Hybrid Deep Reinforcement Learning: Combine the best of Evolutionary Computation and Policy Gradient Methods

Zhou, Yin
Lu, Ziyu
Shi, Yuliang

August 20, 2018

1 Introduction

In the recent years, with the development of deep learning and reinforcement learning, deep reinforcement learning (DRL) has become an important research area in the AI research. The main idea of deep reinforcement learning is to apply deep neural network to traditional reinforcement learning problems, mirroring how humans learn to best exploit the environment from exploring and interacting with the environment. Among different DRL algorithms, Policy Gradient methods have been popular among scientists for years, and a lot of variations, including Actor-Critic[1], TRPO[2], PPO[3], have been proposed since the emergence of the vanilla Policy Gradient method, REINFORCE[4]. These methods are all based on the the idea of Markov Decision Process, and the neural networks are optimized through back-propagating gradients. Nevertheless, recent works have shown that a class of black box optimization algorithms inspired by natural evolution and selection, such as Evolution Strategies (ES)[5] and Genetic Algorithms (GA)[6], can serve as a strong alternative to the Policy Gradient methods. Performing direct policy search, these algorithms require no policy differentiation and no back-propagation. Different as they are, the two families of algorithms have their respective advantages and disadvantages. For example, while convergence is usually guaranteed for Policy Gradient methods, there is always concern for the policy getting stuck in local optima; and while ES and GA explore more aggressively in the policy space than Policy Gradient methods, they suffer from poorer sample complexity compared to their gradient-based counterpart on harder environments. In the summer of 2018, under the supervision of Prof. Keith Ross, Dean of Engineering and Computer Science at NYU Shanghai, and Che Wang, Ph.D. candidate in Computer Science at NYU, we have been experimenting with various approaches of combining these two families of algorithms together, hoping to find an approach that can use the advantage of one kind of algorithms to address the problem of the other, or even better, that can combines their advantages together.

2 Combine Evolution Strategy with vanilla Policy Gradient

Algorithm 1 ES/PG, with adaptive weight

Input: ES learning rate α, noise standard deviation σ, initial policy parameters θ_0, initial policy gradient weight w_{pg}
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \cdots, \epsilon_n \sim N(0, I\sigma)$
  Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \cdots, n$
  Compute policy gradient g of θ_t
  $\theta_{t+1} \leftarrow \theta_t + w_{pg}g + (1 - w_{pg})\alpha\frac{1}{n\sigma}\sum_{i=1}^{n} F_i\epsilon_i$
  $w_{pg} \leftarrow 10 * w_{pg}$

The idea of adaptive weights comes from the observation that this combination generates decent results only when the weight for policy gradient is sufficiently small. Experiment results with some fixed weights on the InvertedPendulum-v2 environment also indicates that smaller weights for policy gradient tend to generate better results. Those experiment results can be seen in the figure 1.
Note: "no pg" refers to vanilla ES; w_pg refers to the weight for policy gradient in each experiment.

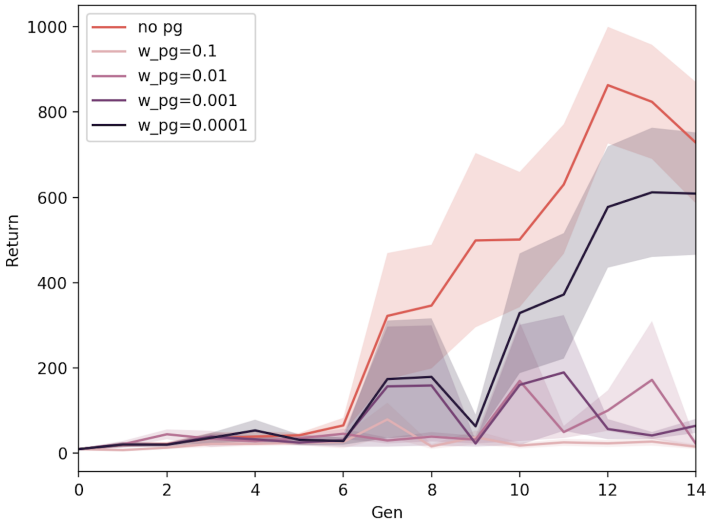


Figure 1

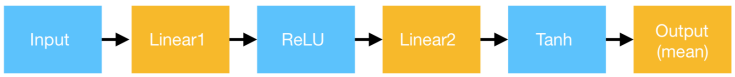
In addition, there is another observation that the performance of Evolution Strategy can still be volatile after the optimum is reached. This observation probably comes from the fact that Evolution Strategy doesn't guarantee convergence. Since Policy Gradient

methods usually have a more stable performance improvement and the gradient of a policy tends to zero as the policy reaches the optimum, we hope that by increasing the weight for policy gradient as more generations have been generated, the policy can be more stable after it reaches the optimum.
Since we want the weight for policy gradient to be small enough at the beginning and gradually get larger, and the sum of the weight for the gradient and the weight for Evolution Strategy update to be one, in each experiment, we started with the weight for policy gradient set to be less than or equal to $10^{-\text{total gen}}$, and the weight for policy gradient becomes 10 times larger after each generation.

2.1 Experiment results on InvertedPendulum environment

Below are some results we got on the InvertedPendulum-v2 environment as well as the settings we used:

- Neural network architecture:



- Standard deviation for sampling actions: 0.1
- Number of hidden units: 4
- Population size: 100
- α: 5
- σ: 1
- Policy gradient batch size: 1000
- Results were averaged over 7 random seeds (1,2,3,4,5,6,7)

2.1.1 ES/PG, with adaptive weight

Note: "no pg" refers to vanilla ES; w_pg refers to the initial weight for policy gradient in each experiment; the learning curves "no pg" and "w_pg=1e-25" overlap each other.

In figure 2, it is clear that experiments with the weight for policy gradient initialized to 1e-20 or 1e-25 give decent results. Meanwhile, the performance of Algorithm 1 is not so satisfactory if the initial weight for policy gradient is set to 1e-15. In particular, its performance suffers from significant decrease in the last few generations, as the weight for policy gradient increases to 0.01 or 0.1.

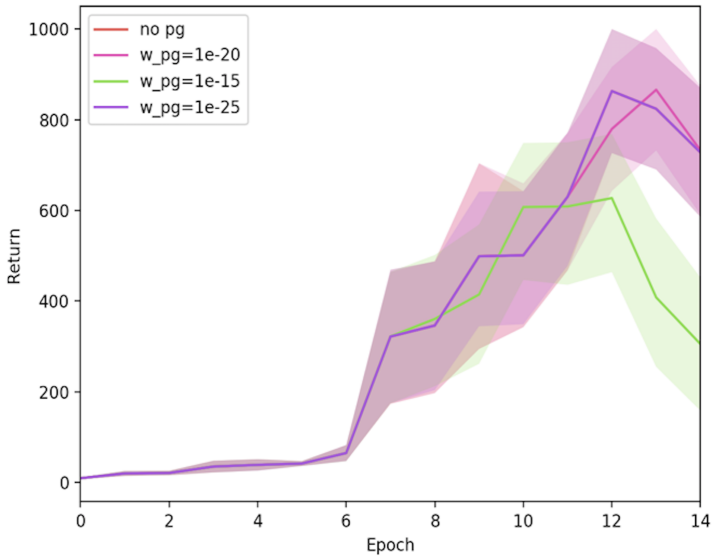


Figure 2

2.1.2 ES/PG, with ES weight decay

To see the impact of the gradients, we also experimented with decreasing the weight for Evolution Strategy update as in Algorithm 1, but without policy gradient computed. The results are shown in figure 3.

Note: The learning curves for "no pg", "w_pg=1e-25", as well as "w_es decay=1e-20" overlap each other; "w_pg decay=1e-15" means that in the nth generation (n$\geq$0) the weight for evolution strategy is equal to $1 - (1e - 15) * (10^n)$.

Among these methods, it seems that "w_es decay=1e-15" gives better result, while "w_pg=1e-20", "w_pg=1e-25", and "w_es decay=1e-20" give decent results as well.

2.1.3 ES/PG, with fixed weight

We also experimented with small fixed weights for policy gradient to see whether fixed weights can give decent results as well if small enough. The results are shown in figure 4.

Among the weights tested, 5e-6 achieves better result than vanilla Evolution Strategy in the first 7 generations, but none of them has performance comparable to that of vanilla Evolution Strategy by the end of the experiment.

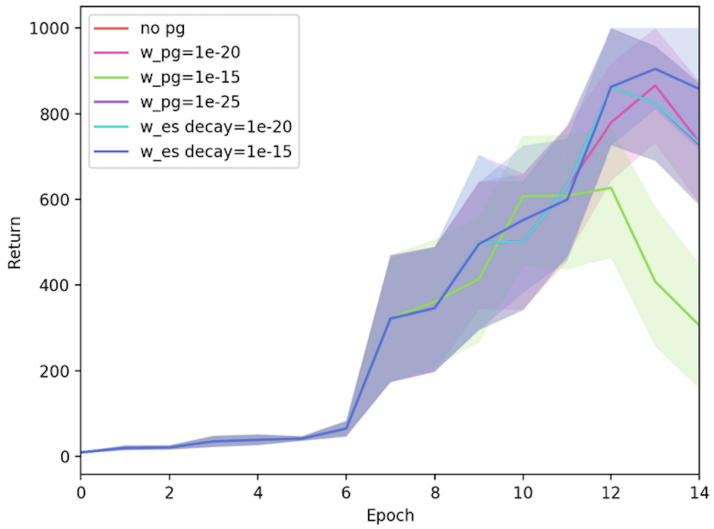


Figure 3

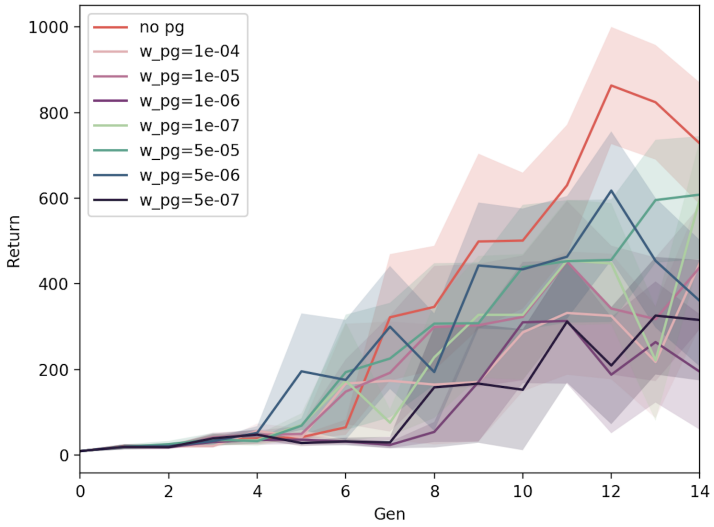


Figure 4

2.1.4 ES/PG, with adaptive weight and fixed weight

Figure 5 compares some results generated with fixed weights and adaptive weights for policy gradient, as well as the result of vanilla Evolution Strategy.

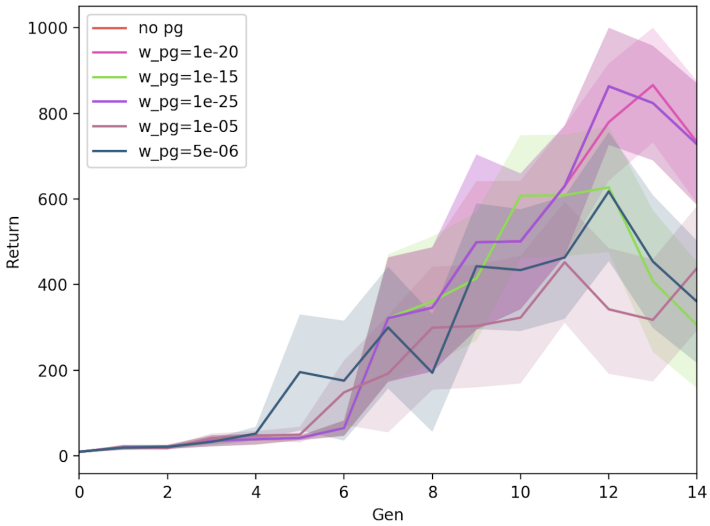


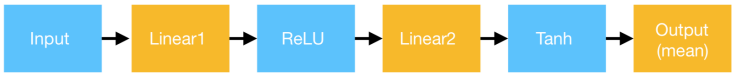
Figure 5

It can be seen from figure 5 that although fixed weight 5e-6 outperforms all other configurations during the first 7 generations, only adaptive policy gradient weights can achieve performances comparable to that of vanilla Evolution Strategy by the end of experiments.

