From Scratch Implementation of Deep Reinforcement Learning Algorithms for Continuous Control

Ruturaj Sambhus

December 9, 2022

1 Abstract

Over the last decade, deep reinforcement learning (RL) algorithms have successfully solved complex control tasks like locomotion. Benchmarking the results is challenging since these algorithms have many hyper-parameters and numerous ways of implementation. A deep neural network is essential to solve continuous control problems which increase the complexity of implementation. As a researcher working with reinforcement learning, it is an important skill to be able to code the algorithms from papers and discover the benefits and pitfalls. To achieve this, a from-scratch implementation of fundamental policy and actor-critic deep RL algorithms for continuous control- Vanilla Policy Gradient (REINFORCE) and Advantage Actor Critic (A2C) is done. The code is well commented for easy understanding which can be valuable for teaching purposes. The developed code would serve as a great starter code to implement advanced algorithms such as PPO, SAC and DDPG. The code can be found here- https://github.com/ruturajsambhusvt/A2C.git

2 Background and Problem Statement

2.1 Fundamentals

Reinforcement learning is based on the assumption of Markov Decision Process (MDP) with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, rewards $r \in \mathcal{R}$, and state transition probability $\mathbb{P}(s'|s,a)$. The next state $s' \in \mathcal{S}$ depends only on the $\{s, a, \mathbb{P}(s'|s,a\}$ and is unrelated to the history of states. A policy $\pi(a|s)$ maps states to actions. The reinforcement learning problem is to find a policy π that maximizes the expected return given by equation 1 over time.

$$G = \mathbb{E}[\sum_{t=0}^{T} \gamma^t R(s_t, \pi(s_t))] \tag{1}$$

where $\gamma \in [0, 1]$ is a future discount factor. The state and action-value functions $V^\pi(s)$, $Q^\pi(s,a)$ are defined as the expected reward obtained by the agent starting from state s (taking action a for Q^π) and following policy π

$$V_\pi(s) = \mathbb{E}[G_t|S_t = s], \;\; Q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{2}$$

The Bellman equations for the value function (tabular setting) are given by-

$$V_\pi(s) = \max_a \sum_{s',r} \mathbb{P}(s', r|s, a)[r + \gamma V_\pi(s')], \;\; Q_\pi(s, a) = \sum_{s',r} \mathbb{P}(s', r|s, a)[r + \gamma V_\pi(s')] \;\; or$$
$$Q_\pi(s, a) = \sum_{s',r} \mathbb{P}(s', r|s, a)[r + \gamma \max_{a'} Q_\pi(s', a')] \tag{3}$$

The Bellman operator $\mathbb{T}$ is a contraction mapping which means the true value can be iteratively solved. Estimating the value function of a given policy is called policy evaluation and using the estimates to update the policy is known as a policy improvement.

2.2 REINFORCE

REINFORCE is also known as the Monte Carlo policy gradient since the temporal difference target is the return of the policy estimated using the Monte Carlo rollout. The policy gradient and the update rule are thus given as per equation 4, where θ are the parameters of the function approximator.

$$\nabla J(\theta_t) \propto \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} \right], \qquad \theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} \tag{4}$$

$$\nabla J(\theta_t) \propto \mathbb{E}_\pi \left[(G_t - b(S_t)) \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} \right], \qquad \theta_{t+1} = \theta_t + \alpha (G_t - b(S_t)) \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} \tag{5}$$

To reduce the variance of the gradient estimate, a baseline value can be subtracted from the return. The baseline can be any function that does not depend on the action. Thus, the update rule is modified as per the equation 5. One of the most intuitive choices for the baseline is the state value function. The state value function $V(S_t, \mathbf{w})$ is learned through the rollouts along with policy updates. The algorithm is provided in figure 1

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

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 $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 1: REINFORCE with Baseline [1]

2.3 Advantage Actor Critic (A2C)

As the name says, the state value function is used to calculate the TD return which is an estimate of the actual return and tells how good the action is. This can be done in one-step TD update or using eligibility traces. The one-step update rule is as per the equation

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{V}(S_t, \mathbf{w}) \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} = \theta_t + \alpha \left(R_{t+1} + \gamma \hat{V}(S_{t+1}, \mathbf{w}) - \hat{V}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t/S_t, \theta_t)}{\pi(A_t/S_t, \theta_t)} \tag{6}$$

The one-step return is superior to the Monte Carlo return in terms of the variance and number of samples required. Thus, actor-critic methods are expected to be sample efficient as compared to the policy gradients. The algorithm is provided in figure 2

One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
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):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Figure 2: One Step A2C [1]

