Asynchronous Methods for Deep Learning

Devesh Nath

December 28, 2024

Abstract

This document provides an overview of Asynchronous Methods for Deep Learning, focusing on Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C) algorithms. These methods are designed to improve the efficiency and performance of reinforcement learning models by leveraging parallelism and asynchronous updates.

1 Introduction

Deep neural networks enable effective reinforcement learning (RL) but were previously considered unstable with simple online RL algorithms such as Q-learning and SARSA. These algorithms update the policy and value estimates based on each new piece of data, which can lead to high variance and instability. Non-stationarity refers to the changing distribution of data due to the evolving policy, making it difficult for the model to learn effectively. Solutions like experience replay stabilize RL by reducing non-stationarity and decorrelating updates. Experience replay stores past experiences and samples them randomly during training, which breaks the correlation between consecutive updates and smooths the learning process. However, experience replay requires off-policy algorithms and more computational resources.
This paper introduces a new paradigm using asynchronous parallel agents, which decorrelates data and supports a broader range of RL algorithms. By running multiple agents in parallel and asynchronously updating a shared model, this method reduces the correlation between updates without the need for experience replay. This method, particularly the asynchronous advantage actor-critic (A3C), achieves superior results on various tasks using standard multi-core CPUs, making it more resource-efficient than previous approaches.

2 Reinforcement Learning Background

We consider the standard reinforcement learning setting where an agent interacts with an environment E over a number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action a_t from some set of possible actions A according to its policy π, where π is a mapping from states s_t to actions a_t. In return, the agent receives the next state s_{t+1} and a scalar reward r_t. The process continues until the agent reaches a terminal state, after which the process restarts. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1]$. The goal of the agent is to maximize the expected return from each state s_t.
The action value $Q^\pi(s, a) = \mathbb{E}[R_t \mid s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The optimal value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ gives the maximum action value for state s and action a achievable by any policy. Similarly, the value of state s under policy π is defined as $V^\pi(s) = \mathbb{E}[R_t \mid s_t = s]$ and is simply the expected return for following policy π from state s.
In value-based model-free reinforcement learning methods, the action value function is represented using a function approximator, such as a neural network. Let $Q(s, a; \theta)$ be an approximate action-value function with parameters θ. The updates to θ can be derived from a variety of reinforcement learning algorithms. One example of such an algorithm is Q-learning, which aims to directly approximate the optimal action value function: $Q^*(s, a) \approx Q(s, a; \theta)$. In 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 i-th loss function is

defined as

$$L_i(\theta_i) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2\right]$$

where s' is the state encountered after state s. The Q-learning update step is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$$

We refer to the above method as one-step Q-learning because it updates the action value $Q(s, a)$ toward the one-step return $r + \gamma \max_{a'} Q(s', a'; \theta)$. One drawback of using one-step methods is that obtaining a reward r only directly affects the value of the state-action pair (s, a) that led to the reward. The values of other state-action pairs are affected only indirectly through the updated value $Q(s, a)$. This can make the learning process slow since many updates are required to propagate a reward to the relevant preceding states and actions.

One way of propagating rewards faster is by using n-step returns. In n-step Q-learning, $Q(s, a)$ is updated toward the n-step return defined as $r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a Q(s_{t+n}, a)$. This results in a single reward r directly affecting the values of n preceding state-action pairs, making the process of propagating rewards to relevant state-action pairs potentially much more efficient.

In contrast to value-based methods, policy-based model-free methods directly parameterize the policy $\pi(a \mid s; \theta)$ and update the parameters θ by performing, typically approximate, gradient ascent on $\mathbb{E}[R_t]$. One example of such a method is the REINFORCE family of algorithms. Standard REINFORCE updates the policy parameters θ in the direction $\nabla_\theta \log \pi(a_t \mid s_t; \theta) R_t$, which is an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$. It is possible to reduce the variance of this estimate while keeping it unbiased by subtracting a learned function of the state $b_t(s_t)$, known as a baseline, from the return. The resulting gradient is $\nabla_\theta \log \pi(a_t \mid s_t; \theta)(R_t - b_t(s_t))$. A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V^\pi(s_t)$, leading to a much lower variance estimate of the policy gradient. When an approximate value function is used as the baseline, the quantity $R_t - b_t$ used to scale the policy gradient can be seen as an estimate of the advantage of action a_t in state s_t, or $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$, because R_t is an estimate of $Q^\pi(a_t, s_t)$ and b_t is an estimate of $V^\pi(s_t)$. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline b_t is the critic.

3 Asyncronous RL Framework

The authors present multi-threaded asynchronous variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic. These methods aim to train deep neural network policies reliably with minimal resource requirements. Despite the differences in underlying RL methods, they use two main ideas to achieve their design goals.

First, they use asynchronous actor-learners with multiple CPU threads on a single machine, eliminating communication costs and enabling Hogwild! style updates for training.

Second, multiple actors running in parallel explore different parts of the environment, using diverse exploration policies to reduce correlation in parameter updates. This approach eliminates the need for replay memory, relying on parallel actors to stabilize learning.