2.2 Experiment results on HalfCheetah environment

Below are some results as well as the settings on the HalfCheetah-v2 environment:

- Neural network architecture:



- Standard deviation for sampling actions: 0.05
- Number of hidden units: 32
- Population size: 100
- α: 0.1
- σ: 0.05
- Results were averaged over 5 random seeds (1,2,3,4,5)

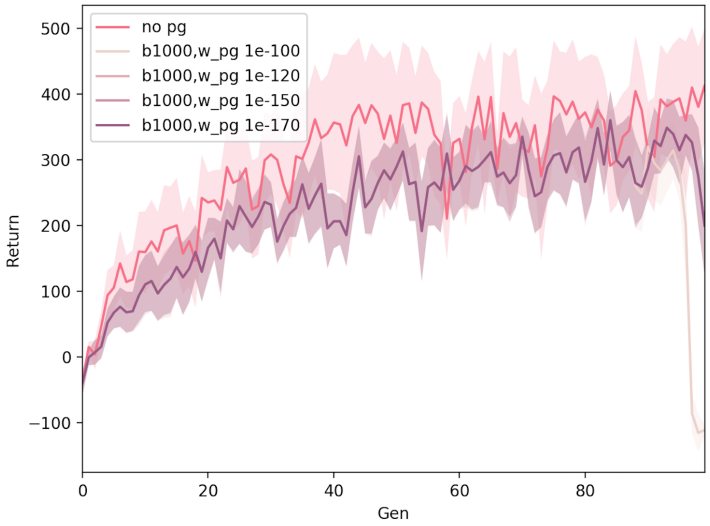


Figure 6: ES&PG, 100 generations, maximum path length 200, policy gradient batch size 1000

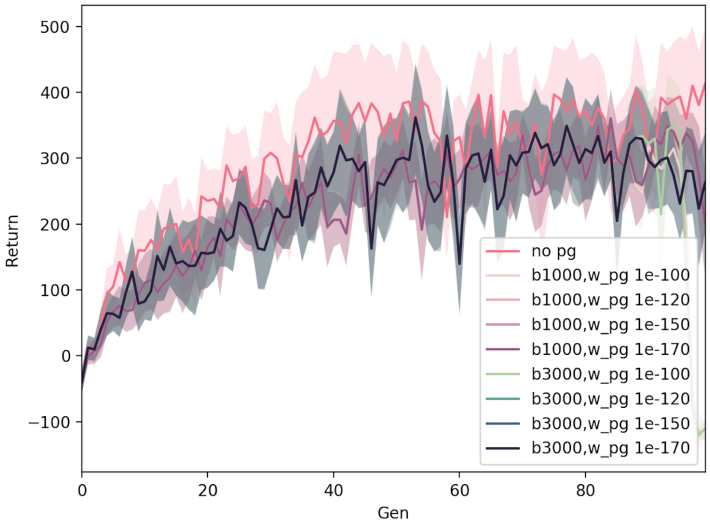


Figure 7: ES&PG, 100 generations, maximum path length 200, policy gradient batch size 1000&3000

Figures 6 and 7 compare performances of vanilla ES and Algorithm 1 with different policy gradient batch sizes.
Among the weights tested, it seems that vanilla ES still gives the best result. And

from figure 7 it seems that larger batch size doesn't make a significant improvement on the performances of Algorithm 1. We didn't test weight decay on this environment, because as the number of generations is large in this case, weight decay for Evolution Strategy update won't be effective until the last few generations.

3 Combine Evolution Strategy with advanced Policy Gradient methods

As vanilla Evolution Strategy always outperforms vanilla Policy Gradient as well as Algorithm 1, it seems to us that this naive combination of ES update and Policy Gradient update is not very promising. We then proposed a new algorithm, which performs one update, either ES or PG, at a time, instead of combining two updates together.

Algorithm 2 ES/PG, with learning goal

Input: ES learning rate α_1, PG learning rate α_2, noise standard deviation σ, initial policy parameters θ_0, learning goal G
for $t = 0, 1, 2, \dots$ **do**
  Compute return $F(\theta_t)$
  if $F(\theta_t) < 0.8 * G$ **then**
    Sample $\epsilon_1, \cdots, \epsilon_n \sim N(0, I\sigma)$
    Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \cdots, n$
    $\theta_{t+1} \leftarrow \theta_t + \alpha_1 \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
  else
    Compute policy gradient $\nabla_\theta J(\theta)$ of θ_t
    $\theta_{t+1} \leftarrow \theta_t + \alpha_2 \nabla_\theta J(\theta)$

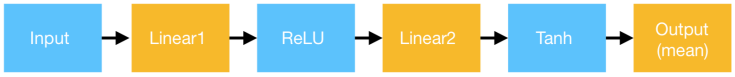
Algorithm 2 will perform Evolution Strategy update when the return of the center network is below the threshold (80% learning goal), typically at the beginning of experiments. By doing so, we are able to take advantage of the observation that Evolution Strategy explores more aggressively and learns faster (especially at the beginning of the experiments) than Policy Gradient methods. Once the return of the center network reaches the threshold, we will follow the policy gradient update in that generation. Since the policy seems to be close to an optimum, it is reasonable to reduce exploration and approaches the optimum in a more steady fashion.
In order to avoid performance collapse after switching to policy gradient update from Evolution Strategy update, we need to find a policy gradient method of which the performance can approximately match that of vanilla Evolution Strategy. Since the performance of vanilla Policy Gradient is not able to match that of vanilla Evolution Strategy, we implemented Algorithm 2 with two more advanced policy gradient methods, the Actor-Critic algorithm and the PPO algorithm.

3.1 Combine Evolution Strategy with Actor-Critic

When implementing Algorithm 2 with Actor-Critic algorithm, we experimented with updating baseline multiple times using mini-batch gradient descent, as well as using different learning rates for the Actor and Critic networks.

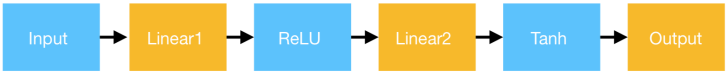
3.1.1 Multiple updates on baseline network

As the prediction accuracy of the baseline network is critical to the performance of the Actor-Critic algorithm, we update the baseline network not only in each policy gradient step, but also in each Evolution Strategy generation, using data generated from fitness evaluation. We also tried mini-batch gradient descent by dividing the training data into several mini-batches and then running one epoch.
Figures 8-13 compare the performances of Actor-Critic algorithm with different learning rates as well as different mini-batch sizes on the HalfCheetah-v2 environment. Neural network architecture and hyper-parameter setting are consistent in all these experiments, and are listed below:
Actor:



- Standard deviation for sampling actions: 0.05
- Number of hidden units: 32

Critic:



- Number of hidden units: 32

Note: Label "0.0001 128" means that the learning curve plots the result of the Actor-Critic algorithm with learning rate 0.0001 and minibatch size 128 for gradient descent. "buo" (baseline update once) means that baseline update was made through batch gradient descent instead of minibatch.
From the graphs we can see that in five out of six learning rate settings, using minibatch gradient descent achieves better result than simply performing batch gradient descent.

Figures 14 and 15 give a further comparison of different learning rates as well as minibatch sizes. From the comparison, it seems that Actor-Critic algorithm with learning rate 0.001 and minibatch size 256 achieves the best performance among all tested settings.

