Parallel Joint PPO

Albert Ge[*]
Department of Computer Science
Cranberry-Lemon University
San Jose, CA 95129
`lbertge@gmail.com`

Abstract

I implement a method for parallelizing a jointly-trained PPO algorithm on the Gym Retro Sonic games, on a single GPU and processor. Tantamount to this algorithm were improved memory efficiency and getting enough sleep, both of which were realized and successfully resolved. Parallel Joint PPO achieves about a 3x speedup on a machine with a GTX 1080 and 32GB of RAM. A working implementation can be found at the repository `https://github.com/lbertge/retro-noob/tree/master/joint-ppo`.

1 Introduction

The OpenAI Retro Contest Nichol et al. (2018) is an online competition aimed at applying transfer-learning algorithms on custom-built Sonic levels, using released Sonic games (SonicTheHedgehog-Genesis, SonicTheHedgehog2-Genesis, SonicAndKnuckles3-Genesis). On submission, an algorithm is packaged into a Docker container and executed remotely for evaluation on a set of test levels.

1.1 Background

In 2018, OpenAI released its Gym Retro repository which enables reinforcement-learning research on a slew of Sega-Genesis games via their standard Gym environment.

1.2 Previous Work

Previous work on transfer-learning for the SonicTheHedgehog franchise has suggested that a jointly-trained Proximal Policy Optimization Schulman et al. (2017) (PPO) algorithm generalizes well to the test set. Joint PPO is identical to normal PPO, with the difference being trained *simultaneously* on all environments. This means that, after playing through each level once, the algorithm produces a list of gradients obtained from each level, averages them using the arithmetic mean, and applies the resulting gradient to the policy. To accomplish this, The authors use a cluster of GPUs to execute the training in parallel, with each GPU assigned to a single level from one SonicTheHedgehog game. This requires resources which is generally above the average machine learning enthusiast's budget, which may also exceed that of this paper's author; thus, this paper seeks to find another, more efficient method for joint training, which is capable of running on more modest hardware.

Reinforcement learning is perhaps no stranger to hardware constraints; previous work on parallelizing the TRPO algorithm Schulman et al. (2015) has demonstrated that distributed RL training can result in speedup by as much as 5x over training Frans and Hafner (2016), which was also done over a home PC. While the work in this paper takes a remarkably similar approach to parallelizing TRPO,

[*]Equal contribution.

Submitted to OpenAI Retro Contest, held April 5, 2018 to June 5, 2018. Distribute all you want.

an important difference is that joint PPO trains over a set of gym environments, which faces its own memory-constraint issues to be able to accomodate for the results from all gym environments. Thus, this paper details and resolves a number of technical challenges involving parallelizing joint PPO.

2 PPO

PPO as an algorithm attempts to produce the same results as TRPO while using only first-order optimization methods, which is significantly less complex than TRPO. At its core, PPO minimizes the following objective:

$$L^{CLIP} = \hat{\mathbb{E}}_t[\text{min}(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

where $r_t(\theta)$ is the probability ratio between the current stochastic policy π_θ and its previous policy $\pi_{\theta_{\text{old}}}$ for action a_t and state s_t, or

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

and $\hat{A}_t$ is the advantage function. Recall that the advantage function approximates how optimal it is to take an action over all other actions in a given state. To get an intuitive sense for this equation, consider two scenarios: one in which the advantage is positive (i.e., good action) and one in which it is negative (bad action).

- If the advantage is positive, the only notable situation is when ratio $r_t(\theta)$ is greater than 1 (the action under the current policy is more probable than the previous policy). This would introduce a multiplying factor (>1) to $\hat{A}_t$. To prevent this factor from becoming too large, the ratio is clipped by a constant hyperparameter ϵ such that the advantage never exceeds a multiple of $1 + \epsilon$, and prevents greedy optimization of making the action too probable.

- If the advantage is negative, and the ratio $r_t(\theta)$ is less than 1 (action is less probable), the objective is clipped such that the advantage never exceeds a multiple of $1 - \epsilon$. Since the objective is negative, This places an upper bound on the objective function, and it forces the policy to not overupdate and make the action too improbable.

In both cases the clipping objective prevents the policy being updated in such a way that capitalizes excessively on improvement. Otherwise, if the policy ratio worsens the behavior, the clipping has no effect, and the objective function penalizes the policy as normal.

3 Architecture

A diagram of the system architecture used can be found in Figure 1, and a description of the algorithm can be found at Algorithm 1. Each iteration begins in the process with id 0, which will be referred to as the master process. The master process spawns worker processes (pids 1, 2, 3). Every process, including the workers, creates its own Tensorflow context, and models are synchronized via message passing through a blocking queue, for concurrency safety.

