DISTRIBUTED DEEP REINFORCEMENT LEARNING WITH ASYNCHRONOUS ADVANTAGE ACTOR-CRITIC USING GENERALIZED ADVANTAGE ESTIMATION

Peter Ebert Christensen (s153758), Nicklas Hansen (s153077)

Technical University of Denmark

1. INTRODUCTION

Deep reinforcement learning (RL) algorithms are generally computationally expensive and require a large number of rollouts in order to converge to a good solution. Unlike supervised learning problems where a fixed dataset is readily available, RL typically relies on training on rollouts generated by simulation. As the model generating rollouts changes over time, the data it is trained on becomes non-stationary, which can destabilize the algorithm.
Deep Q-Networks [1] aim to stabilize learning by sampling from a memory buffer over previously seen transitions, which generally works well but may introduce prohibitively large memory requirements for larger state spaces.

Asynchronous Advantage Actor-Critic (A3C) [2] utilizes distributed learning where a number of workers interact with the environment and update model parameters asynchronously, which effectively removes the need for a memory buffer due to high data throughput. Another advantage of distributed learning is that it allows for parallel generation of multiple rollouts, which often is a performance bottleneck in training of RL systems.
As rollouts are generated on-policy, however, it is likely that all trajectories will be similar as action probabilities gradually become near-zero for all but one action in the discrete case, which ultimately limits exploration. A3C addresses this problem by introducing an entropy-term to the loss function, which is discussed in section 2. We extend the A3C by replacing the advantage estimator used in [2] by the Generalized Advantage Estimate (GAE) as proposed by [3], and evaluate the algorithm on a number of environments.

2. ALGORITHM

As with all actor critic algorithms, the asynchronous advantage Actor-Critic (A3C) learns both a policy $\pi_\theta(a_t|s_t)$ and an estimate of the value function $V_{\theta_v}(s_t)$.

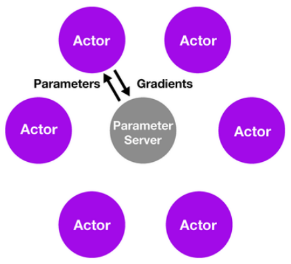

Fig. 1: *General A3C paradigm:* A number of asynchronous workers (denoted *actors* here) with a model parameter down-link and and gradient up-link to a global parameter server. Source: DeepMind.

A3C is an on-policy algorithm as a global model is updated as soon as a worker is done with a rollout or a number of steps t_{max} has passed and thus every worker always uses the same policy. This is in stark contrast to IMPALA [4], for instance, in which worker parameters aren't updated as soon as one worker is done with the rollout and thus workers end up having different policies. The asynchronous part of the algorithm refers to using multiple processes on a single machine for creating multiple copies of the model to generate rollouts and train in parallel.

Having multiple copies running in parallel will have a greater chances of exploring different parts of the environment than a single agent will. Because with effectively different policies in different threads, the overall changes made to the shared parameters by multiple copies applying updates in parallel are likely to be less correlated in time than a single agent applying updates. The update performed by the algorithm for the policy after a rollout is given by:

$$\nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) A_{\theta',\theta}(s_t, a_t) \tag{1}$$

where A is the estimate of the advantage function:

$$\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V_{\theta_v}(s_{t+1}) - V_{\theta_v}(s_t) \tag{2}$$

Another update is needed for the estimate of the value function, as we work with Actor-Critic methods, given by:

$$\partial\left(R - V\left(s_i; \theta'_v\right)\right)^2 / \partial\theta'_v \tag{3}$$

We can add an additional term to the policy loss, that is an entropy term H (the dot product between the log-probabilities and probabilities):

$$\nabla_{\theta'} \log\pi_{\theta'}(a_t|s_t)(R_t - V_{\theta_v}(s_t)) + \beta\nabla_{\theta'} H(\pi_{\theta'}(s_t)) \tag{4}$$

The reason why an entropy term is preferable is that it encourages exploration by penalizing a policy for selecting one particular action with high probability.

