Playing Atari's Frostbite and Space Invaders with Deep Genetic Algorithms

Francesco Ballerini
Università di Bologna
francesco.ballerini3@studio.unibo.it

Abstract

We train a convolutional neural network to play two Atari 2600 games from the Arcade Learning Environment, *Space Invaders* and *Frostbite*. The model, as in DQN, receives raw pixels in input and outputs action values. However, differently from DQN and other popular approaches, it is trained with a gradient-free, population-based method belonging to the family of genetic algorithms, which allows us to achieve interesting results after training the network on a single desktop and with CPU only.

1 Introduction

Four broad families of deep learning algorithms have shown promise on reinforcement learning (RL) problems so far: Q-learning methods (Mnih et al., 2015), policy gradient methods (Mnih et al., 2016), evolution strategies (Salimans et al., 2017), and genetic algorithms (GAs) (Such et al., 2018). The first three approaches can be considered gradient-based methods, as they all calculate or approximate gradients in a deep neural network (NN) and optimize its parameters via stochastic gradient descent/ascent. GAs, on the other hand, update the paramenters of an NN through a population-based approach, and despite their simplicity have been shown to achieve competitive results on deep RL benchmarks (Such et al., 2018).

Intrigued by the idea of a truly gradient-free method in the context of deep RL, we implemented a GA and tested it on two Atari games from the Arcade Learning Environment (ALE) (Machado et al., 2018): *Space Invaders* and *Frostbite*.

2 Algorithm

The core algorithmic component of our work is heavily inspired by Such et al. (2018).

Training procedure A GA *evolves* a population of N individuals, which, in our case, are NN parameter vectors $\boldsymbol{\vartheta}_i, i = 1 \dots N$. At every *generation*, an individual $\boldsymbol{\vartheta}_i$ is produced by selecting a *parent* $\boldsymbol{p}_i$ uniformly with replacement from the population of the previous generation and *mutating* it by adding Gaussian noise:

$$\boldsymbol{\vartheta}_i = \boldsymbol{p}_i + \sigma\boldsymbol{\epsilon} \tag{1}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$ and $\sigma \in \mathbb{R}$. Once an individual has been produced, it is evaluated by computing a *fitness score* s_i. The $(N+1)^{\text{th}}$ individual of the next generation is an unmodified copy of the highest-scoring $\boldsymbol{\vartheta}_i$, which, together with the other top $T-1$ individuals, becomes a parent for the next generation. s_i is obtained by (i) setting the NN parameters to $\boldsymbol{\vartheta}_i$, (ii) playing a full episode of the Atari game, and (iii) summing up the rewards received during the episode; in other words, s_i is the final game score. The output of the algorithm is the parameter vector with the highest score in the last generation. VANILLA-GA in Appendix A provides a pseudocode for the procedure.

Compressed representation Storing each individual as a parameter vector would scale poorly in memory as populations get larger and networks get deeper and/or wider. Therefore, some form of compression is needed. Let $\boldsymbol{\vartheta}_i^g$ and $\boldsymbol{\epsilon}_g, g = 1 \dots G$ be the i^{th} parameter vector and Gaussian noise at generation g, respectively, and let $\boldsymbol{\vartheta}_i^{g-j}, j = 1 \dots g$ be the j^{th}-degree *ancestor* of $\boldsymbol{\vartheta}_i^g$. Equation 1 can be then rewritten as

$$\begin{aligned}
\boldsymbol{\vartheta}_i^g &= \boldsymbol{\vartheta}_i^{g-1} + \sigma\boldsymbol{\epsilon}_g \\
&= \left(\boldsymbol{\vartheta}_i^{g-2} + \sigma\boldsymbol{\epsilon}_{g-1}\right) + \sigma\boldsymbol{\epsilon}_g \\
&= \left(\left(\boldsymbol{\vartheta}_i^{g-3} + \sigma\boldsymbol{\epsilon}_{g-2}\right) + \sigma\boldsymbol{\epsilon}_{g-1}\right) + \sigma\boldsymbol{\epsilon}_g \\
&\vdots \\
&= \boldsymbol{\vartheta}_i^0 + \sigma\sum_{j=1}^{g}\boldsymbol{\epsilon}_j
\end{aligned} \tag{2}$$

where $\boldsymbol{\vartheta}_i^0$ belongs to the population at initialization time (before the first generation); details on initialization can be found in Appendix B. Since both initialization and noise generation are stochastic, they can be expressed as deterministic functions of a random seed $\tau_j \in \mathbb{N}$. Let $\mathcal{I}$ and $\mathcal{E}$ denote the initialization and noise generation function, respectively. Equation 2 can be now be rewritten as

$$\boldsymbol{\vartheta}_i^g = \mathcal{I}(\tau_0) + \sigma\sum_{j=1}^{g}\mathcal{E}(\tau_j) \tag{3}$$

that is, $\boldsymbol{\vartheta}_i^g$ is uniquely determined by the sequence of seeds $\tau_0 \dots \tau_g$. Therefore, instead of storing parameter vectors, we can store sequences of seeds: being the number of generations tipically way smaller than the number of parameters (orders of tens vs order of millions), this allows for a much more memory-friendly representation.

