Combining Policy Gradient and Genetic Approaches for a Hybrid Deep Reinforcement Learning Algorithm

Marshall, Kelly

Saturday 1st September, 2018

1 Introduction

By tapping into the complex pattern recognition abilities of neural networks, the field of Deep Reinforcement Learning has offered a powerful solution to the traditional Reinforcement Learning problem. The problem in question involves optimizing the interaction of an agent with an environment. The environment changes based on a stochastic process which is a function of its current state and the action taken by the agent. The resulting interaction between the agent and environment is cyclical, as the agent uses its policy to provide an action based on the state; Subsequently, the environment uses the chosen action as well as its current state to determine a new state along with a quantity of reward or penalty based on the action selected. The selection of the optimal action at any time step therefore amounts to finding the action which optimizes the total reward to be accrued at every following timestep. In training a policy network to accept a state and in turn output an optimal action, the most commonly used methods involve computing the analytic gradient of the network's parameters. However, recent work has shown genetic algorithms, a gradient-free approach, to be an effective alternative. The goal of this paper is to improve upon the traditional genetic algorithm by performing gradient updates on members of the population.

2 Existing Deep Reinforcement Learning Algorithms

This section covers the aim and methods of both traditional policy gradient algorithms as well as genetic algorithms. The first criteria on which Deep Reinforcement Learning algorithms are typically evaluated is "Wall clock time", for which we assume infinite access to data generation and aim to reduce the time needed for training the policy to optimal performance. Alternatively, we may wish to improve the sample efficiency of an algorithm, for which cases we are able to disregard training time, but wish to minimize training costs associated with collecting data, as in the case of training physical agents prone to wear.

2.1 Vanilla Policy Gradient

In order to use gradient-based methods to train a policy network's performance on some task, one must select a cost function which can serve as a metric of the policy's success. One is then able to repeatedly compute the gradient of the cost function with respect to the network's parameters and shift the parameters in the direction opposite this gradient. In the case of DRL, our stated goal is to maximize expected reward, meaning the quantity we wish to minimize for any given action decided upon by our policy is the total reward accrued after that action multiplied by -1. Using this method, each iteration of the policy gradient algorithm uses data from several trajectories sampled by allowing the policy to interact with the environment for several trials, then taking a single gradient step.

By itself, vanilla policy gradient tends to perform relatively poorly with regard to both sample efficiency as well as wall-clock time, primarily due to each trajectory only providing a single sample -and therefore extremely high-variance- estimate of the reward gained from a particular pair of states and actions. However, there exist many gradient-based algorithms which greatly improve upon the performance of Vanilla PG, most of which use a second "Critic" network to give a better estimate of the relative value of each state.

The Proximal Policy Optimization(PPO) and Actor Critic with Experience Replay (ACER) algorithms both further improve upon the sample efficiency of policy gradient by using importance sampling methods to reuse old trajectories and fully take advantage of all data collected rather than only using each trajectory for a single gradient update[1] [2]. Additionally, the Asynchronous Actor Critic (A3C) algorithms speeds up training time by simultaneously running several agents in parallel and combining the gradients computed by each to compute more reliable and frequent gradient steps[3].

Algorithm 1 Policy Gradient Algorithm [4]

while not done **do**
  sample s_i, a_i from $\pi_\theta(a_t|s_t)$
  Calculate reward-to-go R_i for each timestep and normalize rewards
  $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) R_i$
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end while

2.2 Genetic Algorithm

Owing their recent popularity to Uber AI's 2017 paper[5], genetic algorithms have proved to be a strong alternative to existing DRL algorithms. Genetic algorithms are an almost completely unique group of methods for training neural networks due to the fact that they do not involve computing any gradient. Rather than beginning with a single policy network and iteratively improving this policy, genetic algorithms instead begin by randomly initializing a "population" of numerous policy networks and iteratively improving the entire population. This is done by running trajectories using each policy in the population and scoring the policies based on the average reward that they were able to attain. The next generation is generated by randomly selecting members from the previous generation and adding random noise to their parameters; this can be thought of as the mutation which allows for natural selection. However, the improvement of the population from one generation to the next lies in the fact that not all members of the previous population are eligible to be selected. Only those whose average sampled reward ranks them amongst a pre-fixed number of survivors can influence the next generation, leading to successively higher fitnesses over time. Additionally, to ensure monotonic improvement, the single highest ranking member of the previous generation -called the 'elite' - is included unmutated in the next generation. While it would seem that maintaining a population of several policies would be extremely inefficient compared to a single policy, the power of parallelization allows the operations for each member to be completed simultaneously.

