PPO or SAC: A comparison between algorithms Progress Report

Kaan Atesel[1*] , Kursat Ahmet Akkaya[2*] , Eduin Hernandez [3†] , and Yigit Erkal [4*]

[*]Bilkent University, [†]National Yang-Ming Chiao-Tung University

Student ID: [1]21703694, [2]21703879, [3]22104094, [4]21601521

***Abstract*—Amongst the recent Deep Reinforcement Learning methods, Proximal Policy Optimization (PPO) and Soft Actor Critic (SAC) are algorithms that tackle the shortcomings of earlier methods: scalability, complexity, data efficiency, sample efficiency, and robustness. PPO offers a simpler versatile algorithm that could be trained with multiple samples in parallel, providing data efficiency. SAC offers sample efficiency by training on previous data and employs random behaviour to improve robustness. Our paper seeks to evaluate the trade-offs between employing these two algorithms in terms of their cumulative reward and policy entropy. We will argue that PPO is versatile in terms of providing good performance under different environments and tasks while SAC is an excellent algorithm to employ for complex tasks.**

I. Introduction

In the last few years, Deep Reinforcement Learning (RL) has become a popular subfield. By incorporating multiple hidden layers to optimize the policy and rewards, agents are able to make decisions without requiring engineers to fully design the state space, further allowing to solve more complex tasks. It has been widely used for robotics, video games, natural language processing, computer vision, just to name a few.

However, many early algorithms suffered from scalability, complexity, data efficiency, sample efficiency, and robustness. Amongst the recent deep RL methods, Proximal Policy Optimization (PPO) [1] and Soft Actor Critic (SAC) [2] algorithms tackle this shortcoming. The aim of PPO is to obtain similar performance to Trust Region Policy Optimization (TRPO) [3] while reducing its complexity and to provide data efficiency during the training. Conversely, SAC seeks to maximize the entropy, so that the actor can complete its task under different circumstance. It benefits from the exploration and becomes more robust, requiring even less samples for training.

Our paper seeks to evaluate the trade-offs between employing these two algorithms in terms of their cumulative reward and policy entropy. We argue that PPO is versatile in terms of providing good performance under different environments and tasks while SAC is suitable algorithm to employ for complex tasks. We will compare PPO and SAC in the following games: 3D Ball, Wall Jump, PushBlock, Hallway, Worm, Crawler and Walker using the assets and default training configuration provided by Unity ML-Toolkit Agent [4]. The details of these environments and tasks are provided in Sec. III, the results of training using PPO and SAC on these tasks in Sec. IV, and our conclusions in Sec. V.

II. Background

A. Proximal Policy Optimization

Proximal Policy Optimization (PPO) [1] is an on-policy Deep RL algorithm which switches extracting data from the interaction with the environment and optimizing the objective function via Stochastic Gradient Descent (SGD). It seeks to achieve both data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) [3] while reducing its complexity. Unlike earlier policy gradient methods, PPO enables batch gradient update and provides a better sample complexity. For many RL tasks and environments, PPO is sufficient as it generally performs well enough and has a stable learning.

In TRPO, the following objective function is maximized,

$$\max_\theta \widehat{\mathbb{E}}_t[r(\theta)\widehat{A}_t] \tag{1}$$

$$\text{s.t. } \widehat{\mathbb{E}}_t[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2}$$

$$\widehat{A}_t = \sum_{i=t}^{T-1} (\gamma\lambda)^{i-t}\delta_i \tag{3}$$

$$\delta_i = r_i + \gamma V(s_{i+1}) - V(s_i) \tag{4}$$

where π_θ is a stochastic policy, $\widehat{A}$ an estimator of the advantage function at time-step t, V the value iteration function, γ discount factor, λ Generalized Advantage Estimation (GAE) parameter, θ_{old} a vector of policy parameters before the update, $r_t(\theta)$ is the probability ratio between the old and new estimation, and $r(\theta_{old}) = 1$. (1) uses a KL divergence bound to ensure the policy updates are relatively small, however this is impractical to solve as consequence of the considerable number of constraints.

