The Knowledge Engineering Review (2022), **37**, e9, pp. 1–46
doi:10.1017/S0269888922000042

RESEARCH ARTICLE

A scalable species-based genetic algorithm for reinforcement learning problems

Anirudh Seth[1,2], Alexandros Nikou[2] and Marios Daoutis[2]

[1]KTH, Brinellvägen 8, Stockholm 114 28, Sweden;
E-mail: aniset@kth.se
[2]Ericsson, Torshamnsgatan 23, Stockholm 164 83, Sweden;
E-mails: alexandros.nikou@ericsson.com, marios.daoutis@ericsson.com

Received: 26 August 2021; **Revised:** 17 June 2022; **Accepted:** 28 June 2022

Abstract

Reinforcement Learning (RL) methods often rely on gradient estimates to learn an optimal policy for control problems. These expensive computations result in long training times, a poor rate of convergence, and sample inefficiency when applied to real-world problems with a large state and action space. Evolutionary Computation (EC)-based techniques offer a gradient-free apparatus to train a deep neural network for RL problems. In this work, we leverage the benefits of EC and propose a novel variant of genetic algorithm called SP-GA which utilizes a species-inspired weight initialization strategy and trains a population of deep neural networks, each estimating the Q-function for the RL problem. Efficient encoding of a neural network that utilizes less memory is also proposed which provides an intuitive mechanism to apply Gaussian mutations and single-point crossover. The results on Atari 2600 games outline comparable performance with gradient-based algorithms like Deep Q-Network (DQN), Asynchronous Advantage Actor Critic (A3C), and gradient-free algorithms like Evolution Strategy (ES) and simple Genetic Algorithm (GA) while requiring far fewer hyperparameters to train. The algorithm also improved certain Key Performance Indicators (KPIs) when applied to a Remote Electrical Tilt (RET) optimization task in the telecommunication domain.

1. Introduction

Gardner Murphy, an influential social and personality psychologist, defined learning as something that covers every modification in the behavior to meet the requirements of the environment (Proshansky & Murphy, 1942). Humans and animals rely heavily on the process of trial-and-error to acquire new skills. Experiences like learning how to walk, riding a bicycle, teaching a dog new tricks are all motivated by performing actions that garner positive rewards (e.g., moving forward, not falling, happiness, and getting treats). Such interactions become a significant source of knowledge about new things and uncertain environments. Reinforcement Learning (RL) is a paradigm of Machine Learning (ML) that teaches an agent, a learner, to make sequential decisions in a potentially complex, uncertain environment with the fundamental goal to maximize its overall reward.

Exponential growth in computational power, custom-developed Application Specific Integrated Circuit (ASIC) (Barr, 2007), Tensor Processing Units (TPUs)[1] combined with the availability of enormous training datasets has accelerated training workloads and led to an unprecedented surge of interest in the topic of Deep Reinforcement Learning (DRL), especially in domains like telecommunications

[1] https://cloud.google.com/tpu.

Cite this article: A. Seth, A. Nikou and M. Daoutis. A scalable species-based genetic algorithm for reinforcement learning problems. *The Knowledge Engineering Review* 37(e9): 1–46. https://doi.org/10.1017/S0269888922000042

(Nikou *et al.*, to appear; Luong *et al.*, 2019; Yajnanarayana *et al.*, 2020), networking (Kavalerov *et al.*, 2017; Luong *et al.*, 2019), game-playing agents (Bellemare *et al.*, 2013; Mnih *et al.*, 2015; Silver *et al.*, 2016; Brockman *et al.*, 2016), robotic manipulations (Abbeel *et al.*, 2006; Kalashnikov *et al.*, 2018), and recommender systems (Zheng *et al.*, 2018; Munemasa *et al.*, 2018).

1.1. Problem

Deep Q-Network (DQN) (Mnih *et al.*, 2015) uses Neural Networks (NNs) as function approximators to estimate the Q-function, it does so by making enhancements to the original Q-learning algorithm by employing an experience replay buffer and a separate target network that is updated less frequently. Actor-critic algorithms like Asynchronous Advantage Actor Critic (A3C) utilize Monte Carlo samples to estimate the gradients and then learn the parameters of the policy using gradient descent. Revised variants of these algorithms have been published over the past few years, which have improved performance on several complex problems and industry standard benchmarks (Lillicrap *et al.*, 2015; van Hasselt *et al.*, 2016; Wang *et al.*, 2016; Silver *et al.*, 2017). However, all of these approaches entail expensive computations, such as calculating gradients and backpropagation, to train neural networks. Training such models take several hours, even multiple days (Mnih *et al.*, 2016; Mania *et al.*, 2018), to obtain satisfactory results, mainly when applied to problems that have a very large state space. Some other drawbacks of these algorithms include the sensitivity to the different hyperparameters, poor sample efficiency (Yu, 2018), and limited parallelizability.

Recent publications, notably from OpenAI (Salimans *et al.*, 2017) and UberAI (Conti *et al.*, 2018), exhibited the use of Evolutionary Computation (EC) to train deep neural networks for RL problems. The combined use of EC and RL offers several advantages. These gradient-free methods are more robust to the issues of dense and sparse rewards and are also more tolerant to longer time horizons. The inherent design of such algorithms makes them well suited for distributed training which has shown significant speedup of overall training time (Salimans *et al.*, 2017; Conti *et al.*, 2018).

1.2. Research question

The research questions we are looking to address in the current work are:

1. How effective are gradient-free evolutionary-driven methods, such as variants of Genetic Algorithms (GAs), in training deep neural networks for RL problems with a large state space? How do they compare to gradient-based algorithms in terms of performance and training time?
2. To what extent can evolution-based RL algorithms be scaled? Are the gains worth the extra number of workers?
3. How effective are evolution-based methods when applied to RL problems in different domains like the optimization of Remote Electrical Tilt (RET) in telecommunications?

1.3. Contribution

In this work, a novel species-based GA that trains a deep neural network for RL specific problems is proposed. The algorithm which is *model-free*, *gradient-free* only utilizes simple genetic operators like mutation, selection, recombination, and crossover for training. A memory efficient encoding of the neural network is presented which simplifies the application of genetic operators and also reduces bandwidth requirement making this approach highly scalable.

1.4. Purpose

The key benefits of using EC-based techniques for training neural networks are the simplicity of implementation, a faster rate of convergence due to distributed training, and the reduced number of hyperparameters. Population-based methods, if implemented efficiently, are massively scalable due to

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

their inherent design and can achieve a significant parallel speedup. The reduced training time makes them well suited for RL problems that require frequent retraining and for those that require multiple policies. Since the method is gradient-free, it does not suffer from known issues of exploding, vanishing gradients and is particularly well suited for non-differentiable domains (Cho *et al*., 2014; Xu *et al*., 2016; Liu *et al*., 2018). Some additional advantages of these black-box optimization techniques include: the improved exploration (Conti *et al*., 2018) due to their stochastic nature and the invariance to the scale of rewards (sparse/dense) which is something that has to be addressed explicitly in current state-of-the-art methods.

Our work can also be combined with existing Markov Decision Process (MDP)-based methods to develop a hybrid approach. The use of EC-based training for neural networks can also be extended to supervised learning tasks. Lastly, neuroevolution and its application to RL is thought to be one step toward artificial general intelligence, which is a topic of significant interest to researchers all over the world.

1.5. Ethics and sustainability

A recent research study (Strubell *et al*., 2019) highlighted that training a deep learning model can generate over 600 000 pounds of CO_2 emissions. In comparison, an average Swedish person generates around 22 000 pounds[2] of CO_2 emissions(as of 2020) per year. These numbers, when compared, are pretty concerning. Deep learning models for RL rely on specialized hardware like Graphics Processing Units (GPUs) and TPUs which are not only expensive but also consume significant amount of power (Jouppi *et al*., 2017). The long training times (Mnih *et al*., 2015; Mnih *et al*., 2016; van Hasselt *et al*., 2016) makes the matter much worse. GPUs require an efficient cooling system and maintenance, or else it can thermal throttle, which adversely effects its efficiency and life.

The algorithm proposed in this work does not compute gradients and does not perform backpropagation, both of which are computationally expensive tasks. Lack of gradients means that low precision older hardware and only Central Processing Units (CPUs)s can be used as workers during training. Although a parallel implementation utilizes multiple workers, this is well compensated by the significant reduction of training time as evidenced in the experiments. This approach can also benefit researchers and organizations that do not have access to specialized hardware and cloud-based instances.

1.6. Delimitations

The work aims not to break the records on the Atari 2600 benchmarks but to serve as a proof of concept and explore the potential of a novel approach.

The experiments rely on two open-source libraries, gym[3] for the environments used for benchmarking and rllib[4] for the reference implementation of DQN, A3C, and ES. The exact replication of the results is dependant on the specific versions of these libraries.

Due to limited resources and time constraints, the same hyperparameters are used for all Atari 2600 games and a single run is performed. Training is limited to a maximum of 25 million training frames on 16 games—some chosen at random while others were chosen from the publication Deep Neuroevolution (Conti *et al*., 2018).

2. Background

2.1. Reinforcement learning

RL is inherently different from supervised tasks where labeled samples containing both inputs and targets are available for training. For control problems that involve interaction with an entity or an

[2] http://www.swedishepa.se/Environmental-objectives-and-cooperation/Swedish-environmental-work/Work-areas/Climate/How-can-I-reduce-my-carbon-footprint-/.

[3] https://gym.openai.com/.

[4] https://docs.ray.io/en/master/rllib.html.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

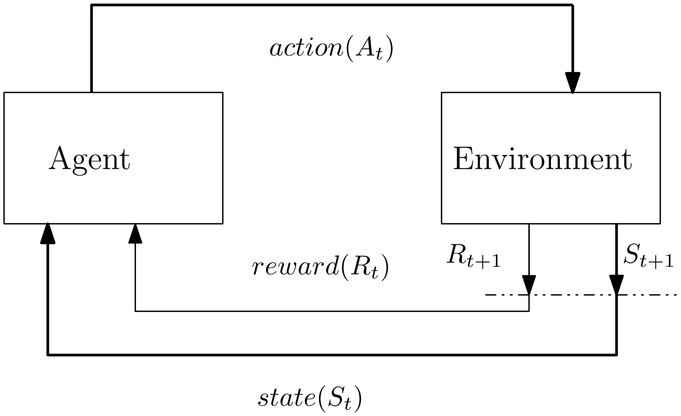


Figure 1. *Interaction of an agent with the environment at a time step t. Image credits (Sutton & Barto, 2018)*

environment, obtaining samples that are both representative and correct for all situations is often unrealistic. RL tackles this problem by allowing a goal-seeking agent to interact with an uncertain environment. This agent is responsible for maintaining a balance between exploration and exploitation to build a rich experience which it uses as a memory to make better decisions progressively.

A RL system can be characterized by the following elements:

- **Policy:** The agent interacts with an environment at discrete time steps $t = 0, 1, 2, 3, \ldots$. The policy is a rule that guides the agent on what actions (a_t) to take. It does so by creating a mapping from states (s_t), a perceived representation of the environment, and possible actions. Policies can either be deterministic, usually denoted by μ, $a_t = \mu(s_t)$, or they can be stochastic, which are usually denoted by π, $a_t \sim \pi(\cdot \mid s_t)$, representing the probability of taking an action at a particular state.
- **Rewards:** are responses from the environment. They represent the immediate and intrinsic appeal of taking a particular action in a specific state.
- **Value function:** is the maximal expected reward that can be accumulated over the future, starting from a particular state. This takes into account the long-time desirability of the state, which makes it different from rewards that provide an immediate response.
- **Model:** It represents the system dynamics of the environment. The presence of a model can help the agent make better predictions and learn faster.

The agent interacts with the environment in a sequence of time steps. At each time step, it observes a representation of the environment, that is, its state $s_t \in \mathcal{S}$, and then decides to take action $a_t \in \mathcal{A}$ by following a policy. As a consequence of its action, the agent receives an immediate reward from the environment, r_t, and transitions to the next state, s_{t+1}. The agent's objective is to learn a policy to maximize its cumulative reward over a time horizon. This fixed horizon is also known as a *trajectory (τ)* or an *episode*.

$$\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots \tag{1}$$

This interaction can be seen in Figure 1.