3 Experiments

3.1 Setup

Preprocessing Following the preprocessing of Mnih et al. (2015), each $3 \times 210 \times 160$ Atari frame is (i) converted to grayscale, (ii) bilinearly downsampled to 84×84, (iii) appended to the three previous frames, and (iv) fed in input to the model as a $4 \times 84 \times 84$ tensor.

Architecture Let $C_{\text{out}} \times C_{\text{in}} \times H \times W$ denote a convolutional layer consisting of C_{out} filters with C_{in} channels, height H, and width W. The convolutional neural network (CNN) we experimented with is the DQN from Mnih et al. (2015), namely: (i) a $32 \times 84 \times 8 \times 8$ layer with stride 4, (ii) a $64 \times 32 \times 4 \times 4$ layer with stride 2, (iii) a $64 \times 64 \times 3 \times 3$ layer with stride 1, (iv) a linear layer with output dimension 512, and (v) a linear layer with output dimension 18 (i.e. the number of Atari actions). The output of each layer (except the last one) goes through a ReLu function.

GPU acceleration Our implementation can execute some sections of code on GPU, mainly the forward pass of the CNN. However, after some preliminary, smaller scale experiments, we noticed that GPU-enabled runs were slightly slower than their CPU-only counterparts, possibly because the overhead of moving model and data to the GPU is higher than the actual gain obtained by running computations on the GPU itself. As a result, all our models were trained on CPU only. It is worth noting that our GA would probably benefit more from a distributed implementation across multiple CPUs than it does from a single GPU, since, instead of running forward + backward passes on one NN, it runs forward passes only on *many* NNs, one for each individual in the population. (See Appendix C for additional hardware information.)

3.2 Vanilla GA

First, we tested our vanilla implementation described in Section 2 on Space Invaders and Frostbite. For each game (i) the algorithm ran for 16 generations, (ii) its output parameters were loaded into a fresh CNN, and (iii) the resulting model was evaluated by letting it play 30 episodes and averaging the final scores[1]. Results are shown in Table 1.

Although some learning certainly occurred (as the comparison with the random policy shows), we noticed that during training the algorithm was reaching a peak score around generation 10 and then stopped improving. What was even more suspicious is that the score registered during training was much higher that the one obtained at evaluation time, as shown in Table 2.

What we realized was happening is that, at a certain generation, one individual of the current population played a very "lucky" episode, where it got an unusually high score (due to the stochastic nature of the environment), therefore becaming the top individual and surviving unmodified until the next generation (as described in Section 2). Once part of the next generation, its score was not matched by any new individual, and it therefore survived again and again, until the last generation. When tested at evaluation time, however, the environment reacted differently to its actions and its score got lower.

3.3 Elite selection

In oder to make the selection of the top individual more robut to the random fluctuations of the environment, we implemented what we call *elite selection*: right before the end of each generation, the top 10 individuals play 30 additional episodes each, and the one with the highest average score gets to be the survivor, i.e. the $(N+1)^{\text{th}}$ individual of the next generation (or the output of the algorithm, if the current generation is the last one). ELITE-GA in Appendix A provides a pseudocode for the procedure.

To be fair, Such et al. (2018) make no distinction between VANILLA-GA and ELITE-GA, as elite selection is already included in their base implementation. However, we initially decided to remove it, in order to test the capabilities of an even simpler algorithm than Such et al.'s, and then added it back to try to solve the issue decribed in Section 3.2. And indeed, as Table 1 shows, elite selection improves the score in both games, especially Frostbite.

3.4 Epsilon-greedy

Although the performance improves sligthly on Space Invaders by adding elite selection, we noticed, by looking at *how* the model was playing the game, that it was adopting quite a "lazy" strategy: instead of trying to avoid the aliens' lasers, the agent does not move from its starting position and keeps shooting until it catches the "mistery ship", which at one point crosses the top of the screen and gives bonus points if destroyed. Once the ship is destroyed, then the agent starts to move to the left/right and defend itself against the enemy.