3 Hybrid Algorithm

It was the goal of this project to improve upon the genetic algorithm described above by performing gradient steps on the members of the population. The logic behind the algorithm's design is that the rollouts generated to evaluate the fitness of population's members provide all the data necessary to perform policy gradient updates on the members of the population. Thus, any performance increase due to taking these gradient step would come at no cost with respect to sample efficiency. This addition to the original genetic algorithm would increase the time taken by each iteration due to the additional computational complexity needed to perform backpropagation. However, the aim of adding policy-gradient computation is to slightly improve the fitness of the population each generation, causing the population to converge to an optimal solution in fewer generations without increasing the amount of data required per generation. We therefore disregard wall-clock time and use sample efficiency as our only criteria.

3.1 Algorithm Design

Algorithm 2 Vanilla Genetic Algorithm (GA)

for $g = 1, 2, .., G$ generations **do**
  for $i = 1, 2, .., pop - 1$ **do**
    if g=1 **then**
      $\theta_i^{g=1} = \phi()$
    else
      k=uniformRandom$(1, T)$
      $\theta_i^g = \psi(\theta_k^{g-1})$
    end if
    Sample set of trajectories $T = \tau_1, \tau_2, .., \tau_N$
    Evaluate $F(\theta_i) = (\sum_{\tau \in T} r(\tau))/N$
  end for
  Sort $\theta_1^g, .., \theta_i^g$ with descending order by $F(\theta_i)$
  if g=1 **then**
    $\theta_{pop}^{g=1} = \phi()$
  else
    Set Elite $\leftarrow \theta_1^{g-1}$
    $\theta_{pop}^g \leftarrow$ Elite
  end if
end for

3.2 Results

Network Hyperparameters:
Policy network architecture: One hidden layer, with 32 units, using ReLU activation.
Population Size: 100
The below graphs display the relative sample efficiency of the hybrid algorithm with different learning rates compared to the vanilla genetic algorithm.

Environment: HalfCheetah-v2

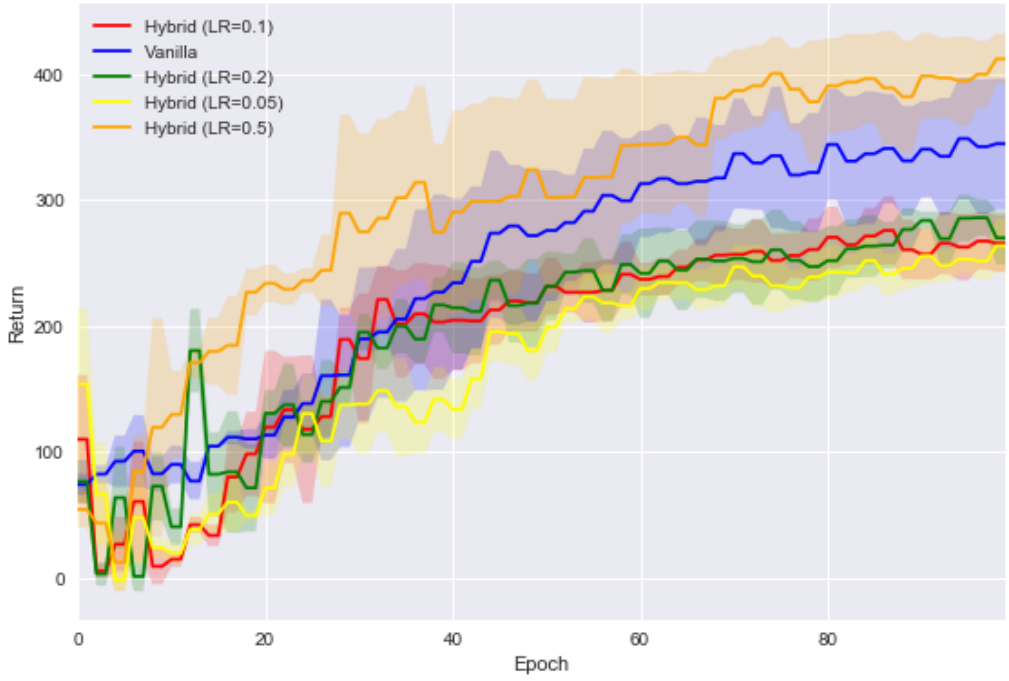

Figure 1: Half Cheetah with σ=.5,population size=100, and survivors =30

Algorithm 3 Policy-Gradient Genetic Algorithm Hybrid (PGGA)

Hyperparameters:
Number of iterations: N
Population size: pop
Number of generations: G
Number of Survivors: S
Sigma: σ
Learning Rate: α
Assumed Functions:
mutation function: ψ
policy initialization method: ϕ

