Research Internship Report

Exploration of Advanced Reinforcement Learning Algorithms in a Robotics Environment

Submitted by

Atharv Sonwane
Undergraduate Student
BITS Pilani

Under the guidance of

Prof G C Nandi
Center of Robotics and Machine Intelligence
Indian Institute of Information Technology, Allahabad

July 2020

1 Introduction

Over the course of this internship, I explored the field of Deep Reinforcement with a special focus on how it can be utilised in Robotics. I implemented and tested multiple different RL algorithms and was able to learn about key concepts in RL such as actor critic methods, Q-learning methods, soft policies and distributed RL. I benchmarked these algorithms in robotics oriented environment to gauge their suitability for real world applications.

2 Policy Gradient

Initially I implemented a simple versions of policy gradient. Through this, I learnt about the policy gradient theorem and about using a critic as a baseline to compute gradients for the policy. In these methods we, are trying to maximise the expected return

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [R(\tau)] \tag{1}$$

To optimize this objective via gradient ascent, we compute its gradient with respect to the policy parameters θ. In its most basic form, this is

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)] \tag{2}$$

However this form produces very high variance gradients which can make training unstable. Thus, more advanced policy gradient methods introduce many modifications and constraints to stabilize training, the most popular being the addition of a critic $V_\phi : S \mapsto \mathbb{R}$ to estimate the advantage. Thus the gradient becomes

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\nabla_\theta \log(\pi_\theta(a_t|s_t) A(s_t, a_t)] \tag{3}$$

3 Q Learning

In this method, there is no explicit policy. Rather, we try to learn a Q function $Q : S \times A \mapsto \mathbb{R}$. The optimal such Q function, Q^* satisfies the Bellman equation

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s' \sim P} [r(s, a) + \gamma \max_{a'} Q^*(s', a')] \tag{4}$$

The Q function is learnt indirectly through Temporal Difference Learning by optimising a bellman loss derived from the above equation –

$$\mathcal{L} = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} [(Q_\phi(s,a) - (r + \gamma(1-d)\max_{a'} Q_\phi(s',a')))^2] \tag{5}$$

4 Advanced Methods

After having thoroughly understood the two main kinds of Deep RL algorithms, I moved on to more advanced once which add additional tricks to improve stability and efficiency

4.1 Double Deep Q Network (DQN)

The simplest one of these was the Double DQN [1] which solves the optimistic value estimation issue with regular DQN by having two separate Q networks and selecting the one with the lower estimate. Other improvements include having a target network that is used to compute the Bellman targets and using an epsilon greedy exploration strategy. I found that these additions significantly improved the performance of DQN on even simple environments.

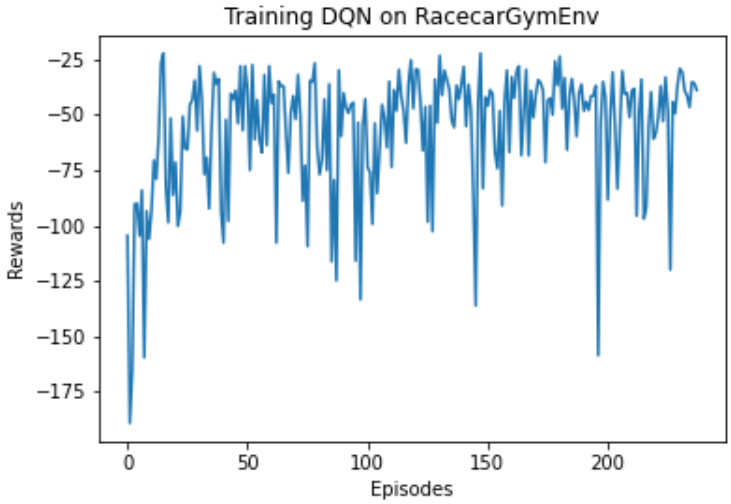

Figure 1: Double DQN converged to a good reward on Racecar Environment within 50 episodes. However it remained noisy after convergence.

4.2 Deep Deterministic Policy Gradient (DDPG)

Algorithms such as DDPG [2] and TD3 cannot be classified into either on policy (policy gradient) or off policy (Q learning). These methods are especially designed

