COMP 579 Final Project - Group 050

William Zhang
McGill University
260865382

Ding Ma
McGill University
260871301

Michael Li
McGill University
260869379

Mikhail Baranchuk
McGill University
260866078

1 Introduction

In the last decade, reinforcement learning (RL) has been extensively studied for solving continuous control tasks by allowing agents to learn a policy that maximizes cumulative reward through interactions with the environment [2]. This project focuses on the Hopper (MuJoco) task from the OpenAI Gym environment where a hopper with 3 joints and 4 body parts has to hop forward as fast as possible without falling on the floor. This is a continuous control task, because both action and state spaces are continuous. In this report, we will discuss the various baselines and state-of-the-art approaches we investigated to solve this problem.

2 Methods

The methods implemented for solving the Hooper-v2 (Mujoco) task will be discussed in this section. An emphasis has been made on gradient-based policy search methods where we try to learn a *parametrized policy* π. We follow the notation from [10] where the policy's parameter vector is denoted by $\theta \in \mathbb{R}^{d'}$. Hence, $\pi(a|s,\theta) = Pr(A_t = a|S_t = s, \theta_t = \theta)$ stands for the probability that action *a* was taken at time step *t* given where the environment is in state *s* and the policy π has parameters θ. We also learn a value function whose parameter vector is denoted by $\mathbf{w} \in \mathbb{R}^d$. These policy gradient methods try to learn the best policy parameters θ that optimize some scalar performance measure $J(\theta)$. Hence, these methods seek to *maximize* $J(\theta)$ using *gradient ascent*. An update step for the parameters of the policy is illustrated in equation (1):

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{1}$$

where $\nabla J(\theta_t) \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure J with respect to the policy parameter θ_t [10].

The reason why we chose to focus on policy gradient methods is because they can be naturally extended to continuous state and action spaces. Furthermore, optimizing the policy directly solves the true objective of the problem as opposed to learning a policy off an estimated value function. Often, policies are easier to learn than value functions or the environments' dynamics. Finally, policy gradients allows to learn stochastic policies which is ideal in the case that the optimal policy is stochastic rather than deterministic. The continuous policy parameterization allows the action probabilities to change smoothly as a function of the learned parameter as opposed to the case of $\epsilon - greedy$ where an arbitrary change in the estimated action values may result in a drastic change in the action probabilities, because a different action might have the maximal value [10]. Consequently, policy gradient methods benefit from stronger convergence guarantees than action-value methods due to the smoothness of $J(\theta)$ with respect to θ.

The policy gradient theorem [11] establishes for an episodic task that any differentiable policy $\pi_\theta(s,a)$, let d_0 be the starting distribution over states in which we begin an episode. Then, the policy gradient of $J(\theta) = \mathbb{E}[G_0|S_0 \sim d_0]$ is:

$$\nabla_\theta J(\theta) = E_{\pi\theta}\left[\sum_{t=0}^{T} \gamma^t q_{\pi_\theta}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t, S_t) | S_0 \sim d_0\right] \tag{2}$$

where

$$\begin{aligned} q_\pi(s,a) &= E_\pi\left[G_t | S_t = s, A_t = a\right] \\ &= E_\pi\left[R_{t+1} + \gamma q_\theta(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \end{aligned} \tag{3}$$

This allows the gradients of the policy to be independent of the Markov Decision Process (MDP) environment dynamics and sampled through interactions with the environment.

2.1 REINFORCE: Monte Carlo Policy Gradient with Baseline

The baseline method that we tried was the classic REINFORCE algorithm. Recall that the goal is to approximate $\nabla J(\theta_t)$ via samples whose expectation is proportional to the actual gradient of $J(\theta)$. The sample gradients only need to be proportional since any constants can be absorbed by the step size parameter α in equation (1). REINFORCE uses the complete discounted return from time t which includes all future rewards up until the end of the episode discounted by γ. Therefore, REINFORCE is a Monte Carlo algorithm. Discounting helps reduce variance as it improves the estimate of $Q^{\pi}(s, a)$, because an action might have more (or less) influence on reward nearby than later in the episode. Hence, the discount factor γ is a hyperparameter that needs to be tuned according to the task. The algorithm below represents the REINFORCE algorithm we've implemented as a baseline for the Hopper challenge.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^{t} \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

In order to reduce variance and increase sample efficiency, a baseline b is used to evaluate the action value of doing action a in state s at time step t. The baseline could be any function or random variable as long as it does not vary with respect to action a. [13] have shown that the policy gradient theorem remains valid, because the expected value of adding the baseline is 0. However, adding this baseline has a large effect on variance reduction and thus faster learning. In our implementation, we chose the baseline to be an estimate of the state-value, $\hat{v}(S_t, \mathbf{w})$ where $\mathbf{w} \in \mathbb{R}^d$ is the weight vector of a multilayer perceptron. $\mathbf{w}$ is learned through minimizing the mean squared error between the discounted return from time step t (e.g. the Monte Carlo return), G, and its estimated value of the state S_t, $\hat{v}(S_t, \mathbf{w})$.