3 Algorithm Implementation

3.1 Class based structure

Reinforcement learning algorithms have many moving parts and can be very complicated to debug. To make the code modular and capable of extending to other advanced algorithms, it is organized in terms of classes inspired from [2]. The three main classes are- *Networks, Memory* and *Agent*. The dataflow between these is as shown in figure 3. The *Networks* class consists of PyTorch-based neural network policy and value function along with functionalities such as sampling actions from the policy network, saving the network parameters, and gradient clipping. The *Memory* class is used to organize the $\{s, a, r, s'\}$ tuple obtained from policy evaluation steps. It has functionalities to calculate the discounted returns for the states to be used for REINFORCE. Finally, the *Agent* class has instances of *Memory* and *Networks*. The agent has two main functions responsible for policy evaluation and policy improvement, inspired from [3]. The *policy evaluation* function plays the environment for a specified number of steps and stores the $\{s, a, r, s'\}$ tuple in the memory. Note that this is different than the original REINFORCE algorithm which plays the complete episode. It was found that the time steps to play an episode are typically much larger than that required to get stable convergence, thus saving time and computation. The *Policy Update* functions calculate the respective TD errors and perform the neural network parameter update steps. The class has additional functionalities to learn the evaluation and update for the specified number of learning steps and save the models.

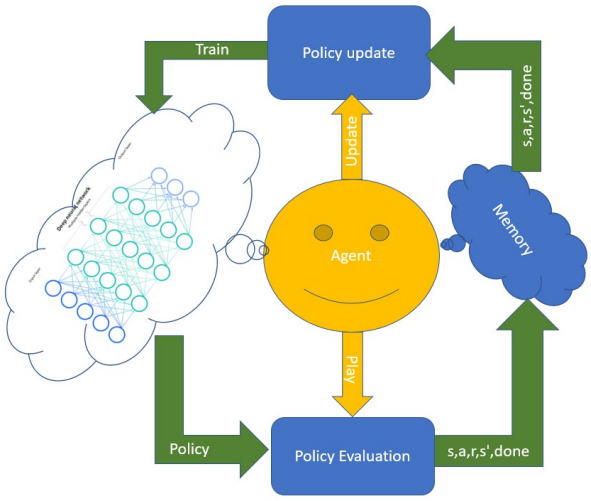

Figure 3: Framework

Parameters	Values
Policy-network	{n(state),64,64,n(actions)}
Value-network	{n(state),64,64,1}
Activation	ReLu
Actor-lr	0.0005
Critic-lr	0.005
Discount γ	0.95
Evaluation Batch (timesteps)	64 (REINFORCE), 32 (A2C)
Gradient clip norm	0.5
Policy Standard Deviation Limits	min=1e-3, max = 50

Table 1: Parameters for Gym Pendulum-v1 Environment

3.2 Neural Networks and Hyper-parameters

We choose PyTorch for implementing the neural networks since it was found to be easier for installation and usage. The policy and value networks are simple Multi-Layer Perceptron networks with two layers of depth. The value network takes in input and outputs the state value. The Policy Network is more challenging to implement since it would be a normal distribution with mean and variance are given by the network. There are different ways to implement this. We adopted the way listed in the ICLR blog post [4] where the variance is exponential of the log std parameters. It is important to note that the variance of the policies is clamped to a maximum value since having policies with a very large variance would essentially mean random policies. The values of various hyperparameters used are shown in the table 1

4 Results and Discussion

4.1 Experiments

The experiments were performed on various continuous control tasks from OpenAI Gym and Pybullet physics engine simulator. The OpenAI Pendulum-v1 is the simplest environment with one degree of freedom action space. The pendulum has to reach an angle of zero degrees (vertical) starting from a random

initial position. The environment is considered to be solved if the episodic reward reaches -200. This was chosen to study the stability of the convergence of the algorithms. Since the algorithms to be demonstrated are preliminary algorithms, the simplest environment is chosen for evaluation.

