Learning to walk a quadruped robot: continuous control with reinforcement learning

Himanshi Allahabadi - 1320246
Department of Mathematics and Computer Science

Contents

Chapter 1

Introduction

1.1 Unity Game Development Platform

Unity has been a popular choice as a platform for game development since 2005, owing to the flexibility of a general-purpose engine which supports 2D and 3D games and simulations, as well as augmented and virtual reality. Unity comes with the Nvidia PhysX 3.3 physics engine [3]. Together with its features for simulating detailed 3D environments and forces such as torque, this makes for an ideal platform for solving continuous control tasks.

1.2 ML-Agents

The Unity ML Agents Toolkit provides the framework needed to use the simulated environment to obtain observations, and perform actions through an Agent script. The actions to be performed by the robot are provided to the Agent via a Brain. The Brain can be configured to be heuristic, or in our case, learning-based. This means that we train reinforcement learning models to obtain these actions, using well-designed reward functions.

1.3 Project Description

For this project, we use a simulation of a 4-legged spider robot. Each of the legs has 2 joints. At every time step, the environment outputs a 129-dimensional state vector. The action space is continuous and the agent outputs a 20-dimensional vector of continuous values. 10 identical agents are trained in parallel. The goal is to walk, as efficiently as possible, towards a dynamic target which can be placed at any location relative to the robot. The agent is rewarded for altering its direction to face the target and moving towards it. Conversely, it is penalized for facing away, and moving away from the target. It is also penalized for taking a long time to reach the target, so that it learns to walk faster. We use two different algorithms to solve this problem (chapter: Methods) and test their accuracy (chapter: Training Results). Finally, we compare and discuss the results (chapter: Discussion).

Chapter 2

Methods

2.1 Reinforcement Learning

The notion of reinforcement learning is described by Pavlov as the strengthening of behaviour patterns in animals, in response to some reinforced stimuli. The effect of reinforcing events upon the tendency to select actions, is called the Law of Effect[]. In the context of computer science, the goal of reinforcement learning is optimal decision-making by estimating the value of actions and/or states .
However, this is not strictly necessary for solving reinforcement learning problems. Methods such as genetic programming and simulated annealing do not estimate value functions. Instead, they apply multiple static policies, each interacting over an extended period of time with a separate instance of the environment. Such evolutionary methods are good when the policy space is small, or can be structured so that good policies are easily found, or if we do not care about how our method performs on time. In continuous control problems, the search space for good policies is large due to the continuous nature of action space.

We train agents to maximise the reward objective, which can be computed empirically as the sum of discounted returns:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \tag{2.1}$$

Using the value of action-state pairs, an action-value function can be calculated as:

$$q_\pi(s, a) = E_\pi[G_t|s, a] \tag{2.2}$$

Policy-based methods make use of this function to directly obtain an optimal policy:

$$\pi(s) = argmax_{a \in A(s)} q(s, a) \tag{2.3}$$

The premise of policy-based methods is simple, can learn true stochastic policies, and is well-suited for problems in continuous spaces.

For this project, we measure the performance of our agent after training with both approaches: policy-based and off-policy. In the rest of the report, we describe the methods used and plot the relevant training statistics.

2.2 Proximal Policy Optimization

PPO is a state-of-the-art RL algorithm which showed considerable improvement over other policy-gradient methods, for Atari game playing as well as continuous control problems such as robotic

locomotion. [5]. It uses importance sampling to estimate the gradient of current policy using the old policy gradient. With the conjecture that the old and current policies are reasonably close to each other, function approximation is used to compute these gradients. The algorithm is provided in figure fig:ppo.

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **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

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}) \right] \right]$$

end for

Figure 2.1: Proximal Policy Optimization [1]

2.3 Off-Policy Methods

As discussed in the previous section, the paradigm of policy-based methods tries to directly optimize the quantity of interest, while function approximation ensures that the training is stable. A disadvantage is that it require on-policy learning, which makes it sample-inefficient. By contrast, off-policy learning can learn through episodes sampled from the same or identical environments, through experience replay.

2.4 Entropy-Regularized RL

The connection between inference and control has been reported in previous works by [7] and [8]. By the general definition, entropy refers to disorder or uncertainty, and is correlated with exploration in reinforcement learning. The agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep [8]. This encourages exploration of the agent, and helps prevent early convergence to suboptimal policies. The resulting policies are also said more robust against changes or disturbances in the environment[6], which is relevant to our problem statement.

2.5 Soft Actor Critic Method

In contrast with PPO, SAC is off-policy, which means it can learn from experiences collected at any time during the past. Experiences are collected in an experience replay buffer and randomly drawn during training. This makes it significantly more sample-efficient than PPO. However, it tends to require more model updates, and may hinder performance by exploring more than exploiting the known high-reward action sequences[4]. In a way, SAC is an abstraction over a pure stochastic policy optimization (as observed in TRPO/PPO), and a purely deterministic approach (as in DDPG).