---

for $g = 1, 2, .., G$ generations **do**
  for $i = 1, 2, .., pop - 1$ **do**
    if g=1 **then**
      $\theta_i^{g=1} = \phi()$
    else
      k=uniformRandom$(1, T)$
      $\theta_i^g = \psi(\theta_k^{g-1})$
    end if
    Sample set of trajectories $T = \tau_1, \tau_2, .., \tau_N$
    Evaluate $F(\theta_i) = \sum_{\tau \in T} r(\tau)/N$
    Evaluate $J(\theta_i) = \sum_{\tau \in T} \sum_{a_t, s_t \in \tau} (\log \pi_\theta(a_t|s_t) \sum_{t' > t} r_t)$
  end for
  Sort $\theta_1^g, .., \theta_i^g$ with descending order by $F(\theta_i)$
  for $i = 1, 2, ..., T$ members of current population **do**
    Compute $\text{grad}_i = \nabla_{\theta_i^g} J(\theta_i^j)$
    $\theta_i^g \leftarrow \alpha * \text{grad}_i$
  end for
  if g=1 **then**
    $\theta_{pop}^{g=1} = \phi()$
  else
    Set Elite $\leftarrow \theta_1^{g-1}$
    $\theta_{pop}^g \leftarrow$ Elite
  end if
end for