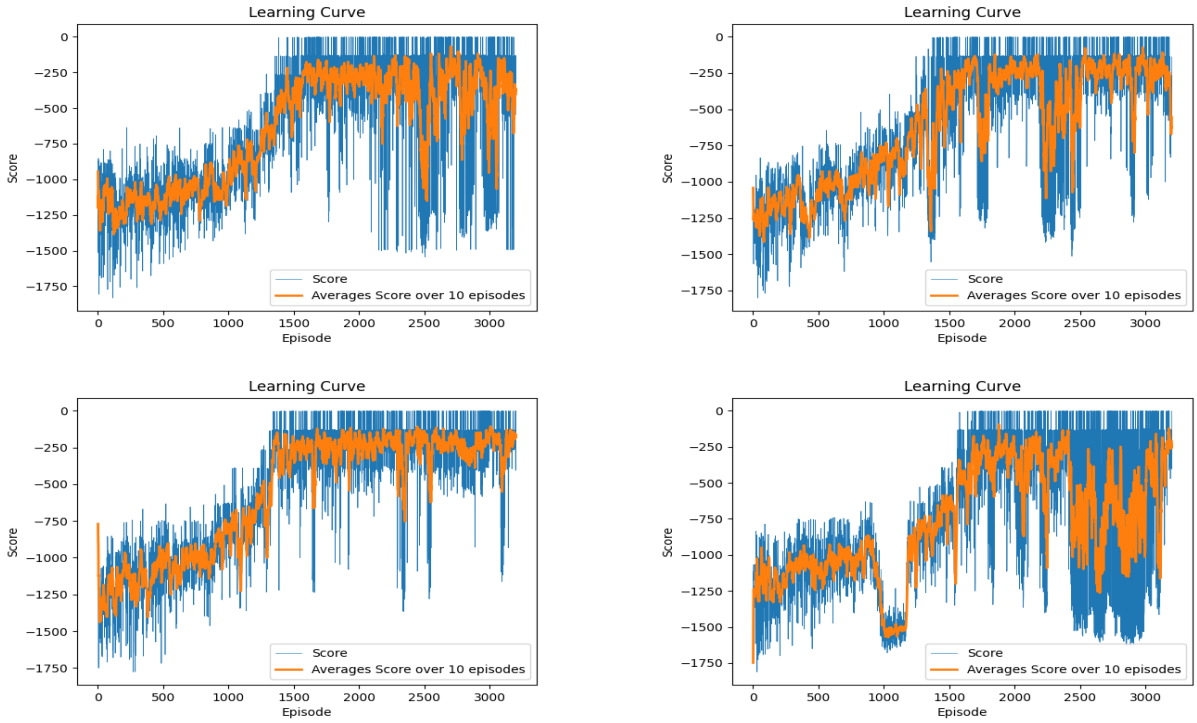

Figure 4: REINFORCE learning curves for Pendulum-v1

Figure 4 shows the learning curves with REINFORCE. The average reward is plotted over the last 10 episodes. Numerous runs were performed and the algorithm consistently learns to improve the reward. The agent starts getting the best average reward after about 1500 episodes. The convergence of A2C on the Pendulum-v1 environment is shown in figure 5. It is observed that the environment learns consistently across multiple runs, although occasionally having poor performance. It takes about 300 episodes to have improved average rewards. It is clearly observed that A2C takes much fewer episodes to learn (about $1/5$th) and is sample efficient compared to REINFORCE. This agrees well with the theory and analysis about the benefits of temporal difference updates. In terms of consistency of learning, REINFORCE was observed to have less variance in learning curves as compared to A2C. This is reflected in Figure 6 which shows the performance of the learned policies. The mean reward of REINFORCE is better than A2C although both don't solve the environment, the average reward is better than a random agent.

The algorithms were evaluated on difficult sparse reward Gym environment of MountainCarContinuous-v0 and PyBullet environments of Inverted Double Pendulum, CartPoleContinuous, and Walker2D. Figure 7 and Figure 8 present the learning curves for REINFORCE and A2C respectively on these environments. Both the algorithms failed to solve these environments, although reward improvement was observed. The same trend of A2C being sample efficient as compared to REINFORCE was observed across these. A2C was observed to have better performance for Walker 2D in terms of average rewards.

