Realizing an efficient parallel implementation of an evolutionary strategy algorithm

Stefano Nolfi * **Sami Sellami** **

* *CNR-ISTC, Roma, Italy, (e-mail: stefano.nolfi@istc.cnr.it).*
** *Innopolis University, Innopolis, Russia (e-mail: s.sellami@innopolis.university).*

Abstract: This present paper aims to realize an efficient parallel implementation of an evolutionary strategy algorithm, we make use of the evolutionary strategy implemented by the authors (`https://github.com/snolfi/evorobotpy`) and scale it up to many parallel workers, the algorithm is parallelized at the level of the evaluation of the individuals forming the population, and uses the shared random seed strategy which reduces the bandwidth required for communication between the workers, the strategy was tested on the double pole pendulum problem on a 4-core CPU machine and resulted in a gain in computation time of about 38%

Keywords: Evolutionary strategy, parallel processing, reinforcement learning, double pole pendulum

1. INTRODUCTION

Evolution Strategies (ES) is a class of black box optimization algorithms that are heuristic search procedures inspired by natural evolution: At every iteration ("generation"), a population of parameter vectors ("genotypes") is perturbed ("mutated") and their objective function value ("fitness") is evaluated. The highest scoring parameter vectors are then recombined to form the population for the next generation, and this procedure is iterated until the objective is fully optimized. Algorithms in this class differ in how they represent the population and how they perform mutation and recombination (Salimans et al., 2017).

While Reinforcement Learning algorithms require a reward to be given to the agent at every timestep, ES algorithms only care about the final cumulative reward that an agent obtain at the end of its rollout in an given environment

One of the main advantage of evolutionary strategies over reinforcement learning is that they are highly parallelizable , ES only requires workers to communicates few scalars between them while in RL it is necessary to synchronize entire parameter vectors (which can be millions of numbers) this is because one can control the random seed on each worker so that each worker can reconstruct the perturbation of other workers making them communicating only the result fitness of each perturbation.

2. EVOLUTION STRATEGIES

Evolution strategy can be seen as a black box optimization, the parameters which describe the policy network comes in the box and one parameter comes out, which is the total reward, the optimization consist in finding the best setting of parameters that yields to the greatest reward, Mathematically, we would say that we are optimizing a function $f(w)$ with respect to the input vector w (the parameters / weights of the network), but we make no assumptions about the structure of f, except that we can evaluate it (hence "black box"). The algorithm is a guess and check process, at each step we take a parameter vector w and generate a population of slightly different parameter vectors $w_1 ... w_n$ by modifying w with Gaussian noise. We then evaluate each one of the n candidates independently by running the corresponding policy network in the environment for a while, and add up all the rewards in each case. The updated parameter vector then becomes the weighted sum of the n vectors, where each weight is proportional to the total reward.

below is one example of a pseudo-code which describe the rollout of an agent in an OpenAI Gym environment where we only care about the cumulative reward, (Ha, 2017)

```
def rollout(agent, env):
  obs = env.reset()
  done = False
  total_reward = 0
  while not done:
    a = agent.get_action(obs)
    obs, reward, done = env.step(a)
    total_reward += reward
  return total_reward
```

We can define the rollout to be the objective function that maps the model parameters of an agent into its fitness score, and use an ES solver to find a suitable set of model parameters.

Let F be the objective function acting on parameters θ, the population is represented by a distribution over parameters $p_\psi(\theta)$ itself parameterized by ψ the task is to maximize the average objective value $E_{\theta\ p_\psi} F(\theta)$ over

the population by searching for ψ with stochastic gradient ascent (Salimans et al., 2017), the gradient step on ${}_{p}si$ is then estimated as follow

$$\bigtriangledown E_{\theta\ p_\psi} F(\theta) = E_{\theta\ p_\psi}\{F(\theta)\bigtriangledown_\psi \log p_\psi(\theta)\} \quad (1)$$

If we set $E_{\theta\ p_\psi}\{F(\theta) = E_{\epsilon\ N(0,1)}F(\theta+\sigma\epsilon)$ where $\sigma^2 I$ is the covariance of the distribution p_ψ, we can optimize over θ directly using stochastic gradient ascent with the score function estimator Salimans et al. (2017)

$$\bigtriangledown E_{\epsilon\ N(0,1)} F(\theta+\sigma\epsilon) = \frac{1}{\sigma} E_{\epsilon\ N(0,1)}\{F(\theta+\sigma\epsilon)\epsilon\} \quad (2)$$

3. SCALING AND PARALLELIZING ES

ES is parallelizable, it operates on complete episodes, thereby requiring only infrequent communication between workers, if the seed used to generate the vector of the vector of random numbers is shared between the processes before optimization, each worker knows what perturbation the other worker used thus the only parameters that need to be communicated between the workers to agree on a parameter update is the scalar return F_i. Algorithm (1) resume the procedure followed in the parallelization process