In a more general way, TRPO can be said to maximize the "surrogate" objective,

$$L^{CPI}(\theta) = \widehat{\mathbb{E}}_t[r(\theta)\widehat{A}_t] \tag{5}$$

where CPI is the conservative policy iteration. However, without its original constraint from (1), the policy update can become large, thus authors in [1] propose a modification that penalizes changes in $r_t(\theta)$ that are away from 1 as follows,

$$L^{CLIP}(\theta) = \widehat{\mathbb{E}}_t[min(r_t(\theta)\widehat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t]. \tag{6}$$

By using clipping, the KL constraint from TRPO is removed and simpler constraints are used, bounding the policy update between $[1-\epsilon, -1+\epsilon]$, a constraint more practical to compute. It is generally suggested to use a $\epsilon = 0.2$, which has numerically proven to be sufficient. This is the objective function which PPO maximizes, which is also referred to as pessimistic estimate of the performance of the policy due to its clipping. Clipping makes sure that the update does not distort the learning.

The PPO algorithm is defined in Algorithm 1. For each iteration, N actors collect data from T timesteps, making a total of NT timesteps worth of data. After this collection phase, the surrogate objective is constructed using this data and the updated policy model is computed through minibatch SGD or Adam for a span of K epochs.

Algorithm 1 PPO [1], Actor-Critic Style

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{old}}$ in environment for $T$ timesteps
        Compute advantage estimates $\widehat{A}_1$, ..., $\widehat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{old} \leftarrow \theta$
end for
```

B. Soft Actor Critic

Soft Actor Critic (SAC) [2] is an off-policy Deep RL algorithm rooted in the maximum entropy framework [5]. According to that framework, the actor seeks to maximize both the expected reward and the entropy. In brief, with this method, the actor tries to successfully complete its task while also behaving randomly as much as possible to benefit from exploration.

On-policy learning is known to provide better stability, but have a poor sample complexity as new samples are required for each update of the policy. With an increasing task complexity, a larger quantity of samples are necessary for an effective policy, which does not scale well. Off-policy learning attempts to use the past experiences, but do not perform well with higher complexity tasks, especially with continuous states and action spaces. SAC seeks to overcome this limitation by providing i) sample-efficient learning and ii) the benefits from entropy maximization for stability and robustness. This is done by including the following: i) an actor-critic structures that employs both a policy function neural network (NN) and a value function NN, ii) off-policy structure which allows the usage of past samples, iii) and entropy maximization.

Algorithm 2 SAC [2], Soft Actor-Critic