2.1.1. Markov decision process

The elements of an Reinforcement Learning (RL) problem, as described in Section 2.1 can be formalized using the Markov Decision Process (MDP) framework (Puterman, 1994; Boutilier *et al.*, 1999). Markov

Decision Process (MDP) can efficiently model the interactions between the environment and the agent by defining a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, where:

- $\mathcal{S}$ represents the finite set of states, $s_t \in \mathcal{S}$ with dimensionality N, that is, $|\mathcal{S}| = N$. Recall, a state is a distinctive characteristic of the environment or the problem being modeled.
- $\mathcal{A}$ represents the finite set of actions, $a_t \in \mathcal{A}$ with dimensionality K, that is, $|\mathcal{A}| = K$. Each action a_t is used as a control to interact with the environment at time t. A set of available actions at a particular state s is denoted by $A(s)$, where $A(s) \subseteq \mathcal{A}$.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the stochastic transition function, where $p\,(s' \mid s, a)$, also denoted as $p^a_{s,s'}$ is the probability of transitioning to state s' by taking action a at state s. These transition probabilities can also be *stationary*, that is, they are independent of time t.

$$p\left(s' \mid s, a\right) = p^a_{s,s'} = \mathbb{P}\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\}. \tag{2}$$

For this transition function to represent a proper probability distribution the following properties must hold:

$$p\left(s' \mid s, a\right) \geq 0 \text{ and } p\left(s' \mid s, a\right) \leq 1 \tag{3}$$

$$\sum_{s' \in S} p\left(s' \mid s, a\right) = 1 \tag{4}$$

Since MDPs are controlled Markov chains (Markov, 1906; Markov, 2006), the distribution of state at time $t + 1$ is independent of the past given the state at time t and the action performed by the agent.

$$\mathbb{P}\left[s_{t+1} = s' \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots\right] = \mathbb{P}[s_{t+1} = s \mid s_t, a_t = a] \tag{5}$$

- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the state reward function. It is a scalar value, $r(s, a, s')$, also denoted as $r^a_{s,s'}$, which is awarded to the agent for transitioning from state s to s' by performing an action a.

$$r\left(s, a, s'\right) = \mathcal{R}^a_{s,s'} = \mathbb{E}\left\{R_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\}. \tag{6}$$

Alternatively, rewards can also be defined as $R : S \times A \rightarrow \mathbb{R}$ where a reward is awarded for taking an action in a state.

$$r(s, a) = t_{s,a} = \mathbb{E}\{R_{t+1} \mid s_t = s, a_t = a\}. \tag{7}$$

2.1.2. Policies and optimality criteria

In the previous section, we have defined a mathematical formalism for RL problems using an MDP. To formally determine the *expected return,* G_t that is, objective that the policy seeks to optimize, two classes of MDPs are discussed:

- **Finite Horizon :** Given a fixed time horizon of **T**, the objective is to find a sequential decision policy π, that maximizes the expected return until the end of the time horizon. If we represent the sequence of rewards received after time t, by $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then the cumulative reward is simply:

$$G_t \doteq r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{8}$$

- **Infinite Horizon—Discounted rewards :** Given an endless time horizon $T = \infty$, the objective is to find a sequential decision policy, π that maximizes the expected discounted return.

$$G_t \doteq r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \cdots = \sum_{l=0}^{\infty} \lambda^l r_{t+l+1} \tag{9}$$

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

where λ is a parameter, $0 \leq \lambda < 1$, which discounts the future rewards, the rewards received after l time steps are worth λ^{l-1} times their original value. The discounting of rewards also ensures that the expected return converges. The following recursive relation between the rewards of successive time steps is the key result used in several RL algorithms.

$$\begin{aligned} G_t &\doteq r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \lambda^3 r_{t+4} + \cdots \\ &= r_{t+1} + \lambda \left(r_{t+2} + \lambda r_{t+3} + \lambda^2 r_{t+4} + \cdots \right) \\ &= r_{t+1} + \lambda G_{t+1} \end{aligned} \tag{10}$$

2.1.3. Value functions and Bellman equation

A stochastic policy π, is a mapping from a state $s \in \mathcal{S}$ to action $a \in \mathcal{A}(s)$. It represents the probability of taking action a in state s, $\pi(a \mid s)$. The state-value function, or simply value function, for a state s under a policy π is denoted as $v_\pi(s)$. It is the expected reward that can be accumulated starting with state s and following the policy π thereafter. For an infinite horizon model as described in the previous section, the *state-value function* can be formalized as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{l=0}^{\infty} \lambda^l R_{t+l+1} \mid S_t = s\right] \tag{11}$$

Similarly, *state-action-value function* $Q : S \times A \rightarrow \mathbb{R}$ is defined as the expected reward that can be gathered starting with state s, taking an action a and following the policy π thereafter.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{l=0}^{\infty} \lambda^l R_{t+l+1} \mid S_t = s, A_t = a\right] \tag{12}$$

A recursive formulation of Equation (11) which relates the value function of a state ($v_\pi(s)$) to the value function of its successor states ($v_\pi(s')$) is a fundamental result utilized throughout RL. It is known as the *Bellman equation* (Bellman, 1958):

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \lambda G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right)\left[r + \lambda \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \lambda v_\pi\left(s'\right)\right] \end{aligned} \tag{13}$$

A policy π is considered better than π' *iff* $v_\pi(s) \geq v_{\pi'}(s)$ for all states $s \in \mathcal{S}$. The policy which is better than all is known as the *optimal policy*: π_*, its value function, denoted by v_*, can be formulated as:

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}. \tag{14}$$

The *optimal state-action-value function*, denoted by q_*, can be formulated as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{15}$$

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

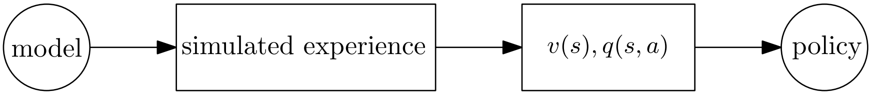


Figure 2. *Workflow for model-based learning techniques*

The *Bellman optimality equation* declares that the value function of a state under the *optimal policy* is the expected return received from the *'best (greedy)'* action in that state.

$$\begin{aligned} v_*(s) &= \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[r_{t+1} + \lambda G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[r_{t+1} + \lambda v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \lambda v_*(s')\big]. \end{aligned} \tag{16}$$

Similarly, the bellman optimality equation for q_*, can be formulated as:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\Big[r_{t+1} + \lambda \max_{a'} q_*\big(S_{t+1}, a'\big) \mid S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p\big(s', r \mid s, a\big)\Big[r + \lambda \max_{a'} q_*\big(s', a'\big)\Big] \end{aligned} \tag{17}$$

2.2. RL algorithms

Model-based RL algorithms rely on the availability of the transition and reward function. The critical component of such techniques is *'planning.'* The agent can plan and make predictions about its possible options and use it to improve the policy (Figure 2).

A major challenge of these techniques is the unavailability of the ground truth model, resulting in a bias where the agent performs poorly when tested in the real environment. In this work, we primarily focus on model-free RL.

Model-free methods don't require a model and primarily rely on *'learning'* the value functions directly.

2.2.1. Dynamic programming

When the model for the problem is known, dynamic programming (Bellman, 1954), a mathematical optimization technique, can be utilized to learn the optimal policy. Generalized Policy Iteration (GPI) is such an approach, and it uses an iterative two-step procedure to learn the optimal policy.

1. **Policy evaluation:** The first step evaluates the state-value function v_π using the recursive formulation in Equation (13).
2. **Policy improvement:** The second step uses the value function to generate an improved policy $\pi' \geq \pi$ using a greedy approach formulated in Equation (17).

Since the improved policy π' is greedy in actions to the current policy, that is, $\pi'(s) = \arg\max_{a\in\mathcal{A}} q_\pi(s, a)$

$$\begin{aligned} q_\pi(s, \pi'(s)) &= q_\pi(s, \arg\max_{a\in\mathcal{A}} q_\pi(s, a)) \\ &= \max_{a\in\mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s) \end{aligned} \tag{18}$$

this iterative process is guaranteed to converge to the optimal value (Sutton & Barto, 2018).

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

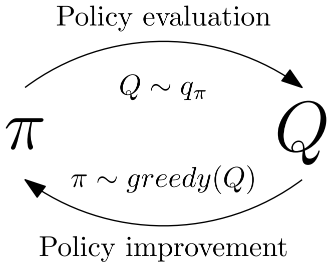


Figure 3. *GPI iterative process for optimal policy*

2.2.2. Monte Carlo techniques

Monte Carlo techniques estimate the value functions without the need of system dynamics of the environment making it a model-free method. They rely on *complete episodes* to approximate the expected return, G_t of a state. Recall, the value function, $v_t(s) = \mathbb{E}[G_t|S_t = s]$. In an episode, $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$, the occurrence of a state s is known as the visit to s. First-Visit Monte Carlo algorithm estimates the value function as the average of the returns (G_t) following the first visits to s whereas Every-Visit Monte Carlo algorithm estimates it as the average of the returns following every visit to s.

The optimal policy is then evaluated using the same GPI explained previously. The schematic is shown in Figure 3.

2.2.3. Temporal-difference methods

Temporal Difference (TD) methods are model-free. Unlike MC methods, where the agents wait until the end of the episode for the final return to estimate the value functions, TD methods rely on an estimated final return, $r_{t+1} + \lambda v(s_{t+1})$ known as the *TD target*. The state-value function and state-action-value function are estimated at every step using the below formulation:

$$v(s_t) \leftarrow v(s_t) + \alpha(r_{t+1} + \lambda v(s_{t+1}) - v(s_t)) \tag{19}$$

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_{t+1} + \lambda q(S_{t+1}, a_{t+1}) - q(s_t, a_t)) \tag{20}$$

where α is a hyperparameter called the *learning rate*, which controls the rate at which the value gets updated at each step. This technique is called *bootstrapping* because the target value is updated using the current estimate of the return $(r_{t+1} + \lambda v(s_{t+1}))$ and not the exact return, G_t.

2.2.4. Policy gradient

The methods discussed so far estimate the state-value, state-action-value functions respectively to learn what actions to take to maximize the overall return. Policy gradient consists of a family of methods that learn a *parameterized policy* without computing these value functions. A stochastic policy, parameterized with a set of parameters, $\theta \in \mathbb{R}^d$, can be represented as, $\pi(a \mid s, \theta) = p(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \theta)$, that is, the probability of taking an action at given state with a set of parameters θ at time t.

The parameters are learned by computing the gradient of an objective function, $J(\theta)$, a scalar quantity that measures the performance of the policy. The goal of these methods is to maximize the performance by updating the parameters of the policy using *stochastic gradient ascent* (Bottou, 1998) as

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_t} \tag{21}$$

where $\nabla_\theta J(\pi_\theta) \in \mathbb{R}^d$ is an estimate whose expectation is equivalent to the gradient of the policy's performance, known as *policy gradient*.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

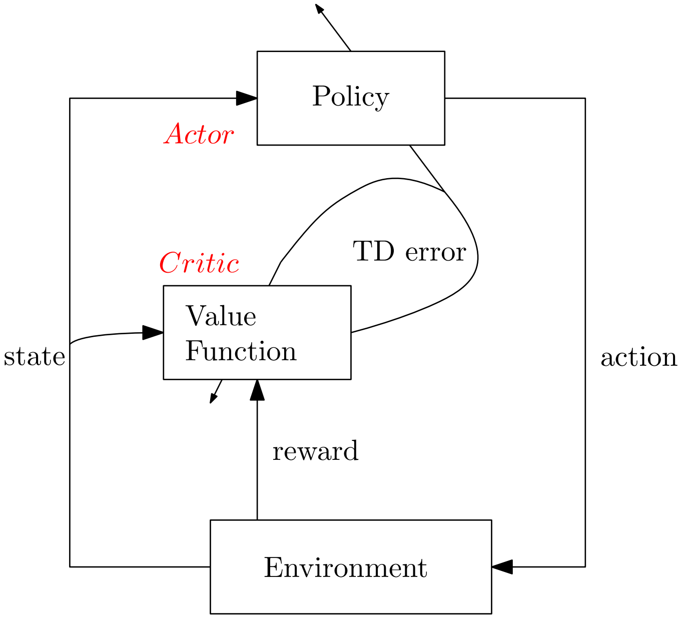


Figure 4. *Architecture for actor-critic methods (Sutton & Barto, 2018)*

2.2.5. Actor-critic methods

They consist of algorithms that rely on estimating the parameters of the value functions (state-value or state-action-value function) concurrently with the policy. The two components include:

1. **Actor:** is responsible for the agent's policy, that is, it is used to pick out which action to perform. The underlying objective is to update the parameters of the policy using Equation (21) guided by the estimates from the critic as the baseline.
2. **Critic:** is responsible for estimating the agent's value functions. The objective is to update the parameters, ϕ, of the state-value function, $v(s; \phi)$, or the state-action-value function, $q(s, a; \phi)$ by computing the TD error, δ_t as

$$\delta_t = r_{t+1} + \lambda v(s_{t+1}) - v(s_t) \tag{22}$$

Figure 4 shows the working of an actor-critic method.

2.3. Artificial neural networks

Artificial Neural Networks (ANNs), often referred to as NNs, were first proposed several decades ago (Fitch, 1944). The formulation of these methods is heavily inspired by the human brain, a highly intricate organ that is made up of over 100 billion neurons (Herculano-Houzel, 2009). Each neuron is a specialized nerve cell that transmits electrochemical signals to other neurons, gland cells, muscles, etc., which further perform specialized tasks. With such an interconnected network of neurons, our brain is able to carry out quite complex tasks. This mechanism is what ANNs are based on.