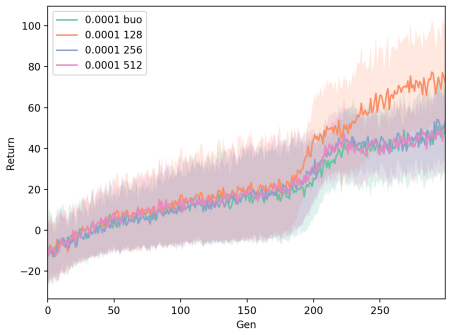


Figure 8

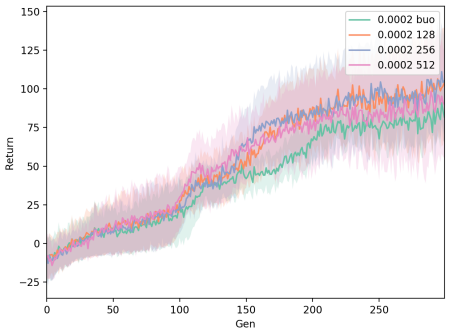


Figure 9

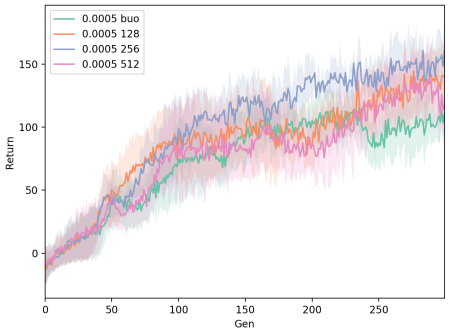


Figure 10

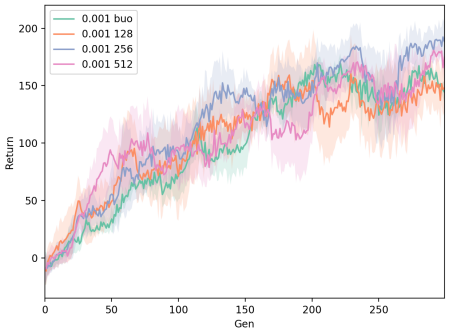


Figure 11

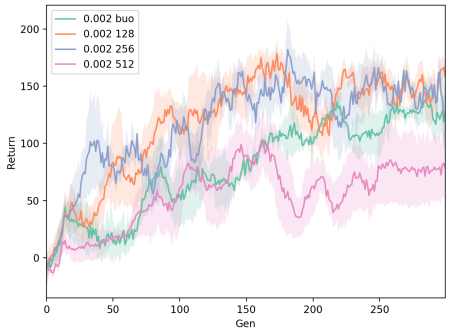


Figure 12

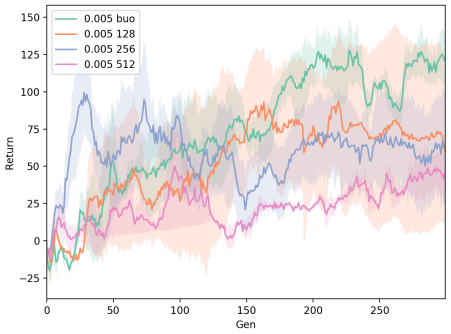


Figure 13

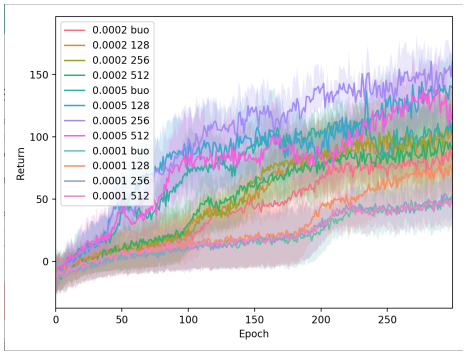


Figure 14

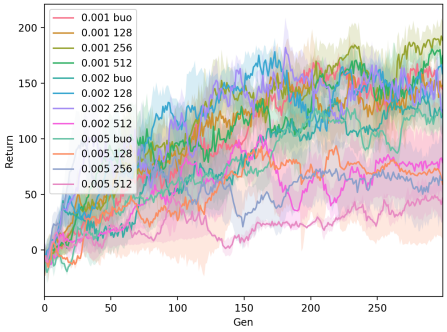


Figure 15

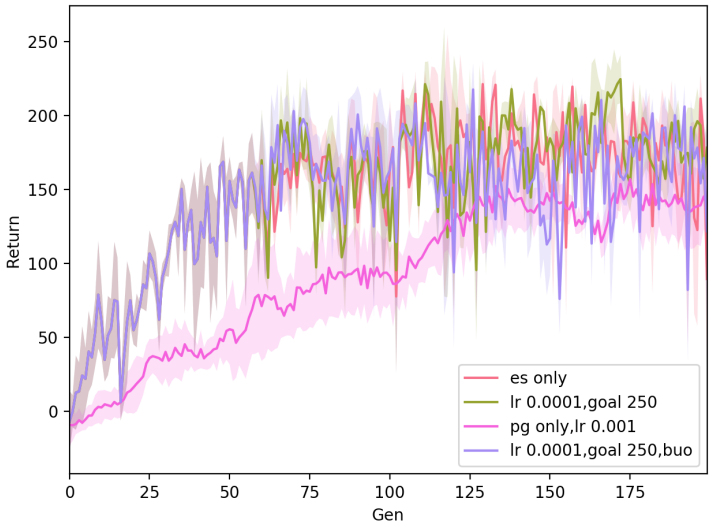


Figure 16

Figure 16 offers a comparison of the performances of Actor-Critic algorithm with learning rate 0.0001, vanilla Evolution Strategy, as well as Algorithm 2.
Note: "es only" is the learning curve for vanilla Evolution Strategy; "pg only, lr 0.001" is the learning curve for Actor-Critic algorithm with learning rate 0.001 and batch gradient descent (the vanilla Actor-Critic algorithm with the best performance); "lr 0.0001, goal 250" is the learning curve for Algorithm 2, with learning rate 0.0001, learning goal 250, and minibatch size 256; "lr 0.0001, goal 250, buo" is the learning curve for Algorithm 2, with learning rate 0.0001, learning goal 250, and batch gradient descent. The hyper-parameters for Evolution Strategy are the same as the ones specified in Section 1.2.
From the graph we see that vanilla Evolution Strategy still outperforms vanilla Actor-Critic algorithm. The performance of Algorithm 2 matches that of vanilla ES, and is

even more stable than that of vanilla ES once the threshold in this algorithm, 80%learning goal, is achieved. Comparing the performances of Algorithm 2 with and without minibatch gradient descent, it seems that minibatch gradient descent helps to improve the stability of performance.

3.1.2 Different learning rates for Actor and Critic

To further improve the performance of Algorithm 2, we experimented with using different learning rates for the Actor and Critic networks separately. Experiment results are shown in figures 17-22. Neural network architecture and hyper-parameters in these experiments are the same as in Section 2.1.1.
Note: "Alr 0.0001, Clr 0.0001" means that the learning curve plots the result of Algorithm 2, with Actor network learning rate 0.0001 and Critic network learning rate 0.0001. Figures 17-19 each compares results with one fixed Actor network learning rate and six different Critic network learning rates, while figures 20-22 each compares results with one fixed Critic network learning rates and six different Actor network learning rates
However, among the various different combinations of learning rates we have tested, none of them seems to achieve significantly better result than a shared learning rate between the Actor and Critic networks. Based on these results, we concluded that using different learning rates for the Actor and Critic networks separately is not very helpful in improving the performance of Algorithm 2.

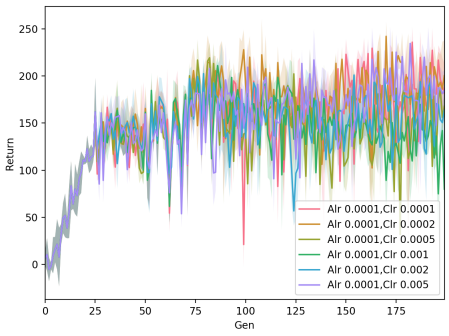


Figure 17

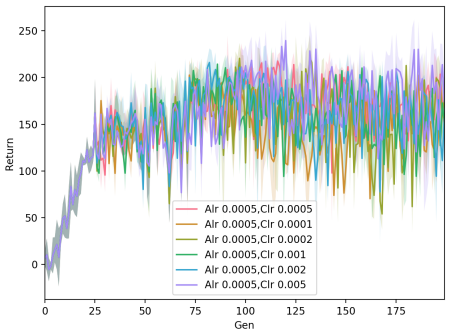


Figure 18

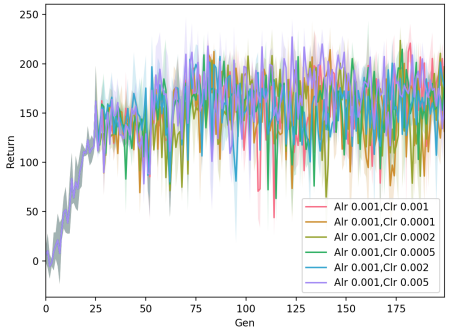


Figure 19

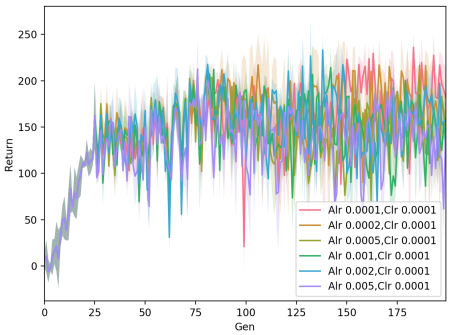


Figure 20

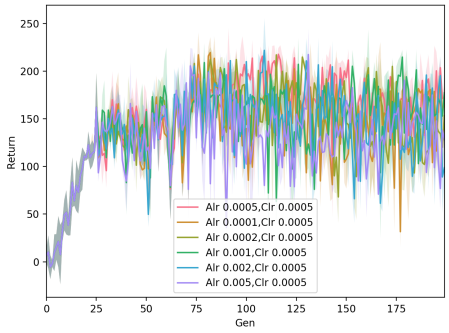


Figure 21

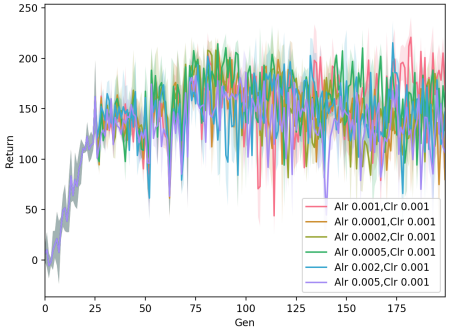


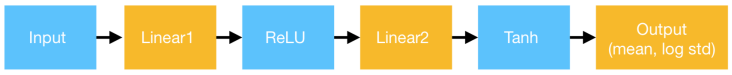
Figure 22

3.2 Combine Evolution Strategy with PPO

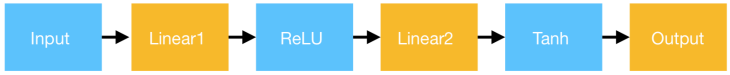
We also implemented Algorithm 2 with PPO algorithm, one of the most powerful Policy Gradient methods known now, in order to improve the performance of Algorithm 2. Since PPO algorithm has the reputation for being sample efficient, this PPO version of Algorithm 2 also has the potential to outperform vanilla Evolution Strategy in terms of sample efficiency. Among the environments we tested, the sample efficiency (amount of data generated when running a fixed number of generations/iterations) of PPO algorithm is always better than that of Evolution Strategy. Thus we expected Algorithm 2 to be more sample efficient than vanilla Evolution Strategy as well.
The hyper-parameter setting of PPO algorithm as well as neural network architecture is shared in all the following experiments, and is listed below:

- episode length: 100

- minibatch size per update: 64
- number of epoch per iteration: 10
- learning rate: 3e-4
- weight decay rate: 1e-3
- lambda value in generalized advantage estimator: 0.95
- Neural network architecture:
  Policy network:



  Baseline network:



3.2.1 Experiment results on InvertedPendulum environment

Below are the additional hyper-parameters we used on the InvertedPendulum-v2 environment:

- α: 5
- σ: 1
- Number of hidden units: 32
- Number of generations: 50

We experimented with different population sizes for Evolution Strategy as well as different number of episodes per PPO iteration. Results are shown in figures 23 and 24. Note: "IP_es_pop100_505000" means that the learning curve plots the result of vanilla ES with population size 100 on the InvertedPendulum-v2 environment, and the total time steps of this experiment is 505000. "IP_pg_nep21_105000" means that the learning curve plots the result of PPO algorithm with 21 episodes generated per iteration, and the total time steps is 105000. All results were averaged over three random seeds. The graphs imply that enlarging population size in Evolution Strategy and increasing the number of episodes generated per PPO iteration both have a positive effect on the performances of the algorithms. Comparing the two graphs, it's clear that the learning curves of vanilla ES grow much faster than those of PPO, especially at the beginning of the experiments. However, the curves of PPO are much more smooth than those of ES, implying a more stable policy performance improvement and a stronger tendency to converge. What's more, the amount of data generated in each experiment using PPO

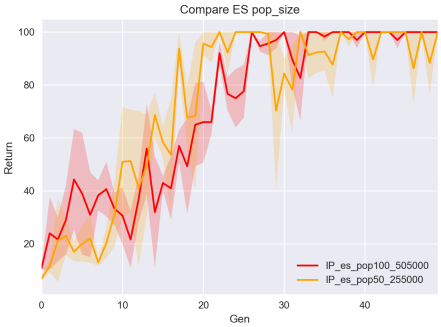


Figure 23

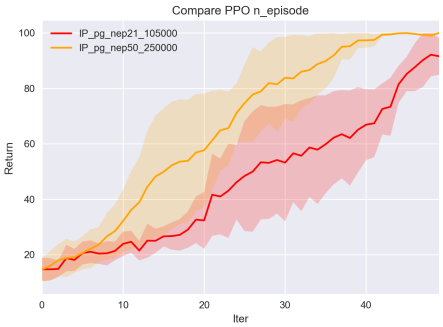


Figure 24

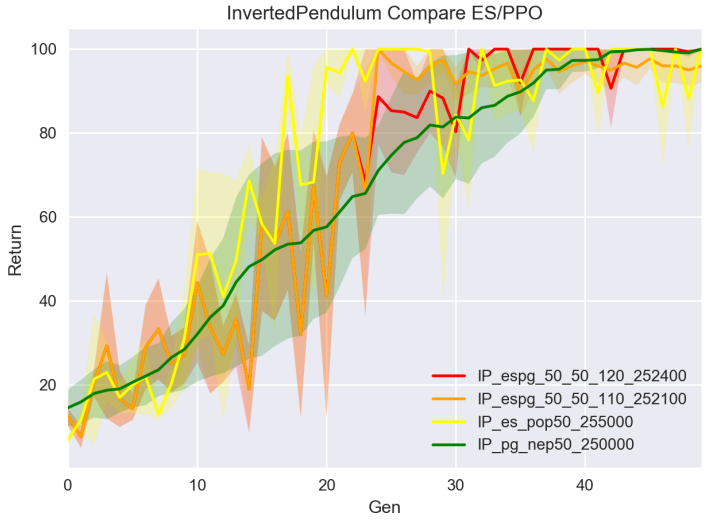


Figure 25

algorithm is generally less than that of vanilla ES.

Note: "IP_espg_50_50_120_252400" means that the learning curve plots the result of Algorithm 2, with population size 50 per ES generation and 50 episodes per PPO iteration, learning goal 120, and total time steps 252400 per experiment.