Up to this point, experiments were performed by selecting actions greedily. Thus, in an attempt to favor the exploration of alternative strategies, we implemented an ε-greedy policy; specifically, three sets of values for ε were tested:

$$\varepsilon = 0.5, 0.4, 0.3, 0.2, 0.1, 0, \ldots, 0 \qquad \text{for } g = 1, 2, 3, 4, 5, 6, \ldots, G \tag{4}$$

$$\varepsilon = 0.1, 0.01, 0.001, 0, \ldots, 0 \qquad \text{for } g = 1, 2, 3, 4, \ldots, G \tag{5}$$

$$\varepsilon = 0.01, 0, \ldots, 0 \qquad \text{for } g = 1, 2, \ldots, G \tag{6}$$

The ε-values of Equation 5 proved to be the most effective when applied to ELITE-GA, though the average score at evaluation time got significantly worse than its greedy counterpart, as shown in Table 1 (the ε-greedy policy was tested on Space Invaders only).

It is worth noting that in Such et al. (2018) there is no mention of how actions are selected once the CNN output has been computed, which seems to suggest that a greedy policy was adopted. After all, at least intuitively, exploration should be guaranteed by the extent of the population itself, i.e. by the fact that not just one but N models take part in the training process.

[1]The version of ALE we experimented with (v5) implements *sticky actions*, i.e. instead of always simulating the action passed to the environment, there is a small probability (0.25) that the previously executed action is used instead, effectively making Atari games stochastic (Machado et al., 2018).

	Average score	
	Space Invaders	Frostbite
Random policy	176.0	68.7
VANILLA-GA	683.0	464.7
ELITE-GA	700.2	2755.7
ELITE-GA + ε-greedy	420.2	–

Table 1: **Performance of GAs on the two Atari games Space Invaders and Frostbite.** Scores are averaged over 30 episodes. Models were trained by running the corresponding algorithm for 16 generations ($\sim$10 hours each on our machine). Random policy results are reported as a baseline.

	Space Invaders	Frostbite
Best training score	1300	2980
Generation	12	10

Table 2: **Highest scores reached by VANILLA-GA during training and generations at which they were first recorded.** By "training score at generation g" we mean the highest sum of episode rewards achieved by an individual of that generation. Notice how much higher these scores are compared to those obtained at evaluation time by the same algorithm (Table 1).

4 Conclusions

We applied GAs to train a CNN to play Atari's Space Invaders and Frostbite from raw input pixels. The results obtained by the model trained on Frostbite are quite impressive, especially considering the simplicity of the algorithm and the fact that it does not rely on gradient-based optimization. Space Invaders, on the other hand, might benefit from techniques that enable stronger exploration within the framework of GAs; we experimented with ε-greedy policies, which however did not appear to be an effective solution. Overall, GAs allowed us to obtain relatively satisfying results in a deep RL task with limited computational resources, which is certainly the most attractive feature of this methodology.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *ICML'16*, page 1928–1937, 2016.

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

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, 2018, arXiv:1712.06567v3.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *JAIR*, 61(1):523–562, Jan 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *JMLR*, 9:249–256, Jan 2010.

A Pseudocodes

In the following, G denotes the number of generations, N the population size, and T the truncation size. Arrays are indexed starting from 1.

```
VANILLA-GA($G, N, T$)
 1  for $g = 1$ to $G$
 2      if $g == 1$
 3          $models = [\text{INIT}()$ for $i = 1$ to $N]$
 4          $scores = [\,]$
 5      else
 6          $parents = models[1 \ldots T]$
 7          $models = parents[1]$
 8          $scores = scores[1]$
 9      for $i = 1$ to $N$
10          if $g == 1$
11              $s = \text{PLAY-EPISODE}(models[i])$
12          else
13              $p = \text{RAND-CHOICE}(parents)$
14              $m = \text{MUTATE}(p)$
15              $\text{APPEND}(models, m)$
16              $s = \text{PLAY-EPISODE}(m)$
17          $\text{APPEND}(scores, s)$
18      sort $models$ with descending order by $scores$
19  return $models[1]$
```

```
ELITE-GA($G, N, T$)
1  for $g = 1$ to $G$
2      execute lines 2–18 of VANILLA-GA
3      $candidates = models[1 \ldots 10]$
4      $elite = \text{argmax}_{c \in candidates} \frac{1}{30} \sum_{j=1}^{30} \text{PLAY-EPISODE}(c)$
5      $\text{SWAP}(models[1], elite)$
6  return $models[1]$
```

B Initialization and hyperparameters

Following Such et al. (2018), we (i) apply Xavier initialization (Glorot and Bengio, 2010) with normal distribution for weights and initialize biases to zero (line 3 of VANILLA-GA), (ii) set $N = 1000$, $\sigma = 0.002$, $T = 20$, and (iii) use integer random seeds in the range $[0, 2^{28})$.

C Hardware

All our experiments were run on a desktop with CPU AMD Ryzen 5 3600 (6 cores), GPU NVIDIA GeForce GTX 1660 SUPER (6 GB), and 16 GB RAM.

Although most of our work is based on Such et al. (2018), their computational budget is far from being easy to replicate: their "single machine" implementation consists of 4 GPUs and 48 CPU cores, whereas their distributed version runs on 720 CPU cores across "dozens of machines".