Learning to Play Soccer Using Reinforcement Learning

Chaman Singh[1]
[1]Northeastern University
360 Huntington Avenue
Boston, Massachusetts 02115
{singh.ch}@northeastern.edu

Abstract

The goal of this project is to use the TensorFlow library, which was built exclusively for machine learning tasks, to apply a modern multi-agent reinforcement learning algorithm to the soccer Twos multi-agent model in the Unity 3D environment. Soccer Two is an adversarial game in which two artificial neural networks compete in a simple game of soccer. The neural networks are trained using deep reinforcement learning. Each player in soccer is an agent, with two of them having the brains of a goalkeeper and two of them having the brains of a striker. Three current algorithms for handling the multi-agent reinforcement learning problem are discussed in this work. These methods were tweaked to work with the ”Soccer Twos” multi-agent model. The Soccer Twos model's built-in ML gents were adjusted for the job, and modeling tools for reinforcement learning trials were studied.

Introduction

Physics-based competitive games have many different state spaces, making them an intriguing learning platform. They create a difficult environment in which players must employ a variety of methods to react to the opponent's tactics. The addition of an opponent to an agent increases the complexity of an already complicated environment, resulting in more fascinating behaviors. In competitive games, there are a variety of strategies that can be used, including playing passively and waiting for the opponent to make a mistake, playing aggressively to drive the opponent to make a mistake and even leveraging the environment objects to an agent's benefit. Because of the huge number of possibilities, it's impossible for a programmer to account for all of them and develop a rule-based, intelligent, and credible AI Agent. While all the studies show novel detailed breakthroughs in the use of Machine Learning and Reinforcement Learning in complicated video game settings, the current paper adds a new game scenario, a detailed iterative approach to the reward mechanism design, and a focus on believability.

Using various Deep RL algorithms, this project will create agents that learn to play Unity Soccer Twos and conduct intrinsic behaviors such as protecting their goal and attacking the ball using reward functions.

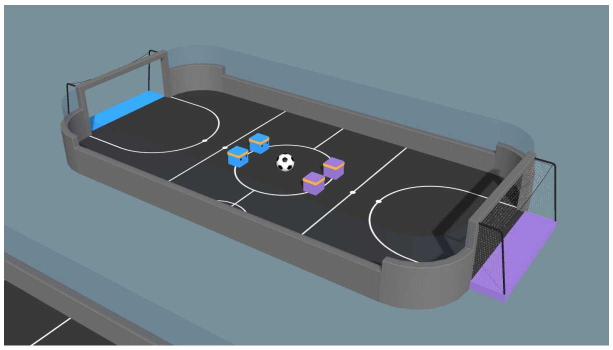


Figure 1: Unity ML Agent Soccer Twos Game Environment

Background

The project is being built using Unity, the world's most popular development engine for creating 2D and 3D multiplatform games and interactive experiences.

It's difficult to create responsive and clever non-playable game characters and virtual players. Especially if the game is difficult. Developers have had to resort to writing a lot of code or utilizing highly specialized technologies to achieve intelligent behaviors. It is no longer necessary to ”write” emergent behaviors. With Unity Machine Learning Agents (ML-Agents), this herculean task to has downed to teaching intelligent agents to ”learn” through a combination of deep reinforcement learning and imitation learning. Developers can use ML-Agents to generate more engaging gameplay and a better gaming experience.

The growth of artificial intelligence (AI) research is contingent on identifying difficult challenges in actual environments utilizing current standards for AI model training. However, as these problems are ”fixed,” other contexts become necessary. However, developing such environments is time-consuming and necessitates specific topic knowledge. Therefore, we designed AI settings that are physically, aesthetically, and cognitively rich using Unity and the ML-Agents toolkit. They can be used for benchmarking as well as the development of new algorithms and methodologies.

The Unity Ml Agents toolkit allows to create and compare reinforcement learning algorithms. This in turn allows customization and application of ML-Agents to meet our specific requirements. Using the self-play method, the agents are taught using RL and the PPO algorithm to place them in a two-versus-two situation. The agents learn how to deploy themselves defensively or offensively and work together to score a goal on the opponent while avoiding conceding a goal.