Algorithm 1: A3C with GAE for a single worker.

```
1  // Assume global shared parameter vectors $\theta$ and $\theta_v$
       and global shared counter $T = 0$
2  // Assume thread-specific parameter vectors $\theta\prime$ and $\theta_v\prime$
3  Initialize thread step counter $t \leftarrow 1$
4  while not done do
5      reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
6      Synchronize thread-specific parameters $\theta\prime = \theta$ and
         $\theta_v\prime = \theta_v$
7      $t_start = t$
8      Get state $s_t$
9      if not in terminal $s_t$ or not $t - t_{start} == t_{max}$ then
10         Perform $a_t$ according to policy $\pi_{\theta\prime}(a_t|s_t)$
11         Receive reward $r_t$ and new state $s_{t+1}$
12         $t \leftarrow t + 1$
13         $T \leftarrow T + 1$
14     end
15     R= $\begin{cases} 0 & \text{for terminal states}_t \\ V_{\theta_v\prime}(s_t) & \text{for non-terminals}_t \end{cases}$
16     for $i \in \{t-1, ..., t_{start}\}$ do
17         $R \leftarrow r_i + \gamma R$
18         Acculumate gradients wrt
             $\theta\prime$:$d\theta \leftarrow d\theta + \nabla_{\theta'} \log\pi_{\theta'}(a_i|s_i)(A^{GAE(\gamma,\lambda)}) + \nabla_{\theta\prime} H(\pi_{\theta\prime}(s_t))$
19         Acculumate gradients wrt
             $\theta'_v$:$d\theta_v \leftarrow d\theta_v + \partial\left(R - V\left(s_i; \theta'_v\right)\right)^2 / \partial\theta'_v$
20     end
21     Perform asynchronous update of $\theta$ using $d\theta$ and of
         $\theta_v$ using $d\theta_v$
22 end
```

This has the cost of efficient exploitation (for a non-stochastic environment at least), but for stochastic environments a certain amount of entropy is beneficial. However as noted in the paper the generalized advantage estimation may improve our results and thus we add it to the update equation:

$$\nabla_{\theta'} \log\pi_{\theta'}(a_t|s_t)(\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}) + \beta\nabla_{\theta'} H(\pi_{\theta'}(s_t)) \tag{5}$$

Here $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)}$ is a sum of exponentially weighted TD-targets. [3]:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{6}$$

See Algorithm 1 for pseudo-code.

3. EXPERIMENTAL DETAILS

The network architectures used in this study is shown in Figure 2. For image-based environments – such as Arcade Learning Environment (ALE) – the convolutional architecture is used. For all other environments the fully-connected network is used.
Architecture and number of parameters has been held constant at 256 neurons in each fully-connected layer throughout the experiments. The first 3-4 layers (depending on implementation) are shared between the actor and critic to force a common state representation and reduce overall number of parameters required. A Long Short-Term Memory (LSTM) cell is used to process temporal dynamics and in the image-based case convolutional layers process spatial dynamics. Only environments using the non-convolutional architecture are considered in the following.

Experiments are conducted on LunarLander-v2, MountainCar-v0 and Hopper-v2 (Mujoco[1]) from OpenAI Gym[2]. Unless otherwise is stated explicitly, experiments have been run using 64 workers, Adam [5] optimizer with a learning rate of 1e-4, updates every 20 steps, fixed appropriate γ for each environment, $\beta = 0$, $\lambda = 1$, reward scaling of 1e-3 and random seeds. The lock-free Hogwild [6] algorithm is used for parameter updates.

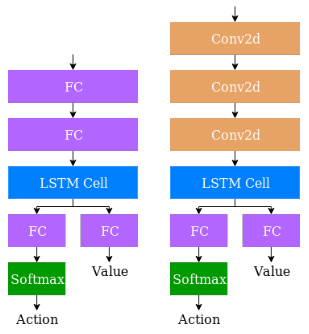

Fig. 2: The two networks used for our implementation of A3C. Softmax is applied to the actor output to compute the action probability distribution from which an action is sampled.

[1]Proprietary software – see `http://mujoco.org/`.
[2]Details are available here: `https://gym.openai.com/`.

4. RESULTS

A number of empirical results are discussed in this section. Due to various sources of randomness such as weight initialization, initial states, action sampling, sequence of parameter updates and even operating system and fluctuations in CPU load results are not replicable but should be reproducible to some extent – see [7].

4.1. Exploration by entropy

Penalizing the actor for low entropy in action probabilities encourages an agent to explore. On the MountainCar-v0 environment, it is observed that a model with $\beta = 0.01$ discovers a solution much faster than for $\beta = 0.00$ but also results in greater variance and converges at a less rewarding local minima. Fixing $\beta = 0.01$ for the first 1000 episodes and setting $\beta = 0.00$ for the remainder of the training achieves both efficient exploration and better convergence, which suggests that e.g. linear annealing of β may be beneficial. Results from the experiment can be seen in Figure 3.