2.3.1. Mathematical model

A NN is a computational model that consists of an input layer, hidden layers (any number), and an output layer. Each layer consists of several artificial neurons, the fundamental units of the network, which define a function on the inputs. Any network that uses several hidden layers (usually more than 3) is referred

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

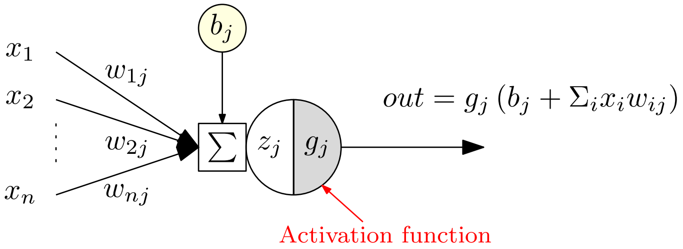


Figure 5. *Activation of a single neuron in the neural network*

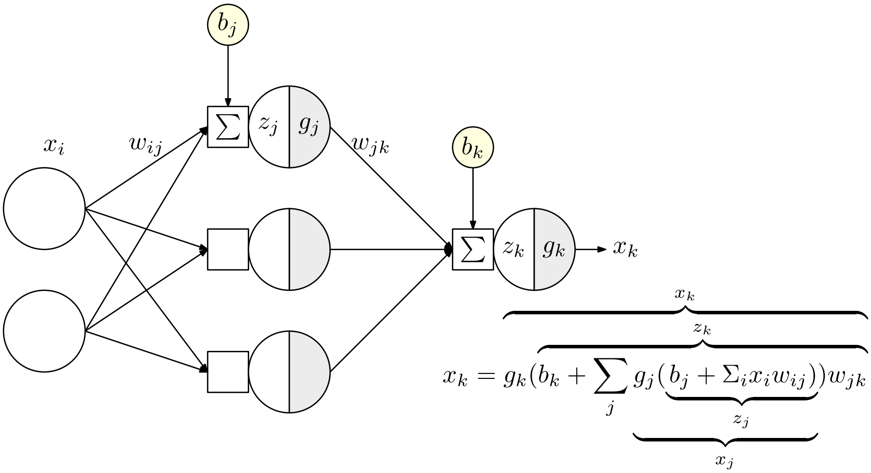


Figure 6. *Forward propagation in an artificial neural network with a single hidden layer*

to as a Deep Neural Network (DNN), and the paradigm of ML that deals with DNN is known as Deep Learning (DL) (Goodfellow *et al*., 2016).

In a feed-forward network, the interconnections between the nodes, that is, the neurons, do not form a cycle. Network topologies where the connections between neurons include a directed graph over a temporal sequence are known as Recurrent Neural Networks (RNNs). In this work, we primarily focus on a feed-forward network which is trained by performing a *forward pass*, also referred as forward propagation and a subsequent backward pass, also known as *backpropagation*.

Forward Propagation: When training a NN, the forward pass is the *calculation phase*, sometimes referred to as the *evaluation phase*. The input signals are forward propagated through all the neurons in each network layer in a sequential manner. The activation of a single neuron is shown in Figure 5, where the input signals $x_1, x_2 \ldots x_n$ are first multiplied by a set of weights, $w_{i,j}$, and then combined with a bias term, b_j to obtain a pre-activation signal, z_j. The bias is usually utilized to offset the pre-activation signal in case of an all-zero input; at times, this can alternatively be seen as an additional weight that is applied to an input, that is, a constant 1.

An activation function, g_j is then applied to z_j to obtain the final output signal of the neuron. The process is then repeated across all neurons of each layer one by one until the final output is received. Figure 6 shows the forward propagation in a NN with a single hidden layer of 3 neurons. The final output, x_k, is the prediction made by the NN given the input. This completes the forward pass of the network.

Backpropogation: When a neural network is created, the parameters, θ including the weights and biases, are often initialized randomly or else sampled from a distribution. The prediction of the NN from the forward pass is compared with the *true* output, computing an error, E, using a loss function. The resultant scalar value indicates how well the parameters of the NN perform the specific task.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

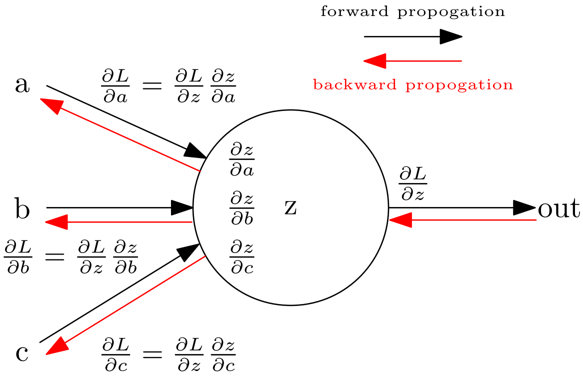

forward propogation
backward propogation
a
b
c
z
out
$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial a}$
$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b}$
$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial c}$
$\frac{\partial z}{\partial a}$
$\frac{\partial z}{\partial b}$
$\frac{\partial z}{\partial c}$
$\frac{\partial L}{\partial z}$

Figure 7. *Backpropagation of gradients by application of chain rule*

Training a NN involves adjusting the parameters of network θ, by first computing the derivative of the loss function and then using that information to update the parameters in the direction where the error is reduced. This is called gradient descent. Formally, the parameters are updated as:

$$\theta_L^{t+1} = \theta_L^t - \alpha \frac{\partial E}{\partial \theta_L^t} \tag{23}$$

where α is the learning rate. The term $\frac{\partial E}{\partial \theta_L^t}$ represents the partial derivative of the error with respect to the parameters of a layer L. These gradients can be computed by applying chain rule (shown in Figure 7) and involves the gradients of subsequent neurons and layers. Repeated application of chain rule can be used to calculate these gradients at every layer, all the way from the final layer, by navigating through the network backward, hence the name *backpropogation* (Rumelhart *et al*., 1986). This is a particular case of Automatic Differentiation (Rall, 1981).

A cost function is another name for a loss function when applied to either the entire training data or a smaller subset known as the batch or mini-batch. Some commonly used cost functions include mean squared error, mean absolute error, cross-entropy, hinge loss, Kullback-Leibler divergence (Kullback & Leibler, 1951) etc.

2.4. Activation Function and Weight Initialization

Neural Network (NN) with at least one hidden layer can serve as universal function approximators (Cybenko, 1989). A NN without an activation function acts like a linear regressor, where the output is a linear combination of its inputs, which restricts its overall learning capacity. An activation function is often used to introduce nonlinearity in the network, which enables it to model much more evolved and complex inputs like images, videos (Ma *et al*., 2018), audio (Purwins *et al*., 2019), text (Radford *et al*., 2019), etc. Some of the widely used nonlinear activation functions are shown in Figure 8.

The initialization of the network parameters, primarily the weights and biases of the different layers, plays a critical role when training a neural network. Situations where the activation output and the gradients explode (i.e., become significantly large) or vanish (i.e., become significantly small) due to these parameter values can significantly affect the convergence of the NN as highlighted in Sousa (2016). Some of the widely used weight initialization strategies and their properties are summarized below:

- **Normal:** The values for the parameters are drawn from a normal distribution, $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean, and σ^2 is the variance. The probability distribution function for normal distribution is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{24}$$

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

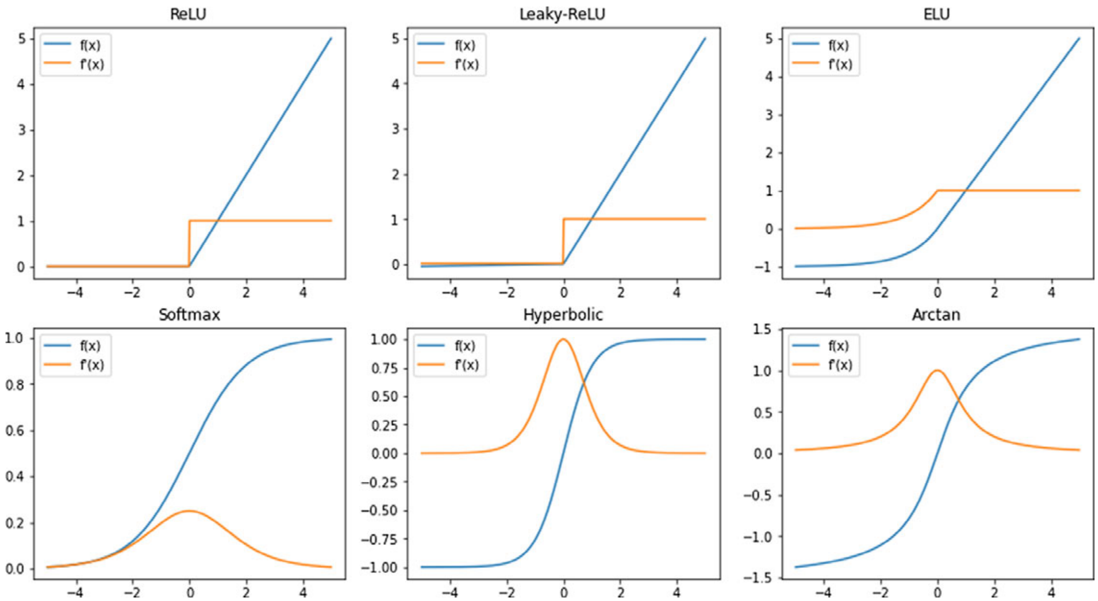


Figure 8. *Commonly used activation functions, f(x) and their derivatives, $f'(x)$ for $x \in [-5, 5]$*

- **Uniform:** The values for the parameters are drawn from a uniform distribution, $\mathcal{U}(a, b)$, where a is the lower bound and b is the upper bound for the distribution. The probability distribution function for *continuous* uniform distribution is:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \le x \le b, \\ 0 & \text{for } xb \end{cases} \tag{25}$$

- **Xavier:** This is also known as *Glorot initialization* (Glorot & Bengio, 2010) and overcomes the problem of vanishing, exploding gradients by adjusting the variance of the weights. *Xavier uniform initialization* samples the weight matrix, W, between the layer j and $j + 1$ from a uniform distribution.

$$\mathcal{U}(-a, a), \quad \text{where} \quad a = \text{gain} \times \sqrt{\frac{6}{n_j + n_{j+1}}} \tag{26}$$

The parameter *gain* is an optional scaling factor and, n is the number of neurons of the respective layer. *Xavier normal initialization* draws samples from a normal distribution.

$$\mathcal{N}\left(\mu, \sigma^2\right), \quad \text{where} \quad \mu = 0, \sigma^2 = \text{gain} \times \sqrt{\frac{2}{n_j + n_{j+1}}} \tag{27}$$

- **Kaiming:** This is also known as He initialization (He *et al*., 2015b), and is well suited for NN with asymmetric and nonlinear activation functions, for example, ReLU, Leaky-ReLU, ELU, etc. *Kaiming uniform initialization* samples the weight matrix, W, between the layer j and $j + 1$ from a uniform distribution.

$$\mathcal{U}(-a, a), \quad \text{where} \quad \text{a} = \text{gain} \times \sqrt{\frac{3}{\text{fan\_mode}}} \tag{28}$$

The parameter *gain* is an optional scaling factor, and *fan_mode* is the number of neurons of the respective layer. *fan_mode* $= n_{j+1}$ preserves the magnitude of the variance for the weights during the backward pass whereas, *fan_mode* $= n_j$ preserves it in the forward pass. *Kaiming*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

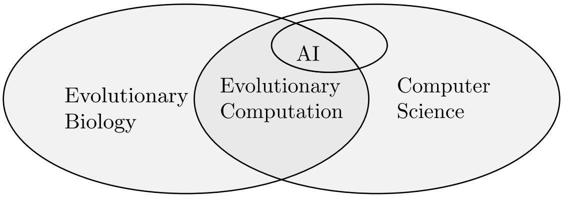


Figure 9. *Evolutionary computation has its roots within computer science, artificial intelligence and evolutionary biology*

normal initialization draws samples from a normal distribution.

$$\mathcal{N}\left(\mu, \sigma^2\right), \quad \text{where} \quad \mu = 0, \text{std} = \frac{\text{gain}}{\sqrt{\text{fan\_mode}}} \tag{29}$$

2.5. Evolutionary computation

EC draws inspiration from nature and utilizes the same concepts to solve computational problems. This family of algorithms has its root spread out in computer science, artificial intelligence as well as evolutionary biology, Figure 9.

EC belongs to a class of population-based optimization techniques where a set of solutions are generated and iteratively updated through *genetic operators*. A new generation of solutions is produced by stochastically eliminating 'less fit' solutions and introducing minor random variations in the population. At a high level, it is simply adapting to the changes within the environment by means of small random changes. The combined use of EC and DL has led to very auspicious results on several complex engineering tasks and drawn great academic interest (Salimans *et al*., 2017; Conti *et al*., 2018).

EC consists of several meta-heuristic optimization algorithms, some of which are explained in the following subsections.

2.5.1. Genetic algorithm

GAs, first developed in 1975 (Holland, 1992a), is a widely used class of evolutionary algorithms (Mitchell, 1996) that takes inspiration from Charles Darwin's theory of evolution (Darwin, 1859). It is meta-heuristic-based optimization technique that follows the principle of '*survival of the fittest*'.

The key components of any GA (Mitchell, 1996) are:

- **Population:** is the subset of all possible solutions to a problem. Every GA starts with an initialization of a set of solutions that become the first population. Each solution, individual from the population, is known as a *chromosome* which consists of several *genes*.
- **Genotype:** is the representation of the individuals of a population in the computation space. This is done to ensure efficient processing and manipulation by the computing system.
- **Phenotype:** is the actual representation of the population in the space of solutions.
- **Encoding and Decoding:** For more straightforward problems, the genotype and the phenotype may be the same. However, complex problems rely on a translation mechanism to encode a phenotype to a genotype and a decoder for the reverse task.
- **Fitness function:** is a tool to measure the performance of an individual on a specific problem.
- **Genetic operators:** are tools utilized by the algorithm to produce a new set of solutions from the existing ones.