Figure 25 compares the performances of vanilla ES, PPO, and Algorithm 2 on the InvertedPendulum-v2 environment. The results were averaged over three random seeds. In our experiments, the maximum episode length was set to be 100. Since the maximum reward in this environment with episode length 1000 is 1000, we set the learning goal to be 120 and 110 in our experiments. From the results it's hard to tell whether Algorithm 2 performs better than vanilla ES, which is probably due to the fact that it takes PPO more iterations to achieve result that is as good as vanilla ES, and thus the

transition from ES to PPO is not so smooth. We also note that at the end of 50 generations, the final result of the experiment with learning goal 110 is still slightly lower than the result of vanilla ES as well as Algorithm 2 with learning goal 120. We then speculate that if the threshold is set too low and we switch to PPO too soon, the current policy and the real optimum will still be far and the improvement afterwards will be slow.

3.2.2 Experiment results on HalfCheetah environment

Below are the additional hyper-parameters we used on the HalfCheetah-v2 environment:

- α: 0.1
- σ: 0.05
- Number of hidden units: 64
- Number of generations: 150

Comparisons of different population sizes in Evolution Strategy and different number of episodes per PPO iteration are shown in figures 26 and 27.

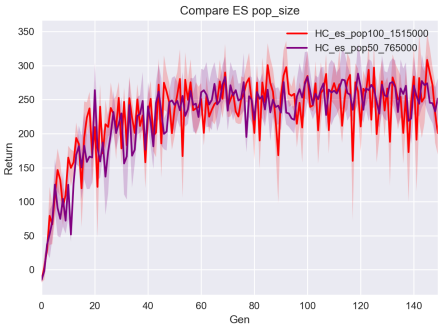


Figure 26

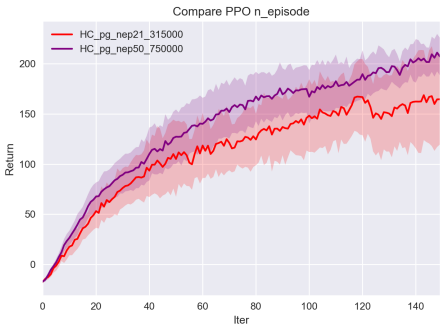


Figure 27

Figures 28 and 29 compare the performances of vanilla ES, PPO, and Algorithm 2 on the HalfCheetah-v2 environment. The results were averaged over three random seeds. In our experiments, the maximum episode length was set to be 100. Since the current benchmark reward for this environment with maximum episode length 1000 is around 3000, we conducted experiments with learning goals 375 and 350. In both graphs, learning curves for Algorithm 2 (the red, yellow, and orange ones, plotted with different pop_size/n_episode ratios) generally grow as fast as that for vanilla ES, but are less volatile. From the curves of different pop_size/n_episode ratios, we can see that larger population size tends to generate less volatile result. We also note that the total times steps of Evolution Strategy doubles that of PPO, and that integrating PPO into Evolution Strategy indeed helps to improve sample efficiency of vanilla ES.

Note: "HC_espg_50_50_375_761733" means that the learning curve plots the result of Algorithm 2, with population size 50 per ES generation and 50 episodes per PPO iteration, learning goal 375, and total time steps 761733 per experiment.
Figure 30 compares the results of vanilla ES and Algorithm 2 with learning goals 350 and 375. Learning goal doesn't seem to affect the performance of the algorithm much in this case. However, figure 30 shows that the total time steps in the experiment with learning goal 350 is less than that with learning goal 375 by 15%. We then speculate that by reasonably lowering the learning goal and thus increasing PPO iterations, we can see further drop in the amount of data used.

3.2.3 Experiment results on Hopper environment

The hyper-parameters on Hopper-v2 environment are the same as the ones on HalfCheetah-v2 environment.

Note: "HP_espg_50_50_250966" means that the learning curve plots the result of Algorithm 2, with population size 50 per ES generation and 50 episodes per PPO iteration, and total time steps 250966 per experiment.
Figure 31 compares different population sizes in Evolution Strategy and figure 32 compares different numbers of episodes per PPO iteration. The two figures show that both vanilla ES and PPO can get good results within few generations/iterations on this environment.
Figure 33 compares the performances of vanilla ES, PPO, and Algorithm 2 on the Hopper-v2 environment. The results were averaged over three random seeds. In our experiments, the maximum episode length was set to be 100. Since the results of both vanilla ES and PPO became stable around 220 after 50 generations/iterations, we ran our new algorithm for 50 generations and set the learning goal to be 265. Similar to the HalfCheetah environment, vanilla ES learns faster than PPO at the beginning of the experiments. But the stabilizing effect of PPO in Algorithm 2 is clearer on this environment. From our understanding, this improvement in stabilization is resulted from the fact that PPO is able to achieve performance as strong as that of vanilla ES within a similar number of iterations on this environment.

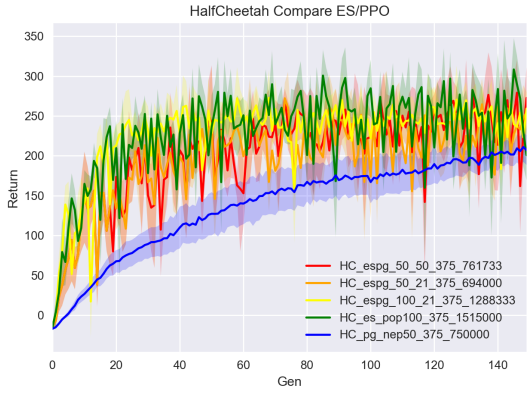


Figure 28

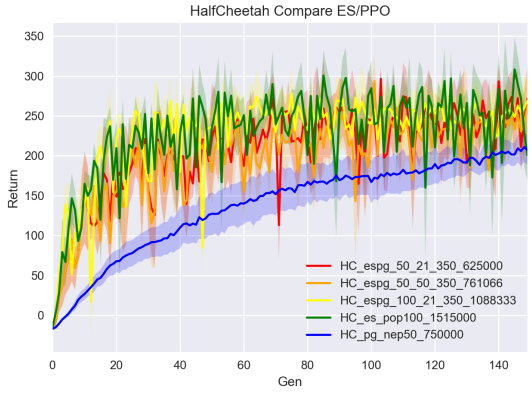


Figure 29

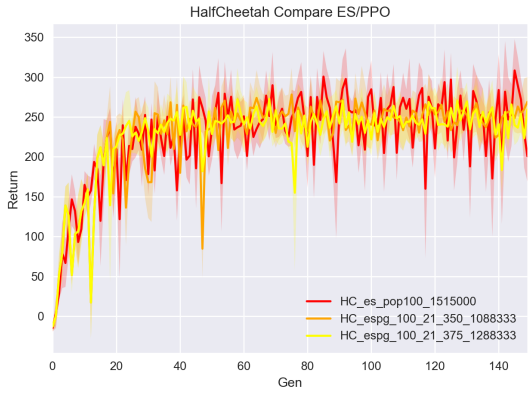


Figure 30

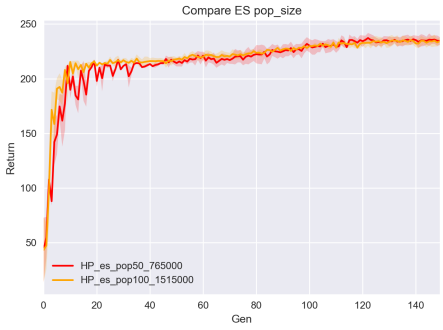


Figure 31

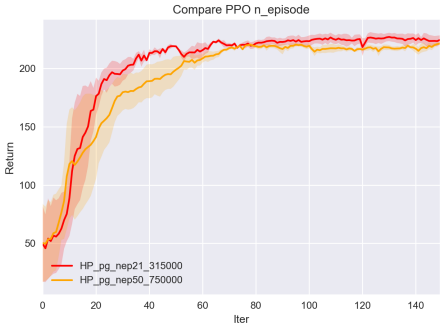


Figure 32

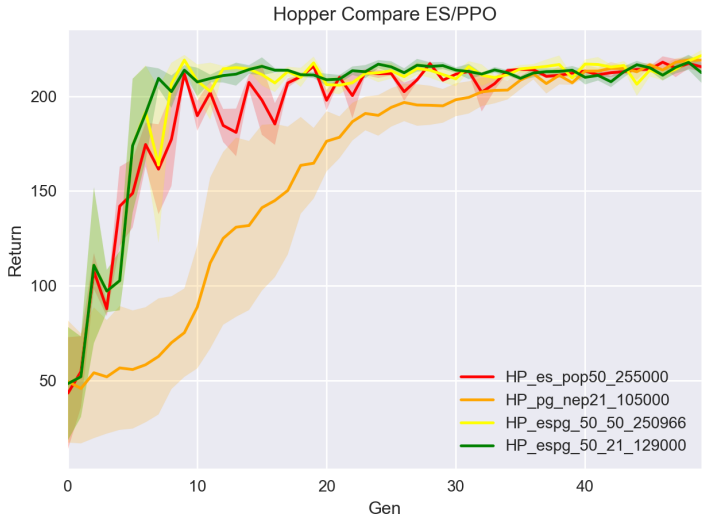


Figure 33

4 Combining Genetic Algorithm with Policy Gradient

4.1 Introduction

The PGGA(Policy Gradient with Genetic Algorithm) algorithms that we have tested and are testing can be categorized into two genres. Each genres of those has been proved helpful to some extent in some cases.

The first kind of PGGA algorithms is to run Genetic Algorithms for a suitable amount of generations and then start Policy Gradient Based Algorithms.

The second ones are to start Policy Gradient Based Algorithm and, when meet some bad situations, run Genetic Algorithms for a effective number of generations.

In the following sections, we are going to explain PGGA with its goals, designing of

algorithms, results of experiments and some basic analysis based on the results.

4.2 Goals of PGGA Algorithm

4.2.1 To avoid low rewards and gradient vanishing situations

In vanilla policy gradient, it is not rare to run into a bad situation where not only the rewards is low but also the gradients is close to zero. In this situation, it is very hard to escape and, even, needs almost forever to get back to previous progress of training. One of PGGA's goals is targeting at this situation. By running Genetic Algorithm for some generations right before running into this situation in policy gradient algorithm, we are expecting to avoid this situation. To achieve this goal we combined Vanilla Policy Gradient Algorithm with Vanilla Genetic Algorithm to show the improvements over Vanilla Policy Gradient Algorithm.

4.2.2 To enhance the stability in the early stage when using Policy Gradient Based Algorithm

Generally speaking, Genetic Algorithms are more stable than Policy Gradient based algorithm in the early stage of training, which we will show later on by empirical results. To take advantages over this, we run several generations of Genetic Algorithm in the early stage and than start the Policy Gradient based Algorithm.

4.2.3 To speed up training by searching for the best parameter choices around current position in parameter space with Genetic Algorithm

When the training progress is not ideal, running some generations of Genetic Algorithm could be an solution in terms of speeding up the training. We think this goal could be achieved sometimes, because Genetic Algorithm is efficient in terms of looking for the best parameter choice in a given region (or around one position) in parameter space. Our results indicate that.

4.3 Building Blocks of PGGA Algorithms

The purpose of this section is to introduce the specific policy gradient algorithm and genetic algorithm we were using and there performance. By doing this we expect to let readers know not only some implementation details but also show the original performance of policy gradient and genetic algorithm.

4.3.1 Policy Gradient Algorithms

The policy Gradient Algorithm we were using is modified from Li, Mufei's implementation of Policy Gradient. We are going to show the performance of it in difference parameter settings.