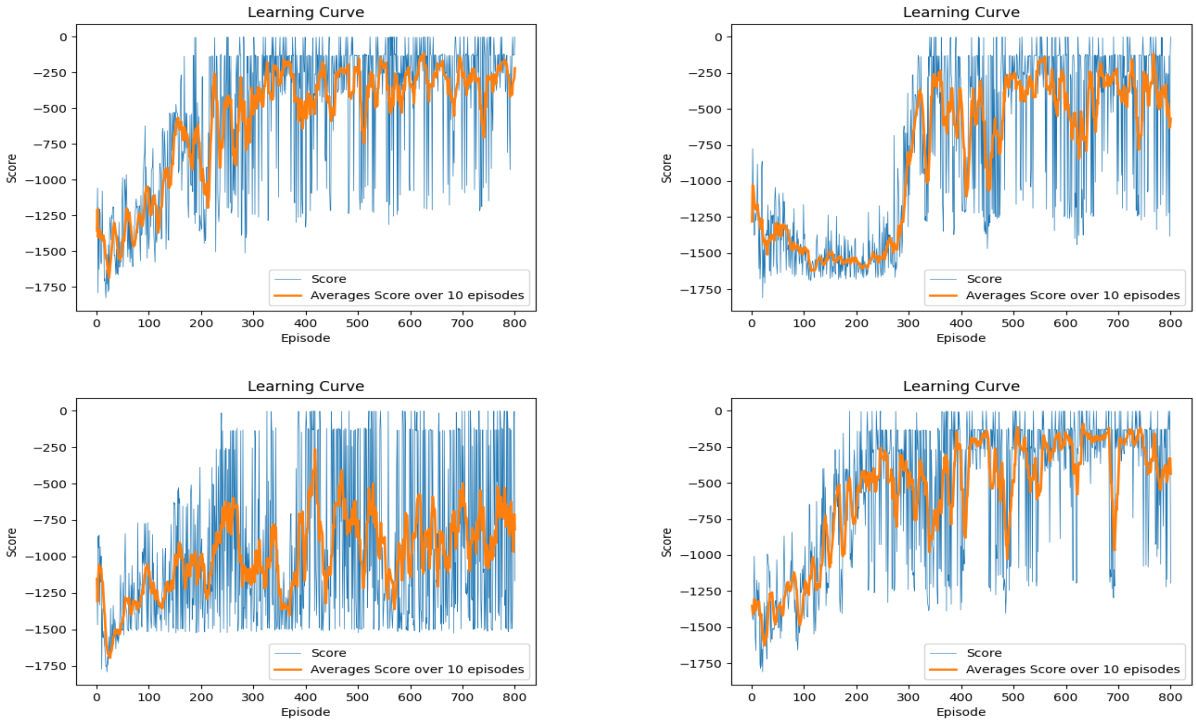

Figure 5: A2C learning curves for Pendulum-v1

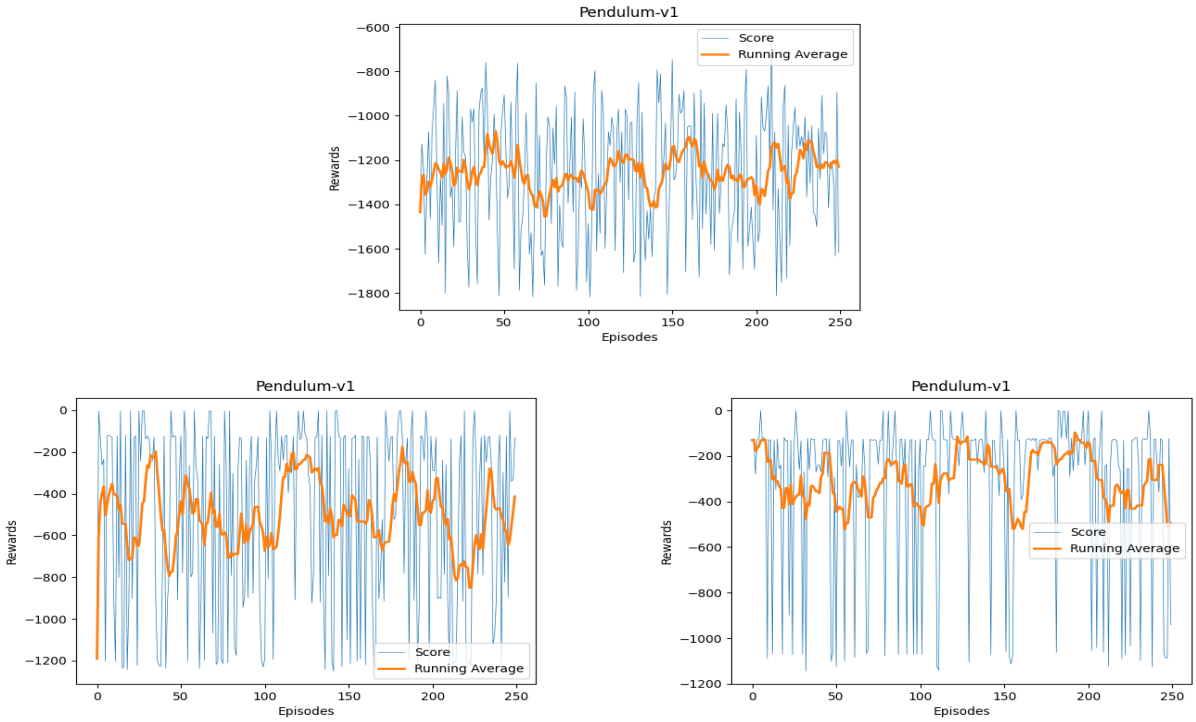

Figure 6: Performance of Learned Policy for REINFORCE and A2C compared to random agent (top)