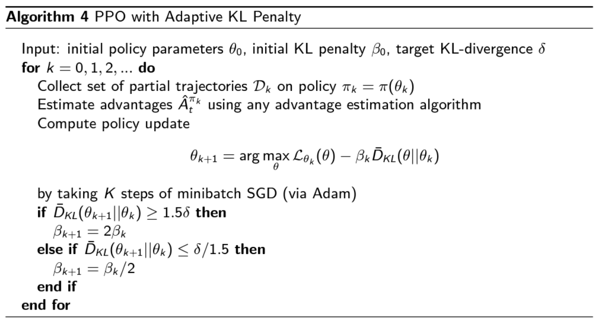
A. PPO (Proximal policy optimization)

PPO is a method that is easier to implement because it is an on-policy algorithm. PPO approximates the ideal function that maps an agent's observations to the optimum action an agent can take in each state using a neural network. TensorFlow is used to implement the ML-Agents PPO algorithm, which runs in a separate Python process (communicating with the running Unity application over a socket). PPO attempts to find a compromise between ease of implementation, sample complexity, and tuning ease by computing an update at each step that minimizes the cost function while ensuring a modest divergence from the preceding policy. The objective function is defined as:

$$L^{\text{CLIP}}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t]$$

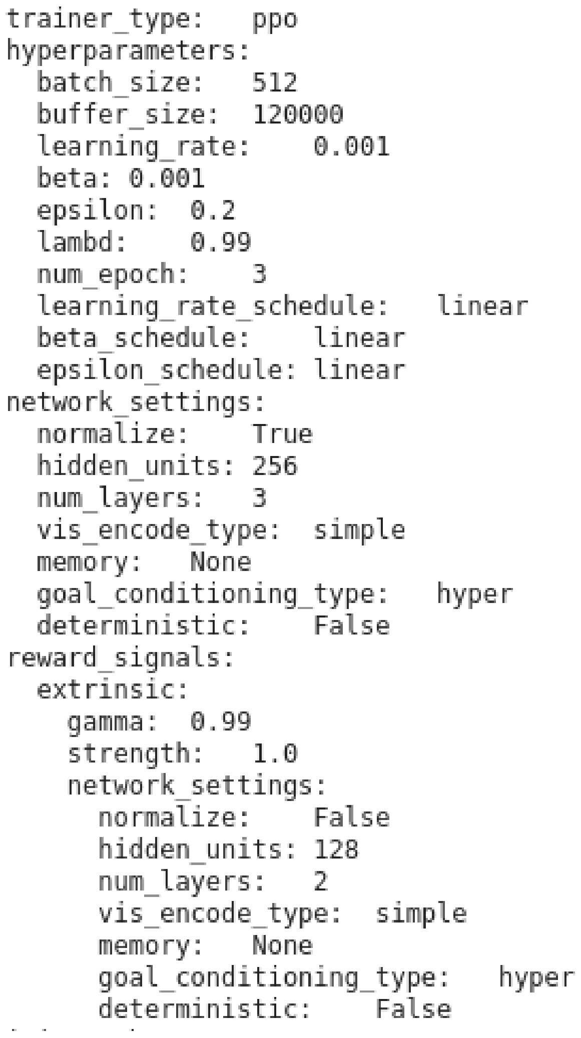
- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- A_t is the estimated advantage at time t
- ϵ is a hyperparameter, usually 0.1 or 0.2

This method outperformed ACER on continuous control tasks and, although being significantly easier to develop, came close to matching ACER's performance on Atari. We've constructed interactive agents based on PPO policies — we can use the keyboard to specify new target locations for an agent in a soccer two's environment; despite the input sequences being different from what the agent was trained on, the agent manages to generalize.



```
Algorithm 4 PPO with Adaptive KL Penalty
Input: initial policy parameters $\theta_0$, initial KL penalty $\beta_0$, target KL-divergence $\delta$
for $k = 0, 1, 2, ...$ do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
    Compute policy update
        $\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta||\theta_k)$
    by taking $K$ steps of minibatch SGD (via Adam)
    if $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \geq 1.5\delta$ then
        $\beta_{k+1} = 2\beta_k$
    else if $\bar{D}_{KL}(\theta_{k+1}||\theta_k) \leq \delta/1.5$ then
        $\beta_{k+1} = \beta_k/2$
    end if
end for
```