The following graph describes its performance without Neural Network Baseline:

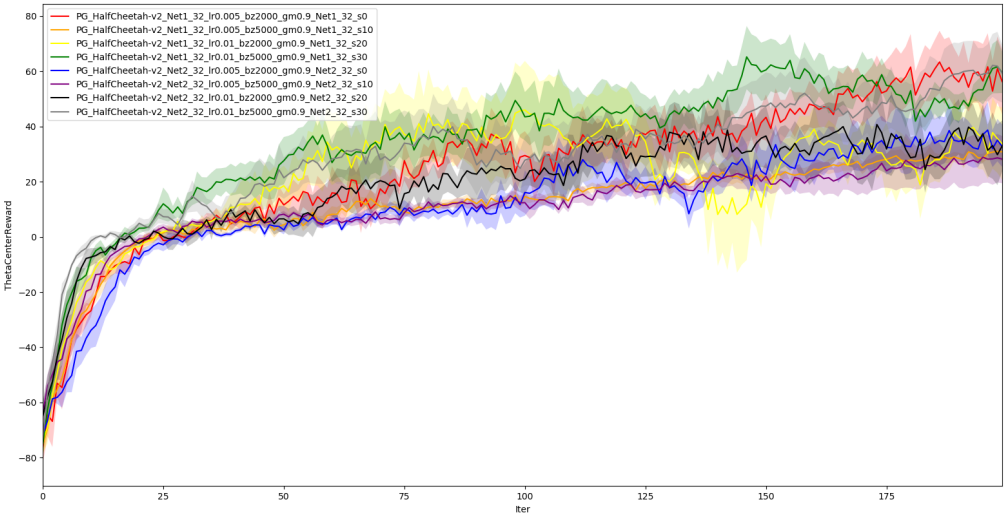


Figure 34: : HalfCheetah-v2 Environment

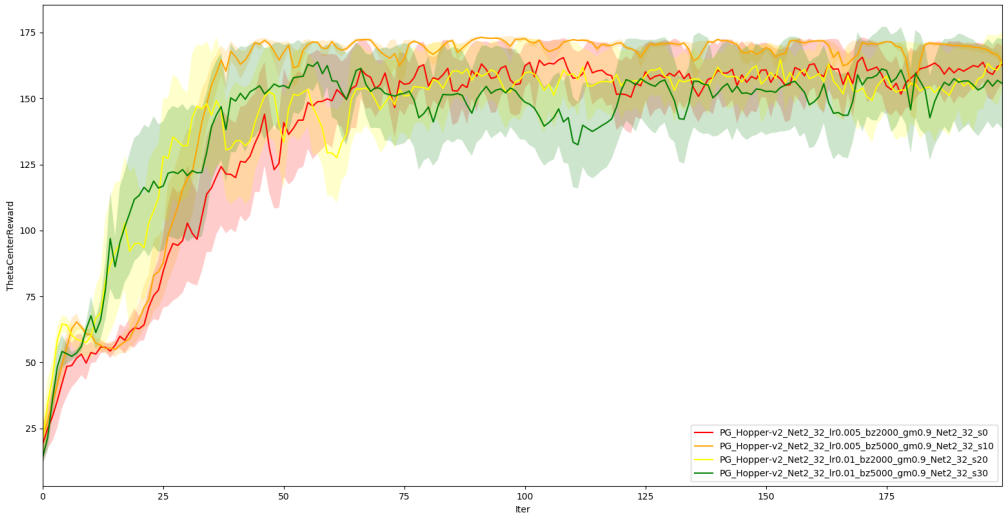


Figure 35: : Hopper-v2 environment

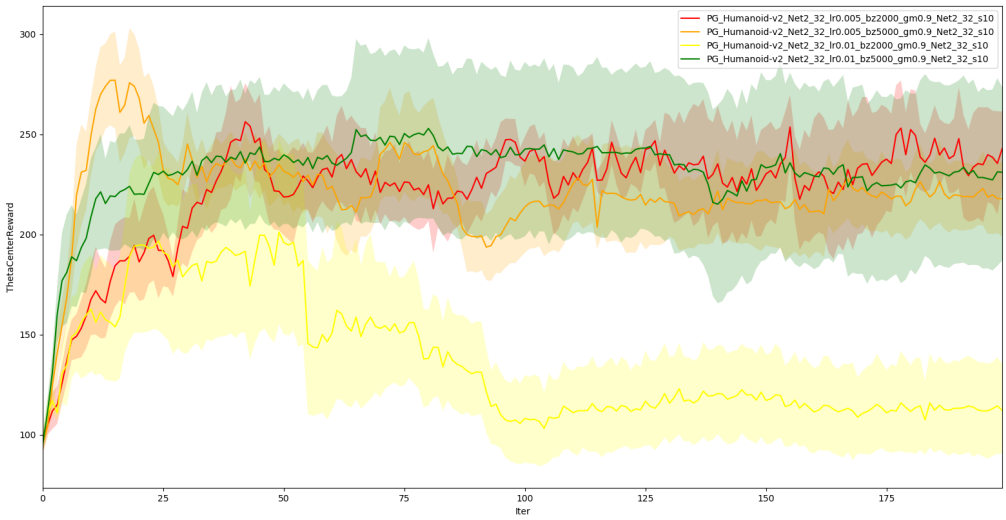


Figure 36: : Humanoid-v2 Environment

Figure 34 is the results of running experiments in HalfCheetah-v2 environment. Figure 35 is the result of running experiments in Hopper-v2 environment. Figure 36 is the results of running experiments on Humanoid-v2 environment. Those graphs are from experiments of running vanilla policy gradient without neural network baseline all in 150 time steps. Since this report is not focusing on vanilla policy gradient algorithm, we will not dive into careful analysis over those graph here.

Note: PG_Humanoid-v2_Net2_32_lr0.005_bz5000_gm0.9 means the following: Experiment of Policy Gradient algorithm in Humanoid-v2 environment with Hyperparameter choice: using a net with 2 layers and 32 units per layer, learn rate (lr) is 0.005, batch size (bz) is 5000, and discounting factor (gm) is 0.9. Those four environment are relatively complex and are all with continue action and observation space. Also, all the experiments are running with 200 training iterations.

The set of graphs below are those from running vanilla policy gradient algorithm with neural network baseline:

Figure 37: : HalfCheetah-v2 Environment

Figure 38: : Hopper-v2 environment

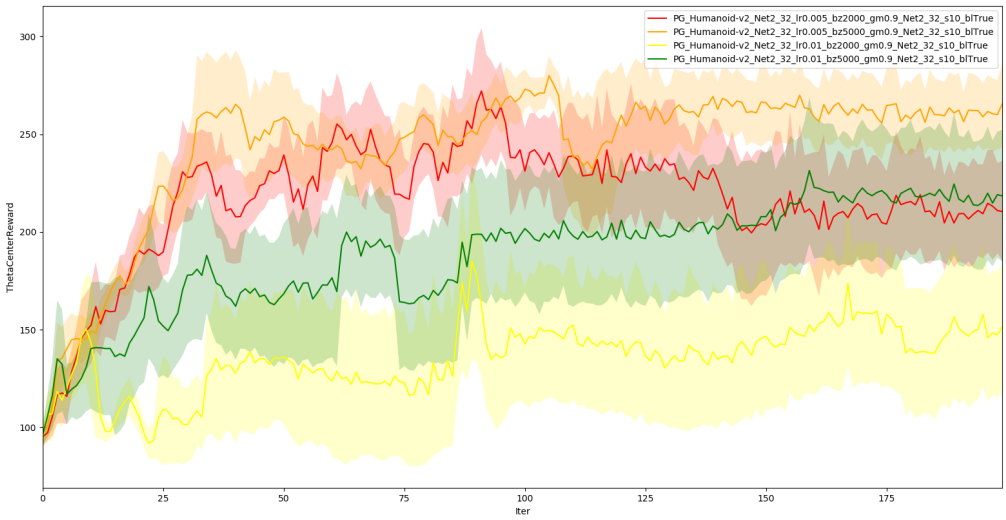


Figure 39: : Humanoid-v2 Environment

Note: PG_Humanoid-v2_Net2_32_lr0.005_bz5000_gm0.9_blTrue means the following: Experiment of Policy Gradient algorithm in Humanoid-v2 environment with Hyperparameter choice: using a net with 2 layers and 32 units per layer, learn rate (lr) is 0.005, batch size (bz) is 5000, discounting factor (gm) is 0.9, and using neural network baseline (blTrue).

4.3.2 Genetic Algorithms

The Genetic Algorithm we are using is the basic and popular one proposed by [6] without novelty search. We tried different hyper-parameter settings and the results of those are showing in the following graphs.

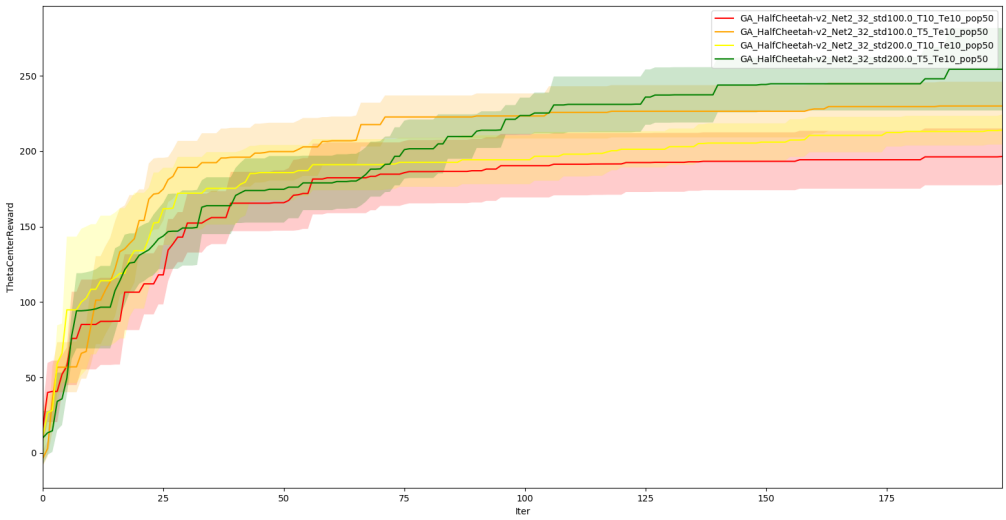


Figure 40: : HalfCheetah-v2 Environment

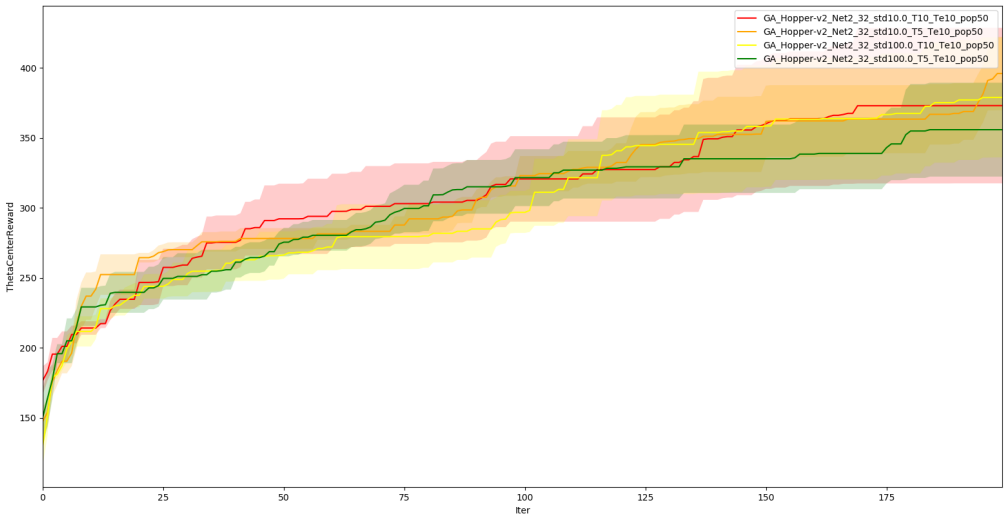


Figure 41: : Hopper-v2 environment

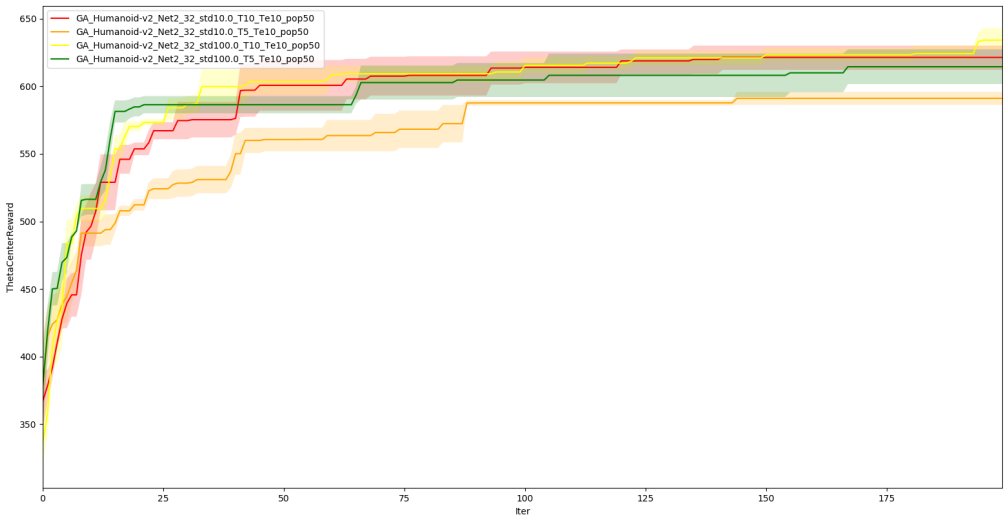


Figure 42: : Humanoid-v2 Environment

Note: GA_HalfCheetah-v2_Net2_32_std100.0_T10_Te10_pop50 represents for running experiment with Genetic Algorithm in HalfCheetah-v2 environment with hyperparameter settings: using a neural network with 2 layers of 32 nodes, standard deviation 100.0, choosing the top 10 in every generation, with an offspring population of 50. Also, all the results are based on running 150 time steps in each environments (HalfCheetah-v2, Hopper-v2, Humanoid-v2).