Figure 2: Half Cheetah with σ=.05,population size=100, and survivors =30

Environment: Pendulum-v0

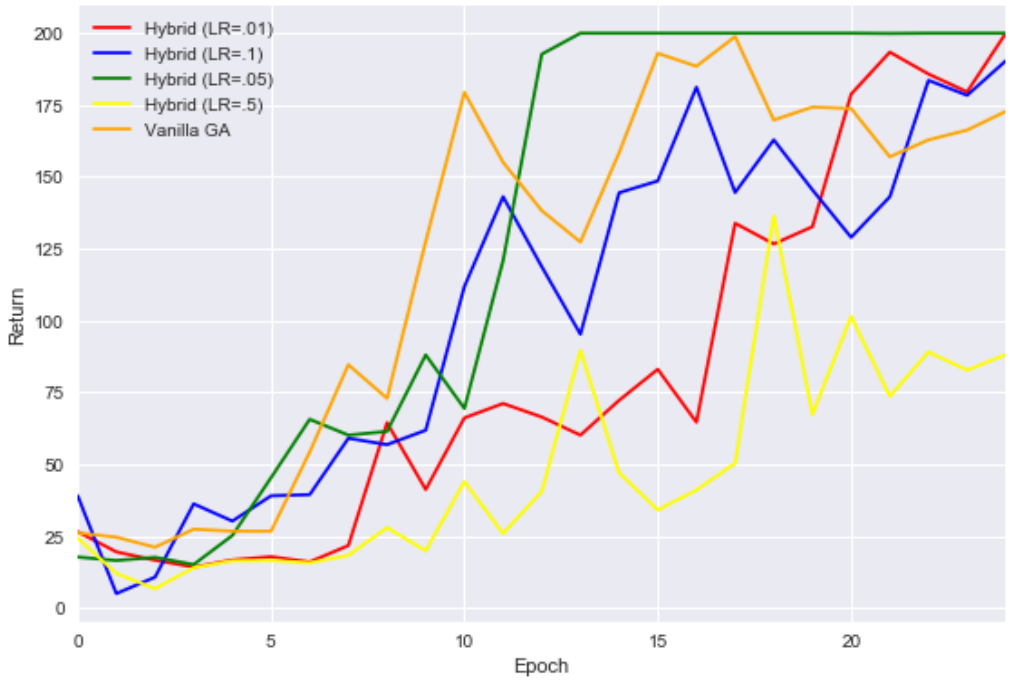

Figure 3: Inverted Pendulum with σ=.5,population size=100, and survivors =30

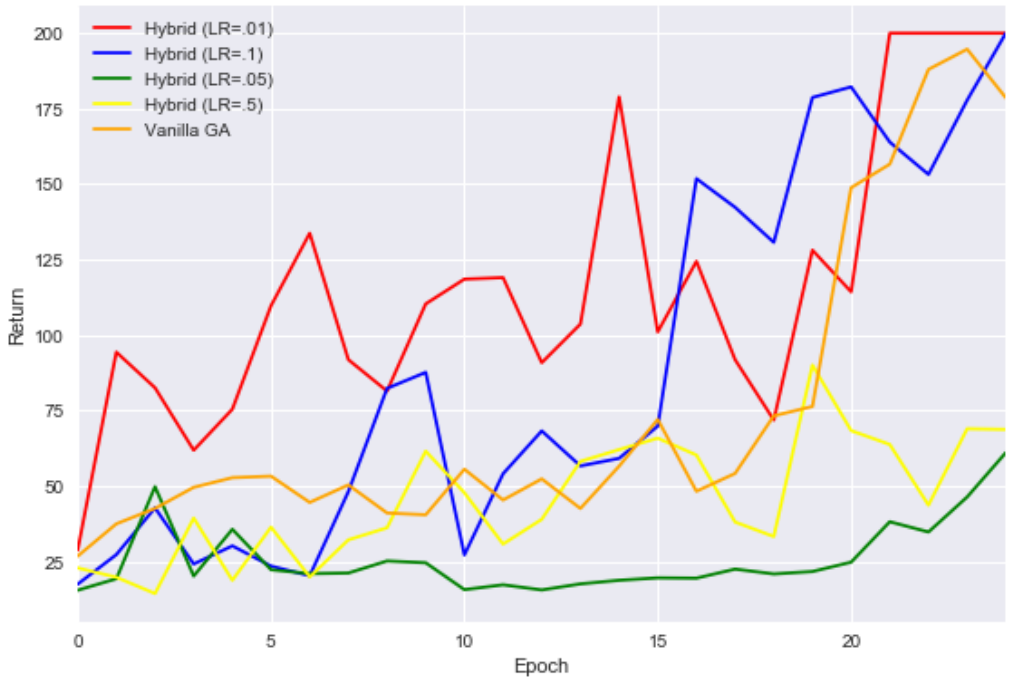

Figure 4: Inverted Pendulum with σ=.05,population size=100, and survivors =30

Environment: Humanoid

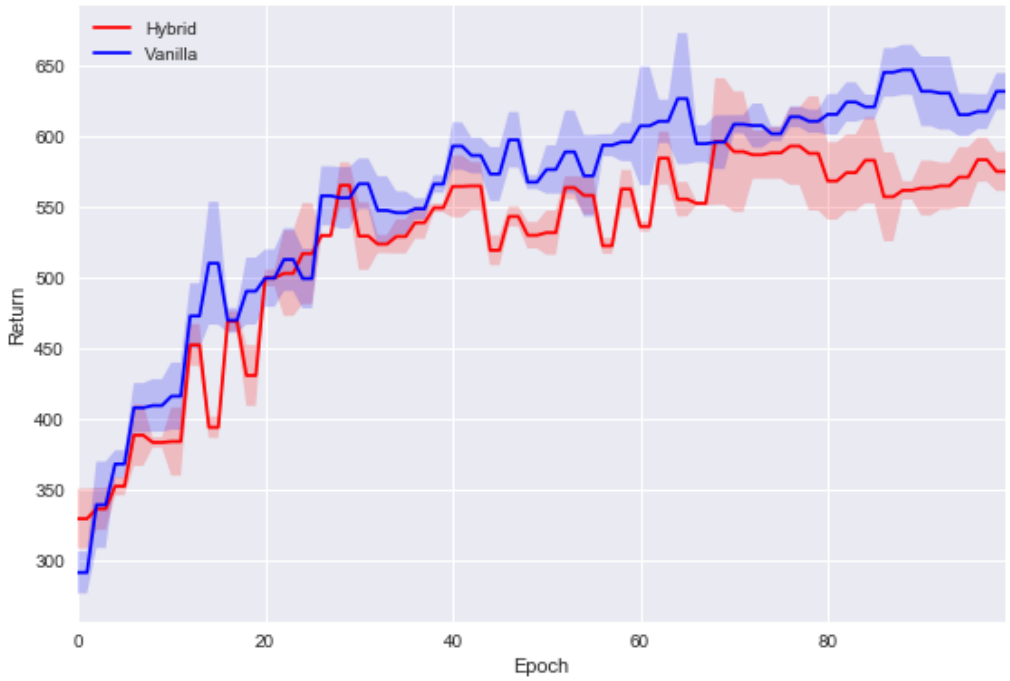

Figure 5: Humanoid task with σ=.5,population size=100, and survivors =30

3.3 Conclusion

Based on the results of testing our hybrid algorithm on MuJoCo environments, it seems the algorithm fails to improve upon the sample efficiency of the vanilla genetic algorithm. This implies that the gradient steps taken more often did harm than good and were not able to improve the members of the population. This is most likely due to the high variance involved in estimating the gradient. Additionally, it proved difficult to reliably train a critic network for the policies, leading to even greater variance.
In order to address this problem, future work could focus on using importance sampling to obtain a more reliable estimate for the gradient. By incorporating elements of the PPO algorithm, the hybrid algorithm could more reliably shift its population members in beneficial directions. Additionally, further research could be done to determine the best manner for training critic networks to accurately assess members of the population.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[2] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[4] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[5] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.