Parallel Trust Region Policy Optimization with Multiple Actors

Kevin Frans, Danijar Hafner

September 2016

Abstract

We show a method for parallelizing the Trust Region Policy Optimization algorithm, using multiple actors with their own environments to simultaneously collect experience. We trained the model on a variety of continuous control tasks and consistently demonstrate a speedup when compared to the benchmark of a single-threaded implementation. We also find that a majority of time spent in a TRPO iteration is in collecting experience, rather than computing gradients to the policy, so additional cores are best suited to parallelize the actors rather than the learners.

1 Introduction

In recent years, a variety of reinforcement learning algorithms have been detailed that may take different approaches to finding an optimal policy. Trust Region Policy Optimization [1] is one such method that consistently performs well on a variety of continuous control tasks. However, it uses a Monte-Carlo estimation that is reliant on collecting a large amount of data after every policy update. Compared to methods such as Q-learning [6] that update the policy after every timestep, TRPO may experience a few thousand timesteps before updating the policy. A major amount of the time spent in every iteration is simply collecting experience, rather than computing gradients for the policy. Therefore, there is less of a reliance on hardware such as a GPU which can quickly compute gradients, and more on the efficiency of a CPU which can run many episodes of an environment quickly. In an effort to improve the practicality of TRPO, we present a method that uses multiple actors to significantly reduce the time needed for each policy iteration. This takes advantage of the multiple cores present in most modern computers, to train faster on a variety of tasks.

2 Related Work

Gorila [3], short for General Reinforcement Learning Architecture, introduced a setup utilizing separate actor processes and separate learner processes. Each actor contains a copy of a Q-network, which periodically updates from a master parameter server. These actors continuously run simulations and collect transition data, which is stored in a distributed replay memory. Learner processes then sample from this replay memory and send gradients to the parameter server, which updates the policy accordingly. Gorila was shown to make big improvements in terms of wall-time on Atari games using the DQN structure.

Minh et al [2] introduced asynchronous versions of four common reinforcement learning algorithms: one-step Sarsa, one-step Q learning, n-step Q learning, and actor-critic. The setup is able to reduce training time in a near linear correlation to the number of threads used. Furthermore, using multiple actors eliminates the need for an experience replay, since the data is decorrelated due to actors observing different parts of the environment.

3 Background

3.1 Reinforcement Learning

A standard reinforcement learning setup consists of an environment in which an agent can observe a set of observations known as a state. The agent can then interact with the environment by taking a set of actions, either discrete or continuous. An agents policy $\pi(a|s)$ represents a probability distribution over all possible actions, given an observation. For every timestep, the agent receives an observation from the environment, and takes an action according to its policy. In addition, the agent receives a scalar reward. This continues until the environment reaches a terminal state. A return is the total sum of all future rewards, which may discount rewards in later timesteps. Typical ways of representing returns are a state-value function $V(s)$, which keeps track of expected return from any state s, or an action-value function $Q(s, a)$, which contains the expected return when taking action a from state s. In practice, these value functions are usually represented using function approximators such as neural networks, as it is impractical to use a naive table lookup when an environments state space is huge or even infinite. An agents objective is to find a policy that will result in the greatest possible return from all of the possible starting states of the environment. The policy may be a neural network

of its own, mapping from states to actions, or it may simply be to act greedily towards the action-value function $Q(s,:)$.

3.2 Trust Region Policy Optimizion

A recent algorithm, Trust Region Policy Optimization (TRPO), tries to solve the classic reinforcement learning problem of maximizing return by iteratively optimizing a surrogate function. Similarly to policy gradient methods, TRPO uses a distinct parameterized policy $\pi(a|s)$ separate from any value function. To improve its policy, TRPO attempts to maximize the expectation of Q-values, over the distribution of states and actions given by θ_{new}.

$$\max_{\theta} \sum_{s} P_{\theta}(s) \sum_{a} \pi_{\theta}(a|s) Q_{\theta_{\text{old}}}(s,a)$$

$$\text{subject to } \overline{D}_{\text{KL}}^{p_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \leq \delta$$

This objective can be approximated by using an importance-sampled Monte Carlo estimate of Q values, with a distribution of states sampled from policy θ_{old}. However, theres a constraint to updating theta: the average KL divergence between the new policy and old policy cannot be greater than a constant δ. This acts as a limiter on the step size we can take on each update, and can be compared to the natural gradient [4]. The theory behind TRPO guarantees gradual improvement over the expected return of a policy, and the practical algorithm performs well on a wide variety of tasks.