Using multiple parallel actor-learners offers practical benefits: a reduction in training time proportional to the number of parallel learners and the ability to use on-policy RL methods like Sarsa and actor-critic for stable neural network training. The authors describe their variants of one-step Q-learning, one-step Sarsa, n-step Q-learning, and advantage actor-critic.

3.1 Asynchronous One-Step Q-Learning

The pseudocode for Asynchronous One-Step Q-Learning is shown in Algorithm 1. Each thread interacts with its own copy of the environment and computes the gradient of the Q-learning loss at each step. A shared and slowly changing target network is used to compute the Q-learning loss, as proposed in the DQN training method. Gradients are accumulated over multiple timesteps before being applied, similar to using minibatches, which reduces the chances of multiple actor learners overwriting each other's updates.

Accumulating updates over several steps also allows for a trade-off between computational efficiency and data efficiency. Each thread uses a different exploration policy to improve robustness and performance through better exploration. In particular, ϵ-greedy exploration is used, with ϵ periodically sampled from a distribution by each thread.

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

```
1: **Assume** global shared $\theta$, $\theta^-$, and counter $T = 0$
2: Initialize thread step counter $t \leftarrow 0$
3: Initialize target network weights $\theta^- \leftarrow \theta$
4: Initialize network gradients $d\theta \leftarrow 0$
5: Get initial state $s$
6: **repeat**
7:     Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
8:     Receive new state $s'$ and reward $r$
9:     $y \leftarrow \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
10:    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
11:    $s \leftarrow s'$
12:    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
13:    **if** $T \bmod I_{\text{target}} = 0$ **then**
14:        Update the target network $\theta^- \leftarrow \theta$
15:    **end if**
16:    **if** $t \bmod I_{\text{AsyncUpdate}} = 0$ or $s$ is terminal **then**
17:        Perform asynchronous update of $\theta$ using $d\theta$
18:        Clear gradients $d\theta \leftarrow 0$
19:    **end if**
20: **until** $T > T_{\max}$
```

3.2 Asynchronous One-Step Sarsa

Each thread interacts with its own copy of the environment and computes the gradient of the Sarsa loss at each step. A shared and slowly changing target network is used to compute the Sarsa loss. Gradients are accumulated over multiple timesteps before being applied, similar to using minibatches, which reduces the chances of multiple actor learners overwriting each other's updates. Accumulating updates over several steps also allows for a trade-off between computational efficiency and data efficiency. Each thread uses a different exploration policy to improve robustness and performance through better exploration. In particular, ϵ-greedy exploration is used, with ϵ periodically sampled from a distribution by each thread.

3.3 Asynchronous n-Step Q-Learning

The pseudocode for Asynchronous n-Step Q-Learning is shown in Algorithm 2. The algorithm operates in the forward view by explicitly computing n-step returns. This approach is easier when training neural networks with momentum-based methods and backpropagation through time. To compute a single update, the algorithm first selects actions using its exploration policy for up to $t_{\max}$ steps or until a terminal state is reached. This results in the agent receiving up to $t_{\max}$ rewards from the environment since its last update. The algorithm then computes gradients for n-step Q-learning updates for each of the state-action pairs encountered since the last update. Each n-step update uses the longest possible n-step return, resulting in a one-step update for the last state, a two-step update for the second last state, and so on for a total of up to $t_{\max}$ updates. The accumulated updates are applied in a single gradient step.

Algorithm 2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

```
1: Assume global shared $\theta$, $\theta^-$, and counter $T = 0$
2: Initialize thread step counter $t \leftarrow 0$
3: Initialize target network weights $\theta^- \leftarrow \theta$
4: Initialize network gradients $d\theta \leftarrow 0$
5: Get initial state $s$
6: repeat
7:     Initialize $n$-step reward buffer $R \leftarrow 0$
8:     for $i = 1$ to $t_{\max}$ do
9:         Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
10:        Receive new state $s'$ and reward $r$
11:        Store $(s, a, r)$ in buffer
12:        $R \leftarrow R + \gamma^{i-1} r$
13:        if $s'$ is terminal then
14:            Break
15:        end if
16:        $s \leftarrow s'$
17:    end for
18:    for $i = t_{\max}$ down to 1 do
19:        $y \leftarrow \begin{cases} R & \text{for terminal } s' \\ R + \gamma^i \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
20:        Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
21:        $R \leftarrow (R - r_i)/\gamma$
22:    end for
23:    $T \leftarrow T + t_{\max}$ and $t \leftarrow t + t_{\max}$
24:    if $T \bmod I_{\text{target}} = 0$ then
25:        Update the target network $\theta^- \leftarrow \theta$
26:    end if
27:    if $t \bmod I_{\text{AsyncUpdate}} = 0$ or $s$ is terminal then
28:        Perform asynchronous update of $\theta$ using $d\theta$
29:        Clear gradients $d\theta \leftarrow 0$
30:    end if
31: until $T > T_{\max}$
```

3.4 Asynchronous Advantage Actor-Critic

The algorithm, called asynchronous advantage actor-critic (A3C), maintains a policy $\pi(a_t \mid s_t; \theta)$ and an estimate of the value function $V(s_t; \theta_v)$. Similar to the n-step Q-learning variant, the actor-critic variant operates in the forward view and uses a mix of n-step returns to update both the policy and the value function. The policy and value function are updated after every $t_{\max}$ actions or when a terminal state is reached. The update performed by the algorithm can be seen as $\nabla_{\theta'} \log \pi(a_t \mid s_t; \theta') A(s_t, a_t; \theta, \theta_v)$, where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$, with k varying from state to state and upper-bounded by $t_{\max}$. The pseudocode for the algorithm is presented in Algorithm 3.

