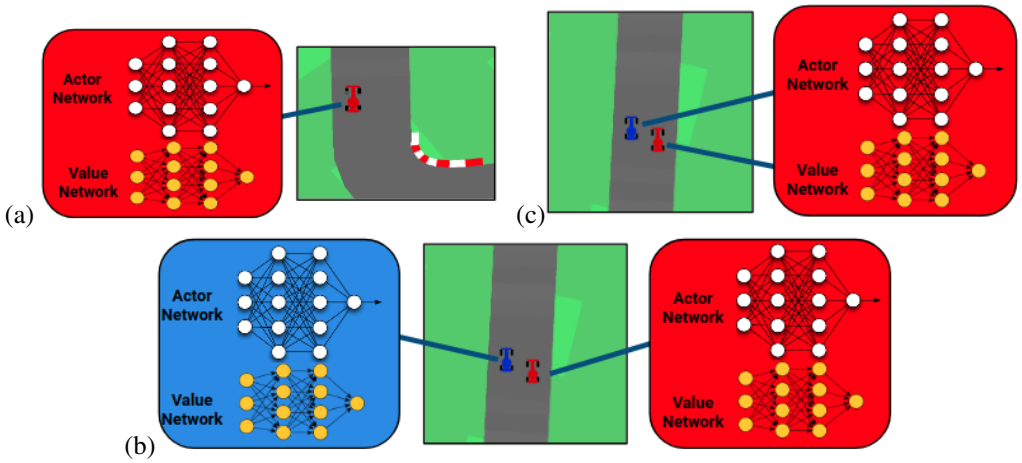

Figure 1: Our MARL training schemes. From left to right: (a) Single-agent (one car, one PPO agent), (b) Multi-Agent (two cars, two PPO agents), (c) Self-Play (two cars, one PPO agent). This work empirically evaluates the competitive performance of each training methodology through an Elo Rating-based competition system.

Through direct competition between trained agents, we can make direct empirical performance comparisons between these training methodologies, providing us with important insights of the extent to which competition during training improves performance, as well as sample efficiencies between training methodologies, which we hope generalizes beyond this MARL task.

To implement our Proximal Policy Optimization (PPO) agents, we leverage TensorFlow Agents (Sergio Guadarrama, 2018) through Python (Van Rossum & Drake, 2009), a novel API for creating scalable and efficient reinforcement learning frameworks without the need for building reinforcement learning algorithms and environments from the ground up.

2 Background & Related Work

This work largely builds on two novel concepts in reinforcement learning: actor-critic methods and multi-agent reinforcement learning frameworks.

2.1 Actor-Critic Methods

Actor-critic methods (Konda & Tsitsiklis, 2000) are a family of policy gradient (Sutton et al., 2000) algorithms. Policy gradient algorithms in turn are a set of reinforcement learning algorithms that learn optimal, parametric policies directly, rather than through a value function, as compared to value-based methods such as Q-Learning (Watkins & Dayan, 1992). At a general level, policy gradients aim to optimize a parametric policy, denoted π_θ, using gradient ascent methods and Monte Carlo estimates of the gradients of advantage functions with respect to policy:

$$\begin{aligned} \theta^{(t+1)} &\leftarrow \theta^{(t)} + \eta \hat{g}^{(t)} \\ \hat{g}^{(t)} &= \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t] \end{aligned} \tag{1}$$

Where η denotes the gradient ascent learning rate, and $\hat{A}_t$ denotes an advantage estimate. In our multi-agent PPO framework, this parametric policy is implemented as two neural networks: an actor (which maps observations to distributions over actions), and a critic (which maps observations to predicted returns). Although policy gradients have advantages over value-based methods such as Tabular Q-Learning, such as the ability to handle continuous state and action spaces, without having a way to estimate values of states, it can become difficult for trained policy gradient agents to select action trajectories that maximize rewards in both single and multi-agent environments. Actor-critic methods overcome this challenge by augmenting this parametric policy with a parametric value function (a "critic") to estimate values of states. These value functions, like their policy ("actor")

counterparts, can be implemented as neural networks, as is done in this paper. Our Proximal Policy Optimization learning algorithm leverages actor-critic architectures for learning optimal behaviors in our environment.