The advantage estimate δ is the difference between the discounted return from time step t, G, and the baseline $\hat{v}(S_t, \mathbf{w})$. Intuitively, δ tells us how good the current action is compared to the average (i.e. what is its advantage?). The update step of the parameters of the policy on the last line is proportional to the product of the advantage estimate with the gradient of the log probability of taking action a in state s at time step t. $\nabla ln\pi(A_t, S_t, \theta)$ stands for the direction in the parameter space that most increases the probability of taking action A_t on future visits to state S_t. Therefore, intuitively, the update increases the parameter θ in this direction proportional to how the return resulted from action A_t, G, was better or worse than the average return from that state given by $\hat{v}(S_t, \mathbf{w})$). Therefore, we want the parameter to move in the direction that favor actions that yield higher than the average return. α^{θ} and α^{w} are the learning rate for the approximate policy and state-value functions respectively. The parameter updates for $\mathbf{w}$ and θ are done at the end of each episode. Since the return G is not a bootstrap estimate, we get a completely unbiased policy gradient estimate. In order to adapt to continuous action space, the policy is parameterized to learn statistics of the probability distribution. In fact, the policy is defined as a normal probability density over a real-valued scalar action, with mean and standard deviation given by the function approximators that take the state as an input.

$$\pi(a|s, \theta) \doteq \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \theta))^2}{2\sigma(s, \theta)^2}\right)$$

The policy's parameter vector θ is divided into two parts, $\theta = [\theta_\mu, \theta_\sigma]$. The first part is used to approximate the mean and the second part the standard deviation. The mean is approximated using a standard 2 layers multi-layer perceptron (MLP). The standard deviation must always be positive, hence it is approximated as the exponential of a linear function.

$$\mu(s, \theta) \doteq \theta_\mu^\top x_\mu(s)$$

and

$$\sigma(s, \theta) \doteq \exp\left(\theta_\sigma \top x_\sigma(s)\right)$$

Using this formulation, policy gradient methods can be safely applied to select real-valued actions.

2.2 One-step Actor-Critic

In the REINFORCE algorithm, the advantage estimate δ is G - $\hat{v}(S_t, \mathbf{w})$. In fact, G was essentially used to estimate the action-value $Q^{\pi}(s, a)$ from a *single* roll out. However, this yields a high variance estimate since it is based on a single sample. In

order to reduce variance, One-step actor critic uses function approximation, e.g. the one-step TD as a target, hence $\delta = R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$. The estimated value of the *second* state, discounted and added to the reward represents the one-step return, $G_{t:t+1}$. This is a better way to estimate the return and assess the action. [8] have shown that this can be extended to *n*-step TD where *n* is a hyperparameter indicating the number of steps to look ahead. Although this bootstrapped estimate introduces bias towards the previous policy and could lead to instability, it yields lower variance and more efficient learning due to faster updates (at each step). This one-step actor-critic is now fully online and incremental where the states, actions, and rewards are processed as they occur and are never revisited [10]. Therefore, the parameter updates for $\mathbf{w}$ and θ are done at at each step. Methods that learn approximations to both policy and value functions are called *actor-critic methods*, where the *actor* is a reference to the learned policy, and the *critic* refers to the value function [10]. Note that in the online scenario $\hat{v}(S_{t+1}, \mathbf{w}) \doteq 0$ if S_{t+1} is terminal. Although one-step Actor-Critic is an online algorithm, we decided to update at the end of each episode as we found that doing so is way more stable than updating the parameters at every step.