```
Input: $\theta_1, \theta_2, \phi$
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$
  $\mathcal{D} \leftarrow \emptyset$
  for each iteration do
      for each environment step do
          $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
          $\mathbf{s}_{t+1} \sim \rho(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
          $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{r}(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1}\}$
      end for
      for for each gradient step do
          $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
          $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
          $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
          $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
      end for
  end for
Output: $\theta_1, \theta_2, \phi$
```

In Maximum Entropy RL, the following objective function is maximized,

$$\pi^* = arg \max_\pi \sum_t \mathbb{E}_{(s_t, a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{7}$$

where α is the temperature parameter that establishes the importance of the entropy term $\mathcal{H}$. This objective function fosters exploration, but forfeits unfavorable actions and also takes advantages of equally rewarding actions.

SAC uses both a soft Q-function $Q_\theta(s_t, a_t)$ and a tractable policy $\pi_\phi(a_t|s_t)$, modeled with θ and ϕ NNs respectively. The soft Q-function are learned by minimizing the soft Bellman residual,

$$\begin{aligned} J_Q(\theta) = \mathbb{E}_{(s_t, a_t)\sim\rho_\pi}[\frac{1}{2}(Q_\theta(s_t, a_t) \\ -(r(s_t, a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\theta}(s_{t+1})}])^2] \end{aligned} \tag{8}$$

where γ is the discount factor so the expected reward and entropy are finite, $\bar{\theta}$ the exponential moving average of the weights, and V the value iteration function.

The policy parameters are learned by minimizing the following objective function,

$$\begin{aligned} J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}, \epsilon_t\sim\mathcal{N}}[\alpha log\pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) \\ -Q_\theta(s_t, f_\phi(\epsilon_t; s_t))] \end{aligned} \tag{9}$$

$$f_\phi(\epsilon_t; s_t) = a_t \tag{10}$$

Environment	NN Layers		Hidden Units		
	2	3	128	256	512
3D Ball Hard	✓		✓		
Crawler		✓			✓
Hallway	✓		✓		
Push Block	✓			✓	
Walker - PPO		✓			✓
Walker - SAC		✓		✓	
Wall Jump					
Worm		✓			✓

Table I: Neural Network Settings for each algorithm trained in the provided environment. Both PPO and SAC default configurations for each environment match with the exception of "Walker".

where ϵ_t is the input noise vector sampled from a distribution and $\mathcal{D}$ the replay pool.

α is updated by minimizing the following function,

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi_t}[-\alpha log\pi_t(a_t|s_t) - \alpha\bar{\mathcal{H}}]. \quad (11)$$

The full SAC algorithm is provided in Algorithm 2, using two soft Q-functions with θ_1, θ_2 trained to optimize (8), a policy with parameter ϕ trained on (9), and a temperature parameter α on (11).

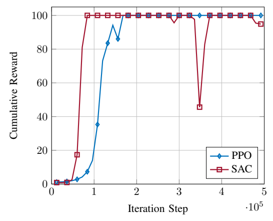
III. Experimental Setup

For evaluating these actor-critic algorithms, we will be using the Unity Engine [6], training the agents with the corresponding algorithms. In particular we will be using ML Agent Toolkit [4] to configure and train these agents on the following learning environments[1]:

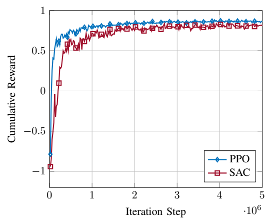
- 3D Ball Hard
  - Description: Agents balance a ball on top of their head.