Our learning algorithm, Proximal Policy Optimization, builds off of recent advances in Trust Region Policy Optimization (TRPO) methods (Schulman et al., 2015), a family of policy gradient algorithms designed to learn policies in a stable and approximately monotone fashion. Compared to TRPO, PPO algorithms exhibit better sample efficiency, are simpler to implement, and more general (Schulman et al., 2017). In particular, our PPO variant, which uses an adaptive KL Divergence penalty, was designed to mitigate issues with penalty sensitivity that were empirically observed with using a fixed KL penalty TRPO variant. (Schulman et al., 2017).

2.2 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning approaches form a broad class of algorithms that involve learning optimal behaviors in environments containing multiple intelligent agents. Because different policies can be learned simultaneously between agents, these environments are generally non-stationary (Hernandez-Leal et al., 2017), since the values of states for an agent can depend significantly on the behavior, and therefore adaptive policies, of other agents. Specifically, for this paper, we conceptualize our multi-car racing environment as a (Partially-Observable) Markov Game (Littman, 1994). Techniques such as self-play (Bansal et al., 2017), which we leverage as one of our multi-agent training paradigms, have been used successfully in tandem with PPO for solving other MARL tasks, such as hide and seek (Baker et al., 2019).

Other MARL approaches have been used successfully for visual predictive control tasks such as our car racing task. The first of these is the Multi-Agent Deep Q-Network (MADQN) (Egorov), which leverages convolutional neural networks for predicting state values using four observation channels: (1) Background environment, (2) Ally agents, (3) Opponent agents, and (4) Self agent. Unlike PPO, MADQN is strictly designed for completely-observable environments, and therefore may not be robust to partial observability. RIAL and DIAL (Foerster et al., 2016) are other multi-agent deep reinforcement learning approaches that have been used for multi-agent visual input tasks. While these approaches enjoy sample-efficient learning through shared communication channels between agents, they are designed for learning behaviors in cooperative, rather than competitive, environments, and thus are not amenable for use in our car racing environment.

In our learning framework, we build upon these related ideas presented on Actor-Critic Methods and Multi-Agent Reinforcement Learning.

3 Environment Setting: Multi-Car Racing

To implement our PPO agents to solve a visual predictive control task, we developed a multi-agent reinforcement learning environment based off of the OpenAI Gym car racing environment (Brockman et al., 2016). Our multi-agent car racing environment can be characterized as a Partially-Observable Markov Game composed of the following:

i. A discrete set of states, denoted $\mathcal{S}$

ii. A discrete set of observations, denoted $\mathcal{O}$

iii. A continuous action space, denoted $\mathcal{A}$

iv. A joint (N agent), continuous transition function denoted $T : \mathcal{S} \times A_1 \times \cdots \times A_N \to \mathcal{S}$

v. A joint (N agent), continuous reward function denoted $R : \mathcal{S} \times A_1 \times \cdots \times A_N \to \mathbb{R}^N$

Though this environment is not entirely zero-sum, it is still a competitive environment, because in order to maximize rewards, an agent must cross tiles on the track before other agents, incentivizing agents to drive faster than their opponents. Our goal with our PPO agents will be to learn a policy, denoted π_θ, and a value function, denoted v_ϕ, that can together learn an approximate mapping $\pi_\theta : \mathcal{O} \to \mathcal{A}$ that maximizes the total discounted reward for the agent.

3.1 VISUAL CONTROL INPUT SETTING

The input visual setting employed in this paper corresponds to the discrete observation space $\mathcal{O}$ introduced above. The components of this observation space $\mathcal{O}$ correspond to only a subset of the total components of the state space of our car racing environment, which introduces partial observability. This partial observability occurs because any state $s \in \mathcal{S}$ in this multi-agent environment depends on the positions of all agents in the environment, and these positions may not be visible to all other agents. Therefore, as introduced above, our environment can be conceptualized as a Partially-Observable Markov Game. This introduced partial observability provides motivation for our actor-critic neural network architecture, which leverages both Long Short Term Memory (LSTM) encoders (Hochreiter & Schmidhuber, 1997) for maintaining state, as well as convolution layers for feature extraction from our high-dimensional observation space.