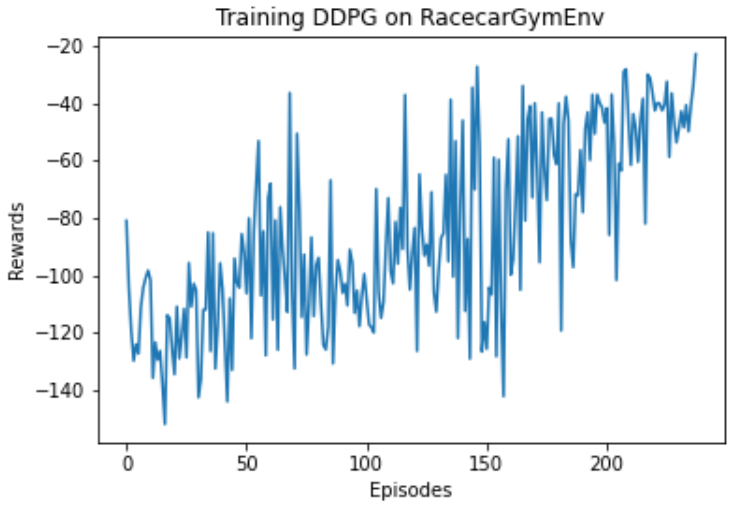

Figure 2: DDPG gradually converged to a good reward on Racecar Environment. This may be due to sub-optimal hyper parameters.

for continuous action spaces and learn both a Q function Q_ϕ and a policy π_θ concurrently.

Unlike in DQN where the Q function only takes in a state and outputs the q values for all possible actions, the Q function in DDPG takes both the state and action as input and outputs a value estimate. This change is due to the fact that DDPG has been designed for continuous action spaces.

A consequence of this is that finding the optimal action for a certain state becomes a non-trivial optimisation problem. This is why we train a separate policy network which maps states to actions to directly optimise this Q function. This whole process can trained in an off policy manner thus improving sample efficiency.

4.3 Twin Delayed DDPG (TD3)

TD3 [3] improves upon DDPG by using two different Q functions and using the lower value for bellman targets to solve optimistic value estimation problem. It also delays the policy updates with respect to the Q function updates and adds noise to target action to prevent the policy exploiting small inaccuracies in the Q function.

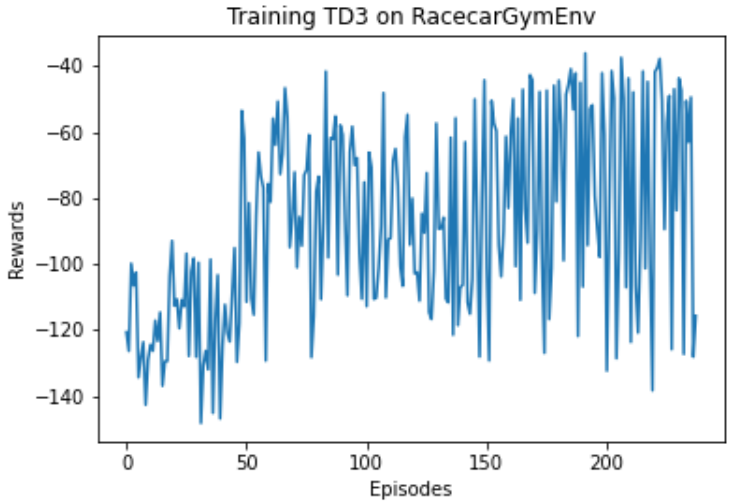

Figure 3: TD3 learned faster than DDPG but remained very inconsistent.

4.4 Soft Actor Critic (SAC)

SAC [4] introduces an entropy regularisation term (weighted by a constant) to both the learner objects for the Q function and the policy. Thus the objective becomes

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi_\theta(.|s_t))] \tag{6}$$

where H denotes the entropy of a distribution, which in this case is the policy π_θ. This introduces stochasticity into the policy and improves exploration and stability. We can even learn the weighting term α to further balance exploration vs exploitation trade off.

The optimisation of (6) corresponds to minimising a KL divergence between the π_θ and a distribution based on Q_ϕ. This optimisation can be performed by reparameterising the the policy for differentiable sampling $a_\theta \sim \pi_\theta$. The final gradient being

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta (Q_\phi(s, a_\theta(s)) - \alpha \log \pi_\theta(a_\theta(s)|s_t))] \tag{7}$$

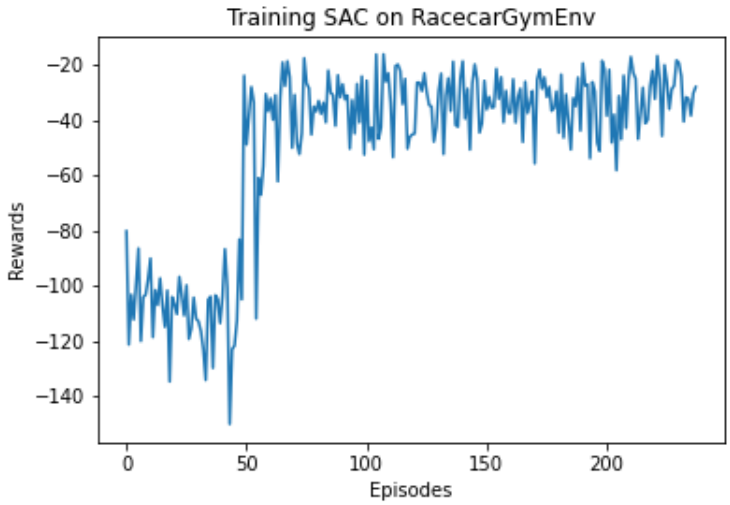

Figure 4: SAC was the fastest and most stable algorithm. The sudden spike occurs since the agent collects random experience without updating its weights for initial 50 episodes after which it starts learning

5 Distributed RL

In all of the above algorithms, we make the implicit assumption that the actor and learner are the same entity and there is only one such entity. However there may be cases where we are able to create multiple actors or learners. For example, when training an agent to play a video game, we can make use of multiple machines running in parallel to collect experience to train on, thus drastically increases the time required to learn.