The algorithm starts by initializing a set of solutions, the population, which is also known as a *generation*. At each iteration, the fitness function is used to evaluate the performance of the entire population.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

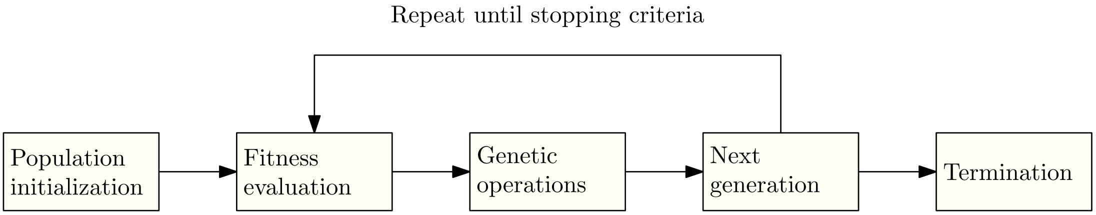


Figure 10. *Workflow of a simple genetic algorithm*

The solutions with higher fitness (Baluja & Caruana, 1995) are stochastically selected as the parents of the current generation. Genetic operators are applied to the parents to generate the next generation. This process is repeated until the required level of fitness is achieved. The complete workflow of a simple GA is shown in Figure 10.

The *genetic operators* are fundamental components of this algorithm that guides it toward a solution. The widely used genetic operators are summarized in the following subsections.

Selection Operators: The selection operator uses the fitness values as a measure of performance and picks a subset of the population as the current generation's parents. A common strategy is to assign a probability value proportional to the fitness scores, with the goal that *more fit* individuals have a higher chance of being selected. Some other strategies include random selection, tournament selection, and roulette wheel selection, proportionate selection, steady-state selection, etc. (Haupt & Haupt, 2003). The best performing individual from the current generation is selected and carried to the next generation without any alterations. This strategy is known as *elitism* (Baluja & Caruana, 1995) and ensures that performance does not decrease over iterations.

Crossover operator: This operator picks a pair from the solutions shortlisted by the selection operator. The selected pair then *mates* and generates new solutions, which are known as the *offspring*. Each offspring is produced stochastically and contains genetic information, that is, features from either parent which is analogous to the crossover of genes in biology.

Single-point crossover is a commonly used crossover operator. A random *crossover point* is picked on the encoding of both parents. The offspring are then produced by recombination of the left partition of one parent with the right split of the other and vice versa. The same strategy can be extended to more than one crossover point. Figure 11 shows *single-point crossover* and *two-point crossover* between two binary encoded parents. In situations, where the encoding is an ordered list (Larranaga *et al.*, 1999), the solutions generated with the methods mentioned above, may be invalid. To overcome such issues, specialized operators like order crossover (Larranaga *et al.*, 1996), cycle crossover (CX) (Larranaga *et al.*, 1999) and partially mapped crossover have been published.

Mutation operator: The mutation operator makes small arbitrary changes to an individual or its encoded representation which promotes the algorithm to explore the search space and find new solutions. As a result, the diversity within the population is improved during successive generations, and the algorithm can avoid getting stuck at local minima. Some commonly used mutation operators include Gaussian operators, uniform operators, bit flip operators (binary encoding), and shrink operators (Da Ronco & Benini, 2014). These operators are applied with a relatively small probability known as the *mutation rate* and have a few hyperparameters that can be tuned to specific problems.

2.5.2. Evolution strategies

Evolution Strategy (ES) is a black-box optimization technique that draws inspiration from the theory of natural selection, a process in which individuals from a population adapt to the changes of the environment.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

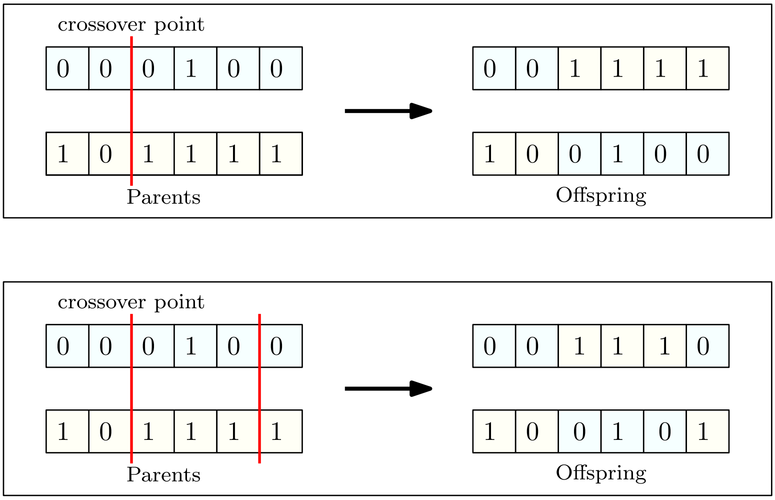


Figure 11. *Single-point crossover (top) and two-point crossover (bottom) between two parents to generate offsprings for the next generation*

Consider a 2D function, $f:\mathbb{R}^2 \rightarrow \mathbb{R}$, for which the gradient, $\nabla f:\mathbb{R}^n \rightarrow \mathbb{R}^n$, can not be computed directly, and we wish to obtain the parameters x^* and y^*, such that $f(x^*, y^*)$ is the global maximum. The ES algorithm starts with a set of solutions for the problem, which are sampled from a parameterized distribution. An objective function then evaluates each solution and returns a single value as a fitness metric which is then utilized to update the parameters of the distribution.

Gaussian Evolution Strategy: One of the simplest techniques is to sample from a normal distribution, $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean and σ^2 is the variance. For the function defined above, one could start with $\mu = (0, 0)$ and some constant, $\sigma = (\sigma_x, \sigma_y)$. After one generation, the mean can be updated to the best performing solution from the population. Such a naive approach is only applicable to simple problems and also prone to getting stuck at local minima since it is greedy with respect to the best solution at each step of the training process.

Covariance Matrix Adaptation Evolution Strategy: The simple Gaussian strategy explained above assumes a constant variance. In some applications, it is worthwhile to explore the search space by having a larger variance initially and fine-tuning it later on when we are close to a good solution. Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) samples the solutions from a multivariate Gaussian distribution. It then adapts both the mean as well as the covariance matrix from the fitness metrics of the solutions. This algorithm is widely applicable to non-convex and nonlinear optimization problems (Hansen & Auger, 2011), especially in the continuous domain. There is no need to tune the parameters for a specific application since optimizing the parameters is incorporated within the design of this algorithm.

Natural Evolution Strategy: Simple Gaussian ES and CMA-ES both have the identical drawback of utilizing only the best individuals from the population and discarding all the others. At times, it might be beneficial to incorporate the information from low-scoring individuals as well, since they contain the vital information of what not to do. Natural Evolution Strategy (NES) (Wierstra *et al.*, 2008) utilize the information from the entire population to estimate the gradients, which are utilized to update the distribution from which the solutions were sampled. For an objective function, F with parameters θ which are sampled from a parameterized distribution $p_\psi(\theta)$. The algorithm's goal is to optimize the average objective $\mathbb{E}_{\theta\sim p_\psi}[F(\theta)]$. The parameters of the distribution are iteratively updated using stochastic gradient ascent. The gradient of the objective is computed as follows:

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Algorithm 1: Evolution strategy for RL by OpenAI (Salimans *et al.*, 2017)

```
Input: learning rate $\alpha$, noise standard deviation $\sigma$, initial neural network parameters $\theta_0$
for t = 0,1 ... do
    Sample $\epsilon_1, \epsilon_2 \dots \epsilon_n \sim N(0, I)$
    Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, 2 \dots n$
    Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
```

$$\begin{aligned}\nabla_\psi\left(\mathbb{E}_{\theta\sim p_\psi}[F(\theta)]\right) &= \nabla_\psi\left(\int_\theta p_\psi(\theta)\cdot F(\theta)\cdot d\theta\right)\\ &= \int_\theta \nabla_\psi\left(p_\psi(\theta)\right)\cdot F(\theta)\cdot d\theta\\ &= \int_\theta p_\psi(\theta)\cdot\nabla_\psi\left(\log p_\psi(\theta)\right)\cdot F(\theta)\cdot d\theta\\ &= \mathbb{E}_{\theta\sim p_\psi}\left[\nabla_\psi\left(\log p_\psi(\theta)\right)\cdot F(\theta)\right]\end{aligned} \tag{30}$$

Equation: 30 uses the same log-likelihood technique as the popular *reinforce algorithm* (Sutton & Barto, 2018). The expectation can be computed using Monte Carlo approximations by sampling points as shown in Equation (31).

$$\begin{aligned}\nabla_\psi\left(\mathbb{E}_{\theta\sim p_\psi}[F(\theta)]\right) &= \mathbb{E}_{\theta\sim p_\psi}\left[\nabla_\psi\left(\log p_\psi(\theta)\right)\cdot F(\theta)\right]\\ &\approx \frac{1}{\lambda}\sum_{k=1}^{\lambda}\left[\nabla_\psi\left(\log p_\psi(\theta)\right)\cdot F(\theta)\right]\end{aligned} \tag{31}$$

Evolution Strategy for RL: Researchers in Salimans *et al.* (2017) utilized Natural Evolution Strategy to find the optimal parameters of a neural network for RL problems. The parameters represent the weights and biases, and the objective is the stochastic return from the environment. The parameterized distribution, $p_\psi(\theta)$ is assumed to be multivariate gaussian with mean ψ and a fixed noise of $\sigma^2 I$.

$$\theta \sim \mathcal{N}\left(\psi, \sigma^2 I\right) \text{ equivalent to } \theta = \psi + \sigma\epsilon, \epsilon \sim \mathcal{N}(0, I) \tag{32}$$

This parametrization allows to rewrite the average objective as $\mathbb{E}_{\theta\sim p_\psi} F(\theta) = \mathbb{E}_{\epsilon\sim N(0,I)} F(\theta + \sigma\epsilon)$. Given the revised definition of the objective function, the authors derive (Salimans *et al.*, 2017) the gradient of the objective function in terms of θ as

$$\nabla_\theta \mathbb{E}_{\epsilon\sim N(0,I)} F(\theta + \sigma\epsilon) = \frac{1}{\sigma}\mathbb{E}_{\epsilon\sim N(0,I)}\{F(\theta + \sigma\epsilon)\epsilon\} \tag{33}$$

The gradients are approximated by generating samples of noise ϵ during training and the parameters are then optimized using stochastic gradient ascent as shown in Algorithm 1.

2.5.3. Neuroevolution

Neuroevolution is a form of Artificial Intelligence (AI) that combines EC with NNs. The objective is to either learn the topology of the NN or its parameters, that is, weights and biases by utilizing algorithms discussed in Sections 2.5.1 and 2.5.2.

An individual within the population is a NN or its parameters—weights and biases. The trivial method of storing a NN, as a complex data structure, in the parameter space scales poorly in memory, especially with the increasing population size, and makes the application of genetic operators a computationally expensive task. Hence, the need for simplified encoding of the NN.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

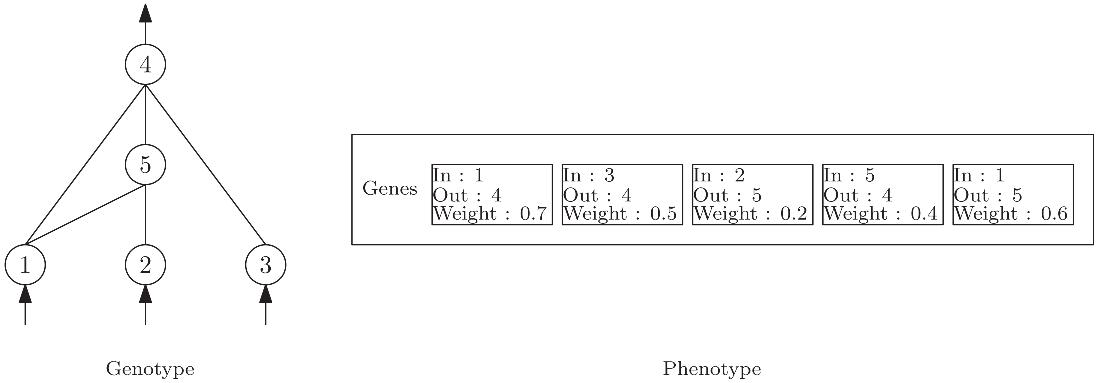


Figure 12. *A simplified version of direct encoding proposed in NEAT (Stanley & Miikkulainen, 2002). Mapping of genotype, a neural network to a phenotype, its encoding*

Direct encoding: In the notable work, 'Evolving neural networks through augmenting topologies', NEAT (Stanley & Miikkulainen, 2002), the authors propose a direct encoding wherein the connections between each neuron are explicitly stored with all the weights and biases, as shown in Figure 12. Such an encoding gives a very high degree of flexibility to the network topology but has a massive search space due to a fine granularity of the encoding. This approach is not well suited for DNN, which consists of several hidden layers, each with thousands of neurons.