To implement this multi-car racing environment, we augmented the original environment to support N cars, observations, and action vectors. Our environment uses a state-space kinematics model, with a continuous action space $\mathcal{A}$ composed of steering angle, acceleration, and braking:

1. Agent action: $\mathbf{a}_i^{(t)} \triangleq \begin{bmatrix} \delta_t & g_t & b_t \end{bmatrix} \in \mathcal{A} \subset \mathbb{R}^3$
2. Steering Angle: $\delta_t \in [-1, 1]$
3. Gas/Acceleration: $g_t \in [0, 1]$
4. Braking: $b_t \in [0, 1]$

Where g_t denotes gas input (acceleration), b_t denotes braking, and δ_t denotes steering angle. These actions together comprise our action space $\mathcal{A}$, a subset of $\mathbb{R}^3$. With N agents, we consider time step action matrices $\mathbf{A}^{(t)}$ over multiple agents:

$$\mathbf{A}^{(t)} \triangleq \begin{bmatrix} \mathbf{a}_1^{(t)} & \cdots & \mathbf{a}_N^{(t)} \end{bmatrix}, \quad \mathbf{a}_i^{(t)} \in \mathcal{A} \subset \mathbb{R}^3 \tag{2}$$

These action matrices are used to step our environment at each time step. Furthermore, the reward signal is another key component of this environment, as we seek to ensure that the behaviors learned are emergent, and not shaped or influenced too heavily by the environment's reward structure. We retained the original structure of the reward signal used in the single-agent OpenAI Gym car racing environment (Brockman et al., 2016), with the only modifications being (1) the addition of a backwards driving penalty, and (2) weighting the tile reward by the number of cars that have already crossed a given tile. For completeness, the environment rewards are constructed according to the following:

- Each agent begins with 0 points.
- For each time step, an agent is given a base reward of -0.1 points. This encourages the agent to find trajectories that maximize its episodic reward quickly.
- For each tile (of K total track tiles) the agent drives over, the agent receives a reward of $\frac{1000}{K}(1 - \frac{\text{previous cars}}{\text{total cars}})$ points, where "previous cars" refers to the number of cars that have already crossed this tile. Once any agent has driven over all track tiles, the episode ends.
- If an agent drives outside the boundary of the environment, the episode ends, and that agent experiences a reward of -100 points.
- If an agent's car angle is more than $\frac{\pi}{2}$ radians away from the angle of the road (our condition for driving backwards), and the agent is driving on the track, the agent is penalized proportionally to the deviation between these two angles:
  $\Delta\theta \triangleq |\theta_{\text{ROAD}} - \theta_{\text{CAR}}|$.

As visual information is the only information an agent receives from its observations, we chose to include the visual indicators used in the original OpenAI car racing environment, as well as add some of our own. These visual indicator signals provide agents with a richer observation space while avoiding direct reward shaping (Wiewiora, 2010). These visual cues are given by the following:

- "Ego cars" (i.e. the cars the agent perceives as itself) are always observed as red, while all other cars are observed as blue. This enables the PPO agent to learn which car corresponds to the car it is controlling through visual feature extraction.

- If the agent begins driving backwards on the track, a blue triangle is rendered in the bottom right corner of the observation.
- Dynamic bar indicators are provided to the agent at each time step. From left to right (excluding the backwards-driving indicator), these bars correspond to: (1) true speed, (2) ABS sensor readings, (3) steering wheel position, and (4) gyroscope (Brockman et al., 2016).

A trajectory from our multi-agent setting can be visualized below, from the observed perspective of the red car.