4.2. Update frequency and the bias-variance trade-off

The number of steps between updates is a bias-variance trade-off; frequent updates biases an update towards a few samples, while infrequent updates greatly increases variance but also has a positive impact on training walltime due to fewer costly back-propagations. As is illustrated in Figure 3, 20 steps between each update appears appropriate on LunarLander-v2, while 100 steps delays convergence and 4 steps introduces too much bias for the model to converge. We apply the same update frequencies to the Hopper-v2 environment, where a large lag seems beneficial in the long run.

4.3. Another bias-variance trade-off: λ

While update frequency directly impacts bias-variance by limiting the number of transitions considered when updating, one might also consider a longer sequence of steps but apply a discount factor λ (distinct from γ) to the GAE to reduce the impact of steps further into the future. In an experiment on LunarLander-v2 where update frequency is fixed at 100 and λ is varied, we found that applying a discount on the advantage destabilizes the algorithm but also learns faster than in the undiscounted case (see Figure 3).

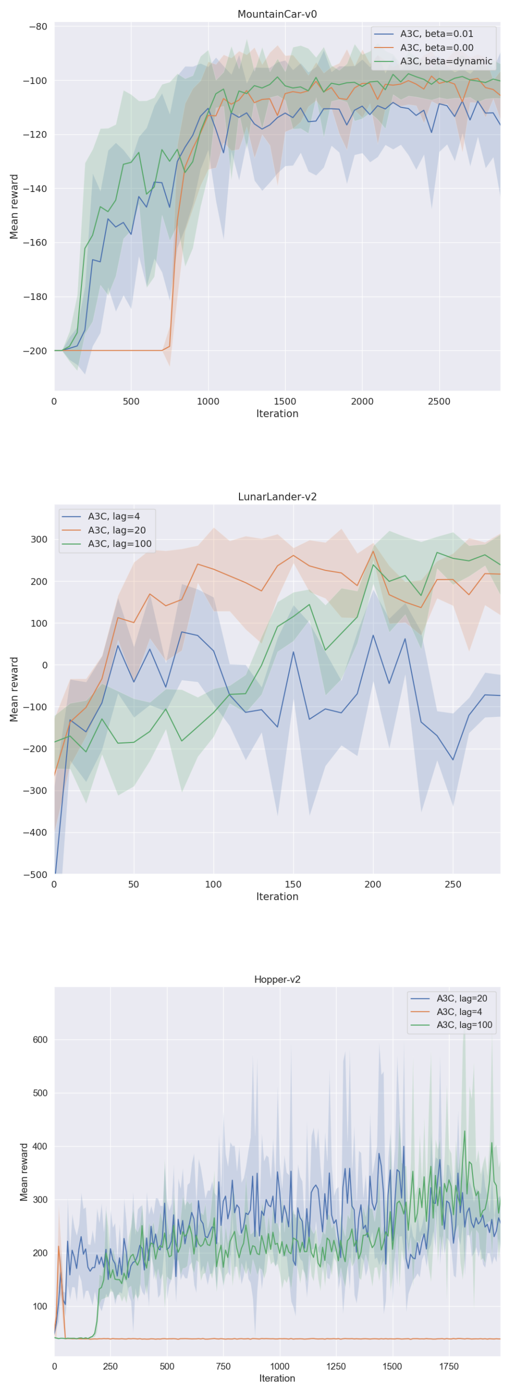

Fig. 3: *Top:* Training of three agents in MountainCar-v0 where only β is varied. In the dynamic case, $\beta = 0.01$ for the first 1000 iterations and zero afterwards. *Mid and bottom:* Training of three agents on LunarLander-v2 and Hopper-v2, respectively, where only the number of steps between each update is varied (denoted *lag*).

4.4. Scalability and number of workers

A3C leverages the CPU compute power of clusters by distributed learning in the form of multiple workers running in parallel. As can be seen in Figure 4, increasing the number of workers does in fact improve stability of the algorithm and results in faster learning – especially during initial exploration. Perhaps surprisingly, there appears to be no significant learning benefit running 64 workers versus 24 workers (aside from gain in stability) on LunarLander-v2, which might be due to 24 workers already exploring the state space sufficiently (recall that trajectories are generated on-policy). It is speculated that on more complex environments, there will in fact be a greater benefit of more workers beyond what is shown here.

4.5. Sensitivity to learning rate

Changing the learning rate is among the many different hyper-parameters that can be tuned for all environments but had significant effect on the final result for the Hopper environment. As illustrated in Figure 4 a large learning rate tends to overshoot the target and hence not improve over time, whereas a small learning rate results in slow learning. The best result was found to lie between the two values.