Figure 2: PPO pseudo-code from the original paper



```
trainer_type:   ppo
hyperparameters:
  batch_size:   512
  buffer_size:  120000
  learning_rate:    0.001
  beta: 0.001
  epsilon:  0.2
  lambd:    0.99
  num_epoch:    3
  learning_rate_schedule:   linear
  beta_schedule:    linear
  epsilon_schedule: linear
network_settings:
  normalize:    True
  hidden_units: 256
  num_layers:   3
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.99
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
```

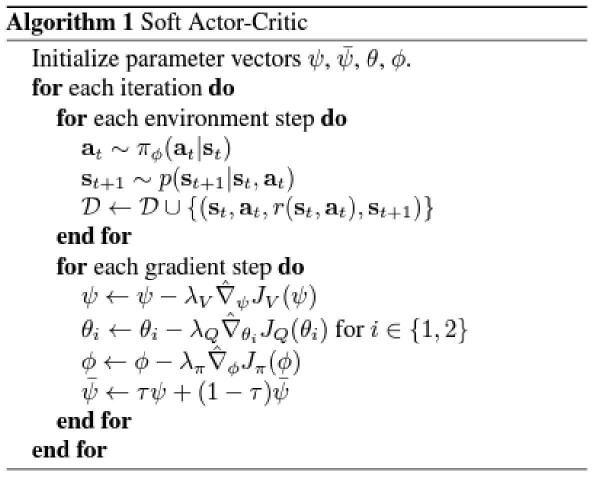
Figure 3: Hyperparameter settings for PPO

B. SAC (Soft Actor-Critic)

Off-Policy Learning is what SAC stands for. If we can reuse data acquired for another purpose, an algorithm is against policy. When designing a new job, we often need to tweak parameters and shape the reward function, and employing an off-policy method allows us to reuse previously acquired data. It optimizes a stochastic policy in an off-policy manner, bridging the gap between stochastic policy optimization and DDPG-style methods. It employs the clipped double-Q technique. SAC employs entropy regularization, in which the policy is taught to maximize the expected return/entropy trade-off (randomness in the policy). SAC learns a policy and two Q-functions at the same time.
Soft actor-critic (SAC), which is detailed below, is a deep RL method that is well-aligned with these requirements while being off-policy and model-free. We show that it is sample efficient enough to perform real-world robot tasks in a few hours, hyperparameter-resistant, and operates on a range of simulated settings with a single set of hyperparameters.
The objective function of SAC is:

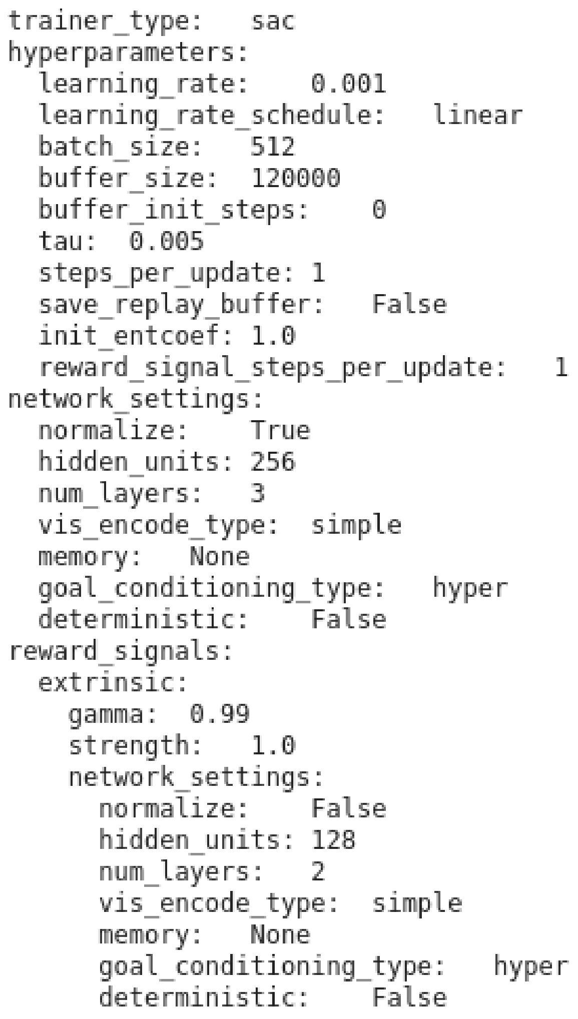
$$J(\pi) = E_\pi \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log(\pi(\mathbf{a}_t|\mathbf{s}_t)) \right]$$