During training, each worker evaluates its model one of 47 SonicTheHedgehog Gym environments specified in the training set, producing a tuple of experiences obtained during T timesteps. Experiences here is defined as observations (raw input), advantage estimates, values, and negative log probabilities. These experiences are passed back to the master process, and gradients are produced for each set of experiences for K epochs and M minibatches. Finally, the gradients are averaged and the resulting gradients is applied to the master's copy of the model, and re-synchronized with the other worker processes.

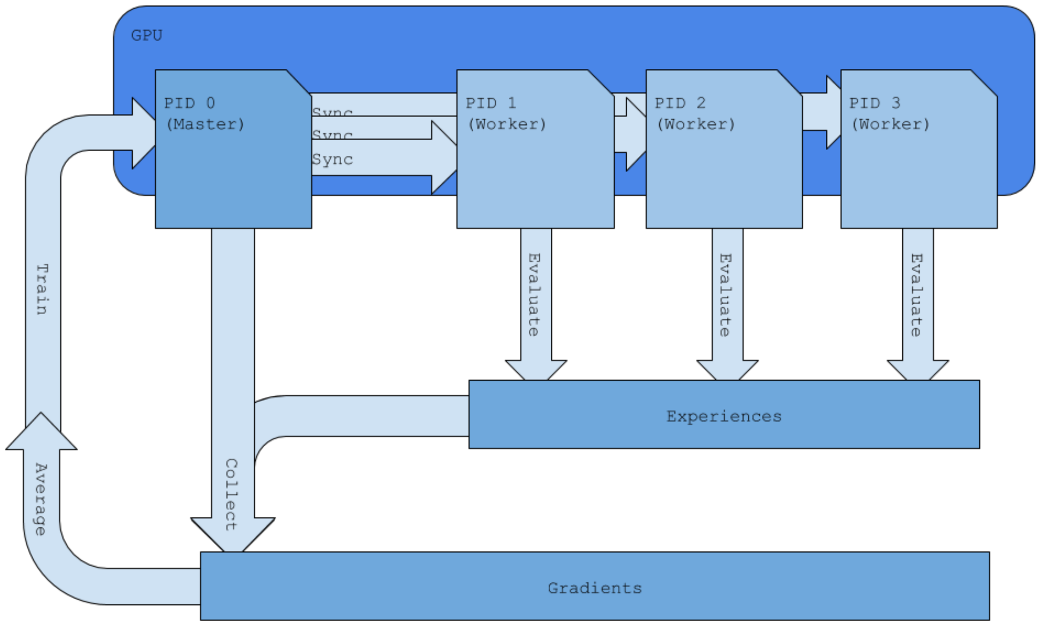

Figure 1: System Architecture for parallelizing joint PPO. Each node in the diagram represents a physical state, and each directed edge refers to an action performed in, or on that physical state. For example, inside of process with `pid 0` (Master), experiences are collected and gradients are computed/averaged.

Algorithm 1: Parallel Joint PPO

```
for iteration=1,2 do
    Enqueue 47 environments in queue
    while queue not empty do
        Run policy $\pi_{\theta_{\text{old}}}$ for T timesteps
        experiences.append(advantage estimates, neglogpacs, observations)
    end
    for experience in experiences do
        compute gradients for $K$ epochs with minibatch size $M$ (optimizing for $L_t^{CLIP+VF+S}$)
    end
    Average gradients
    Apply gradients
end
```

3.1 Preventing memory leaks

The model architecture was specifically designed to prevent memory leaking, and an overview of this design is presented in Figure 2. Two source of memory leaks were discovered, both of which were successfully resolved:

- During the **Sync** step of Figure 2, the master's model with those of the workers, the weights are pickled and sent over the blocking queue to each worker. Each worker will then load the weights by calling `tf.assign`, which is a Tensorflow operation (op) that overwrites the values in the existing graph with those obtained from the queue. However, Tensorflow ops are permanent while the context session is active, meaning that with each iteration, `tf.assign` ops will be continually added, leading to about 30MB additional memory per worker process.

  To resolve this, a single `placeholder` Tensor object and `assign` op is created on graph initialization. Since the placeholder object is only initialized once, and can accomodate array objects of variable size and shape, the weights are first placed inside the placeholder, which in turn executes the assign op to map the weights into the model. Thus, this method prevents runaway memory usage, capping a Worker's memory requirements to about 3GB.

- The second issue arises in the **Collect** step of Figure 2, when attempting to average gradients from all levels together. An important note is that the `apply_gradients` op appears to *consume* the gradient Tensor objects during execution. Thus, if no `apply_gradients` op immediately follows the `compute_gradients` op, the Tensor objects become orphaned in the graph, and it is difficult to explicitly free without killing the entire context session. In this particular case, running the model on each level environment's experiences results in a separate list of gradients, leading to about an additional 600MB per gradient Tensor.