Indirect encoding: This type of encoding relies on a translation function or a mechanism that maps a neural network to its encoded representation. Unlike, direct encoding which is a quick way to encode or decode a neural network, this type of encoding has a processing overhead and limited granularity. Some implementations that employ an indirect encoding include Compositional Pattern Producing Networks (CPPNs) (Stanley, 2007), HyperNEAT (Stanley *et al.*, 2009) and Deep Neuroevolution (Conti *et al.*, 2018).

2.6. Related work

Computing the Q-value for complex problems with a large state-action space using traditional tabular RL algorithms is computationally infeasible. This led to the use of function approximations like neural networks for estimating the Q-values (Melo *et al.*, 2008), which is memory efficient and can also be applied to problems with continuous state and action spaces (Van Hasselt, 2013).

The first significant breakthrough in deep reinforcement learning, the combined use of deep learning and reinforcement learning, was the introduction of DQN (Mnih *et al.*, 2015) which incorporated two critical components in the training process, an *experience replay buffer* and a *separate target network*. The replay memory stores the agent's experience, including the states, actions and rewards over a few episodes and the target network is a periodically updated copy of the main network which is used to decide what actions to perform. During training, samples are drawn from the replay memory to remove the correlations between the observations. The combined use of these two innovative techniques stabilized the learning of Q-values and achieved state-of-the-art performance on many Atari 2600 benchmarks[5] .

Over the past few years, several extensions to DQN have been proposed. A known problem with Q-learning is its tendency to overestimate the targets since it relies on a single estimator for the Q-values. A technique called Double Q-learning (Hasselt, 2010) tries to rectify this issue by utilizing a double estimator. An extension of DQN with double estimators, was implemented in Double DQN (van

[5] https://github.com/openai/atari-py.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Hasselt *et al*., 2016). When sampling from the replay buffer as proposed in DQN, all samples, that is, transitions were given the same significance. Research on a *prioritized experience replay* (Schaul *et al*., 2015) that offered a higher importance to more frequent transitions resulted in improved performance and was included in most subsequent research. A Dueling network architecture (Wang et al. 2016) with two estimators—one for Q-values and the other for state-dependent action advantage function, was proposed and shown to have better generalization properties. Deep Deterministic Policy Gradient (DDPG) (Lillicrap *et al*., 2015) combines the use of deterministic policy gradient and DQN and can be operated on a continuous action space making its application particularly useful in robotics (Dong & Zou, 2020).

The slow rate of convergence and longer duration of training prompted the need for parallelizable algorithms. A distributed framework for RL, Gorilla (Nair *et al*., 2015) was proposed by researchers at Google which consists of several actors, all learning continuously in parallel and syncing their local gradients with a global parameter server after a few steps. This parallelized variant of DQN outperformed its standard counterpart on several benchmarks and reduced the training time.

Asynchronous Advantage Actor Critic (A3C) (Mnih *et al*., 2016) is a parallelized policy gradient method that consists of several agents, independently interacting with copies of environments in parallel. The algorithm consists of one global network with shared parameters and uses asynchronous gradient descent for optimization. In Asynchronous Advantage Actor Critic (A3C), each agent communicates with the global network independently. The aggregated policy from different agents, when combined together may not be optimal. A2C (Mnih *et al*., 2016) is deterministic and synchronous version of A3C, which resolves this inconsistency by using a coordinator that ensures a synchronous communication between agents and global network.

2.6.1. Evolution-based RL

The Q-learning-based methods try to solve the Bellman optimality equation whereas, policy gradient algorithms consist of parameterizing the policy itself to learn the probability of taking an action in a state. Regardless, they usually entail expensive computations, such as gradient calculations and back-propagation, which result in very lengthy (i.e., hours to even days) training in order to obtain desirable results, especially when solving complex problems with large state spaces (Lillicrap *et al*., 2015; Mnih *et al*., 2015; Mnih *et al*., 2016).

Recently we see a surge of a number of alternative approaches to solving RL problems, including the use of EC. These methods have been quite effective in discovering high-performing policies (Whiteson, 2012). A notable contribution is the work by OpenAI on Evolution Strategy (ES) (Salimans *et al*., 2017) to train deep neural networks. ES does not calculate the gradients analytically, instead approximate the gradient of the reward function in the parameter space using minor random tweaks. This massively scalable algorithm, trained on several thousand workers was able to solve the 3D humanoid walking benchmark (Tassa *et al*., 2012) within minutes and also outperformed several established deep reinforcement learning algorithms on many complex robotic control tasks and Atari 2600 benchmarks. Collaborative Evolutionary Reinforcement Learning (CERL) (Khadka *et al*., 2019) is another scalable framework that was designed to simultaneously explore different regions of the search space and creating a portfolio of policies, the results on continuous control benchmarks highlighted improved sample efficiency of such approaches. Another breakthrough work by researchers at UberAI, is Deep Neuroevolution (Conti *et al*., 2018), a *black-box optimization* technique based on principles of evolutionary computation. Deep Neuroevolution utilizes a simple GA to train a convolutional neural network that approximates the Q-function.

Results from these publications highlight improved performance and significant speedup due to distributed training (Salimans *et al*., 2017; Conti *et al*., 2018). Moreover, these gradient-free approaches can be applied to non-differentiable domains (Cho *et al*., 2014; Xu *et al*., 2016; Liu *et al*., 2018) and for training models that require low precision arithmetic—binary neural networks on low precision hardware. In a subsequent paper (Conti *et al*., 2018), the same researchers incorporated quality diversity (QD) (Cully *et al*., 2015; Pugh *et al*., 2016) and novelty search (NS) (Lehman & Stanley, 2008) to ES,

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

and proposed three different algorithms, which improved performance and also directed the exploration. Their work also highlighted the algorithm's robustness against a local minima.

Several population-based evolutionary algorithms have been used to train neural networks or optimize their architecture (Jaderberg *et al*., 2017) and the hyperparameters (Liu *et al*., 2017), but a rather interesting approach was proposed in Genetic Policy Optimization (GPO) (Gangwani & Peng, 2018) algorithm which utilized imitation learning as crossover and policy gradient techniques for mutations. The algorithm had superior performance and improved sample efficiency in comparison to known policy gradient methods. Evolutionary Reinforcement Learning (ERL) (Khadka & Tumer, 2018) is another hybrid approach that uses a population-based optimization strategy guided by policy gradients for challenging control problems.

3. Species-based Genetic algorithm (Sp-GA)

A typical GA (Section 2.5.1) consists of a population of individuals where each individual is a potential solution to the problem. In the context of RL, a solution may represent a policy, a neural network that estimates the Q-function (Mnih *et al*., 2015) or a neural network that predicts what action to take given the state as an input. In this work, we propose a GA wherein each individual is a neural network that estimates the Q-function for the specific RL problem. The algorithm initializes the population of such neural networks from a species-based initialization routine and iteratively evolves the population to search for the best solution. We call this algorithm Sp-GA.

3.1. Species initialization

The notion of speciation was first investigated in NEAT (Stanley & Miikkulainen, 2002), which clustered neural networks with similar topologies into subclasses called *species*. The philosophy behind the approach is to give sufficient time for evolution to new solutions within a smaller subset before they compete with other species. As a result, innovation, novelty, and diversity are protected within the population.

Sp-GA extends the concept of speciation to the weight initialization strategy of a neural network. The initialization of weights is a critical design choice that can adversely affect the network's rate of convergence (He *et al*., 2015b; Sousa, 2016). The choice of initialization sets the starting point of the optimization process and therefore controls the effectiveness of training. The algorithm maintains a pool of such strategies, each having unique properties and particular advantages. When the population is initialized, a neural network is assigned a species uniformly at random, and the weights are initialized as per the strategy for that species. Figure 13 shows the distribution of individuals, that is, neural networks initialized from the proposed methodology wherein five weight initialization strategies were utilized.

3.2. Algorithm hyperparameters

The proposed algorithm has the following hyperparameters:

1. **Population size (N):** controls the number of neural networks in the population. This is an important parameter that controls the search space to find the solution. For more straightforward problems like CartPole from gym environment even a small value $N \approx 50$ gave good results. For more complex problems like the ones addressed in this work, a higher number is preferred.
2. **Mutation Power (σ):** controls the strength of mutation, that is, the perturbations applied to the parameters of the neural network. This operator is responsible for exploration of better solutions in the search space. In case of Gaussian operator, it is the variance of the distribution. A small value between $0.001 \sim 0.003$ showed promising results.
3. **Fraction/Number of elites (E):** control the number of neural networks shortlisted as parents of the current generation and are responsible for producing the next generation. This parameter is responsible for preserving the best performing solution of a generation.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

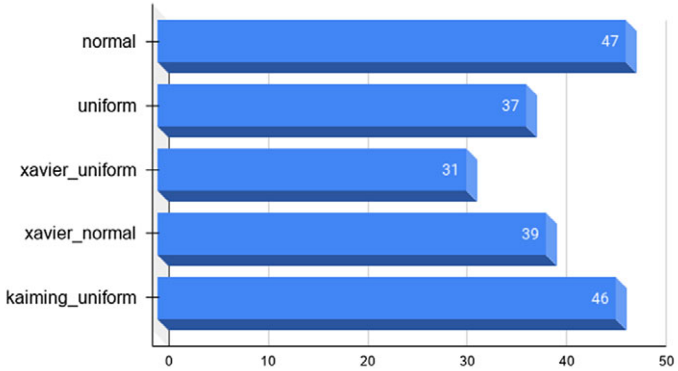


Figure 13. *Distribution of species in a population of size 200. Each species is a unique weight initialization strategy for neural networks*

4. **Mutation probability** (ψ)**:** represents the probability of using the mutation operator. Conversely, 1-ψ is the probability of using the crossover operator.
5. **Number of species** (S)**:** is the number of species spawned in the population. Each species corresponds to a weight initialization strategy of a neural network. During initialization, each neural network is assigned a particular species uniformly at random. The choice of initialization sets the starting point of the optimization process and therefore controls the effectiveness of training.

The pseudocode for Sp-GA is shown in Algorithm 2. The algorithm starts with the initialization of a population P, neural networks, with S different species. Each network is an estimator of the Q-function for the RL problem. The next step is to evaluate the fitness of the entire population. The fitness value of a single neural network is estimated by calculating the cumulative return for a fixed time horizon in the simulated environment. Once the fitness values are evaluated, a fraction of neural networks E, with the highest scores, are selected as the parents of the current generation. The parents then generate the offspring, using genetic operators as explained in Section 3.3.2 to form the new generation. The process is then repeated until the stopping criteria.

3.3. Distributed Sp-GA

A unique advantage of a GA is its ability to evaluate the fitness of the individuals in parallel, making it suitable for a distributed computation. A master-worker segregates the population into smaller subsets which are then sent to slave workers for fitness evaluation. These slave workers can work independently in parallel and return the scalar fitness scores back to the master-worker for the subsequent steps of the algorithm.

Such a naive approach cannot be directly applied to Sp-GA due to several reasons. A large population size would involve sending a lot of neural networks back and forth between workers over a network. A typical deep neural network that is stored as a complex data structure (or an object) takes a few megabytes of memory (Conti *et al.*, 2018) and is not trivially serializable. This would result in a high memory and bandwidth requirement for the communication.

Distributed Sp-GA relies on an encoding mechanism that takes inspiration from Deep Neuroevolution (Conti *et al*., 2018). The encoding of a neural network scales well with memory, can be serialized and therefore utilizes low bandwidth making it ideal for distributed training. This encoding also enables an intuitive application of genetic operators like mutation as well as crossover which was not implemented in the previous work (Salimans *et al*., 2017; Conti *et al*., 2018).