4 Parallelization

One downside to the TRPO algorithm is its on-policy nature, requiring new Q-values after every policy update. We cannot use methods such as experience replay which reuse past information, so we must acquire new Monte Carlo estimates of Q for every new policy. Furthermore, Monte Carlo estimates are known to have higher variance than methods such as one-step TD updates, since the return is affected by independent future decisions. Bringing this variance down requires many episodes of experience per policy update, making TRPO a data-heavy algorithm.

In practice, the biggest bottleneck by a magnitude in a TRPO update is collecting the experience. We present a setup that parallelizes actors, collecting experience from many environments simultaneously. Although the learner could also be parallelized, we find that the additional cores are put to better use as actors, since the learner must wait until all the data is collected before it can operate.

By using multiple agents asynchronously, we can achieve a substantial decrease in training time, since Monte Carlo estimation is well suited for parallelization. Although we need to collect more data, we dont have to compute $Q(s_{\text{i}+1})$ for every transition as in a TD update, which lessens the workload on our single-threaded learner. We also suffer no loss from stale data: each agent is always on the latest version of the policy.

5 Algorithm

Presented is the general outline of a parallel version of TRPO. Each actor process contains a copy of the policy as well as its own environment. When estimating the Q function, we place a number of requests equal to the number of episodes we want to run into a distributed pool. Each actor can then accept requests, and run simulations in their own environments simultaneously. At the cost of some memory, we keep a separate copy of the same up-to-date

Algorithm 1 Parallel Trust Region Policy Optimization

initialize π_0
for i = 0,1,2,... until convergence **do**
    Send actor requests to distributed pool
    Wait for all threads to finish, and collect transition experience
    Estimate Q(s,a) using Monte Carlo
    Find policy update direction to maximize $\sum_s P_\theta(s) \sum_a \pi_\theta(a|s) Q_{\theta_{\text{old}}}(s,a)$
      while $\overline{D}_{\text{KL}}^{p_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \leq \delta$
    Update policy parameters and send to all actors

policy in each actors process to evaluate the policy faster. Once an actor has finished running an episode, it returns the transition experience, containing an observation, the action taken, and a reward for each timestep. The actor can then accept another request, and continue until all requests are completed. In the original TRPO algorithm, episodes are run until a certain amount of timesteps have been recorded. However, the asynchronous nature of our method requires us to determine how many episodes to run before collecting any data. We can estimate this number by dividing a desired number of timesteps by the average episode length using the previous policy.

6 Experiments

To verify our claims, we provide a simple experiment to compare TRPO with varying degrees of parallelization. We tested the model on a variety of continuous control tasks in the Mujoco physics simulator [5], using a setup from OpenAI gym. These tasks range from carefully maneuvering the tip of a two-jointed arm to a desired point, to controlling the joints of a figure and make it walk. The environment follows the classic reinforcement learning scheme: every timestep, the agent receives a vector of real-numbered observations, and takes an action according to its policy. The environment is updated accordingly and returns a scalar reward. In all the experiments, the action space is a continuous vector, not a discrete action. To account for this, we use a parameterized policy mapping an observation vector to the mean and standard deviation of a gaussian distribution. We then sample from this distribution to take the action.