Based on the maximum entropy reinforcement learning framework discussed in Section 2.5, SAC maximizes the trade-off between expected return and entropy in the policy:

$$J(\pi) = E_{\pi}[\sum_{t} R(s_t, a_t) + \alpha H(\pi(\cdot|s_t))] \tag{2.4}$$

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. H is the entropy of policy π at state s_t, given by the negative logarithm of the probability mass function [2]. Mathematically,

$$H(\pi(\cdot|s_t)) = -\sum_{a_t} \pi(a_t, s_t) \times \log(\pi(a_t|s_t)) \tag{2.5}$$

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
$\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
    $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
  end for
end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 2.2: Soft Actor Critic [2]

Chapter 3

Training Results

3.1 Cumulative Reward

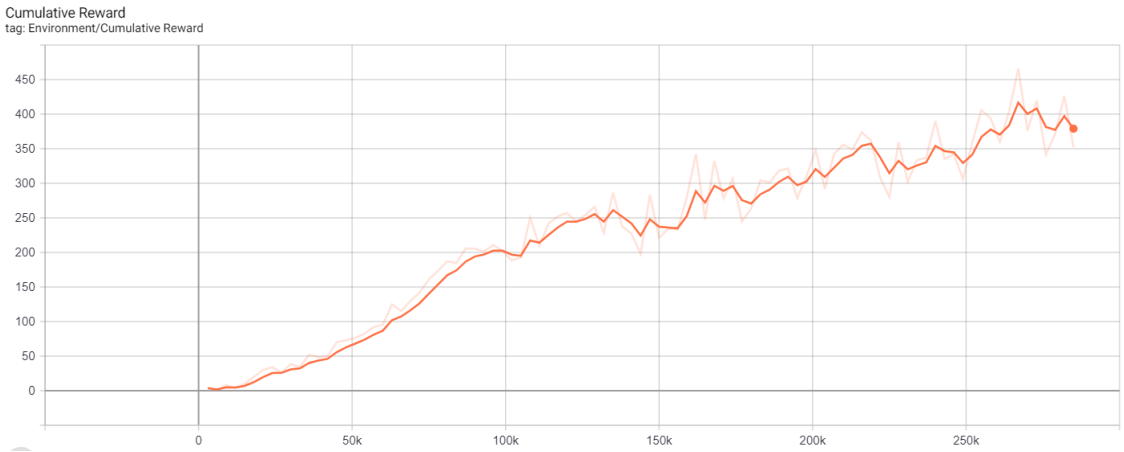

Figure 3.1: Cumulative Reward: PPO

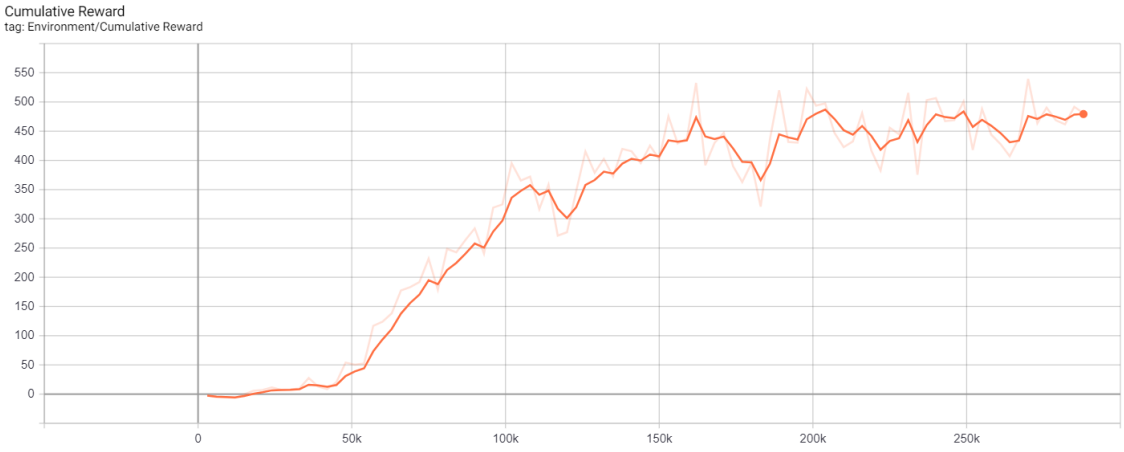

Figure 3.2: Cumulative Reward: SAC

3.2 Entropy

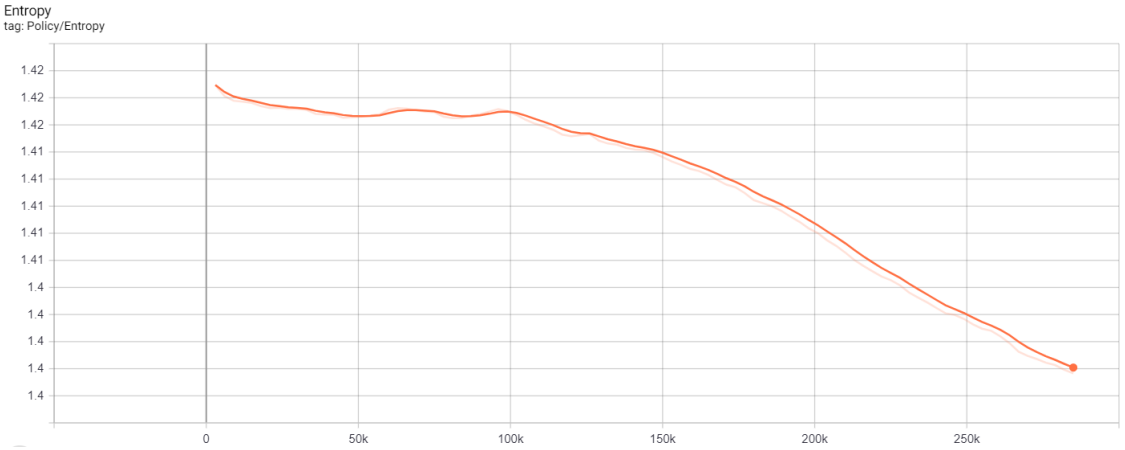

Figure 3.3: Entropy: PPO

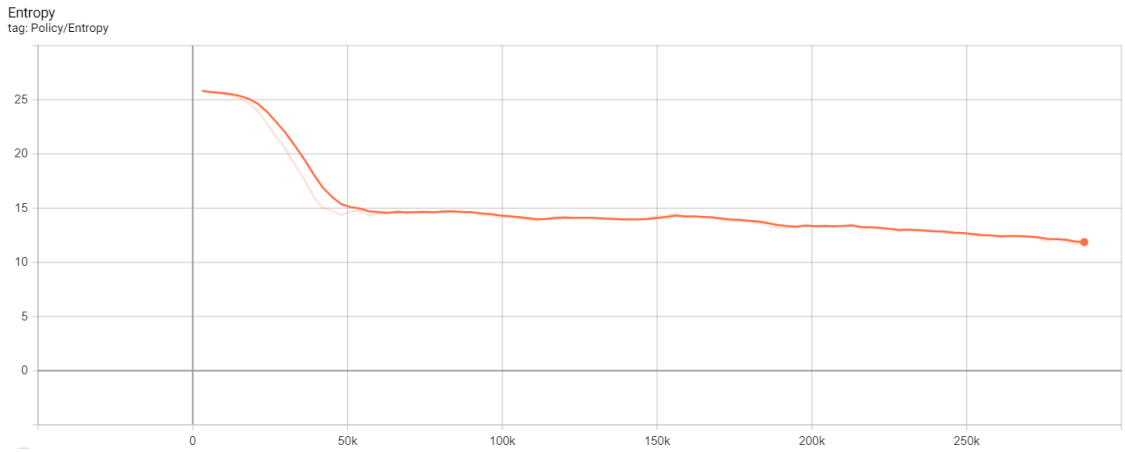

Figure 3.4: Entropy: SAC.