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Algorithm 2: Species-based genetic algorithm (Sp-GA)

```
Input: population size $N$, number of elites $E$, number of generations $G$, species initialisation
       routine $\phi$, mutation power $\sigma$, fitness function $\mathcal{F}$, mutation fraction $\psi$
Output: Elite
for $i$ = 1,2 ... $N$ do
    $P_i^{g=1}$ = $\phi$(uniformRandom(0,$S$))                    // initialize a species of DNN
for $g$ = 1,2 ... $G$ do
    for $i$ = 1,2 ... $N$ do
        $F_i = \mathcal{F}(P_i^{g=1})$                             // evaluate fitness
    Sort $P^{g=g}$ with descending order by $F_i$
    Elite = $P_1^{g=g}$
    if $g$=1 then
        Parents = $P_{i=1...E}^{g=1}$
    else
        Parents = $P_{i=1...E-1}^{g=1}$ ∪ Elite
    for $i$ = 1,2 ... $N$ do
        if uniform(0,1)¿$\psi$ then
            Parent1 = Parents(uniformRandom(0,$E$))
            Parent2 = Parents(uniformRandom(0,$E$))
            $P_i^{g=g}$ = crossover(parent1,parent2)
        else
            Parent = Parents(uniformRandom(0,$E$))
            $P_i^{g=g}$ = mutation(Parent,$\sigma$)
Return: Elite
```

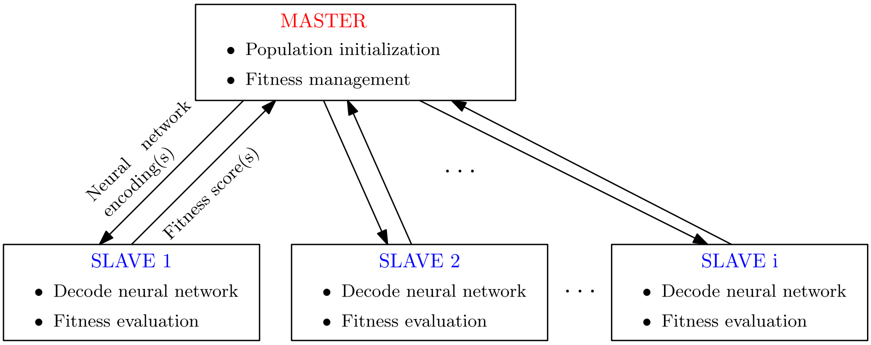


Figure 14. *Distributed framework for Sp-GA*

Figure 14 shows the distributed framework implemented in Sp-GA. The pipeline can be summarized as:

1. The master-worker first initializes the population—a set of NNs. It is responsible for encoding these networks and creating a mapping dictionary.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

$$\boxed{[(\tau_0, species\_identifier), (\tau_1, \sigma_1), (\tau_2, \sigma_2), \ldots\ldots, (\tau_g, \sigma_g)]}$$

Figure 15. *Proposed encoding of a neural network*

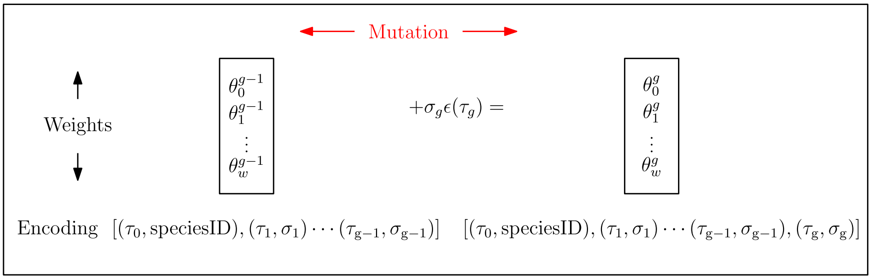

Mutation
Weights
θ_0^{g-1} θ_1^{g-1} $\vdots$ θ_w^{g-1}
$+\sigma_g\epsilon(\tau_g) =$
θ_0^{g} θ_1^{g} $\vdots$ θ_w^{g}
Encoding
$[(\tau_0, \text{speciesID}), (\tau_1, \sigma_1) \cdots (\tau_{g-1}, \sigma_{g-1})]$
$[(\tau_0, \text{speciesID}), (\tau_1, \sigma_1) \cdots (\tau_{g-1}, \sigma_{g-1}), (\tau_g, \sigma_g)]$

Figure 16. *Mutation of a neural network in parameter and encoded space. $\theta_{0:w}^{g-1}$ represent the parameters of the network at $g-1$ generation. $\theta_{0:w}^{g}$ are obtained by utilizing a gaussian operator with σ_g as the mutation power and τ_g as the random seed*

2. During training, the master sends an encoded neural network (or a subset) to the slave workers.
3. A slave worker decodes the encoding to produce the complete neural network, evaluate its fitness and returns a scalar value, the fitness score, back to the master-worker.
4. Once the fitness values of the entire population are available, the subsequent steps of the Algorithm 2 are carried out by the master-worker.

3.3.1. Model encoding

An individual from the population which is a neural network, is stored as list of tuples, as shown in Figure 15.

The first tuple consists of a *randomly selected seed* utilized for initializing the model parameters, and the second entry is the *identifier for the species* of the network. Every subsequent tuple consists of another seed value and the mutation power (σ) utilized by the genetic operators for training the network parameters.

3.3.2. Genetic operators in encoded space

The Gaussian operator is the choice for mutation in this work. Given a neural network with parameters, θ, the Gaussian operator updates all the parameters as:

$$\theta' = \theta + \sigma\epsilon \tag{34}$$

where $\epsilon \sim \mathcal{N}(0, I)$ and where σ is the mutation power. If a parent is selected for mutation, the encoding of the resultant network is simply the original encoding appended with a new tuple containing the seed and the mutation power. This process is shown in Figure 16. When needed, any encoding can be decoded to the actual network by iteratively setting the seed and applying the Gaussian mutations.

The proposed encoding is also well suited for the application of single-point crossover as described in Section 2.5.1. A random point on the encoded representation of the parents is chosen as the crossover point. A new offspring is produced by merging the left and right partitions (and vice versa) of either

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 1. *Some key libraries used for the experiments*

Library	Version
torch	1.7.1
atari-py	0.2.6
gym	0.18.0
numpy	1.19.5
ray	1.2.0

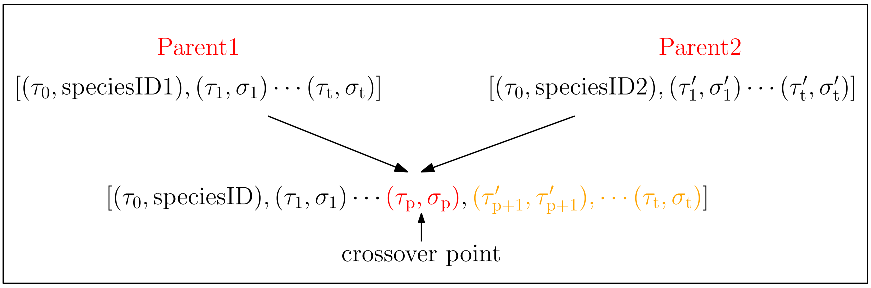

Parent1
Parent2
$[(\tau_0, \mathrm{speciesID1}), (\tau_1, \sigma_1) \cdots (\tau_\mathrm{t}, \sigma_\mathrm{t})]$
$[(\tau_0, \mathrm{speciesID2}), (\tau'_1, \sigma'_1) \cdots (\tau'_\mathrm{t}, \sigma'_\mathrm{t})]$
$[(\tau_0, \mathrm{speciesID}), (\tau_1, \sigma_1) \cdots (\tau_\mathrm{p}, \sigma_\mathrm{p}), (\tau'_{\mathrm{p}+1}, \tau'_{\mathrm{p}+1}), \cdots (\tau_\mathrm{t}, \sigma_\mathrm{t})]$
crossover point

Figure 17. *Crossover of two neural networks in encoded space*

parent. The resulting list represents a *valid encoding* of a neural network. Figure 17 shows this intuitive application of crossover. This approach can also be applied to neural networks with different architectures and took inspiration from the crossover of chromosomes in genetics.

4. Implementation

4.1. Hardware and Software

The entire code was written in Python version 3.8.7. The critical libraries utilized for the implementation are summarized in Table 1.

The development was done within a dockerized environment, and training was performed on a Kubernetes cluster. The quota allocated to this project was limited to 50 CPUs and a memory of 200GB. No GPU was utilized in any experiment. Reference implementations for DQN, A3C, and ES were obtained from rllib package in the ray library.

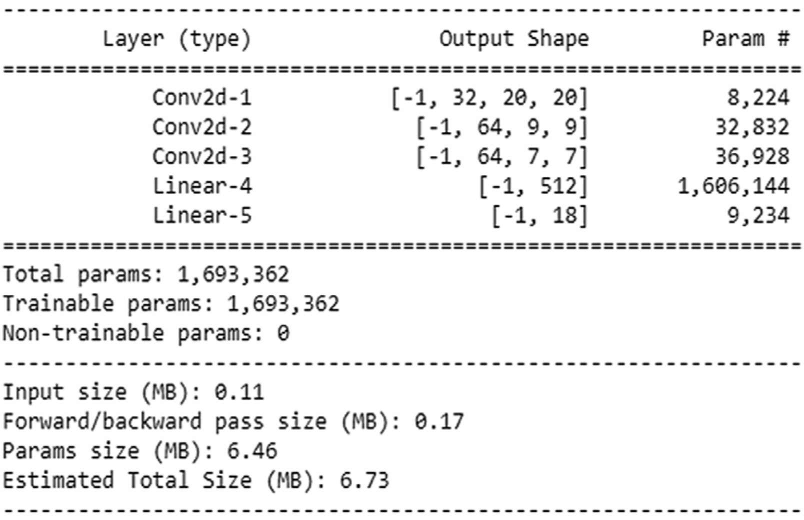
4.2. Models

Two deep neural networks are implemented from scratch for the conducted experiments.

4.2.1. Model 1

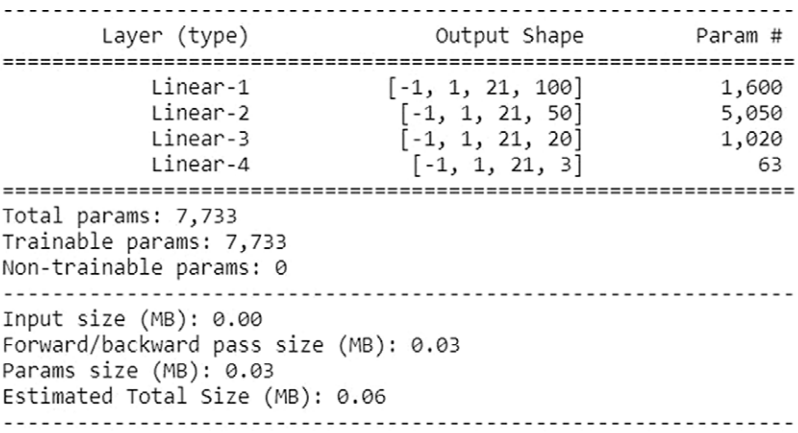
Model 1 is used as the neural network for all Atari 2600 games. This includes Experiments 1, 2, 3, and 4. The architecture of the model is identical to DQN (Mnih *et al*., 2016). The model takes the preprocessed training frame of size 84 × 84 × 4 as the input and outputs the Q-value for all possible actions for that game. There are three layers of convolution. The first convolution layer has 32 filters of size 8 × 8, which are applied with a stride of 4 and ReLU activation function. The second convolution layer has 64 filters of size 4 × 4, which are applied with a stride of 2 and ReLU activation. The last convolution layer has 64 filters of size 3 × 3, which are applied with a stride of 1 and ReLU activation. The penultimate layer is fully connected with an output of 512 neurons. The final output layer is a fully connected layer with the output neurons equivalent to the possible actions in the game. The architecture of the model is shown in Figure 18. The model consists of over 1.7M trainable parameters and has a size of about 6.75MB.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 20, 20]           8,224
            Conv2d-2             [-1, 64, 9, 9]          32,832
            Conv2d-3             [-1, 64, 7, 7]          36,928
            Linear-4                  [-1, 512]       1,606,144
            Linear-5                   [-1, 18]           9,234
================================================================
Total params: 1,693,362
Trainable params: 1,693,362
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.17
Params size (MB): 6.46
Estimated Total Size (MB): 6.73
----------------------------------------------------------------
```

Figure 18. *Neural network used for Atari 2600 games*



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1           [-1, 1, 21, 100]           1,600
            Linear-2            [-1, 1, 21, 50]           5,050
            Linear-3            [-1, 1, 21, 20]           1,020
            Linear-4             [-1, 1, 21, 3]              63
================================================================
Total params: 7,733
Trainable params: 7,733
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.03
Params size (MB): 0.03
Estimated Total Size (MB): 0.06
----------------------------------------------------------------
```

Figure 19. *Neural network used for RET environment*

4.2.2. Model 2

Model 2 is used as the neural network for the Remote Electrical Tilt (RET) simulator by Ericsson, that is, Experiment 5. It is a feed-forward network with four fully connected layers of size 100, 50, 20, and 3 respectively. The architecture of the model is shown in Figure 19. This model has over 7500 trainable parameters and a size of about 0.06MB.

4.3. Dataset

Gym (Brockman *et al.*, 2016) by OpenAI is a widely used open-source library with various environments to train and test you RL agents. Each environment is provided with the *classical* agent–action–environment loop, as shown in Figure 20.

The library has a wide variety of datasets ranging from classical control problems to complex 2D and 3D robotic manipulation tasks. It also exposes its registry and API as wrappers to adapt your custom environments, data, and models. This work relies on two such datasets—Atari 2600 games and a RET simulator developed by researchers at Ericsson as a gym wrapped environment.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

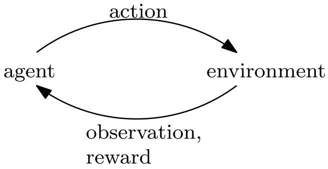


Figure 20. *The classical RL training loop*

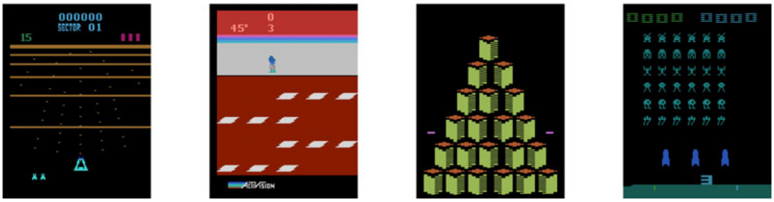


Figure 21. *Game frames from 4 Atari 2600 games—beamrider, frostbite, qbert, spaceinvader*

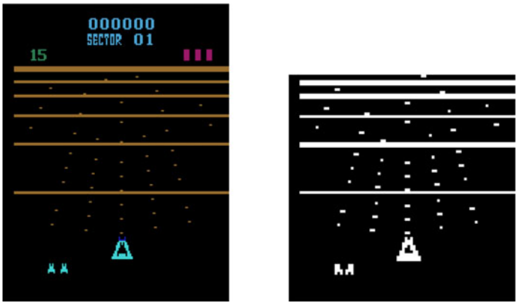


Figure 22. *Original game frame (left) and preprocessed frame (right) using the first three steps*

4.3.1. Atari 2600 games

Atari 2600 games, which are implemented in the Arcade Learning Environment (Bellemare *et al.*, 2013) are available on Gym. These problems are widely used for benchmarking the performance of RL algorithms (Hasselt, 2010; Mnih *et al.*, 2016; Wang et al. 2016; Salimans *et al.*, 2017; Conti *et al.*, 2018). Each game consists of a range of actions, and game scores as rewards. The state of each game is the pixel data from an image. This is also known as a *game frame*. Figures 21 and 22 show examples from the game frames from four Atari 2600 games.