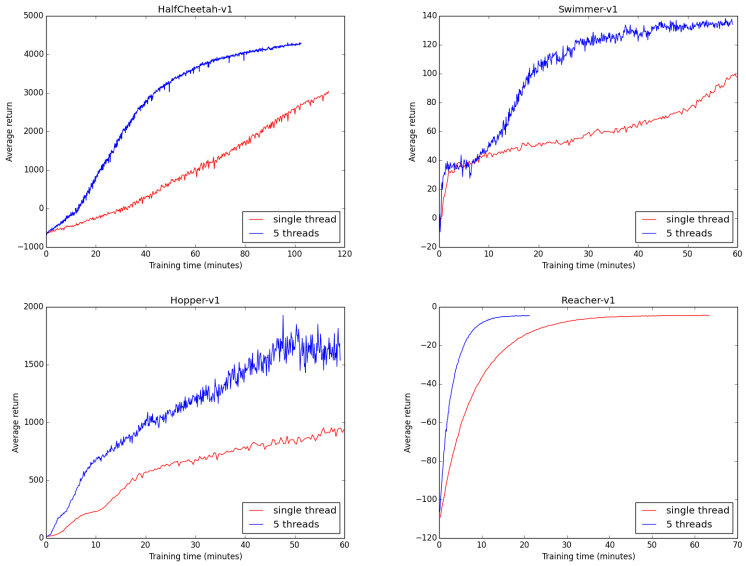

Figure 1: Learning curves of various continuous control environments from OpenAI gym, run using a single threaded implementation and a parallel implementation of TRPO.

The results support our theory. The model with multiple actors is consistently able to outperform the single-threaded model, over a variety of environments. When comparing average return to the iteration count, both models perform evenly with each other, showing the underlying methods are still the same. However, by reducing the wall-clock time required to compute each iteration, the parallel implementation of TRPO is able to learn much faster without any loss in accuracy.

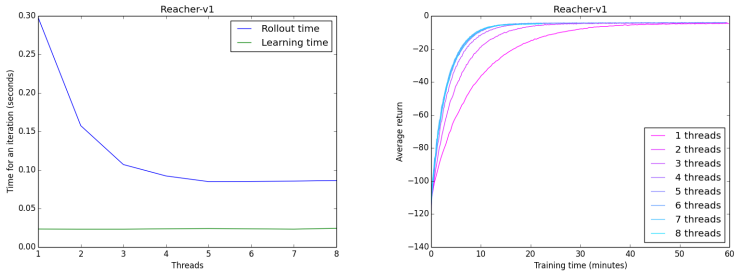

Figure 2: The left graph shows the relationship between number of threads used and the time it took to collect data for 20,000 timesteps on Reacher. A 5 core CPU was used. The right graph shows the learning curve on Reacher when using varying numbers of threads.

To get a sense of how adding more threads affects the time it takes to collect experience, we perform multiple trials on Reacher, one of the environments with a constantly stable learning time. Parallelization always decreases the time it takes to train an iteration, albeit with diminishing returns. These diminishing returns most likely stem from hardware limitations rather

than an inefficiency in the implementation. Although the number of threads was increased, the same 5 core CPU was used in all trials. Using a CPU with more cores or multiple computers would likely provide more detailed results on how much speedup can be attained.

7 Conclusion

We introduced a method of parallelizing the Trust Region Policy Optimization algorithm to achieve a significant decrease in training time. We identified the bottleneck of the single-threaded implementation of TRPO to be collecting experience data. We then used multiple parallel actors, each with their own copy of the policy and their own environment, to quickly simulate many episodes of experience simultaneously. The parallel implementation of TRPO displayed decreasing training time when adding threads. We predict that the number of cores used in a computer will substantially affect how much of a speedup the algorithm will gain from adding more threads. Especially when dealing with reinforcement learning algorithms that require big amounts of experience to train, parallelization is an important step to implementing methods in practice. Most modern computers come with multiple cores, and it taking advantage of them can lead to a significant speedup in wall-clock training time.

References

[1] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, Pieter Abbeel. Trust Region Policy Optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[2] Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783*, 2016.

[3] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, David Silver. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1507.04296*, 2015.

[4] Razvan Pascanu, Yoshua Bengio. Revisiting Natural Gradient for Deep Networks. *arXiv preprint arXiv:1301.3584*, 2013.

[5] Todorov E, Erez T, Tassa Y. MuJoCo: A physics engine for model-based control. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[6] Christopher Watkins, Peter Dayan. Technical Note Q-Learning. *Machine Learning (1992) 8: 279*, 1992.