5. OPPORTUNITIES FOR FUTURE WORK

An extension to our current work would be to implement IMPALA [4] and compare its performance to A3C. IMPALA offloads the learning process of workers to one or more learners running on a GPU, such that workers running on CPUs solely perform rollouts. This approach increases scalability as forward passes and backward passes through large networks become the performance bottleneck in A3C.

6. CONCLUSION

In conclusion the A3C algorithm works very well, even when only being run solely on the CPU. However there have been way to many hyper-parameters to tune including: β, γ, λ, reward scaling and so on which all influence the results. So instead the learner should be on a GPU and IMPALA should be used as it scales better with more computational resources and have fewer hyper-parameters.

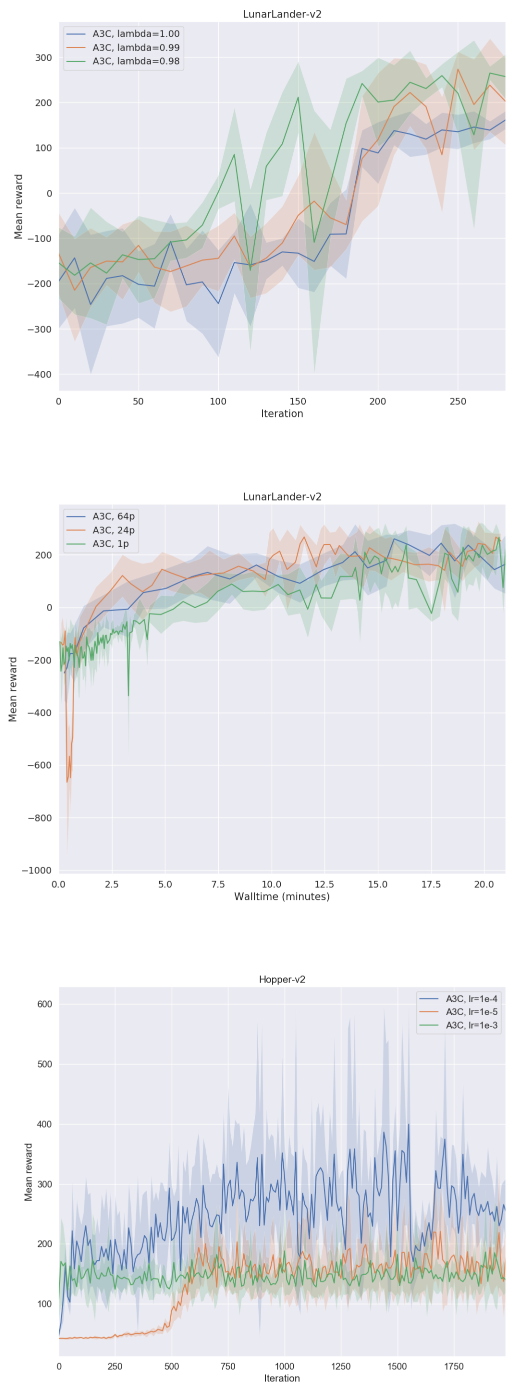

Fig. 4: *Top:* Training of three agents where update frequency is 100 steps and λ is varied.. *Mid:* Training of three agents with 64, 24 and 1 worker, respectively. Note that the horizontal axis displays walltime of training rather than number of iterations. *Bottom:* Training three identical agents with different learning rates.

7. REFERENCES

[1] V. Minh et al., "Playing atari with deep reinforcement learning," *arXiv:1312.5602 [cs.LG]*, 2013.

[2] V. Minh et al., "Asynchronous methods for deep reinforcement learning," *arXiv:1602.01783 [cs.LG]*, 2016.

[3] J. Schulman et al., "High-dimensional continuous control using generalized advantage estimatio," *arXiv:1506.02438 [cs.LG]*, 2015.

[4] L. Espeholt et al., "Impala: Scalable distributed deep-rl with importance weighted actor-learner architecture," *arXiv:1802.01561 [cs.LG]*, 2018.

[5] D. P. Kingma et al., "Adam: A method for stochastic optimization," *arXiv:1412.6980 [cs.LG]*, 2014.

[6] F. Niu et al., "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *arXiv:1106.5730 [math.OC]*, 2011.

[7] P. Nagarajan et al., "Deterministic implementations for reproducibility in deep reinforcement learning," *arXiv:1809.05676 [cs.AI]*, 2018.