Gym has a collection of over 50 games. Each game is implemented as an environment in the library. The agent is provided with the current state and a list of possible actions. Once an action is selected by the agent, it is given a reward (positive or negative) by the model and the next state of the environment. The interaction is repeated until the environment returns a flag indicating the termination of the game. The technical specifications of all the games utilized in this work are available in Appendix A.1.

Data Preprocessing: Most games, have an image frame of size 210 × 160 × 3 pixels which puts a heavy workload when training a NN, especially with several hidden layers. To reduce the dimensionality of inputs, some basic preprocessing steps are applied as suggested by researchers in the implementation of DQN (Mnih *et al.*, 2015).

1. The game frame is first converted to a grayscale.
2. The grayscale image is down sampled to the size 84 × 84.
3. The pixel values of the image are normalized between [0,1].
4. A training frame of size 84 × 84 × 4 is then produced by stacking 4 game frames together.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

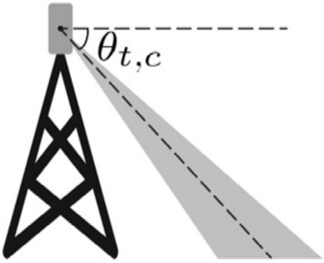


Figure 23. *Antennae downtilt, $\theta_{t,c}$ for cell c at time t. Source Vannella et al. (2021)*

4.3.2. Remote electric tilt simulator by Ericsson

A use case of RL in telecommunications is RET optimization. For a better consumer experience and improved (), networks rely on making some adjustments to their configurations remotely. Antenna downtilt is the angle of inclination between the horizontal plan the radiating beam of the antenna which is shown in Figure 23. RET optimization is a technique to adjust the downtilt as mentioned above to improve certain Key Performance Indicators (KPIs) defined by the network. This framework is well suited for applying RL, and prior work using DQN has shown safe and reliable policies (Vannella *et al*., 2021).

The state of the environment comprises factors like congestion, overlapping, interference, overshooting and Reference Signal Receive Power (RSRP)[6]. The actions include uptilt by 1 degree, downtilt by 1 degree and no tilt. After training, the simulated environment provides the overall improvement on four metrics, the Reward KPI, good traffic, bad traffic, and RRC congestion[7]. The details about metrics are proprietary, however, a positive value for each metric is an indicator of a good policy.

4.4. Evaluation metrics

The metrics used for evaluating the results from the experiments are summarized below:

Speedup: In parallel processing, this notion was established by Amdahl's argument (Amdahl, 1967) as the ratio of the time taken by the best sequential algorithm (fastest) (T_S^*) to the time taken by a parallel algorithm (T_p) for any computational problem. If the best sequential algorithm is not available or well defined, a common practice is to use the time taken by the parallel algorithm with 1 processor (T_1) as its surrogate.

$$S_p = \frac{T_S^*}{T_p} \approx \frac{T_1}{T_p} \quad \text{, p is the number of processors.} \tag{35}$$

Linear speedup, also referred to as *ideal speedup*, is obtained when $S_p = p$ suggesting that the algorithm is perfectly scalable, which is infeasible to achieve in reality.

Efficiency: In parallel processing, it is the ratio of the speedup achieved by the parallel implementation of the algorithm, S_p to the number of processors utilized, p.

$$E = \frac{S_p}{p} = \frac{T_S^*}{pT_p} \approx \frac{T_1}{pT_p} \tag{36}$$

A typical value lies in the range of 0 and 1. Algorithms running on a single processor and parallel algorithms with a linear speedup have an efficiency of 1.

Cumulative reward: The goal of any RL policy is to optimize the expected return. Given a fixed time horizon of **T**, the objective is to find a sequential decision policy π, that maximizes the expected return

[6] https://en.wikipedia.org/wiki/RSRP.

[7] https://en.wikipedia.org/wiki/Radio_Resource_Control.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 2. *Hyperparameters used to train Sp-GA on Atari 2600 games*

Hyperparameter	Value
Population size	200
Mutation power	0.002
Fraction of elites	10%
Mutation probability, ψ	0.75
No. of species	5

until the end of the time horizon. If we represent the sequence of rewards received after time t, by $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then the cumulative reward is simply:

$$G_t \doteq r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{37}$$

Sample Efficiency: For any RL algorithm, sample efficiency is a measure of its efficiency in utilizing the experience gained by the agent, that is, the amount of experience on an average that an algorithm needs to generate in an environment in order to reach a certain level of performance. An algorithm can be termed *sample efficient* if it uses each sample from its experience optimally to generate a policy that achieves a high performance.

4.5. Experiments

Experiment 1: Comparison to gradient-based methods

In this experiment, the performance of Sp-GA is compared with two gradient-based RL methods, DQN (Mnih *et al*., 2015), a sequential algorithm and A3C (Mnih *et al*., 2016), a distributed algorithm. The metrics used for evaluation are cumulative rewards and training time.

Each algorithm was trained on 6 Atari 2600 games—beamrider, assault, spaceinvader, qbert, frostbite, and amidar for a maximum of 5 million training frames. The hyperparameters for DQN and A3C were obtained from the benchmarked results of rllib library[8] and for Sp-GA, they are mentioned in Table 2. The same architecture (Model 1) and preprocessing was utilized. The training started with a random number (maximum upto 30) of no-op action similar to prior work (Mnih *et al*., 2015). A single episode for DQN, A3C, and iteration for Sp-GA was capped at a maximum of 10K training frames.

Experiment 2: Comparison to gradient-free methods

In this experiment, the performance of Sp-GA is compared with two gradient-free techniques that also use EC-based methods for training, a simple GA (Conti *et al*., 2018), and ES (Salimans *et al*., 2017). The metrics used for evaluation are cumulative rewards.

Each algorithm was trained on 16 Atari 2600 games (list available in Appendix A.1) for a maximum of 25 million training frames. The hyperparameters for ES were obtained from the benchmarked results of rllib library. GA was implemented as a special case of Sp-GA without crossover and species-based initialization. All other parameters for GA and Sp-GA were kept the same which are mentioned in Table 2. The same architecture (Model 1) and preprocessing was utilized across algorithms. A single episode for ES and iteration for GA, Sp-GA was capped at a maximum of 10K training frames.

Experiment 3: Scalability Assessment of Sp-GA

In this experiment, the scalability of distributed Sp-GA is assessed by evaluating the speedup and efficiency. The efficacy of encoding is also tested by comparing memory utilization.

This experiment was conducted on a single Atari 2600 game—spaceinvader. Sp-GA was trained for a maximum of 500 000 training frames on clusters with different number of workers. A single iteration

[8] https://github.com/ray-project/rl-experiments.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 3. *Hyperparameters used to train Sp-GA on RET environment*

Hyperparameter	Value
Population size	10
Mutation power	0.002
Fraction of elites	100%
Mutation probability, ψ	0.75
No. of species	5

step was capped at a maximum of 1000 training frames. The population size was reduced to 100; all other hyperparameters were kept the same as mentioned in Table 2.

Experiment 4: Comparison of sample efficiency

In this experiment, the sample efficiency of Sp-GA, A3C, DQN, ES, and GA are compared. The algorithm that consistently obtains higher rewards with the increasing size of training frames is considered as more sample efficient. The results are computed on 3 Atari 2600 games chosen at random.

This experiment was conducted on 3 Atari 2600 games—frostbite, qbert and spaceinvader. The same training pipelines from Experiments 1 and 2 were used. The average rewards from all the algorithms were logged during training up to 5 million frames.

Experiment 5: Performance evaluation on RET optimization

The performance of Sp-GA is evaluated on the RET simulator provided by the host organization (Section 4.3.2) and compared to DQN. The evaluation metrics are provided by the simulator.

This experiment was conducted on RET simulator provided by the host organization. The implementation and hyperparameters for DQN were preconfigured and are mentioned in Appendix 3. Both algorithms were trained for a fixed number of iterations of 1500. The model architecture was kept the same (Model 2). The hyperparameters used by Sp-GA are summarized in Table 3

5. Results

Experiment 1 : Comparison to gradient-based methods

Table 4 summarizes the cumulative rewards achieved by Sp-GA, DQN, and A3C. Sp-GA trains a population of 200 neural networks, whereas DQN and A3C train a single neural network on the same amount of training frames. The cumulative reward of the elite model from Sp-GA is compared to the maximum episodic reward from the gradient-based methods for a fair comparison. Sp-GA outperforms DQN on all the games and A3C on 5 out of 6 games. The cumulative rewards on these benchmarks are not directly comparable to the results in Mnih *et al*. (2016), Salimans *et al*. (2017), Conti *et al*. (2018) due to training size being limited to 5M frames, but they support their findings that gradient-free methods can perform surprisingly well on RL tasks.

Figure 25 shows the maximum episodic reward for DQN and A3C at each iteration for 5 million training frames. Figure 24 shows the elite model's cumulative reward and the entire population's average cumulative reward for each generation of Sp-GA. Two milestones have been highlighted in the figure—5 million training frames and 25 million training frames. With continued training, the performance of Sp-GA continues to improve in most of the games. The rate of improvement is lower for assault and beamrider, but it is still greater than DQN and A3C by a good margin. The decision of using the same hyperparameters across games might have contributed to this discrepancy. A fair comparison of all the algorithms in terms of training time is difficult due to the different number of CPUs utilized. However,

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 4. *Cumulative rewards achieved by DQN, A3C and Sp-GA on 6 Atari 2600 games*

	Sp-GA		A3C		DQN	
Workers	32 CPU		8 CPU		1 CPU	
Training time	0.9 hours		3.45 hours		9 hours	
Game	Elite model's reward	Population average	Max episodic reward	Mean episodic reward	Max episodic reward	Mean episodic reward
beamrider	**948.0**	643.38	708	395.64	660	336.16
assault	**817.0**	497.92	483	314.16	672	359.1
spaceinvaders	**925.0**	471.86	555	188.05	585	216.8
qbert	900.0	264.67	**1150**	226.75	650	270.25
frostbite	**2390.0**	277.21	240	98.59	180	71.19
amidar	**165.0**	45.81	141	40.45	144	46.2

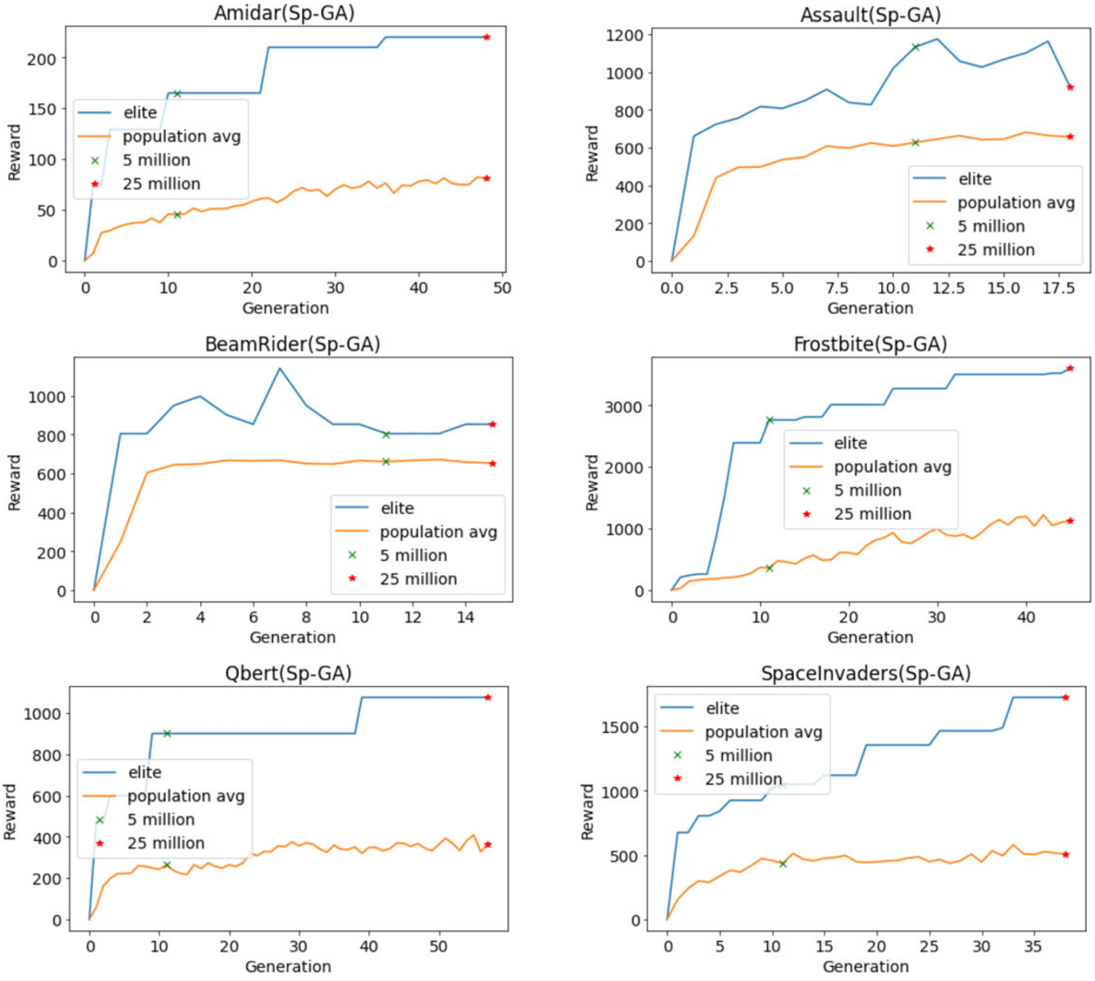


Figure 24. *Elite model's score and population average achieved by Sp-GA on Atari 2600 games for two milestones (5 million, 25 million training frames)*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