where s_t and a_t are the state and the action, and the expectation is taken over the policy and the true dynamics of the system.
In other words, the best policy maximizes not only the expected return (first summand) but also its own predicted entropy (second summand). The non-negative temperature parameter controls the trade-off between the two, and we can always get back to the traditional, maximum expected return target by choosing =0. Soft actor-critic achieves this goal by employing a neural network to parameterize a Gaussian policy and a Q-function, then optimizing them using approximate dynamic programming.



```
Algorithm 1 Soft Actor-Critic
  Initialize parameter vectors $\psi$, $\bar{\psi}$, $\theta$, $\phi$.
  for each iteration do
    for each environment step do
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
    for each gradient step do
      $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
      $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
  end for
```

Figure 4: SAC pseudo-code from the original paper



```
trainer_type:   sac
hyperparameters:
  learning_rate:    0.001
  learning_rate_schedule:   linear
  batch_size:   512
  buffer_size:  120000
  buffer_init_steps:    0
  tau:  0.005
  steps_per_update: 1
  save_replay_buffer:   False
  init_entcoef: 1.0
  reward_signal_steps_per_update:   1
network_settings:
  normalize:    True
  hidden_units: 256
  num_layers:   3
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.99
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
```

Figure 5: Hyperparameter settings for SAC

C. POCA (Multiagent Posthumous Credit Assignment)

ML-Agents can train cooperative behaviors, which are defined as groups of agents working together to achieve a shared objective, with the success of each individual being related to the success of the entire group. Agents are usually rewarded as a group in this situation. For example, if an agent team defeats a rival team, everyone is awarded, including agents who did not actively contribute to the victory. This makes it tough to figure out what to do as an individual because wins for doing nothing and loses for trying the best. MA-POCA (Multiagent Posthumous Credit Assignment), a revolutionary multi-agent trainer that trains a centralized critic, a neural network that works as a "coach" for a collection of agents, is available in ML-Agents. The team is given rewards, and the agents will figure out how to best contribute to the achievement of that reward. Individual awards can also be provided to agents, and the team will work together to assist them to reach their objectives. Agents can be added or removed from the group during an episode, such as when they spawn or die in a game.
If agents are removed from the game in the middle of an episode (for example, if teammates die or are removed from the game), they will still learn whether their actions contributed to the team's victory later, allowing agents to take group-beneficial actions even if it means they are removed from the game (i.e., self-sacrifice). MA-POCA can also be used in conjunction with self-play to teach agents to compete against one another.