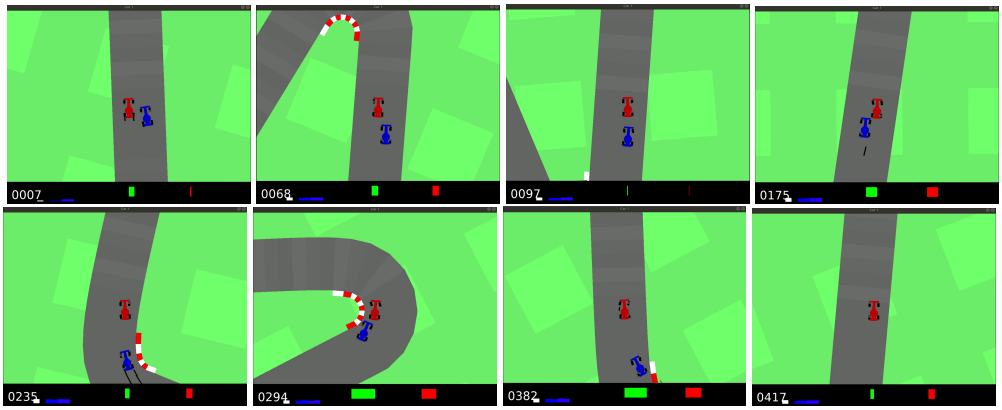

Figure 2: A visualized trajectory for our multi-car racing environment. For all sets of state pixel observations, the ego car is observed as red, while all opponent car(s) are observed as blue.

Together, this interactive framework via a multi-agent OpenAI Gym car racing environment interacts with our PPO framework to support a fully integrated deep reinforcement learning framework for agents to learn how to race against one another. With our environment constructed, we are ready to discuss our training procedure for learning optimal multi-agent behaviors.

4 Learning Through Proximal Policy Optimization

To learn optimal policies, we leverage Proximal Policy Optimization (PPO), a novel class of deep reinforcement learning actor-critic algorithms for learning stable policies (Schulman et al., 2017). Using the framework from the policy gradient section above, we can develop our Proximal Policy Optimization algorithm for our MARL racing task.

4.1 Proximal Policy Optimization through Adaptive KL Penalization

The variation of Proximal Policy Optimization (PPO) we utilize in this work uses an adaptive Kullback-Leibler (KL) Divergence penalty to ensure that policy updates are robust and approximately optimal throughout the training process (Schulman et al., 2017). Like other PPO variants, our adaptive KL-penalty PPO leverages an actor-critic framework for action selection (done by the "actor") and value estimation (done by the "critic"). Let our parametric policy and value function approximators be defined according to:

i. **Policy**: $\pi_\theta : \mathcal{O} \rightarrow \mathcal{A}$, where $\mathcal{O}$ and $\mathcal{A}$ are the observation and action spaces, respectively.

ii. **Value function**: $v_\phi : \mathcal{O} \rightarrow \mathbb{R}$, where $\mathcal{O}$ is the observation space.

We can optimize the parameters in our parametric policy and value function through the surrogate objective function below. Using Monte Carlo estimates over entire trajectories, we apply neural network backpropagation updates using the gradient of the following surrogate objective:

$$L^{\text{KLPEN}}(\theta, \phi) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|o_t)} (R_t - v_\phi(t)) - c_1 L_t^{\text{VF}}(\phi) - \beta \text{KL}(\pi_{\theta_{\text{old}}}(\cdot|o_t)||\pi_\theta(\cdot|o_t)) \right], \quad (3)$$

where o_t is an agent's observation at time t, R_t is the empirical return for an agent, L_t^{VF} is a mean-squared value prediction loss, and β is an adaptive KL penalty coefficient. Using backpropagation and gradient ascent methods, the policy parameters θ and value function parameters ϕ are updated (at a general level) according to:

$$\theta \leftarrow \theta_{\text{old}} + \eta \nabla_\theta L^{\text{KLPEN}}(\theta, \phi), \quad \phi \leftarrow \phi_{\text{old}} + \eta \nabla_\phi L^{\text{KLPEN}}(\theta, \phi), \tag{4}$$

where η denotes the learning rate for these updates. The KL Divergence penalty is also adapted according to how much the sample expectation of KL policy divergence changes relative to a target change between policies, denoted d_{targ}. This update rule is governed by:

$d = \hat{\mathbb{E}}_t \left[\text{KL}(\pi_{\theta_{\text{old}}}(\cdot|o_t) || \pi_\theta(\cdot|o_t)) \right]$
if $d < d_{\text{targ}} \,/\, 1.5$ **then**
  $\beta \leftarrow \frac{\beta}{2}$
else if $d > d_{\text{targ}} \times 1.5$ **then**
  $\beta \leftarrow 2\beta$
end if

This adaptive KL penalty ensures that policy updates of optimal size can be made, improving both robustness and sample efficiency. This PPO training procedure is summarized as follows:

Algorithm 1 PPO with KL Penalization

$\theta \leftarrow \theta_{\text{INIT}}$
$\phi \leftarrow \phi_{\text{INIT}}$
for i in N **do**
  $s_t \leftarrow s_0$
  for t in T **do**
    Collect experience for replay buffer (o_t, a_t, r_t)
    Step environment $s_t \leftarrow s_{t+1}$
  end for
  Compute $L^{\text{KLPEN}}(\theta, \phi) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|o_t)} (R_t - v_\phi(t)) - c_1 L_t^{\text{VF}}(\phi) - \beta \text{KL}(\pi_{\theta_{\text{old}}}(\cdot|o_t) || \pi_\theta(\cdot|o_t)) \right]$
  Update policy: $\theta \leftarrow \theta_{\text{old}} + \eta \nabla_\theta L^{\text{KLPEN}}(\theta, \phi)$
  Update value function: $\phi \leftarrow \phi_{\text{old}} + \eta \nabla_\phi L^{\text{KLPEN}}(\theta, \phi)$
  $d = \hat{\mathbb{E}}_t \left[\text{KL}(\pi_{\theta_{\text{old}}}(\cdot|o_t) || \pi_\theta(\cdot|o_t)) \right]$
  if $d < d_{\text{targ}} \,/\, 1.5$ **then**
    $\beta \leftarrow \frac{\beta}{2}$
  else if $d > d_{\text{targ}} \times 1.5$ **then**
    $\beta \leftarrow 2\beta$
  end if
end for
return π_θ, v_ϕ

With our learning algorithm outlined, we can now discuss our implementation of the actor-critic neural networks used for our PPO agents.

4.2 Neural Network Structure

The policy and value function for our PPO variant are implemented through neural networks. Architecturally, these neural networks consist of:

i. An LSTM encoder (Hochreiter & Schmidhuber, 1997), which we use to maintain a notion of state and velocity.
ii. Convolution layers, which are used for feature extraction on high-dimensional visual observations.
iii. Multilayer Percptrons (MLP), which are used for action selection and value estimation at the output.

We found that using all three of these architectural components was essential for learning appropriate, high-dimensional, temporally-dependent policies. The parameters used for each component of this network can be found in the Appendix. With our environment and learning framework laid out, we are now ready to discuss our MARL experiments and evaluation approach.

5 EXPERIMENTS

Our main contribution of this paper is to apply the PPO agent to our novel multi-agent environment, and evaluate its performance using multi-agent and adversarial self-play training paradigms. Below is a diagram of our different MARL training processes:

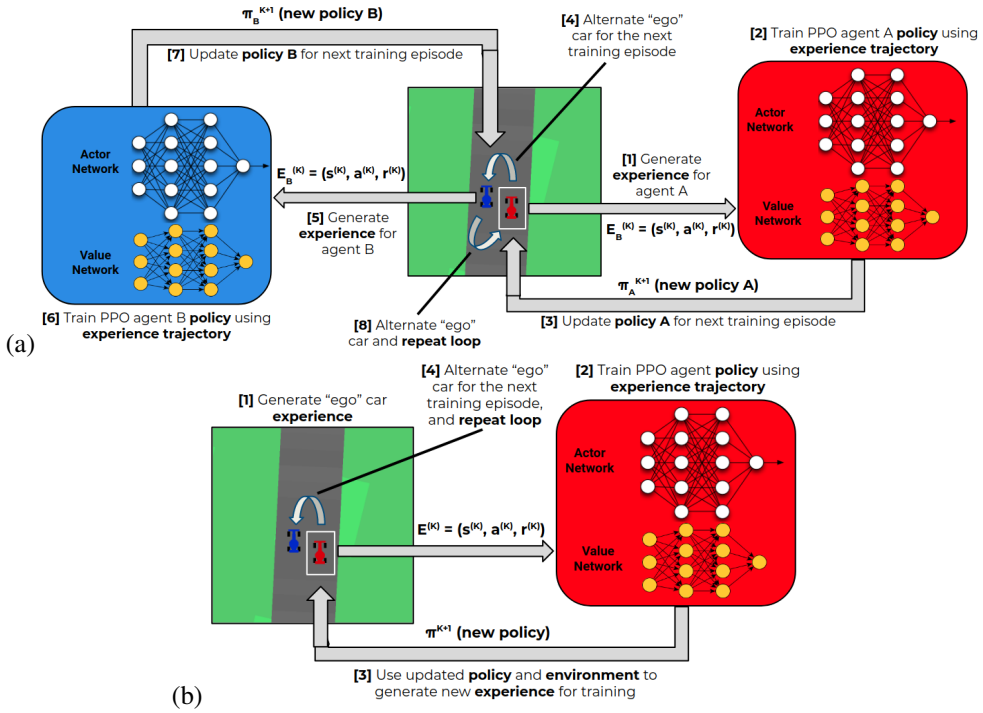

Figure 3: (a) Multi-car, multi-agent training diagram. Each car is controlled by its own PPO agent. (b) Self-play training diagram. Each car is controlled by the same PPO agent, though the agent only observes the environment through one car at a time.

5.0.1 SINGLE-AGENT TRAINING

This training paradigm corresponds to the original single-agent OpenAI Gym car racing (Brockman et al., 2016) environment, with added negative reward for driving on the track in the reverse direction and a visual cue indicating this behavior. Our "single-agent" agents are our baseline we consider for our multi-agent Elo Rating evaluation framework.

5.1 ADVERSARIAL MULTI-AGENT

The first MARL training paradigm we implement for our PPO agent is multi-car, multi-agent, with two simulated cars and two simulated agents. During training, we designate one agent to be the "ego agent", and collect trajectories for the replay buffer from that agent's observations using its exploration policy while it is racing against the other "racing agent". This "racing agent" in turn is using its evaluation policy. After taking an optimization step using an entire rollout from the "ego agent", we then alternate which car becomes the "ego agent". After repeating this alternating training process, we evaluate each agent using their greedy evaluation policies.

5.2 Adversarial Self Play

The second MARL training paradigm we investigate is adversarial self-play (Bansal et al., 2017), in which we run simulations using two cars, and one master PPO agent. The motivation for self-play is that this training paradigm increases sample efficiency (Bansal et al., 2017), and ensures that when training, an agent is always competing with competition most closely aligned in skill level. During training, rather than selecting an ego agent, we select an ego car, and again generate experiences for that car to fill the experience replay buffer with, all while the other "racing agent" is being run independently on the master agent's greedy evaluation policy. Here, turn alternation is performed at the car, but not agent level (since there is only one agent). Thus, the same agent is trained for each episode, using experiences collected from entire rollouts of each car.

6 Evaluation and Results

To evaluate and compare the efficacy and performance of the three training paradigms introduced above, we consider both the agent's returns during training, as well as the agent's performance in our Elo Rating tournament evaluation framework. These evaluation criteria are each summarized below.