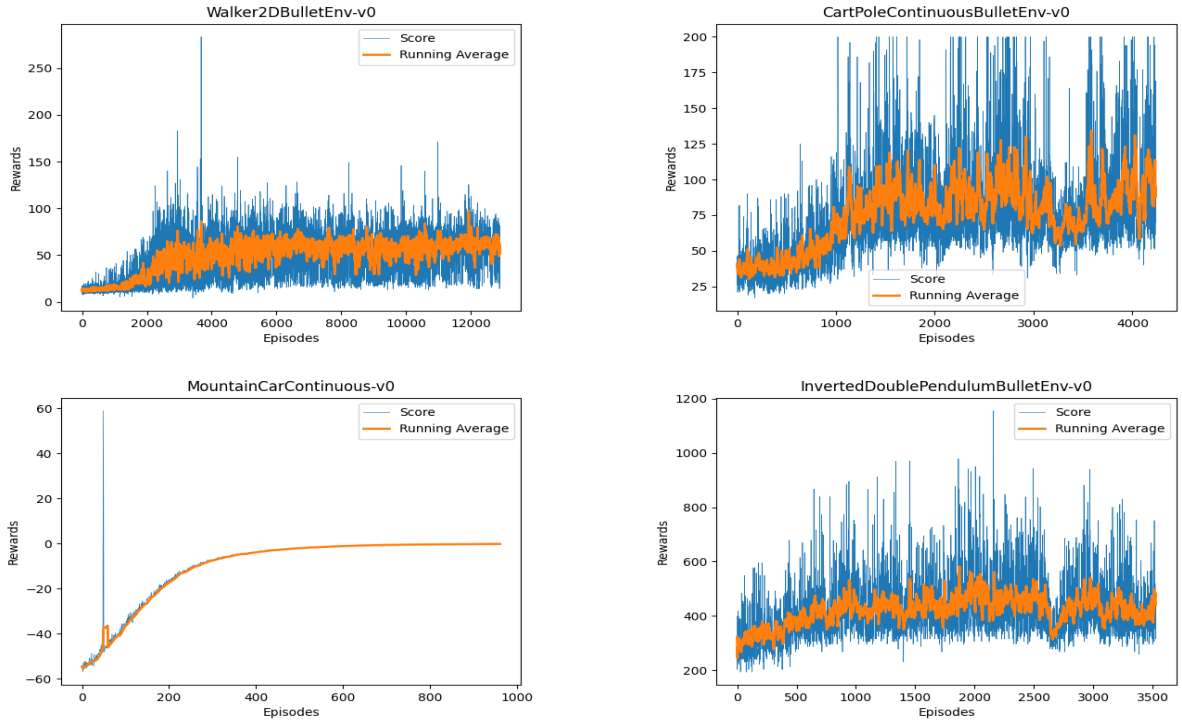

Figure 7: REINFORCE learning curves for other environments

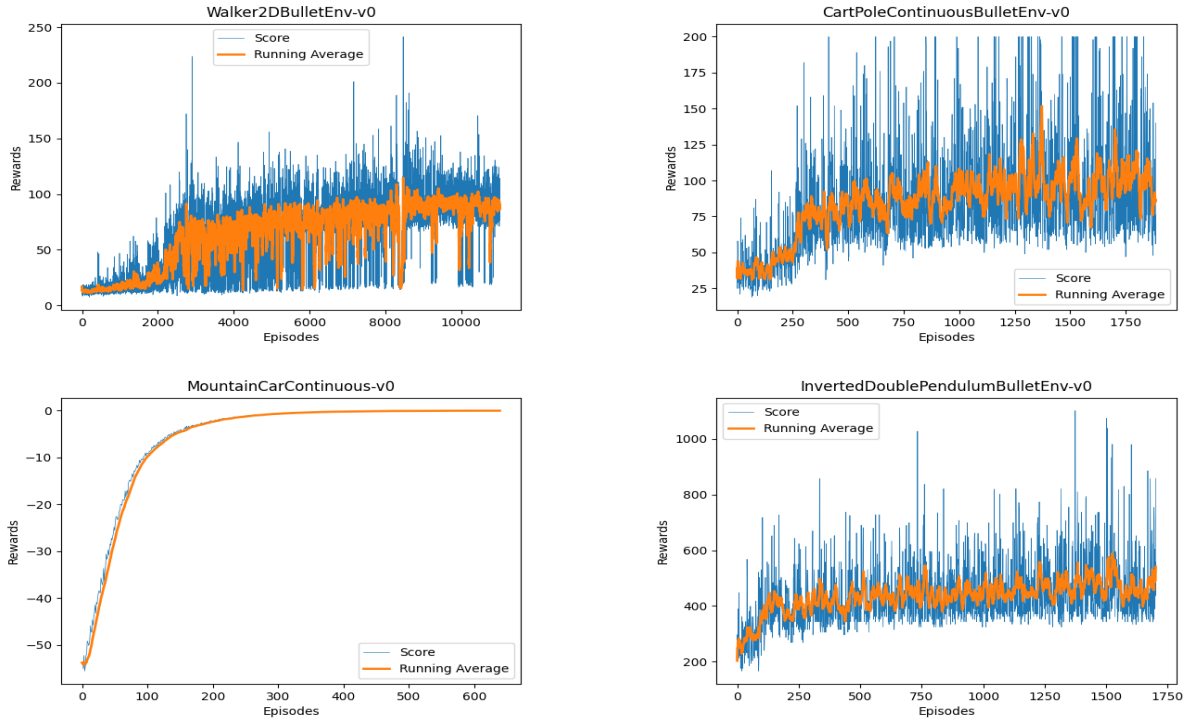

Figure 8: A2C learning curves for other environments

4.2 Sensitivity to Hyper-parameters

Deep RL algorithms have numerous hyper-parameters and the performance is usually brittle to variations in these. In this section, we try changing various hyper-parameters and observe the effects.

An important implementation detail for this work is using gradient clipping while calculating the value and policy gradients for backpropagation. Large neural network gradients can cause the networks to diverge. As per [5], gradient clipping has been found to improve the learning performance for continuous control tasks. The stochastic gradient descent convergence improves due to better local smoothness. It further accelerates the training by reducing the variance of gradient and preventing slow learning due to vanishing gradients. Our experiments (figures 9 and 10) find that gradient clipping is necessary for A2C but not for REINFORCE. In A2C, the TD error is estimated from the critic network which is further used for