Summary on Asynchronous Methods for Deep Reinforcement Learning

Siddharth Nayak

1 Introduction

Generally when training deep reinforcement learning agents, an experience replay buffer is used. But using this requires more memory and also requires off-policy algorithms which have to be updated from the data of an older policy. This paper introduces an algorithm which asynchronously executes multiple agents in parallel. This parallel processing makes the agents experience a variety of different states at any given time-step. Then the authors evaluate and compare DQN, and asynchronous versions of 1-step Q-Learning, 1-step SARSA, n-step Q-Learning and Asynchronous Advantage Critic(A3C) on 57 different Atari games.

2 Reinforcement Learning Background[1]

2.1 One-step Q-Learning

The parameters θ of the action value function $Q(s, a; \theta)$ are learned by iteratively minimizing a sequence of loss functions where the ith loss function is defined by:

$$L_i(\theta_i) = \mathbf{E}\big(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\big)$$

This is called 1-step Q-Learning because it updates the action value $Q(s, a)$ with the one-step return $r+\gamma \max_{a'} Q(s', a'; \theta_{i-1})$. The drawback of using this is that obtaining a reward r only directly affects the value of the state of the state action pair (s, a) that led to that reward which is evident by the form of update. The state action pairs which indirectly affected the update get affected only by $Q(s, a)$. This slows the learning process a lot.

2.2 n-step Q-Learning

To tackle the slow nature of one-step Q-Learning, instead of using the one-step returns we can use n-step returns. In n-step Q-Learning, $Q(s, a)$ is updated by: $r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \max_a \gamma^n Q(s_{t+n}, a)$. This results in a single reward r directly affecting n preceding state action values which makes the propagation more efficient.

2.3 Policy based method (REINFORCE)

These methods directly parametrize policy$\pi(a|s; \theta)$. Standard REINFORCE updates the policy parameters θ in the direction of $\nabla_\theta log\pi(a_t|s_t; \theta)R_t$. We can also use baselines to reduce tha variance of this estimate by keeping it unbiased by subtracting a learned function of the state $b_t(s_t)$. Therefore the resulting gradient is $\nabla_\theta log\pi(a_t|s_t; \theta)(R_t - b_t(s_t))$. A learned estimate of the value function which is commonly used is $b_t(s_t) \approx V^\pi(s_t)$. Also, R_t is an estimate of $Q^\pi(a_t, s_t)$. Therefore if, $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ then the update of the policy parameters become: $\nabla_\theta log\pi(a_t|s_t; \theta)A(a_t, s_t)$.

3 Model

This section focuses on the the asynchronous variants of one-step SARSA, one-step Q-learning, n-step Q-learning, and advantage actor-critic presented in the paper. The authors use multiple CPU threads on a single machine. This single machine training removes communication costs of sending gradients and parameters. Also running multiple agents in parallel helps in exploring more as each of the agent will explore different parts of the environment. This asynchronous method does not require memory replay and relies on the agents' different policies to stabilise the overall convergence. Because there is no need of a replay memory, on-policy algorithms like one-step SARSA and actor critic can be used.

3.1 Asynchronous one-step Q-learning

Each thread interacts with its own copy of the environment and at each step the gradient of the Q-Learning loss is computed. The target network is updated after a a few steps. This reduces the chances of the multiple actors overwriting each other's updates. They also give different exploration policies to the different actors. This improves the robustness of the system.

3.2 Asynchronous one-step SARSA

This almost similar to the asynchronous one-step Q-learning. Except that the target value used is $r + \gamma Q(s', a' : \theta^-)$ where a' is the action taken in state s'.

3.3 Asynchronous n-step Q-Learning

The algorithm works by explicitly computing the n-step returns rather than using backward view techniques like Eligibility Traces. To evaluate a single update the algorithm first selects actions using its exploration policy for up to t_{max} steps or till terminal state is reached. Then the gradients are computed and updated for each stat-action pair encountered since the last update. Here, each of the n-step update uses the longest possible n-step return possible. Which means the last state will have a one-step update, a two step-update for second-last state and so on. All the accumulated gradients are applied in a single step.

[1] Writing this so that I can refer it

3.4 Asynchronous Advantage Actor Critic(A3C)

Here a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$ is maintained. Like the n-step Q-Learning the n-step returns are used to update both the policy and the value function after t_{max} steps or when a terminal state is reached. The update for the algorithm is $\nabla_\theta log\pi(a_t|s_t;\theta)A(a_t,s_t;\theta,\theta_v)$ where $A(a_t,s_t;\theta,\theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1}\gamma^i r_{t+i}+\gamma^k V(s_{t+k};\theta_v)-V(s_t;\theta_v)$. The authors find that adding the entropy of the policy to the loss function improved the overall exploration and hence not making the agent converge to a sub-optimal policy. The gradient for the loss function now becomes: $\nabla_\theta^{'} log\pi(a_t|s_t;\theta^{'})A(a_t,s_t;\theta,\theta_v)+\beta\nabla_{\theta'}H(\pi(s_t;\theta^{'}))$ where H is the entropy.

3.5 Optimization

Three different optimization methods are used - SGD with momentum, RMSProp without share statistics and RMSProp with share statistics. The latter on turns out to be more robust.

4 Experiments

All the different algorithms mentioned before are tested out on different Atari 2600 Games and TORCS 3D car racing simulator. In all the games A3C works the best by getting the highest reward/score with lesser number of training hours. They also examined the performance of the A3C algorithm on tasks with continuous action spaces.

5 Conclusion

The paper presented an asynchronous method to train different algorithms by making multiple agents work on the same task. Having multiple agents helps in making the system more robust as well as having a more exploration. This method of multiple actors also does not require a memory replay buffer which allows us to use on-policy algorithm. This is one of the widely used algorithms in the current RL research.

6 References

V.Mnih A.Badia M.Mirza, A.Graves, T.Harley, T.Lillicrap, D.Silver, K.Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: WCP volume 48.