  - Task: Balance ball on head until the time limit for the episode is reached.
  - Actions (2 continuous variables): Values containing i) x-rotation and ii) z-rotation angle for the agent.
  - Observed States (5 variables): Rotation values of the agent and ball position.
  - Rewards: +0.1 for every step the agent retains the ball on its head, -1.0 if it falls off.
  - Expected Reward[2]: 100
- Crawler
  - Description: An agent having 4 arms and a forearm for each arm.
  - Task: Move towards the reward target without falling.
  - Actions (20 continuous variables): Targeted joint rotation values.
  - Observed States (172 variables): Position, rotation, velocity, and angular velocity of each body segment; acceleration and angular acceleration of body.
  - Rewards: Matching the i) body speed and ii) head direction of the agent with respect to the goal's speed and direction. Both reward values are normalized between 0 and 1 and the overall reward is the product between the two.
  - Expected Reward[2]: 3000
- Hallway
  - Description: A room with two doors an agent can enter, each with a mark opposite to the other door.
  - Task: Pick the correct door to enter based on the answer mark provided in the middle of the room and the marks provided at each door.
  - Actions (discrete branches): rotation (left or right) and motion (forward or backwards). Only one action can be performed at any given time.
  - Observed States (30 variables): Sensors that return the detected objects type (wall, door, or mark).
  - Rewards: +1 for entering correct door, -0.1 for entering incorrect goal, -0.0003 for each step.
  - Expected Reward[2]: 0.7
- Push Block
  - Description: A domain where the agent is able to push a block.
  - Task: Push the block towards the goal.
  - Actions (discrete branches): rotation (left or right), movement (forward, backwards, left, or right), or be idle. Only one action can be performed at any given time.
  - Observed States (70 variables): Sensors that return the detected objects (wall, goal, or block).
  - Rewards: +1.0 for achieving goal, -0.0025 for each step.
  - Expected Reward[2]: 4.5
- Walker
  - Description: Humanoid agent with multiple connected body sections.
  - Task: Move towards the reward target without falling.
  - Actions (39 continuous variables): Targeted joint rotation and strength values.
  - Observed States (243 variables): Position, rotation, velocity, and angular velocity of each body segment; and goal direction.
  - Rewards: Matching the i) body speed and ii) head direction of the agent with respect to the goal's speed and direction. Both reward values are normalized between 0 and 1 and the overall reward is the product between the two.
  - Expected Reward[2]: 2500
- Wall Jump
  - Description: A domain where the agent is able to push a block and jump.
  - Task: Reach the goal, and if there is an obstacle, jump over it if it small, otherwise use the block to jump over it.
  - Actions (discrete branches): rotation (left, right, or no rotation), depth movement (forward, backwards, or no depth movement), horizontal movement (left,

[1]Further details of the environments are provided by [4] in their github repo.

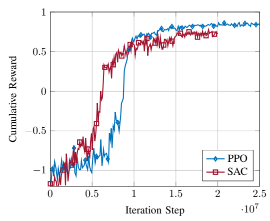
[2]The expected reward is the average reward achieved by the fully trained benchmark agent model provided by the ML-Agent Toolkit at which the agents are able to successfully perform the task.



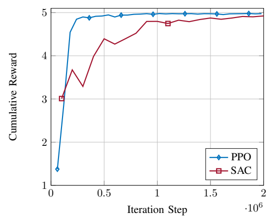
(a)



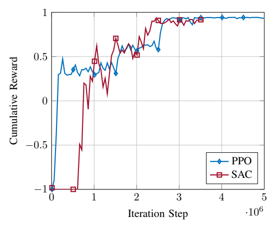
(b)



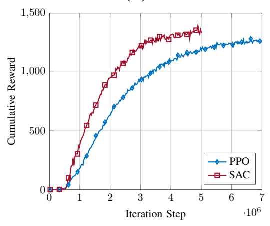
(c)