3.3 Policy Loss

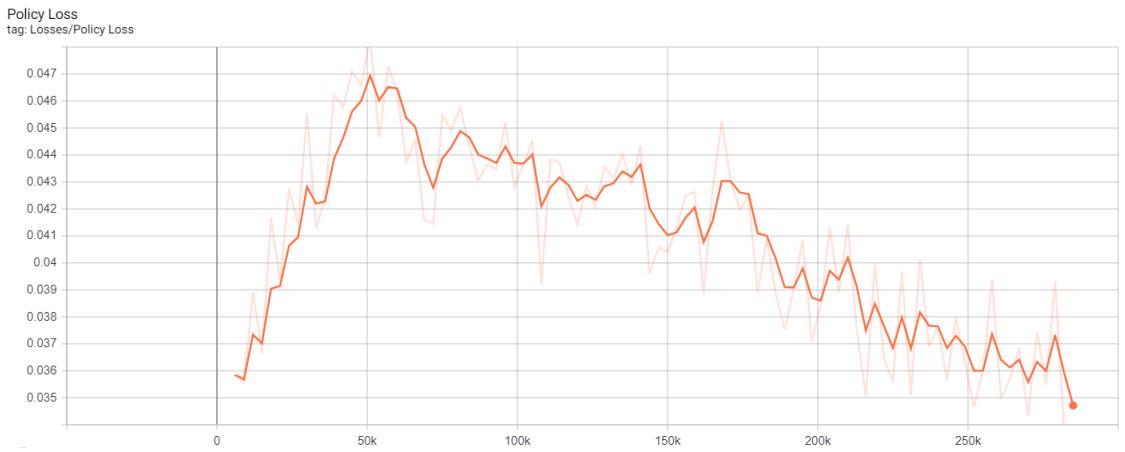

Figure 3.5: Policy Loss: PPO

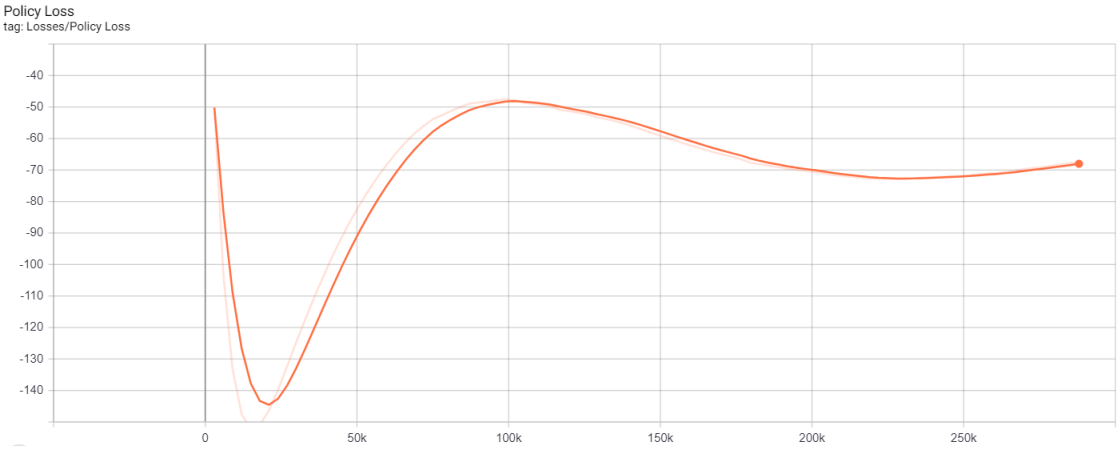

Figure 3.6: Policy Loss: SAC

Chapter 4

Discussion and Conclusion

The cumulative reward is calculated as the mean of total rewards per time step for 10 agents. From fig:cumulativePPO and fig:cumulativeSAC, we can see that SAC reaches the baseline reward of 400 in <150k time steps. PPO takes >250 time steps to reach the same reward. Evidently, SAC is more sample-efficient than PPO. The SAC curve suggests lower rewards the beginning, until it reaches <75k time step, which is when it outperforms PPO. This is due to high exploration rate in the beginning, induced by entropy regularization (eq. 2.4).

During initialization, small amount of noise is added to encourage random actions, which slowly decreases as agents earn more cumulative reward. This SAC has a much higher range of entropy due to the entropy term in the policy updates.

The policy loss metric indicates how much the policy changes from one step to another. We can see in fig:policylossPPO that the policy changes are increasing until step 50k, due to the noise we added to encourage agents to explore the action space and not converge to a suboptimal policy. As the agent starts earning more cumulative reward, the the policy gets more and more stable. In fig:policylossSAC, we see that the magnitude of policy loss increases for 25k steps, and as we see from fig:entropySAC, the system has the highest entropy, more or less constant for the first 25k steps. After this point, the cumulative reward earned by agents starts increasing, and the entropy value is accordingly decreased. The magnitude of policy loss also decreases and eventually stabilises.

Notice that policy loss for PPO is in the range [0, 1] while the SAC policy loss has much higher magnitudes. This is due to the fact that SAC can learn in an off-policy manner, while gradient updates in PPO rely on the fact that the policies from one timestep to the next are close to each other: fig:ppo.

From the results, we derive the conclusion that SAC outperforms PPO for the continuous control task of learning to walk. It attains higher cumulative reward as well as faster convergence due to sample efficiency.

Bibliography

[1] Joshua Achiam. Advanced policy gradient methods, October 2017. v

[2] Tuomas Haarnoja. Soft actor-critic algorithms and applications. January 2019. vi, vi

[3] Greg Kumparak. Unity 5 announced with better lighting, better audio, and "early" support for plugin-free browser games. March 2014. iii

[4] Kyungjae Lee. Tsallis reinforcement learning: A unified framework for maximum entropy reinforcement learning. February 2019. v

[5] John Schulman. Proximal policy optimization algorithms. August 2017. v

[6] Wenjie Shi. Soft policy gradient method for maximum entropy deep reinforcement learning. September 2019. v

[7] Emanuel Todorov. General duality between optimal control and estimation. December 2008. v

[8] Marc Toussaint. Robot trajectory optimization using approximate inference. June 2009. v, v