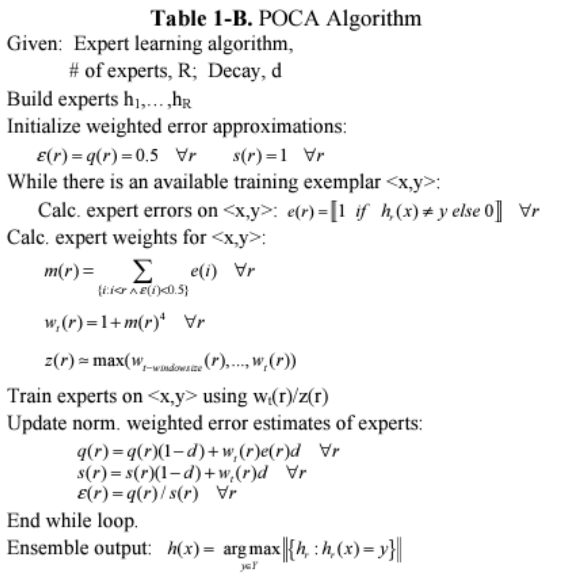


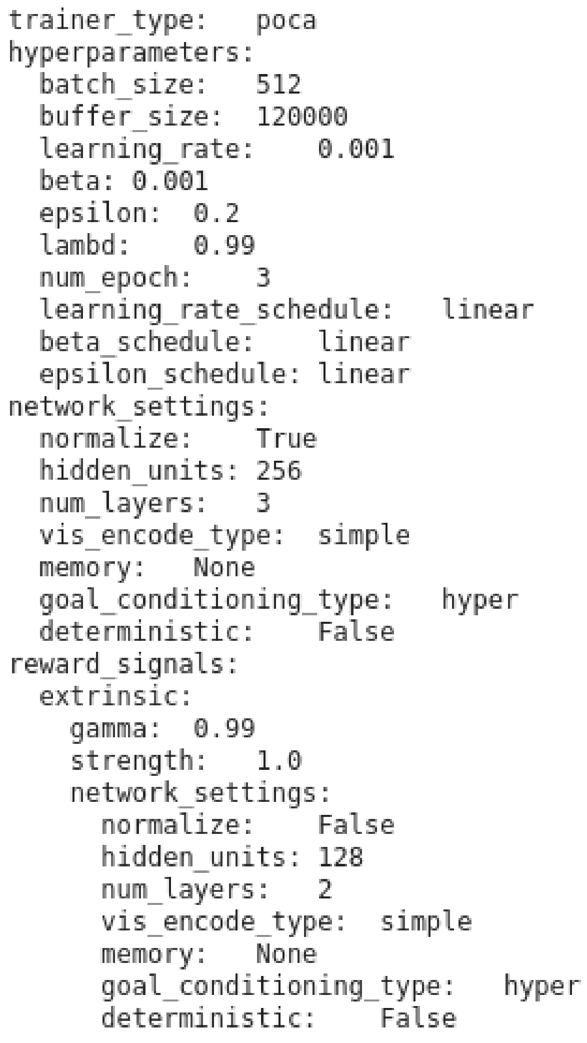
Table 1-B. POCA Algorithm

```
Given:  Expert learning algorithm,
        # of experts, R;  Decay, d
Build experts $h_1,\dots,h_R$
Initialize weighted error approximations:
    $\varepsilon(r) = q(r) = 0.5 \quad \forall r \qquad s(r) = 1 \quad \forall r$
While there is an available training exemplar <x,y>:
    Calc. expert errors on <x,y>:  $e(r) = [\![ 1 \ \text{if} \ h_r(x) \neq y \ \text{else} \ 0 ]\!] \quad \forall r$
Calc. expert weights for <x,y>:
    $m(r) = \sum_{\{i : i<r \wedge \varepsilon(i)<0.5\}} e(i) \quad \forall r$
    $w_t(r) = 1 + m(r)^4 \quad \forall r$
    $z(r) \approx \max(w_{t-windowsize}(r), \dots, w_t(r))$
Train experts on <x,y> using $w_t(r)/z(r)$
Update norm. weighted error estimates of experts:
        $q(r) = q(r)(1-d) + w_t(r)e(r)d \quad \forall r$
        $s(r) = s(r)(1-d) + w_t(r)d \quad \forall r$
        $\varepsilon(r) = q(r)/s(r) \quad \forall r$
End while loop.
Ensemble output:  $h(x) = \arg\max_{y \in Y} \|\{h_r : h_r(x) = y\}\|$
```

Figure 6: POCA pseudo-code from the original paper

Related work

We also considered using TRPO, ACER, DDPG, and DQN Algorithms for our project. Because policy gradient approaches are sensitive to step size, getting good results with



```
trainer_type:   poca
hyperparameters:
  batch_size:   512
  buffer_size:  120000
  learning_rate:    0.001
  beta: 0.001
  epsilon:  0.2
  lambd:    0.99
  num_epoch:    3
  learning_rate_schedule:   linear
  beta_schedule:    linear
  epsilon_schedule: linear
network_settings:
  normalize:    True
  hidden_units: 256
  num_layers:   3
  vis_encode_type:  simple
  memory:   None
  goal_conditioning_type:   hyper
  deterministic:    False
reward_signals:
  extrinsic:
    gamma:  0.99
    strength:   1.0
    network_settings:
      normalize:    False
      hidden_units: 128
      num_layers:   2
      vis_encode_type:  simple
      memory:   None
      goal_conditioning_type:   hyper
      deterministic:    False
```

Figure 7: Hyperparameter settings for POCA

them is difficult. If the step size is too small, progress is impossible to make, and if it is too large, the signal is drowned out by noise. In such cases, there is a significant decline in performance. They also have a low sample efficiency, which means learning basic tasks takes millions or billions of timesteps. Researchers have used techniques like TRPO and ACER to try to fix these issues by limiting or otherwise optimizing the size of a policy change.
These algorithms each have their own set of trade-offs: ACER is significantly more complicated than PPO, involving the addition of code for off-policy corrections and a replay buffer, but only outperforming PPO on the Atari benchmark by a hair. As a result, those methods were ruled out for training. Though effective for continuous control tasks, TRPO is incompatible with algorithms that exchange parameters between a policy and a value function or auxiliary losses, such as those used to solve issues in Atari and other domains with considerable visual input.