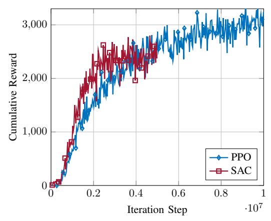
(d)



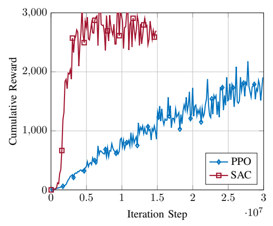
(e)



(f)

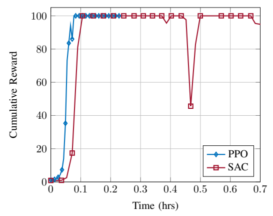


(g)

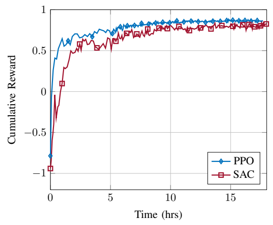


(h)

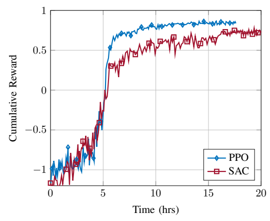
Fig. 1: Cumulative Reward vs step for PPO and SAC in various learning environments: (a) 3D Ball Hard, (b) Wall Jump - Small Wall, (c) Wall Jump - Big Wall, (d) PushBlock, (e) Hallway, (f) Worm, (g) Crawler, and (h) Walker. For all cases, higher values are better.



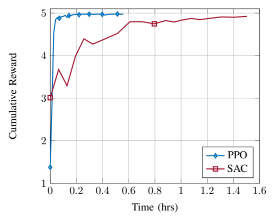
(a)



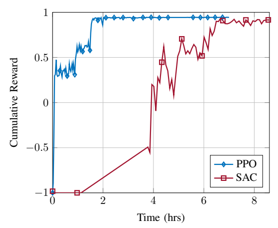
(b)



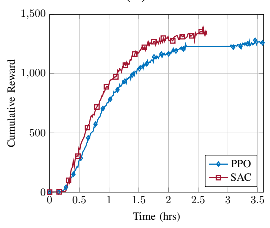
(c)



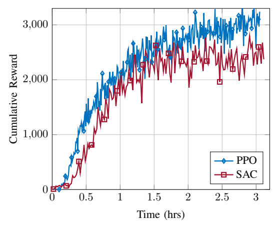
(d)



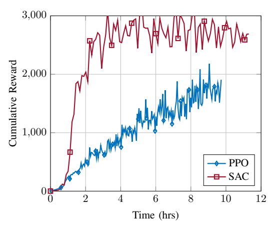
(e)



(f)



(g)



(h)

Fig. 2: Cumulative Reward vs wall time for PPO and SAC in various learning environments: (a) 3D Ball Hard, (b) Wall Jump - Small Wall, (c) Wall Jump - Big Wall, (d) PushBlock, (e) Hallway, (f) Worm, (g) Crawler, and (h) Walker. For all cases, higher values are better.

right, or no horizontal movement), and jump (jump or stay grounded). Multiple actions can be performed at a given step, so long as they belong to separate discrete action branches.
  - Observed States (74 variables): Sensors that return the detected objects (obstacle, goal, block, out-of-bounds); agent's world's position; and agent's jump state.
  - Rewards: +1.0 for achieving goal, -1.0 for falling off the stage, -0.0005 for each step.
  - Expected Reward[2]: 0.8
- Worm
  - Description: An agent with 4 body segments.
  - Task: Move towards reward target.
  - Actions (9 continuous variables): Targeted joint rotation values.
  - Observed States (64 Variables): Position, rotation, velocity, and angular velocity of each body segment; and acceleration and angular acceleration of body.
  - Rewards: Matching the i) body speed and ii) head direction of the agent with respect to the goal's speed and direction. Both reward values are normalized between 0 and 1 and the overall reward is the product between the two.
  - Expected Reward[2]: 800

The default configurations provided by the ML-Agent Toolkit will be used for the training of PPO and SAC under the different environments. The default configurations for the NN are provided in Tab. I.

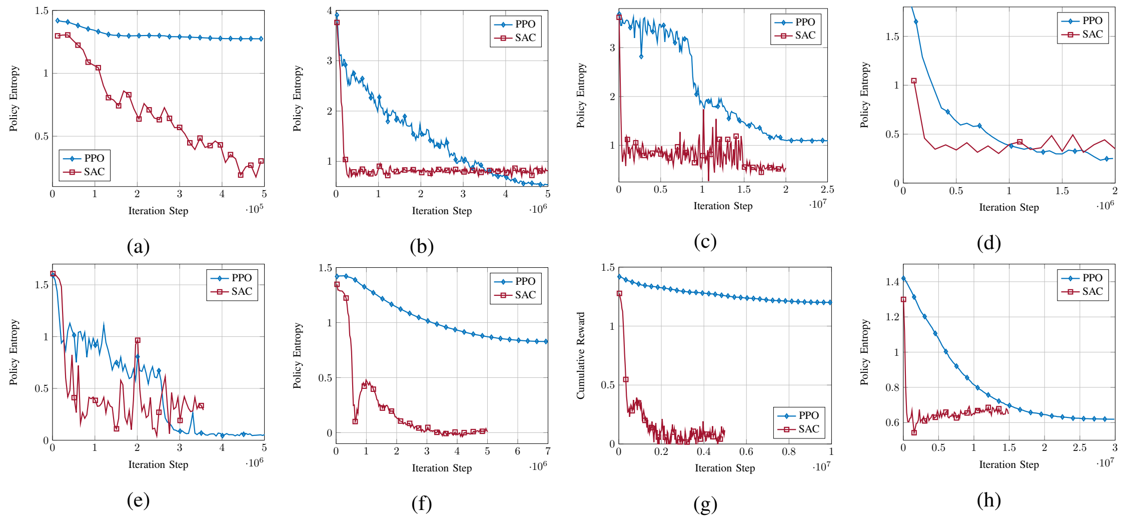


Fig. 3: Policy Entropy for PPO and SAC in various learning environments: (a) 3D Ball Hard, (b) Wall Jump - Small Wall, (c) Wall Jump - Big Wall, (d) PushBlock, (e) Hallway, (f) Worm, (g) Crawler, and (h) Walker

IV. Results

The statistical results are provided in terms of the cumulative reward per iteration step, cumulative reward per wall time, and policy entropy per iteration step in Figs. 1, 2, and 3, respectively. The cumulative reward is the average cumulative episode reward over all agents, which for a successful training, should be increasing. Cumulative reward is provided in terms of both iteration step and wall time to show the speed the actor learns with different units of time and the trade-off from the algorithm's complexity. The policy entropy measures how random the decisions of the model are and for a successful training, should be slowly decreasing.