6.1 Training Performance

The average return from our exploration policies of each of our MARL training schemes can be visualized below:

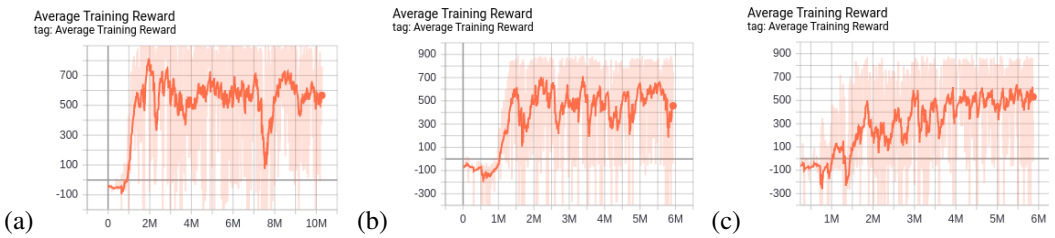

Figure 4: Returns as a function of total training steps for our different training schemes. (a) Single-agent, (b) Multi-agent, (c) Self-play.

Our average greedy policy returns actually yielded lower performance than our exploration returns. We believe this stems from a domain shift in speed between the training and evaluation environments. Since the exploration policy adds noise to its output when selecting an action, cars being controlled by the exploration policy tend to accelerate much less consistently during training when compared to their greedy policy counterparts. As a result, our agents do not learn how to navigate at high speeds with their greedy policies, which often results in cars spinning out, driving off the track, or driving in the wrong direction. To help these agents better perceive their velocities, we will augment the observation space to include not only the current observation, but the past three observations as well, enabling for more robust speed estimation during both training and evaluation.

6.2 Elo Rating

To directly compare our trained policies against one another, we use an Elo Rating (Balduzzi et al., 2018) tournament evaluation framework to assign a rating score to each trained policy after many rounds of competition. This evaluation algorithm proceeds as follows:

Algorithm 2 Elo Rating Evaluation

players $\leftarrow [\text{player}_1, \text{player}_2, \ \dots \ , \text{player}_K]$
elo ratings $\leftarrow [0, 0, \ \dots \ , 0]$
for i in N **do**
  $\text{player}_1, \text{player}_2 \leftarrow$ **choose opponents**(elo ratings)
  $\text{expected score} \leftarrow \left(1 + 10^{\left(\frac{\text{elo ratings}[\text{player}_1] - \text{elo ratings}[\text{player}_2]}{400}\right)}\right)^{-1}$
  score $\leftarrow$ **simulate**(env, $\text{player}_1, \text{player}_2$)
  $\text{elo ratings}[\text{player}_1] \leftarrow \text{elo ratings}[\text{player}_1] + k(\text{score} - \text{expected score})$
  $\text{elo ratings}[\text{player}_2] \leftarrow \text{elo ratings}[\text{player}_2] + k(\text{expected score} - \text{score})$
end for
return elo_ratings

Using this evaluation algorithm, we compared the performance of our trained LSTM policies for single-agent, multi-agent, and competitive self-play against one another. Our evaluation parameters and results from this tournament are given below:

Metric	Value
Evaluation Rounds	200
Update Rate (k)	32
Episode Length (steps)	1000
Single-Agent Elo Rating	-174
Multi-Agent Elo Rating	195
Self-Play Elo Rating	-21

Table 1: Our Elo Rating parameters and results after having our trained single-agent, multi-agent, and self-play policies compete against one another.

7 Discussion and Conclusion

From our training and Elo Rating evaluation results, we conclude that policies trained using multi-car training approaches (in this case, multi-agent and self-play) perform substantially better in multi-agent racing settings than policies trained using the single-agent training paradigm. We believe this is a direct result of our multi-car policies having been trained inside a competitive racing environment.

Our multi-agent training system yielded higher tournament performance than our self-play training approach, which was unexpected, especially given that each agent in the multi-agent system received approximately half as many training steps as the agent trained using self-play. This phenomenon may stem from some variation of over-training/over-fitting, but it remains an open empirical question from this study.

As discussed in our training results above, for all three training paradigms (single-agent, multi-agent, and self-play), the smoothed return of the greedy policy is significantly lower than that of the exploration policy. We believe this is due to a domain shift between the training and evaluation environments, specifically concerning the speed of the cars. Though we are only providing the neural networks with a single observation at a time, because our actor-critic networks leverage LSTM encoders, we believe these networks have the capacity to maintain a belief state, which could include a notion of speed. We plan to investigate if training these LSTM-based policies over more episodes will improve the agents' abilities to perceive their speed, and as a result, their learned driving behaviors. We will also experiment with multi-frame observations to see if this addresses this speed perception problem.