policy network updates. Instability in the value network due to large gradients would magnify the gradient of the policy network leading to divergence. However, REINFORCE uses the value network just as a baseline to reduce variance and a Monte Carlo estimate of return.

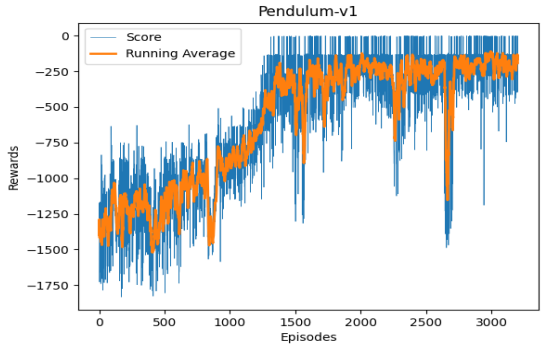

Figure 9: REINFORCE Gradient Clipping = 1e12 for Pendulum-v1

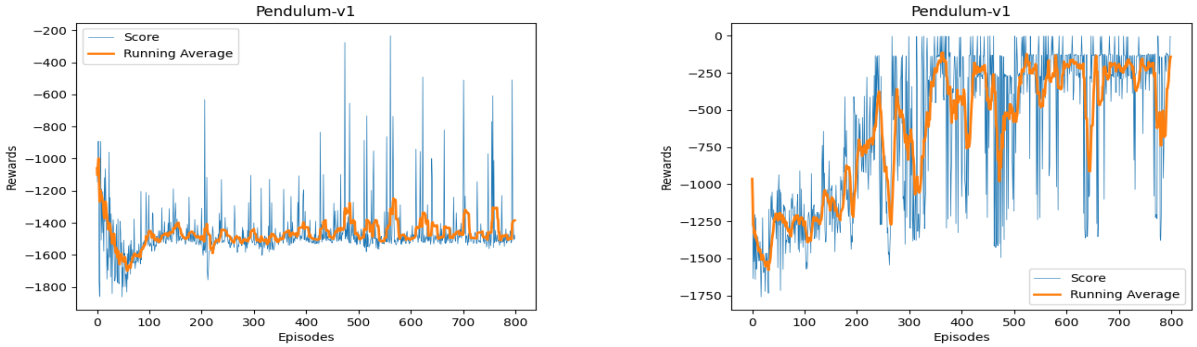

Figure 10: A2C Gradient Clipping = 100 and 10 for Pendulum-v1

Another important hyper-parameter is the evaluation batch size which is the timesteps required for policy evaluation. A larger evaluation batch size would mean better estimates of the TD error and improved performance, but along with increased computation and more samples. Since REINFORCE uses Monte Carlo returns, it would be expected that it needs more timesteps for estimating the return. Our experiments confirm this, as shown in figures 11 and 12. A single sample batch is not sufficient for both algorithms to learn, but A2C achieves learning for a batch size of 8 while REINFORCE needs a batch size of at least 32. Beyond these values, the sample efficiency starts to decrease.