DDPG, like many RL algorithms, is prone to instability and is strongly reliant on finding the right hyperparameters for the current job. This is due to the algorithm's constant overestimation of the critic (value) network's Q values. These estimating mistakes accumulate over time, causing the agent to enter local optima or suffer catastrophic forgetfulness.

DQN (Deep Q learning), their disadvantage is that they significantly delay learning while also increasing sample complexity. DQN also has concerns with stability: various runs of the same network may not converge in the same way. Furthermore, this strategy was not included in the ML agent tool kit, and our attempts to reproduce the learning were unsuccessful. Furthermore, one of the difficulties with the DQN method is that it overestimates genuine rewards; the Q-values believe the agent will receive a bigger return than it will. DQN is a function that approximates a set of values that are highly interconnected. For all these reasons, we decided not to use this strategy.

Approach

We're working with the soccer twos framework's starter environment from the ML agents tool kit. Experimenting with pre-trained models that are flexible enough to attempt alternative techniques and approaches for training agents to expand the advanced AI and research applications. The team trained and constructed a neural network model that delivered the correct behavior using the ML-Agents toolset — especially, deep reinforcement learning. The Proximal Policy Optimization method was the first one we worked on. Recent developments in the use of deep neural networks have relied heavily on policy gradient approaches.

Experiments

The reward function is adjusted through trial and error to gradually create behavioral patterns that increase performance. The results show that agents with a reward function that considers different state-space factors perform better than those with a less defined reward function and state space. Furthermore, the final agent created because of the studies has proven to be convincing and difficult to identify from a human player.

A. Environment setting

We started working on the project by setting the environment of the game. It consists of 4 agents in a 2 vs 2 soccer game where the objective of the agents is to get the ball into the opponent's goal while preventing the ball from entering its own goal. The behavior parameters of the four agents are the same and are as follows:

- 336 observation space corresponding to 11 ray-casting forward distributed over 120 degrees.
- 3 ray-casts backward distributed over 90 degrees each detecting 6 possible object types, along with object's distance.
- The forward ray-casts contribute 264 state dimensions and backward 72 state dimensions over three observation stacks.

Agent Reward function:

- Accumulated time penalty is incremented by 1 when the ball enters the opponent's goal.
- Every fixed update is reset to 0 at the beginning of an episode.
- Award -1 when the ball enters the team's goal.

Discrete vector action space consists of three branched actions corresponding to forward, backward, sideways movements, as well as rotation. Float properties of the game include two components:

- Ball scale: which specifies the scale of the ball uniformly in all the three dimensions.
  - Default value: 7.5
  - Recommended minimum: 4
  - Recommended maximum: 10
- Physics aspect of the game objects
  - Default gravitational pull: 9.81

In our project we decided to stick with the default values for the aforementioned components.