2.3 Generalized Advantage Estimation

The advantage estimate is critical to policy gradient methods as it dictates how methods will learn and perform. The one-step TD yields high-bias, therefore we've implemented the state-of-the-art advantage function estimator called Generalized Advantage Estimation (GAE) [9]. GAE is concerned with producing an accurate estimate $\hat{A}_t$ of the discounted advantage function $A^{\pi,\gamma}(s_t, a_t)$ which is used to estimate the policy gradient estimator in the following form:

$$\hat{g} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{\inf}\hat{A}_t^n \nabla_\theta log\pi_\theta(a_t^n, s_t^n) \qquad (4)$$

The hyperparameters γ and λ allows us to adjust bias-variance trade-offs and smoothly interpolate between a high or low-bias estimator ($\lambda = 0$ or $\lambda = 1$). GAE can be applied to various RL algorithms in both the online and batch setting. In our experiment, we investigated the performance of the REINFORCE algorithm with and without GAE.

2.4 Soft Actor-Critic

Although on-policy methods tend to be more stable, they require a notorious amount of samples, because each sample is discarded after every gradient computation [3]. To improve sample efficiency, off-policy policy gradient methods such as DDPG [6] and SAC [3] sample from a replay buffer [7] thus allowing samples to be reused. In addition, the replay buffer helps address the assumption that samples are independently and identically distributed in neural network optimization. In fact, sampling from a replay buffer minimizes correlation between samples. However, implementing Q-learning using neural networks often leads to divergence and unstability for off-policy methods, especially due to the bias introduced through *boostrapping* [6]. In order to solve this issue, a copy of the actor-critic, whose weights track those of the learned networks, is used to compute the target values [6].

Soft Actor-Critic [3] is a state-of-the-art off-policy policy gradient algorithm for continuous tasks. It optimizes the expected return regularized by the entropy, a measure of the randomness in the policy:

$$J(\pi) = \sum_{t=0}^{T}\mathbb{E}_{(s_t,a_t)\sim\rho_\pi}\left[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right] \quad (5)$$

where $\mathcal{H}$ is the policy entropy and α is the temperature controlling the trade-off between the two objectives. Maximizing entropy means more exploration and focusing on the expected return results in exploitation. This reflects the exploration-exploitation trade-off [10]. We have implemented automatic entropy tuning (AET) [4] which means that the α hyperparameter is learned during training, therefore eliminating the need to tune this crucial hyperparameter. An exploration phase is done at the beginning of each training to prevent the policy from prematurely converging to a bad local optimum [3]. Using function approximators, we approximate the soft Q-function and the policy function. This avoids having to run *policy iteration* and *evaluation* until convergence at every step. Instead, the networks are optimized using stochastic gradient descent. The soft Q-function is modeled using a multilayer perception while the policy is modeled as a squashed Gaussian where a neural network is used to approximate the mean and the standard deviation parameters [3]. The hyperparameters can be found in the *Hyperparameters* section in the Appendix. In our implementation, SAC learns a policy π_θ along with two Q-functions Q_{ϕ_1} and Q_{ϕ_2} to mitigate the positive bias [5] and an

identical copy of this actor-critic is created to generate targets (denoted $Q_{\phi_{targ,1}}$, $Q_{\phi_{targ,2}}$ and $\pi_{\theta_{targ}}$). The Q-functions are trained to minimize the soft Bellman residual [3]:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}[\tfrac{1}{2}(\min_{j=1,2} Q_{\theta_j}(s_t,a_t) - \hat{Q}(s_t,a_t))^2] \quad (6)$$

where

$$\hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma\, \mathbb{E}_{s_{t+1}\sim p}[\min_{j=1,2} Q_{\phi_{\text{targ},j}} - \alpha \log \pi_\theta(a_{t+1}|s_{t+1})] \quad (7)$$

The minimum between $Q_{\phi_{targ,1}}$ $Q_{\phi_{targ,2}}$ is used as a target to update Q_{ϕ_1} and Q_{ϕ_2}. Using the *reparameterization trick*, a sample from $\pi_\theta(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters and independent noise $\epsilon_t \sim N(0,1)$ sampled from a Spherical Gaussian. Since we are modeling the policy as a squashed Gaussian, the samples are obtained according to

$$f_\theta(\epsilon_t; s_t) = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) \odot \epsilon_t)$$

Hence, the policy is optimized by minimizing [3]:

$$J_\pi(\theta) = \mathbb{E}_{\substack{s_t\sim D \\ \epsilon_t\sim N}}[\log \pi_\theta(f_\theta(\epsilon_t;s_t)|s_t) - \min_{j=1,2} Q_{\phi_j}(s_t, f_\theta(\epsilon_t;s_t))] \quad (8)$$

3 Results

Algorithm	Performance	Sample Efficiency
Reinforce with GAE	1521 ± 30	41 ± 13
Reinforce without GAE	1508 ± 38	28 ± 7
One-step Actor Critic Online Update	867 ± 77	40 ± 14
One-step Actor Critic Offline Update	1112 ± 5	48 ± 4
Soft Actor Critic with AET	3685 ± 23	79 ± 26
Soft Actor Critic without AET	4001 ± 38	80 ± 41

We saved the agent's best weights while training and loaded them for evaluation on 50 episodes with 5 different seeds for the performance metric. For the sample efficiency, we trained our agent on 5 different seeds for 100,000 steps while evaluating every 1000 steps for 20 episodes. Hence, there is a total of 10 evaluations. We plot the expected return (y-axis) against the number of time steps (x-axis) and compute the area under the curve (AUC) with the *numpy.trapz* function to find the sample efficiency of each algorithm.