4.3.3 Network

In this section, we are going to show the detail information of the structure of our network, including Policy Net and Value Net (baseline prediction).
First, the Value Net we are using when running experiments over PGGA algorithms:

Figure 43: Value Net

Second, the policy network with two/one hidden layers and two heads (one of them output mean and the other output standard deviation):

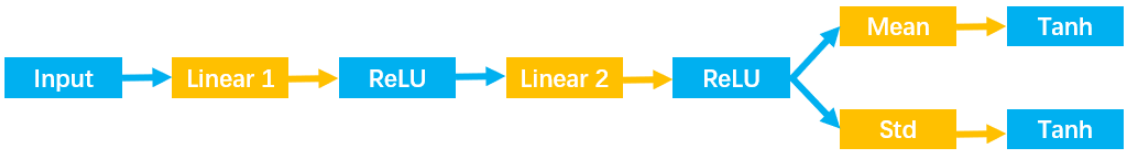


Figure 44: Neural Net with 2 Layer

Figure 45: Neural Net with 1 Layer

4.4 Stabilized Policy Gradient with Genetic Algorithm

4.4.1 Basic Designing

The basic designing of this algorithm is to run genetic algorithm for a fixed amount generations when the results of policy gradient algorithm turn out to be undesirable. Therefore, the choice of which condition will be chosen to determining the point of turning for Genetic Algorithm become important and challenging for us.

4.4.2 Pseudo code of Stabilized Policy Gradient with Genetic Algorithm

(see next page)

Algorithm 3 Stabilized Policy Gradient with Genetic Algorithm

notations:
learning rate: α
number of iteration: n
population size: pop
number of generation: g
top individual: T
sigma: σ
exponential weight: w
exponential weighted average improvement: ave_imp
improvement of objective function (averaged accumulated return): imp
expected return: J
network parameters: θ

Initialize θ_0
$ave\_imp = 0$
for Iter $i \leftarrow 1, n$ **do**
  Sample trajectories by Monte Carlo simulation
  Estimate gradients $\nabla_\theta J(\theta_i)$ and J
  do gradient update: $\theta_{i+1} = \theta_i + \alpha \nabla_\theta J(\theta_i)$
  compute the improvement of accumulated return comparing to the last iteration: $imp = J(\theta_{i+1}) - J(\theta_i)$
  if $imp \leq -|ave\_imp| * 5$ or $imp < 0$ for iteration i and $i-1$ **then**
    Initialize population $\{\theta_j\}_{j=1}^{pop}$ around θ_{i+1} and
    Compute objective function J of each individual
    $\theta_e = \arg\max_{\theta \in \{\theta_j\}_{j=1}^{pop}} \{J(\theta_j)\}$
    for $gen \leftarrow 1, g$ **do**
      Generate new population from the best T individuals from previous generation
      Estimate the objective function J of new individuals
      $\theta_e = \arg\max_{\theta \in \{\theta_j\}_{j=1}^{pop}} \{J(\theta_j)\}$
    Load θ_e back to the network
    $imp = J(\theta_e) - J(\theta_i)$
  $ave\_imp = w * ave\_imp + (1-w) * imp$

4.5 Some of the empirical results

4.5.1 Comparisons Among Different Parameter Choices

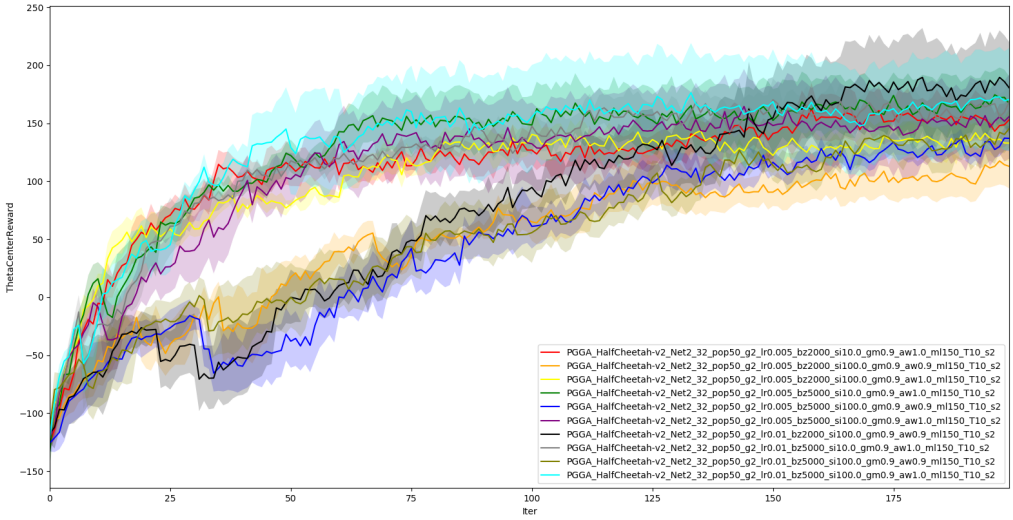


Figure 46: HalfCheetah-v2 environment

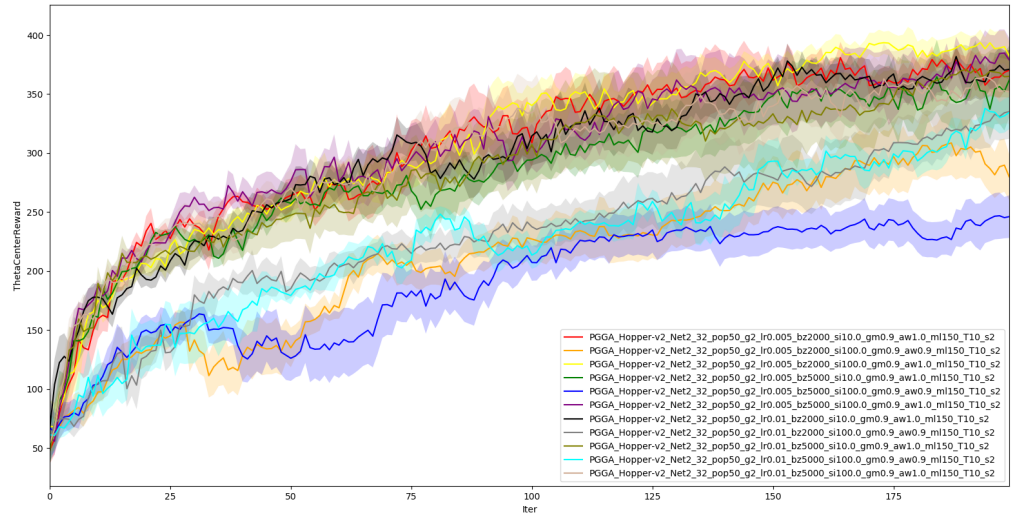


Figure 47: : Hopper-v2 environment

Figure 48: : Humanoid-v2 environment

Note: *PGGA_Humanoid-v2_Net2_32_pop50_g2_lr0.005_bz2000_si100.0_gm0.9_aw0.9_ml150_T10* stands for the following: Running experiment of PGGA algorithm in Humanoid-v2 environment: the neural net is equipped with 2 hidden layer and 32 hidden uinits for each layer (Net2_32); the offspring population size is 50 (_pop50); for each time of calling genetic algorithm we run 2 generations of it (_g2); the learning rate of policy gradient update is 0.005 (_lr0.005); the batch size is 2000 (_bz2000); the standard deviation of genetic algorithm is 100 (_si100.0); the discount factor is 0.9 (_gm0.9); the exponential weight is 0.9 (_aw0.9); the maximum trajectory length is 150 (_ml150); in genetic algorithm we the top 10 into next generation (_T10).

Also, Figure 46 is the results of running experiments in HalfCheetah-v2 environment. Figure 47 is the result of running experiments in Hopper-v2 environment. Figure 48 is the results of running experiments on Humanoid-v2 environment. Those three environments are all continue space.

Some analysis: From those graphs, it could be seen that:

- In each of those three environment and based on the data we got so far, it is shown that the choice of exponential weight is the most important hyper-parameter. The exponential weight is the weight we put on the previous exponential weighted average when calculating the a new one. The empirical results indicates that when the exponential weight is high(e.g. 1), we tends to get better rewards than some lower choices(e.g. 0.9). In other word, the more weight we put on the most recent result, it seems the better we are allowed to reach.
- Unlike Policy gradient algorithm, increasing batch size does not always make the performance better.
- Increasing the sigma (i.e. the range of exploration in genetic algorithm) usually helps but does not have very great improvement. This is very different from pure Genetic Algorithm.
- The learning rate in a suitable range seems has very little effect over the results in the settings have been tested. However, this need more careful investigations, since we have only tried only very limited amount of learning rates.
- The overall performance of this algorithm is better than Vanilla Policy Gradient Algorithm and comparable with Genetic Algorithm with the best hyper-parameter choices. Which will be shown in the following graph and analysis.

4.5.2 Comparisons With Vanilla Policy Gradient Method

We are going to show the result of running experiments over our new algorithm and vanilla policy gradient algorithm in HalfCheetah-v2, Hopper-v2, and Humanoid-v2 environments.