  The solution is similar to the first scenario. In this instance, a set of `tf.Variable` objects are initialized, which are reused when computing each environment's gradients. `tf.Variable` is used instead of `tf.placeholder` because the gradients are aggregated and averaged outside of the Tensorflow context session, which would allow each Worker process to separately compute each gradient, rather than just evaluating and collecting experiences. Due to the poor author's time and budget constraints, that work remains slightly out of the scope of this paper, and hopefully remains on the horizon for future work.

Table 1 shows a list of hyperparameters used for training the algorithm.

4 Performance

Resolving the two memory issues reduces the requirements from unbounded to about 14GB for the master process, and 3GB for each Worker process. Additionally, with 3 worker processes, the parallelized PPO algorithm can process over 21 million total timesteps in about 9 hours, which is about a 3x speedup over a single process (about 7 million timesteps in 9 hours).

5 Sleep

Sleep is really important, and vital for being able to think through technical challenges without going around in circles.

6 Conclusion

This paper documents and explains the issues encountered during implementing a parallelized version of jointly-trained PPO, and the solutions found to overcome them. In particular, memory leak issues were the core issue due to the processing of many Gym environments in parallel, which were resolved by reusing heavily-trafficked nodes and ops inside of the Tensorflow execution graph. Furthermore, which is evaluated on the SONICTHEHEDGEHOG games on the Sega-Genesis. The parallelized algorithm can have a speedup of about 3x over its single-processor counterpart.

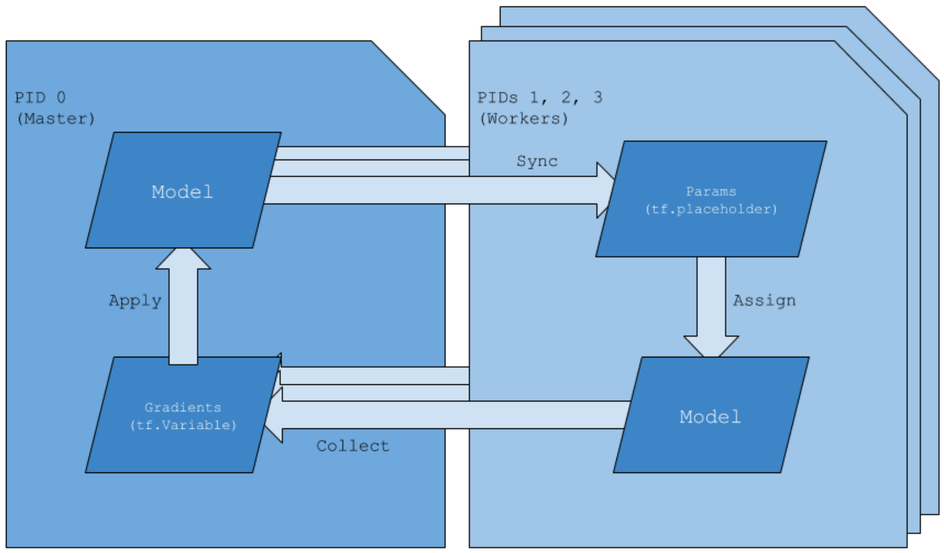

Figure 2: Model Architecture for Joint PPO. Here, each parallelogram refers to a node (or group of nodes) inside of the Tensorflow's execution graph. Each light blue arrow represents a Tensorflow operation performed. Both memory leaks occur when transferring data between Tensorflow sessions, i.e. when syncing the master's model with the those of the worker's, and when collecting experiences/averaging gradients from the workers back to the master process.

Table 1: Hyperparameters for Joint PPO training

Hyperparameter	Value
Workers	3
Horizon	4500
Epochs	4
Minibatch size	1125
Discount γ	0.99
GAE Λ	0.95
Clipping ϵ	0.2
Entropy coeff ϵ	0.2
Reward scale	0.01

6.1 Future Work

There still remains much future work to be done, such as actually evaluating the agent and confirming that it has improved performance over its meta-learning-less counterpart, normal PPO. Initial results look promising and will be released in an updated version of this paper.

Acknowledgments

Thanks to the OpenAI team for releasing and maintainining the Gym Retro environments, the baselines PPO algorithm implementation, and for giving the the author two months of wild speculation about how to improve agent evaluation.

References

Frans, K. and Hafner, D. (2016). Parallel trust region policy optimization with multiple actors.

Nichol, A., Pfau, V., Hesse, C., Klimov, O., and Schulman, J. (2018). Gotta learn fast: A new benchmark for generalization in RL. *CoRR*, abs/1804.03720.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization. *CoRR*, abs/1502.05477.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.