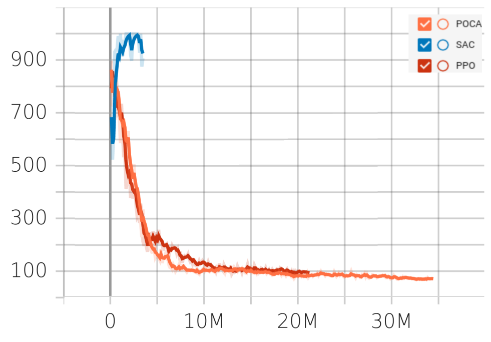


Figure 8: Environment Episode Length

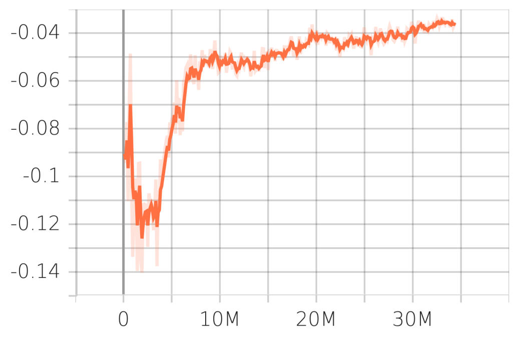


Figure 9: Environment Group Cumulative Reward for POCA

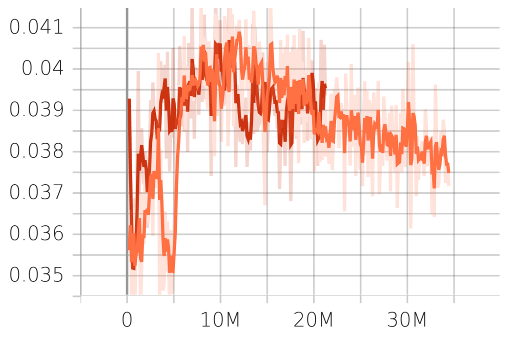


Figure 10: Policy Loss for POCA and PPO

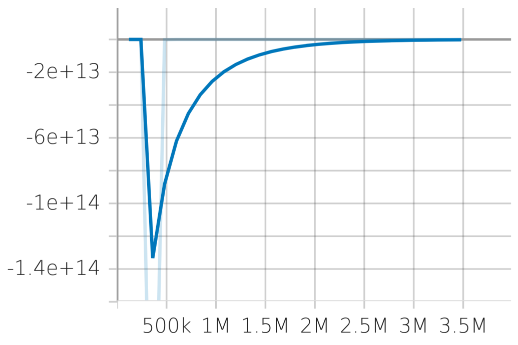


Figure 11: Policy Loss for SAC

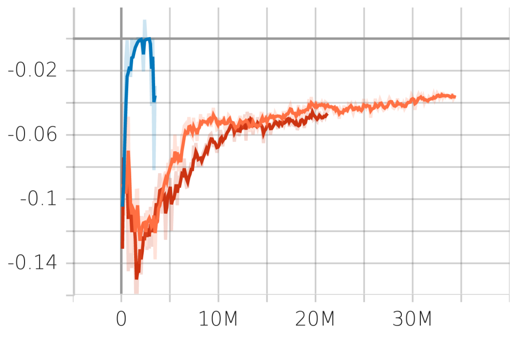


Figure 12: Policy Extrinsic Reward

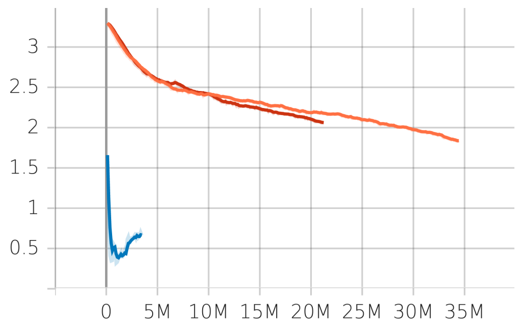


Figure 13: Policy Entropy

Algorithm	Number of Steps (Millions)	Wall Time (Hours)
POCA	34.44	54
PPO	21.24	45.3
SAC	3.48	57.5

Table 1: Training Steps Vs Clock Time

B. Model Training

We then moved to the training stage. As mentioned in previous parts, we trained using three algorithms parameterized as shown in figure 3 (PPO), figure 5 (SAC) and figure 7 (POCA). It is worth pointing out that PPO and SAC do not support receiving group rewards in multiagent settings and hence figure 9 only has group reward for POCA. We have not shown cumulative reward graph as all the rewards converge to zero for all the algorithms. Finally, the different approaches have been compared through graphs and discussed in the conclusion section. The horizontal axis on each of the presented graphs represents number of steps the training was performed for and the vertical axis represents respective quantity values. For each of the algorithms, we fixed the number of hidden layers to 3 and number of hidden units to 256. We use linear reduction in the learning rate starting with 0.001 initially.

Conclusion

The performance of POCA, PPO and SAC algorithms for a multiagent soccer game is compared in this paper. Since the environment corresponds to a zero sum adversarial game, all the rewards converge to zero. At the start of the training, the episodes take longer as the policy is mostly random and maximum number of steps are explored. But with time, the agents learn their policies and the episodes result in a goal earlier.[Figure 8]

SAC is very efficient for energy-based optimization techniques due to entropy factor regularization but we observed that it was the slowest in our environment settings. Although, SAC had better rewards but it being the very slow, we did not run it for more number of steps. Between POCA and PPO, POCA had somewhat better rewards than PPO.[Figure 12] It can be observed that PPO provides a good convergence and performance rate than other techniques but is sensitive to changes while POCA has a slightly more stable policy loss.[Figure 10]