We are going to show the results coming from various environment with different hyper-parameter settings.

source code can be found in https://github.com/YuliangShi/Stablized-Policy-Gradient-with-Genetic-Algorithm.git

Experiments in HalfCheetah-v2:

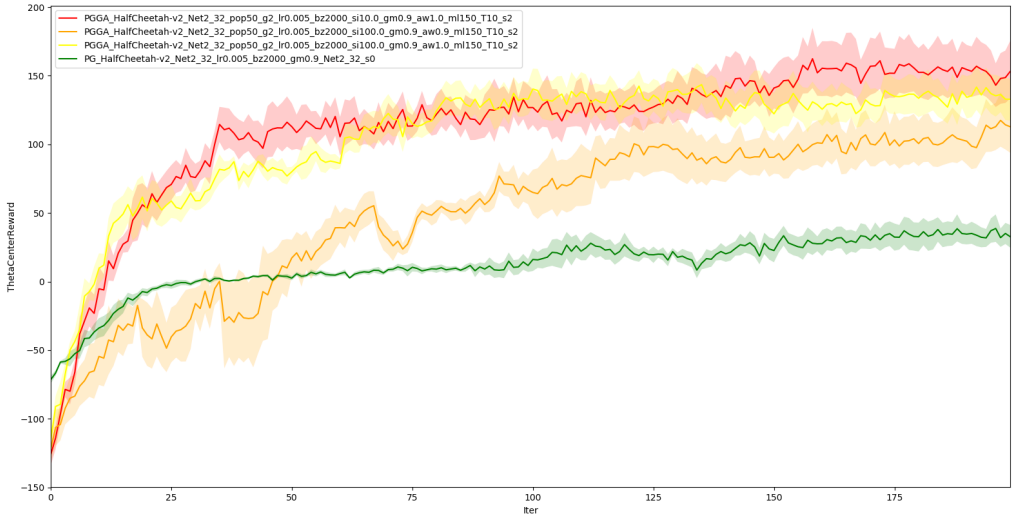


Figure 49: : HalfCheetah-v2 environment (learning rate 0.005; batch size 2000)

Figure 50: : HalfCheetah-v2 environment learning rate 0.005; batch size 5000)

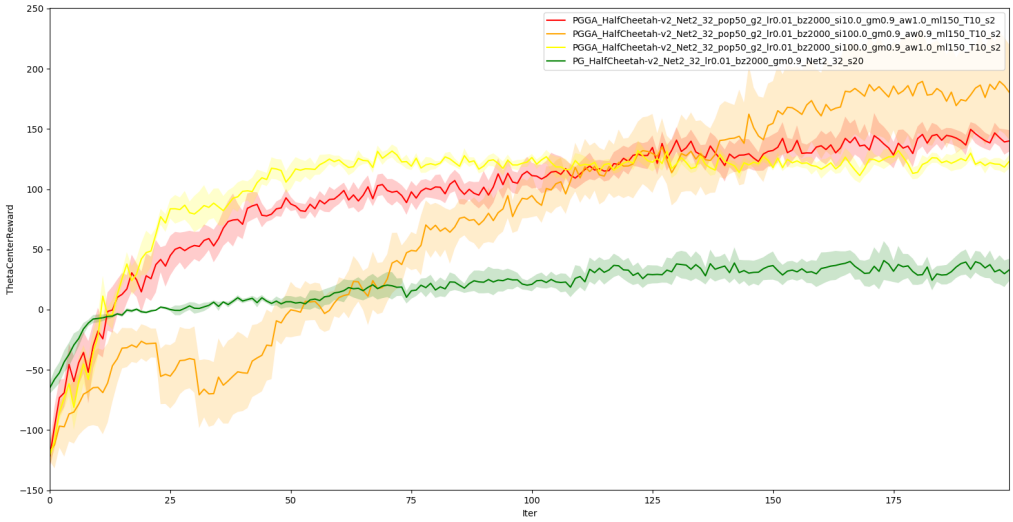


Figure 51: : HalfCheetah-v2 environment (learning rate 0.01; batch size 2000)

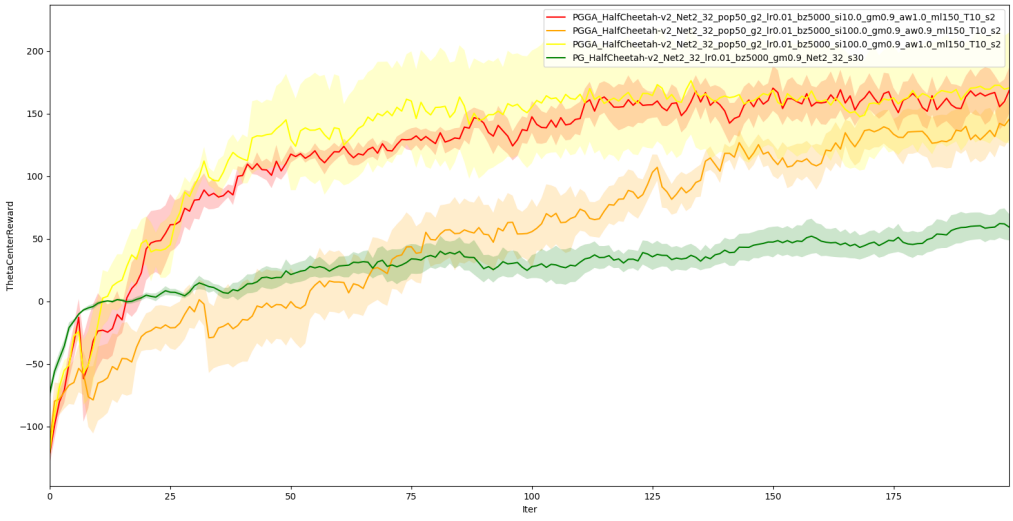


Figure 52: : HalfCheetah-v2 environment learning rate 0.01; batch size 5000)

Experiments in Hopper-v2 environment:

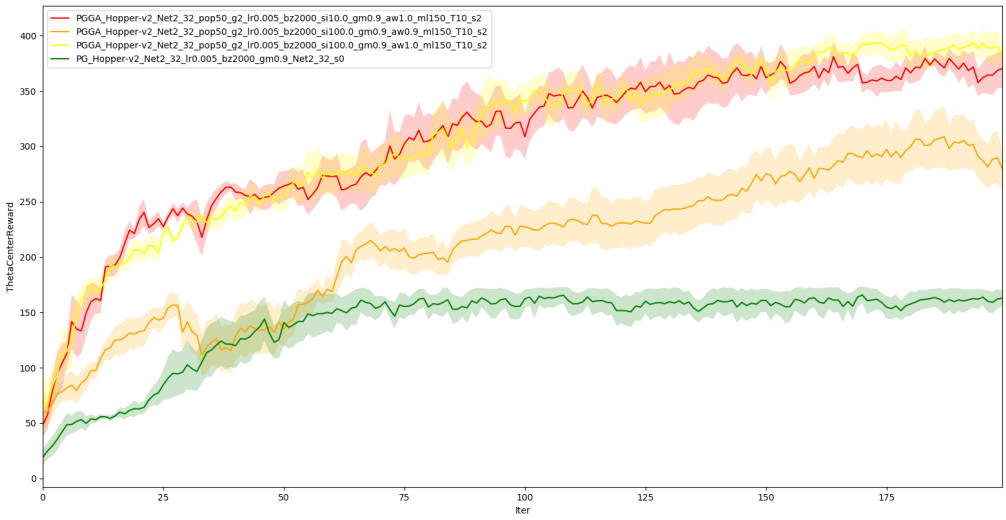


Figure 53: : Hopper-v2 environment (learning rate 0.01; batch size 2000)

Figure 54: : Hopper-v2 environment learning rate 0.01; batch size 5000)

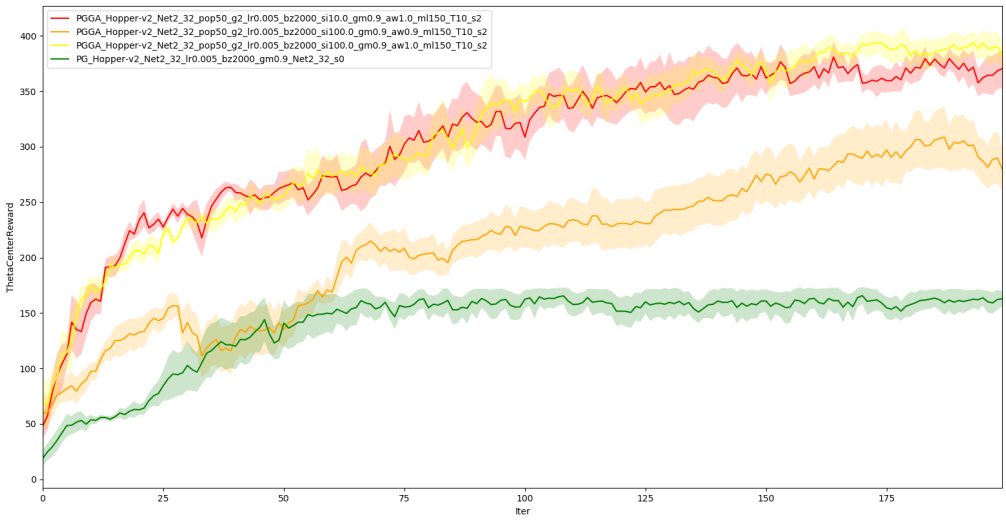


Figure 55: : Hopper-v2 environment (learning rate 0.005; batch size 2000)

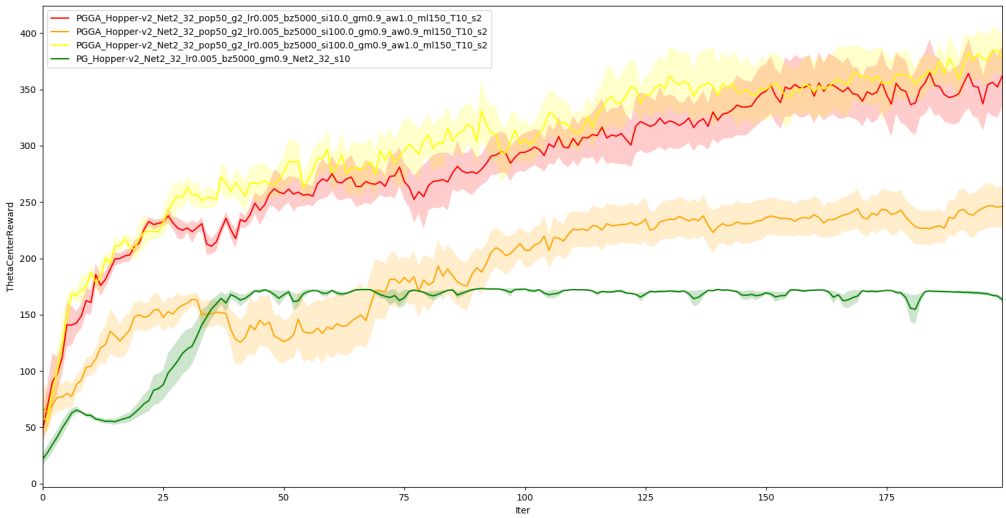


Figure 56: : Hopper-v2 environment learning rate 0.005; batch size 5000)

Experiments in Humanoid-v2 environment:

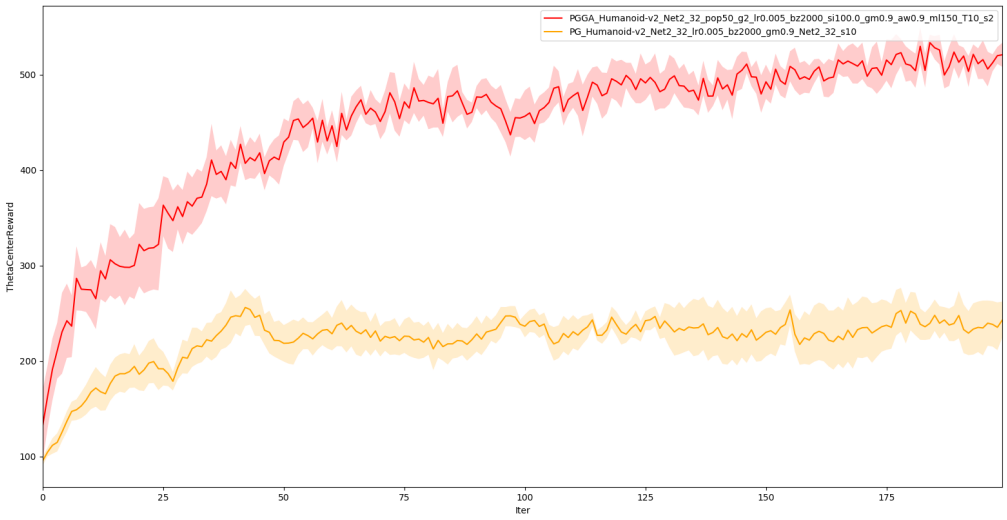


Figure 57: : Humanoid-v2 environment (learning rate 0.005; batch size 2000)

Figure 58: : Humanoid-v2 environment learning rate 0.005; batch size 5000)

Note:
PGGA_Humanoid-v2_Net2_32_pop50_g2_lr0.005_bz2000_si100.0_gm0.9_aw0.9_ml150_T10 stands for the following: Running experiment of PGGA algorithm in Humanoid-v2 environment: the neural net is equipped with 2 hidden layer and 32 hidden uinits for each layer (Net2_32); the offspring population size is 50 (_pop50); for each time of calling genetic algorithm we run 2 generations of it (_g2); the learning rate of policy gradient update is 0.005 (_lr0.005); the batch size is 2000 (_bz2000); the standard deviation of genetic algorithm is 100 (_si100.0); the discount factor is 0.9 (_gm0.9); the exponential weight is 0.9 (_aw0.9); the maximum trajectory length is 150 (_ml150); in genetic algorithm we the top 10 into next generation (_T10).