The environment for 3D Ball Hard is a more challenging task than 3D Ball, but since the learning behaviour was the same, only the results for 3D Ball Hard were provided. In the case for Wall Jump - Small Wall and Big Wall, the difference in the task lied in whether the agent had to jump directly over the wall (small) or push a block to jump over the wall (big). This latter environment is a combination of two tasks that are learned in Wall Jump Small (jumping over the obstacle to reach goal) and Push Block (pushing the block towards goal, but the agent cannot jump). The environments for Worm, Crawler, and Walker provided an ascending complexity for the same tasks by providing more body segments to the agent, thus increasing the degrees of freedom, observable states, and actions.

The figures all show that the algorithms are able to learn the tasks, however, the learning takes different timesteps depending on the complexity. For simpler tasks such as Wall Jump - Small Wall and Push Block, PPO learns faster and attains a higher cumulative reward than SAC. For slightly higher difficult tasks such as Hallway and Wall Jump - Big Wall, SAC is able to learn faster than PPO in terms of iteration steps, but PPO still maintains a slightly higher cumulative reward and in terms of the wall time, PPO provides either an equivalent or faster training. By further increasing the complexity of the tasks as in Worm, Crawler, and Walker, we notice that PPO does not learn as fast as in other environments, decreasing its performance and creating a bigger gap with SAC. PPO requires a significant number of more steps to achieve the same rewards as SAC, if not more as in Figs. 1c, 1g, however, Figs. 2c, 2g suggests that using previous samples to learn causes SAC's algorithm to take roughly the same wall time as just training on more samples with PPO.

The highest complexity environment, Walker, shows an overwhelming advantage of using previous samples to learn with SAC both in Figs. 1h and 2h instead of new samples with PPO.

For all the environments, Fig. 3 show how the randomness for both SAC and PPO change over iteration step, with SAC decreasing faster due to its training on previous samples.

V. Discussion and Conclusion

Despite SAC's ability to decrease the samples required for training by using previous data, it comes at the cost of slowing the training in real time. For most of the training in environments presented, using PPO would have been sufficient to achieve a satisfactory performance, if time complexity is an issue. However, for higher complexity tasks that involve a significant size for the observe states and actions that can be taken, it would be better to rely on SAC instead. Also, SAC can also be consider for cases where there is a deficiency in samples or a sample limit.

It is also important to note that for all the plots provided above, only 1 trial of training is performed for each actor-critic and task. To provide a better and more fair comparison, more trials should be gathered and either average or choose the best among them for the plot comparison.

We conclude this investigation by providing our trained models, training logs, and videos which are all readily available at our github repo and google drive.

REFERENCES

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[2] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[3] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[4] A. Juliani, "Introducing: Unity machine learning agents toolkit," *Arthur Juliani [ ]:– : https://blogs. unity3d. com/2017/09/19/introducing-unity-machine-learning-agents*, 2017.

[5] B. D. Ziebart, A. L. Maas, J. A. Bagnell, A. K. Dey *et al.*, "Maximum entropy inverse reinforcement learning." in *Aaai*, vol. 8. Chicago, IL, USA, 2008, pp. 1433–1438.

[6] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar *et al.*, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2018.