Algorithm 1 Parallelized evolution strategies

procedure
  input: noise standard deviation σ, initial policy parameters θ_0
  initialization: n workers with know random seeds
  while iteration < limit **do**
    for each worker i=1....n **do**
      construct the vector of Gaussian noise $\epsilon_i\ N(0,1)$
      compute the fitness F_i
    end for
    send all fitness values from each worker to every other worker
    for each worker i=1....n **do**
      reconstruct all perturbations ϵ_j using known seeds
      update the policy parameters θ
    end for
  end while
end procedure

4. IMPLEMENTATION

We implemented the scaling up of the evolutionary strategy in the double pole problem (Fig 1) of (`https://github.com/snolfi/evorobotpy`) which is a platform for training robots in simulation through evolutionary and reinforcement learning methods, it comprises the state-of-the-art algorithms and software tools that are used to carry on research in evolutionary robotics and in reinforcement learning robotics.

We made use of the multiprocessor library *mpi4py* to paralellize the work at the level of the population, we modified the */bin/es.py* code as to permit the multiprocessing computations, for this we added an extra argument $n_{workers}$ to specify the number of processes that can be used, to evolve a policy with multiprocessing, one can run

```
$ python3 ../bin/es.py -f Erdpole.ini
-s 11 -w 4
```

Where $-w$ specifies the number of workers. Also, the permission of the */xdpole* folder must be modified to permit all users to save the solution every g generations, for this, navigate to one level above that directory and run the command

```
$ chmod -R 777 xdpole/
```

The population evaluation is split between processes as to reduce the computation time, the *salimans.py* file is modified, the master process contain the implementation of the evolutionary algorithm and n slaves processes take care of evaluating the individuals of the population. Clearly if the population size is not a multiple of the number of workers, some workers will have more trials than the others, also, because the evaluation time may varies between the workers, the master process will have to wait for all the workers to finish before it can proceed.

The workers construct the perturbations and agree on a shared seed value which is basically the seed of the generation plus the rank of each processes, then, they return the fitness value and send it to all the other workers (see algorithm 1) using the *mpi4py* instruction *Allgatherv* (Dalcin, 2020), and finally they update the policy parameters θ using the fitness values received and the shared random seed (to construct the perturbations of all the other workers).

Since the master process have to wait for the slaves to finish to be able to proceed, we decided to include the master in the population evaluation and we verified that this approach decrease the computation time, Finally, the master saves the best solution obtained so far, which can be tested with the instruction.

```
python3 ../bin/es.py -f Erdpole.ini
-t bestgS20.npy
```

5. RESULTS

We Tested the strategy with a 4-core CPU machine on the double pole problem, which consist of a cart with two inverted pendulum attached to it (Fig 1), the task is to balance the double pendulums by only controlling the force applied to the cart, the observations and actions made of numpy array of float32 encoding the angles of the pendulums as well as the position of the cart, regarding the neural network, we used a LSTM recurrent architecture strategy with 10 hidden layers, and finally, we took the reward function as the total steps of the evaluation where the cart managed to keep the two pendulums up.

The algorithm showed a significant gain in the overall computation time, as an example with the following parameters

(1) duration of evaluation episode = 1000
(2) number of evaluation episodes = 10

(3) termination conditions occurring when one of the pendulum exceeds a certain angle
(4) total number of evaluation steps = 100 millions steps
(5) number of replications = 1

the overall benefit in time registered was about 38%, the algorithm of the full solution can be found the author's git repository (`https://github.com/samysellami/Behavioral_Robotics/tree/master/Project`)

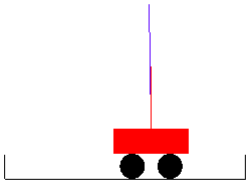

Fig. 1. Double Pole Inverted Pendulum

6. CONCLUSION

This paper presented an efficient paralellization of an evolutionary strategy algorithm, we used the shared random seed strategy to reduce the bandwidth required for communication between the workers and make the fitness value returned the only parameter needed for communication. The processes are capable of updating the policy parameters that will be used in the next generation and the master process ensure the implementation of the main evolutionary strategy, the algorithm showed a gain in time of about 38% when using 4 workers on a 4-core CPU machine.

REFERENCES

Dalcin, L. (2020). *MPI for Python*. URL `https://mpi4py.readthedocs.io/en/stable/`.

Ha, D. (2017). Evolving stable strategies. *blog.otoro.net*. URL `https://blog.otoro.net/2017/11/12/evolving-stable-strategies/`.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.