As with the value-based methods, parallel actor-learners and accumulated updates are used to improve training stability. While the parameters θ of the policy and θ_v of the value function are shown as separate for generality, some parameters are shared in practice. Typically, a convolutional neural network with one softmax output for the policy $\pi(a_t \mid s_t; \theta)$ and one linear output for the value function $V(s_t; \theta_v)$ is used, with all non-output layers shared.

Adding the entropy of the policy π to the objective function improves exploration by discouraging premature convergence to suboptimal deterministic policies. The gradient of the full objective function, including the entropy regularization term, with respect to the policy parameters takes the form $\nabla_{\theta'} \log \pi(a_t \mid s_t; \theta')(R_t - V(s_t; \theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t; \theta'))$, where H is the entropy and β controls the strength of the entropy regularization term.

Algorithm 3 Asynchronous Advantage Actor-Critic (A3C) - pseudocode for each actor-learner thread.

```
1:  **Assume** global shared $\theta$, $\theta_v$, and counter $T = 0$
2:  Initialize thread step counter $t \leftarrow 0$
3:  Initialize network gradients $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
4:  Get initial state $s$
5:  **repeat**
6:      Initialize $n$-step reward buffer $R \leftarrow 0$
7:      **for** $i = 1$ to $t_{\max}$ **do**
8:          Take action $a$ with policy $\pi(a \mid s; \theta)$
9:          Receive new state $s'$ and reward $r$
10:         Store $(s, a, r)$ in buffer
11:         $R \leftarrow R + \gamma^{i-1} r$
12:         **if** $s'$ is terminal **then**
13:             Break
14:         **end if**
15:         $s \leftarrow s'$
16:     **end for**
17:     **for** $i = t_{\max}$ down to 1 **do**
18:         $y \leftarrow \begin{cases} R & \text{for terminal } s' \\ R + \gamma^i V(s'; \theta_v) & \text{for non-terminal } s' \end{cases}$
19:         Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a \mid s; \theta)(y - V(s; \theta_v)) + \beta \nabla_\theta H(\pi(s; \theta))$
20:         Accumulate gradients wrt $\theta_v$: $d\theta_v \leftarrow d\theta_v + \frac{\partial (y - V(s; \theta_v))^2}{\partial \theta_v}$
21:         $R \leftarrow (R - r_i)/\gamma$
22:     **end for**
23:     $T \leftarrow T + t_{\max}$ and $t \leftarrow t + t_{\max}$
24:     **if** $t \bmod I_{\text{AsyncUpdate}} = 0$ or $s$ is terminal **then**
25:         Perform asynchronous update of $\theta$ using $d\theta$
26:         Perform asynchronous update of $\theta_v$ using $d\theta_v$
27:         Clear gradients $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
28:     **end if**
29: **until** $T > T_{\max}$
```

4 Conclusion

Asynchronous methods in deep reinforcement learning (DRL) offer several significant advantages, which are discussed below:

4.1 Scalability and Data Efficiency

Asynchronous methods leverage parallelism by running multiple agents simultaneously, each interacting with its own copy of the environment. This parallelism allows for more efficient use of computational resources, particularly multi-core CPUs, and reduces the overall training time. By decorrelating the data collected by different agents, asynchronous methods improve data efficiency, as the updates to the shared model are based on a more diverse set of experiences. This diversity helps in faster convergence and better generalization of the learned policies.

4.2 Robustness and Stability

The use of multiple parallel actor-learners with different exploration policies enhances the robustness of the learning process. Each agent explores different parts of the state-action space, reducing the likelihood of the model getting stuck in local optima. Additionally, asynchronous updates help in stabilizing the training process by mitigating the non-stationarity problem inherent in reinforcement learning. The Hogwild! style updates, where multiple threads update the shared model asynchronously, further contribute to the stability by ensuring that the model parameters are updated frequently and consistently.

4.3 Importance in DRL

Asynchronous methods have proven to be crucial in advancing the field of DRL. They enable the training of deep neural network policies with minimal resource requirements, making it feasible to train complex models on standard hardware. The ability to use on-policy methods like Sarsa and actor-critic without the need for experience replay is a significant advantage, as it simplifies the training process and reduces memory overhead. The success of algorithms like Asynchronous Advantage Actor-Critic (A3C) in various challenging tasks highlights the effectiveness of asynchronous methods in achieving state-of-the-art performance in DRL. In summary, asynchronous methods in DRL provide a scalable, data-efficient, robust, and stable framework for training deep reinforcement learning models. These methods have been instrumental in pushing the boundaries of what is achievable with DRL, enabling the development of more sophisticated and capable agents.