It is possible to observe that using the Generalized Advantage Estimate (GAE) for REINFORCE increases the performance and sample efficiency while reducing the variance. For the One-step Actor critic (1AC), the offline version of the algorithm obtained reward of 1112 with a sample efficiency of 48 while the online version obtained 867 and 40 proving that updating the parameters of the value and policy functions at the end of an episode (offline) is better than updating them at every step (online). Although REINFORCE has higher variance, it has shown to be better than 1AC as the latter yields high bias estimates due to function approximation. We can clearly see that SAC outperforms all the other algorithms as expected since it is the state-of-the-art. Training SAC with AET reduces significantly the variance. In addition, we observe that SAC is significantly more sample efficient than its counterparts due to its replay buffer allowing it to reuse previous interactions. We were able to finely tune SAC (without AET) to obtain a performance of 4001 and a sample efficiency of 80.

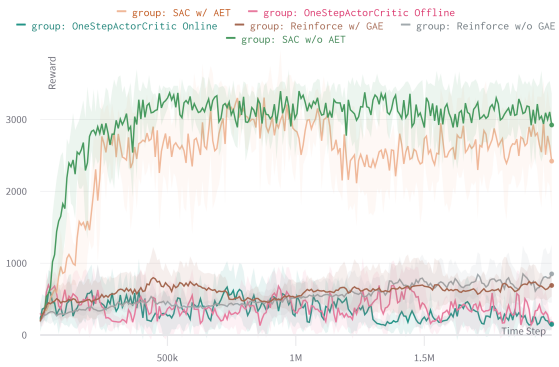

Figure 1: Mean rewards during training for SAC, One-Step AC, and Reinforce

The learning curves of all of the aforementioned algorithm are shown in 1. The Y-axis is the mean reward for the algorithm train on 5 different seeds. The shaded areas show the standard deviation. The results that we obtained here matches those obtained by [3]. Overall, SAC has similar variance than REINFORCE.

4 Conclusion and Future Works

In conclusion, we have implemented Reinforce, One-Step Actor Critic, and Soft Actor Critic for the Hopper challenge. In each algorithm, we tried different variations such as GAE for Reinforce, online and offline update for One-step Actor Critic, and AET for Soft Actor Critic to understand the trade-offs. Through our results, we can clearly see that SAC is the state-of-the-art algorithm and outperforms all the others. Future work include improving exploration through Normalizing Flow policies [12] and adaptability via Continual Meta Policy Search [1].

References

[1] G. Berseth, Z. Zhang, G. Zhang, C. Finn, and S. Levine. Comps: Continual meta policy search, 2021.

[2] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control, 2016.

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[4] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications, 2018.

[5] H. Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2015.

[7] L.-J. Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[9] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[10] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[11] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[12] P. N. Ward, A. Smofsky, and A. J. Bose. Improving exploration in soft-actor-critic with normalizing flows policies. *CoRR*, abs/1906.02771, 2019.

[13] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992.

Appendix

4.1 Hyperparameters

Reinforce

- gamma: 0.99
- policy
  - learning rate: 3e-4
  - momentum: 0
  - weight decay: 0
- value function
  - learning rate: 1e-3
  - momentum: 0
  - weight decay: 0
  - iterations: 80
- lambda: 0.95
- GAE: True/False

One Step Actor Critic

- gamma: 0.99
- policy
  - learning rate: 3e-4
  - weight decay: 0
- value function
  - learning rate: 1e-3
  - weight decay: 0
- Online: True/False

Soft Actor Critic

- seed: 0
- replay size: 1 million
- gamma: 0.99
- polyak 0.995
- alpha: 0.3
- start step: 1000
- update after: 500
- update every: 50
- batch size: 512
- policy

learning rate: 3e-4
weight decay: 0

- q function weight decay: 0
- automatic entropy tuning: True/False
- actor critic MLP size: 256, 256