Some Analysis:

- First of all, as shown by the graphs above, in all the experiments we ran, the performances of the new algorithm in HalfCheetah-v2, Hopper-v2, and Humanoid-v2 with certain hyper-parameter choices are in general much better than the performance of vanilla policy gradient algorithm.
- It seems to us in most experiments we ran, vanilla policy gradient algorithm tends to be trapped and stop increasing at some point, while the new algorithm, combining Policy Gradient and Genetic Algorithm, could not only help the policy update getting out of that undesirable position in the parameter space, but also speed up the training progress, in terms of how much is the improvement per training iteration.
- However, the weakness of this new algorithm is that usually it takes more than twice as much as the training time than Vanilla Policy Gradient algorithm. Also, low sample efficiency is another problem occurred to us and we are still looking for solutions.

4.5.3 Comparisons With Genetic Algorithm

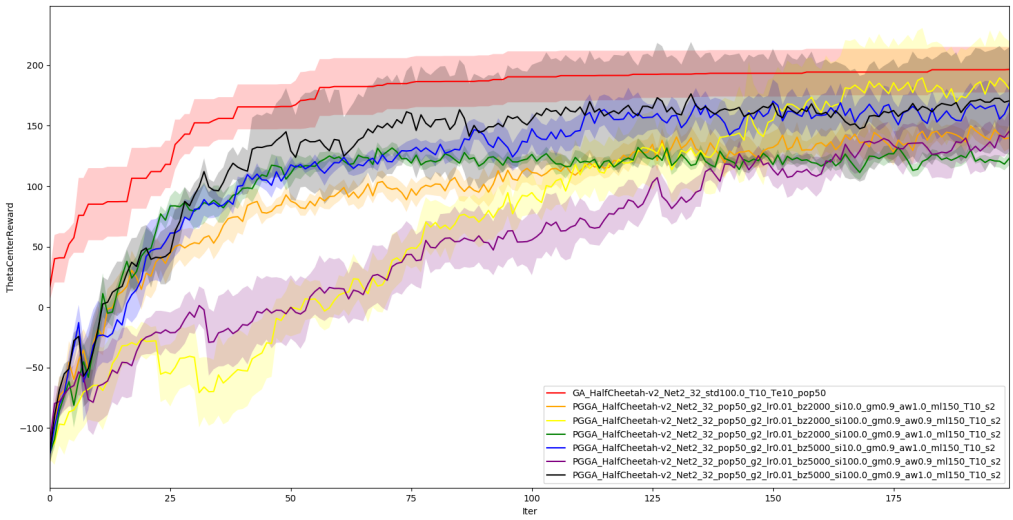


Figure 59: : HalfCheetah-v2 environment

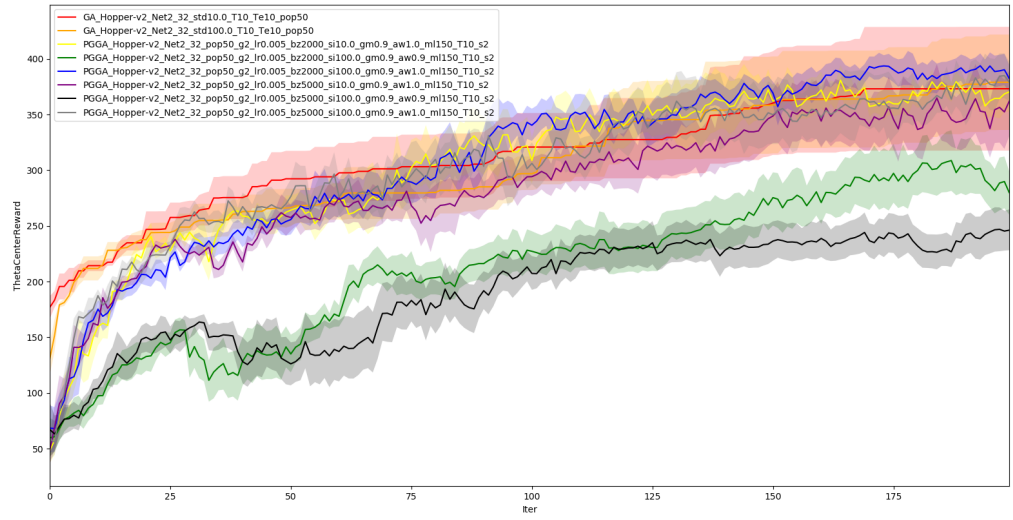


Figure 60: : Hopper-v2 environment

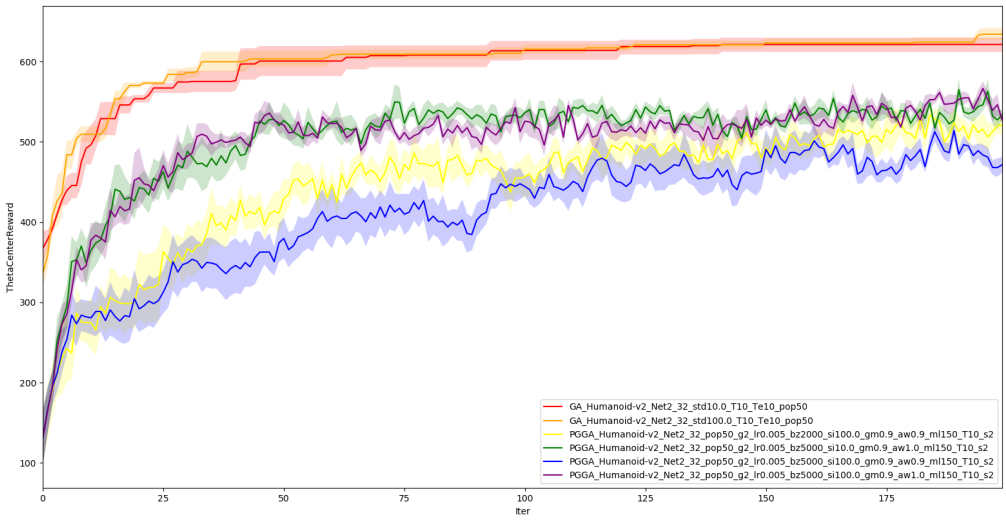


Figure 61: : Humanoid-v2 environment

Note:
PGGA/GA_Humanoid-v2_Net2_32_pop50_g2_lr0.005_bz2000_si100.0_gm0.9_aw0.9_ml150_T10 stands for the following: Running experiment of PGGA/GeneticAlgorithm(GA) algorithm in Humanoid-v2 environment: the neural net is equipped with 2 hidden layer and 32 hidden uinits for each layer (Net2_32); the offspring population size is 50 (_pop50); for each time of calling genetic algorithm we run 2 generations of it (_g2); the learning rate of policy gradient update is 0.005 (_lr0.005); the batch size is 2000 (_bz2000); the standard deviation of genetic algorithm is 100 (_si100.0); the discount factor is 0.9 (_gm0.9); the exponential weight is 0.9 (_aw0.9); the maximum trajectory length is 150 (_ml150); in genetic algorithm we the top 10 into next generation (_T10).

Also, Figure 59 is the results of running experiments in HalfCheetah-v2 environment. Figure 60 is the result of running experiments in Hopper-v2 environment. Figure 61 is the results of running experiments on Humanoid-v2 environment.

- Based on the data we collected, an important thing to notice is that the performance of the new algorithm is usually slightly lower than or similar to Genetic Algorithm. (However, in Humanoid-v2 environment, it is obvious that the results of new algorithm is less ideal than pure genetic algorithm.) We think the reason for that is the vanilla policy gradient is not strong enough and thus we are working on combining PPO with genetic algorithm. We hope this will bring good news to this new algorithm and even make PPO even more stable.

- Also, we could see that the new algorithm is more sensitive to some changes of various hyper-parameters, while genetic algorithm is more stable in that sense.
- To compare the running time of those two algorithms, the run time of the new algorithm is $\frac{1}{2}$ longer than Genetic Algorithm. The reason for that is very possibly to be the following: every time when the algorithm turns to Genetic Algorithm because of some non-ideal performance of policy gradient update, a new population need to be initialized around current position in the parameter space. This is a very time-consuming process. we think possible ways of solving this problem are, first, trying to find a more careful condition to control the switch to turn on Genetic Algorithm, and, second, trying to do not initialize the population randomly but using a set of fixed vector to initialize the population around current position in the space.
- An other issue we are concerning about is low sample efficiency problem: in the designing designing of this new algorithm we have not included strategies that allow us to store and use date collected from previous iterations. However, we think it is possible to achieve. The data collected from running Genetic Algorithm in a single generation is much more than the data collected from from one policy gradient update iteration. Those information could be more valuable.

5 Possible Future Work

- We are looking for combining evolutionary methods (Evolution Strategy $\&$ Genetic Algorithm) with more kinds of stronger gradient-based algorithms, e.g. PPO, A3C, Acer.
- More test jobs on both Mujoco and Atari environments needed. Most of the results we have are got from basic Mujoco environments.
- Evolution Strategies (ES) seem to be less powerful compared to Genetic Algorithm when combined with policy gradient methods. In fact, what ES tries to do is another way of doing gradient updates. Its advantage is in shortening wall-clock time compared to other gradient-based algorithms. Therefore, one other possible way would be combine ES with GA, trying to get short wall-clock time as well as easy-implementation ability.
- Another problem occurred in our test is that sample efficiency problem tends to become worse when combining evolutionary methods with policy gradient algorithms, which results in longer time of training (Algorithm 4 is an example, see below). This is a problem we are still looking for solutions. We think parallelizing the work would one option to increase the sample efficiency, which needs further testing.
- Novelty Search would be another option to be combined with existing algorithms. In RL, always we are doing updates on the parameter space which is different from the policy space, the space we really want to explore. Novelty Search would be a way to achieve stable step size and increasing performance.

Algorithm 4 Combine Policy Gradient with Evolution Strategies (PGES)

Hyperparameters:
learning rate: α
number of iteration: n
population size: pop
number of generation: g
top individual: T
sigma: σ
Axioms:
exponential weight: w
exponential weighted average improvement: ave_imp
improvement: imp
expected return: J
network parameters: θ

Initialize θ_0
$last\_return = 0$
for $iter \leftarrow 1, ..., n$ **do**
  Sampling trajectories
  Estimate gradients $\nabla_\theta J$
  $\theta_{i+1} = \theta_i + \nabla_\theta J$
  if $current\_return \leq last\_return * 0.9$ **then**
    Initialize $\{\epsilon_j\}_{j=1}^{pop}$
    then initialize population $\{\theta_j\}_{j=1}^{pop}$ around θ_{i+1} and estimate the expected returns of each individual $\{F_j\}_{j=1}^{pop}$
    $\theta = \theta + \alpha * \frac{1}{n\sigma} \Sigma_{j=1}^{pop} F_j \epsilon_j$
    for $gen \leftarrow 1, g$ **do**
      Generate new population from the center
      Repeat the update process for the center

References

[1] 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.

[2] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[5] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[6] 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.