This work demonstrates the viability of our KL Penalty Proximal Policy Optimization (PPO) variant (Schulman et al., 2017) as a robust deep reinforcement learning algorithm for our novel multi-agent OpenAI Gym car racing environment. Additionally, this work quantitatively assesses the empirical

performance of different MARL training paradigms, namely single-agent, multi-agent, and self-play training. We find that policies trained using multi-agent and self-play Bansal et al. (2017) training substantially outperform policies trained using single-agent training in autonomous racing competitions. This suggests that optimal multi-car racing agents learn emergent behaviors not only for navigation, but also specifically for winning against their opponents.

8 ACKNOWLEDGEMENTS

Thank you to Professor Phillip Isola and James Minor for their valuable feedback and recommendations for this project, as well as AWS Educate and Google Compute Engine for providing us with the compute power necessary to complete this project.

Most importantly, thank you to my research advisors Igor Gilitschenski, Wilko Schwarting, Lucas Liebenwein, Tim Seyde, Sertac Karaman, and Daniela Rus. None of this work would have been possible without them.

REFERENCES

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

David Balduzzi, Karl Tuyls, Julien Perolat, and Thore Graepel. Re-evaluating evaluation. In *Advances in Neural Information Processing Systems*, pp. 3268–3279, 2018.

Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Maxim Egorov. Multi-agent deep reinforcement learning.

Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems*, pp. 2137–2145, 2016.

Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. A survey of learning in multiagent environments: Dealing with non-stationarity, 2017.

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

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pp. 1008–1014, 2000.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pp. 157–163. Elsevier, 1994.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Wilko Schwarting, Tim Seyde, Igor Gilitschenski, Lucas Liebenwein, Ryan Sander, Sertac Karaman, and Daniela Rus. Deep latent competition: Learning to race using visual control policies in latent space. 11 2020.

Oscar Ramirez Pablo Castro Ethan Holly Sam Fishman Ke Wang Ekaterina Gonina Neal Wu Efi Kokiopoulou Luciano Sbaiz Jamie Smith Gábor Bartók Jesse Berent Chris Harris Vincent Vanhoucke Eugene Brevdo Sergio Guadarrama, Anoop Korattikara. Tf-agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. URL `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019].

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Eric Wiewiora. *Reward Shaping*, pp. 863–865. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_731. URL https://doi.org/10.1007/978-0-387-30164-8_731.

9 APPENDIX

9.1 TF AGENTS PPO HYPERPARAMETERS

Below are the hyperparameters used to train our PPO agents using the TensorFlow Agents API.

Parameter	**Value**
Importance Ratio Clipping	0
Policy L_2 Regularization	0
Value Function L_2 Regularization	0
Value Prediction Loss Coefficient (c_1)	0.5
Initial KL β	1.0
Adaptive KL Target	1.0
Adaptive KL Tolerance	0.1
Use GAE	False
Use TD-λ	False
Episode Length	1000 Steps
Replay Buffer Size	1000 Steps
Observation Dimensions	(64, 64, 3)

9.2 NEURAL NETWORK PARAMETERS

For training our PPO agents, we also used a custom neural network architecture for our policy and value networks that serve as our actor and critic, respectively.

Parameter	**Value**
LSTM Units	256
Conv1 Filters	16
Conv1 Kernel Size	8
Conv1 Stride	4
Conv2 Filters	32
Conv1 Kernel Size	3
Conv1 Stride	2
Input Fully Connected Neurons	256
Output Fully Connected Neurons	128
Optimizer	ADAM
Learning Rate	8×10^{-5}
Epochs	5000

9.3 IMPLEMENTATION DETAILS

These experiments were run with the following specifications:

Parameter	Value
Operating System	Ubuntu 18.04 LTS
Instance Type	c2-standard-16 (Google Compute Engine)
vCPUs	16
GPUs	0
Memory	64 GB
Training Time	120 hours