One of the biggest challenges in using Reinforcement Learning for Robotics is that standard algorithms require the agent to collect large amounts of experience to achieve good performance which can be infeasible to do in real life. The solution is often to train the agent in a simulated environment before transferring to a real robot. Distributed RL can be especially useful here since we can significantly reduce train times by running multiple simulations in parallel. However agents, trained in this way often face issues in generalising to many real world conditions which might not have been accurately represented in the simulation.

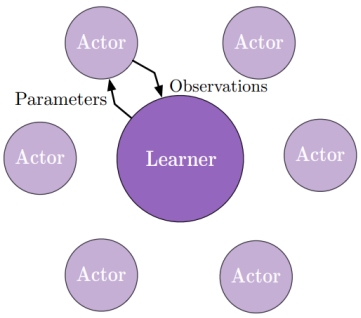

Figure 5: [5]

Another approach for training is training multiple different real robots in parallel. While this may not be feasible for large or autonomously moving robots, it is suited for smaller robots such as robotic arms which have a constrained state space. Distributed RL can again be of use here as multiple robots can be used as actors which collect experience which a central learner uses to update the policy.

5.1 IMPALA

IMPALA [5] is a policy gradient algorithm designed for multiple actors. Previous methods of using policy gradient in Distributed RL had multiple actors running in parallel and when all of them completed their trajectory, the learner would collect the trajectories, learn from them and then update the policy weights of each learner. The main drawback here is that the the time between updates is defined by the slowest actor which means parallelism is not being complete optimised.

In IMPALA however, all actors run independently and update with the central learner asynchronously of one another. The issue that arises here is the when am actor communicates its completed trajectory to the learner, the learner's policy might be multiple steps ahead of the actors (having synced with other actors). To solve this, IMPALA introduces a v-trace off policy correction based on importance sampling.

5.1.1 Implementation Results

Since IMPALA involves multiple actors running asynchronously, implementing it required some amount of multi-threading code which was difficult to write and debug.

Once implemented, the algorithm learnt much faster than the other mentioned in this report. It did however suffer from reward collapse. That is, after having achieved the maximum reward possible for an environment, after a few episodes, the policy would collapse to a sub-optimal one which gave low reward. The implementation was also very sensitive to hyper parameters.

To solve these issues, I employed a time decaying learning rate. However the implementation still remains unstable. Further investigation is required to learn the cause of this behavior and potential solutions.

6 Conclusions

After bench-marking the various algorithms, it is evident that performing real time training of an RL agent on a robot posses many challenges -

- The time required to collect enough experience for the agent could be prohibitive
- For some robots with a large range of movements (autonomous ground robots, UAVs etc..), setting up a safe and sufficiently large environment will be difficult.

Thus training in simulation (at least initially) seems like the most viable option for many use cases.

Distributed RL can be used to drastically speed up training in both simulation and real life by employing multiple different actors. Such as method could make real world training of agents on constrained robots such as robotic arms feasible.

However, even after training converges, in most of the benchmarks we can see that the agents are very unstable and would not be a good fit for tasks requiring high consistency. I also found that RL agents are highly sensitive to changes in implementation and small bugs in the code can lead to large fluctuations in performance. Thus future exploration includes finding ways of making RL algorithms more robust to both hyper-parameters and implementations to make them more suitable for robotics applications.

On a more personal note, I found that exploring different styles of implementations of these algorithms is necessary to gain better understanding of what makes them work as well as ways to make them stable. I found that it is difficult to conduct meaningful experiments without having stable benchmarks. Thus by gaining better understating of these algorithms, I can improve the current implementations I have for use in further research and investigations.

7 Acknowledgements

I would like to thank Ms. Priya Shukla for guiding me through the learning process and providing great advice and practical knowledge regarding how RL is used in Robotics. I would also like to thank Prof. Nandi for his guidance as well as for providing valuable intuition into the field of Reinforcement Learning. I am grateful to IIIT Allahabad for offering this great opportunity.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[3] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[4] Tuomas Haarnoja, Aurick Zhou, Sehoon Ha, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *CoRR*, abs/1812.11103, 2018.

[5] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.