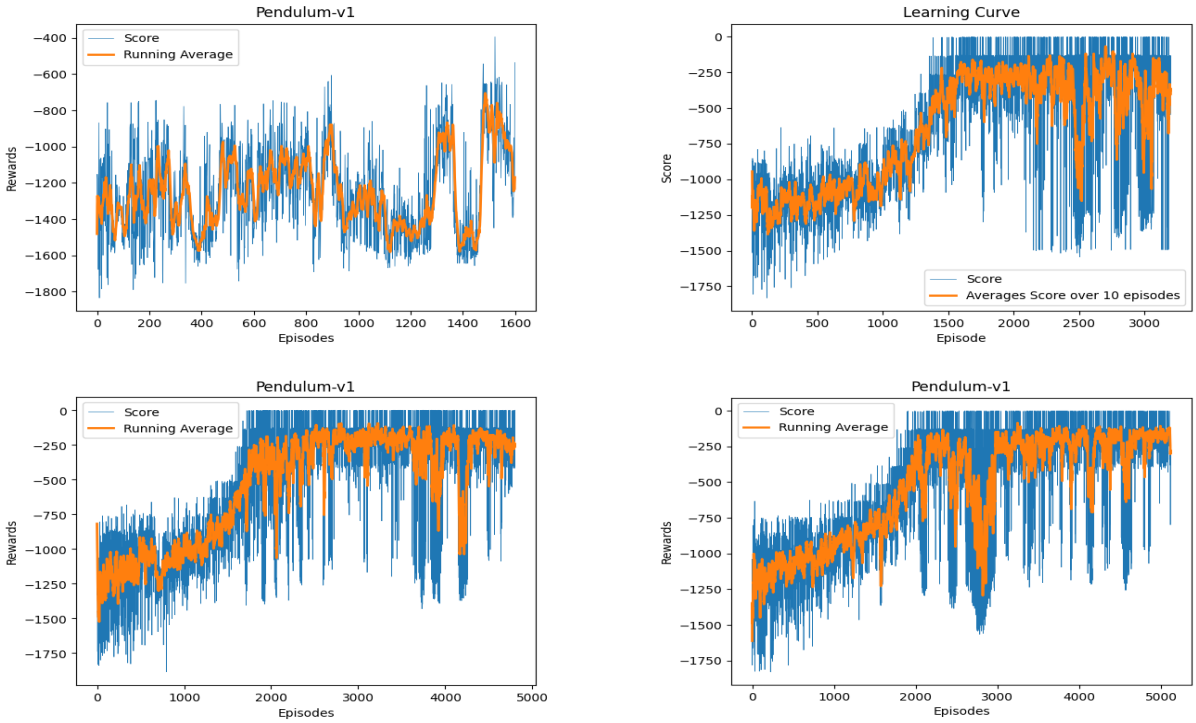

Figure 11: REINFORCE evaluation batch = 32, 64, 128 and 256 for Pendulum-v1

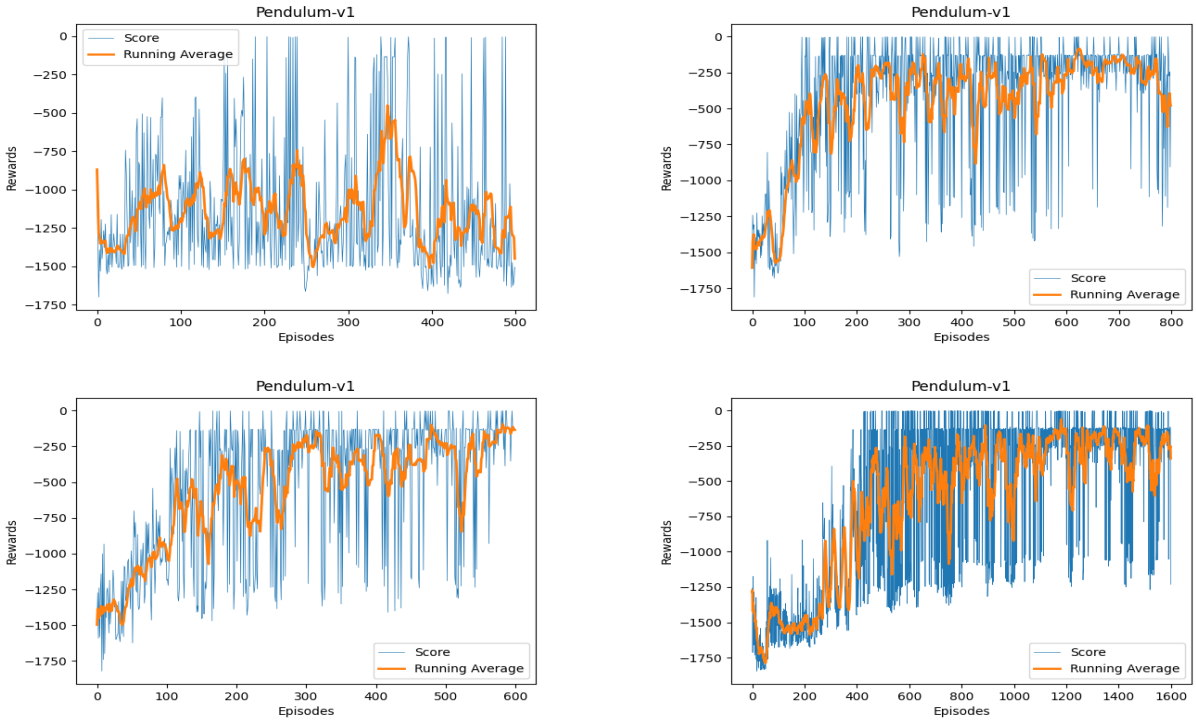

Figure 12: A2C evaluation batch = 1, 8, 16 and 64 for Pendulum-v1

The size of the neural network is another important hyper-parameter. We analyze the effect of the number of neurons per layer for the two-layer deep feedforward network. As shown in figure 13, the larger network size is found to have degraded performance which can be attributed to overfitting.

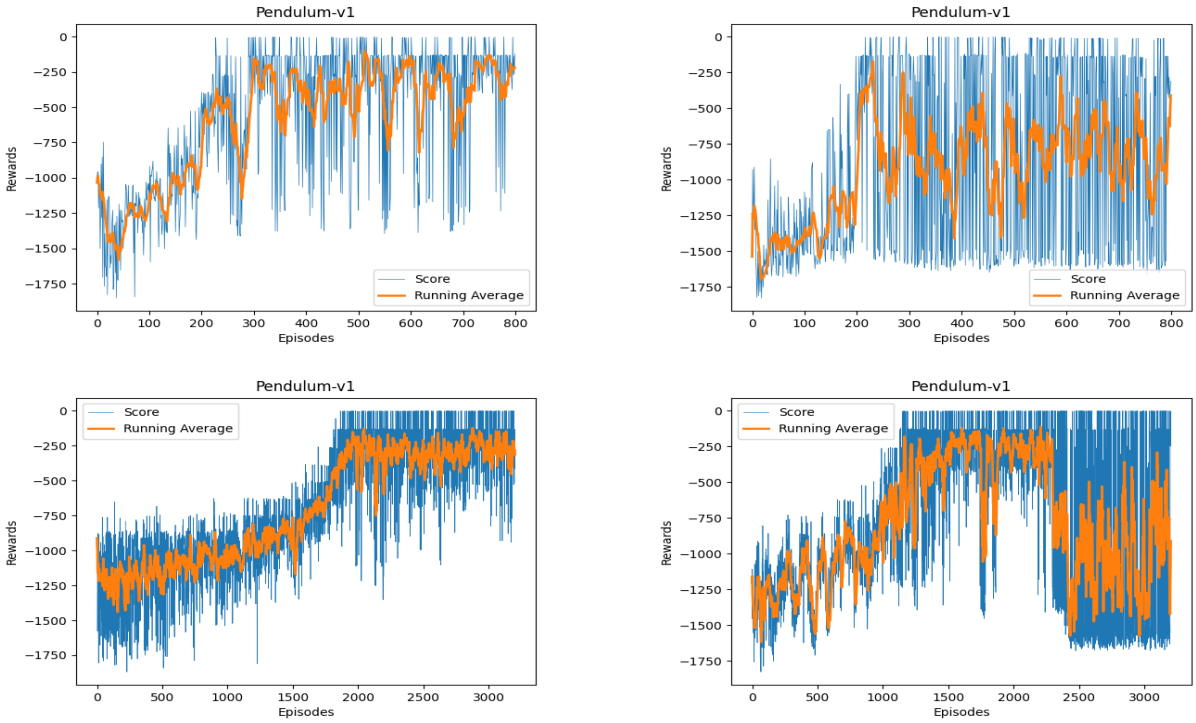

Figure 13: Neurons per layer 32, 128 (A2C and REINFORCE) Pendulum-v1

In conclusion, we find an agreement with the theory in the fact that actor-critic methods are sample efficient compared to pure policy gradients. Further, these deep RL algorithms are unsuccessful in solving complex control tasks, and advanced algorithms are crucial. Finally, the learning is sensitive to hyper-parameters and it would take effort to find the optimal parameters for a given problem. Hence, the generalizability of hyper-parameters would be an important benchmark for future deep RL algorithms. Future work would involve building up on the code to implement advanced algorithms like PPO, SAC, DDPG and benchmarking those against existing libraries.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Philtabor, "Philtabor/youtube-code-repository: Repository for most of the code from my youtube channel."

[3] Hermesdt, "Reinforcement-learning/a2c at master · hermesdt/reinforcement-learning."

[4] "The 37 implementation details of proximal policy optimization."

[5] J. Zhang, T. He, S. Sra, and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," *arXiv preprint arXiv:1905.11881*, 2019.