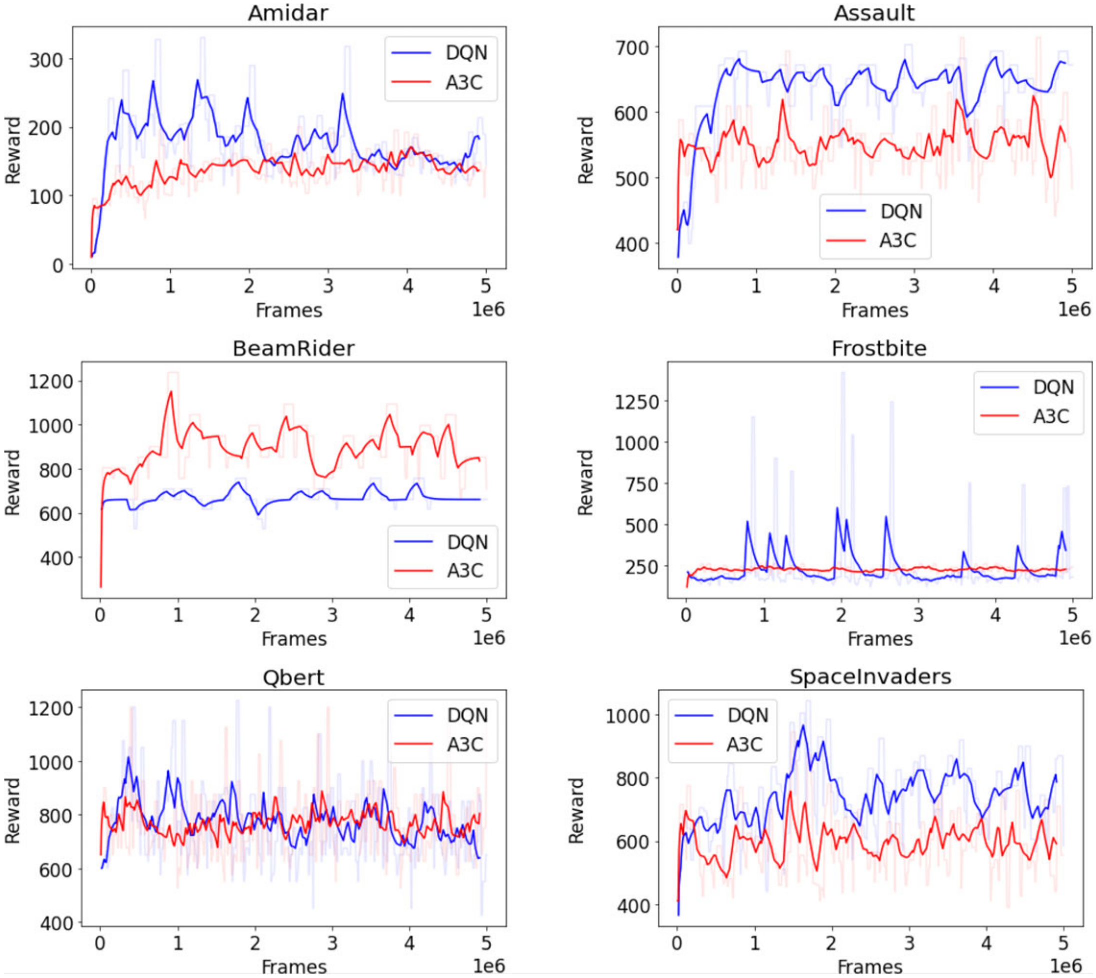


Figure 25. *Maximum episodic reward for DQN, A3C on Atari 2600 games for 5 million training frames. The transparent lines represent the actual value, and the solid ones represent the smoothed values*

the distributed Sp-GA, which is trained on a cluster of 32 CPUs is atleast 10 times faster than DQN and 4 times faster than A3C and trains 200 neural networks all at once.

These results corroborate the findings in Conti *et al*. (2018) that EC-based methods like GAs can provide comparable performance to gradient-based algorithms while reducing the overall training time by a significant amount.

Experiment 2 : Comparison to gradient-free methods

Table 5 summarizes the cumulative rewards achieved by Sp-GA, a simple GA, and ES on 16 Atari 2600 games, each trained for about 25 Million training frames which are equivalent to 100 million game frames. The simple GA is a special case of Sp-GA with a single species and no crossover operator. Since the library implementation of ES only provides the average episodic reward, we compare them to the population's average cumulative reward from Sp-GA and simple GA. The best performing algorithm for each game is highlighted in bold. Clearly, GA-based methods outperformed ES on 10 out of 16 games. Surprisingly, all of the algorithms scored a 0 on venture which is similar to the findings in Bellemare *et al.* (2013). This specific game is known to require long-term planning. Our proposed algorithm and

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 5. *Comparing the performance of Sp-GA with a simple genetic algorithm (GA) and evolution strategies (ES)*

Game	Sp-GA		GA		ES
	Elite's reward	Population average	Elite's reward	Population average	Episode average
assault	920.0	656.62	1340.0	**672.53**	NA*
asterix	1600.0	**1009.45**	1400.0	922.14	545.0
atlantis	64 700.0	23 770.15	61 100.0	21 893.53	**40 312.50**
zaxxon	7200.0	1030.35	7000.0	**1438.31**	1410.0
frostbite	3600.0	**1122.34**	2590.0	413.98	253.0
seaquest	700.0	380.60	760.0	285.57	**473.33**
beamRider	852.0	652.24	1092.0	**668.44**	650.0
asteroids	3220.0	1372.59	3220.0	1499.30	**2029.0**
amidar	220.0	81.05	247.0	**95.41**	NA*
skiing	–8965.0	**–9013.79**	–8083.0	–11 519.20	NA*
qbert	1075.0	363.43	1325.0	403.48	**755.0**
kangaroo	2000.0	**980.10**	2000.0	639.80	600.0
gravitar	850.0	211.20	700.0	190.80	**388.89**
spaceinvaders	1725.0	**505.82**	1280.0	424.68	417.0
venture	0.0	0.0	0.0	0.0	0.0
enduro	91.0	22.20	65.0	**23.15**	0.0

*Configuration for these Atari games are not implemented in the library.

ES doesn't involve a discount term when evaluation the cumulative reward which might have resulted in poor performance. Among GA-based methods, the elite models from Sp-GA outperformed simple GA on 9 out of 15 games (better model for each game is highlighted in green). The decision to use the same hyperparameters for all the games (and a single run being used) could have resulted in these results. The generational rewards for each game for all the algorithms are available in Appendix A.2. The library implementation of ES was twice as fast when compared to Sp-GA and simple GA which suggests some improvements might be needed in our implementation.

This experiment demonstrates how different variants of GAs can be adapted to train neural networks for RL problems while still maintaining comparable performance to other gradient-free algorithms (Salimans *et al*., 2017; Conti *et al*., 2018).

Experiment 3 : Scalability assessment of Sp-GA

Figure 26 shows the average time per generation and total training time, which includes the time it took for communication among the workers, as a function of the number of processors. The total time, as well as the average generation time, is seen decreasing up to 32 CPUs. When the number of CPUs was increased to 40, the time taken rather increases. Possible reasons for this might be, the communication time between the workers posing a bottleneck, or congestion between the workers since these experiments are sent as jobs to clusters.

Figure 27 shows the parallel speedup of Sp-GA as a function of the number of CPUs. The speedup increases as the number of processors utilized are increased. Since the training time for 40 CPUs was higher than 32 CPUs, the speedup did not increase. Figure 28 shows the efficiency of the Sp-GA as a function of processors used. The algorithm is only optimal on a single CPU. With the increasing number of processors, the efficiency is decreasing which can be attributed to the additional communication time and CPUs remaining ideal post the fitness calculation.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 6. *The number of generations, top elite's species and average size of encoding after training Sp-GA for 25M frames*

Game	Avg. size of encoding (in bytes)	No of generations	Elite's species
assault	271.96	19	xavier_normal
asterix	319.60	25	normal
atlantis	314.47	24	kaiming_uniform
zaxxon	348.94	30	xavier_normal
frostbite	453.89	46	xavier_uniform
seaquest	330.51	26	normal
beamrider	247.92	16	kaiming_uniform
asteroid	375.60	31	normal
amidar	481.19	49	xavier_uniform
skiing	623.96	58	uniform
qbert	553.99	58	kaiming_uniform
kangaroo	386.63	49	normal
gravitar	511.48	47	normal
spaceinvaders	425.67	39	normal
venture	215.08	16	uniform
enduro	220.70	14	xavier_normal

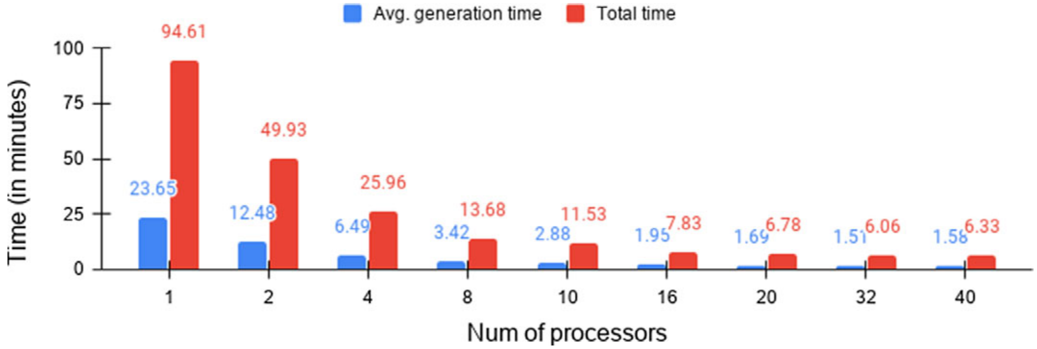


Figure 26. *Total training time (including time for communication between workers) and average time per generation as a function of the number of CPU's.*

Figure 18 shows the architecture of the model utilized for Experiments 1, 2, 3, and 4. Each neural network has a size of 6.73 MB. This becomes a significant memory and bandwidth bottleneck especially when the size of the population is increased. Table 6 shows the average size of an encoding of a neural network after training on 25M frames. The maximum size of an encoding which is ∼600 bytes is still 10 000 folds less than the original size. This shows the efficiency of the encoding implemented for this work.

The results in this experiment demonstrate that GAs if implemented efficiently, are well suited for a distributed training framework that results in a reduction of overall training time.

Experiment 4 : Comparison of sample efficiency

Figure 29 compares the performance of Sp-GA, simple GA, DQN, A3C, and ES on 3 Atari 2600 games till 5M training frames which is equivalent to 20M games frames as experienced by the agent. For DQN,

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

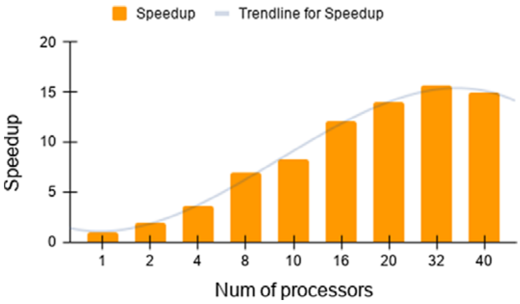


Figure 27. *Parallel speedup as a function of number of processors*

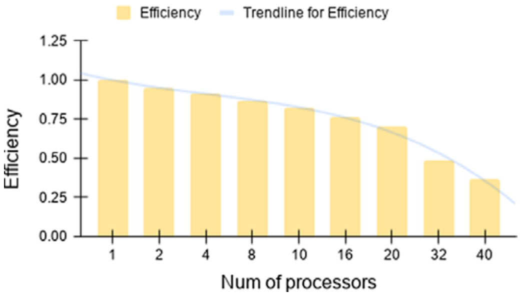


Figure 28. *Efficiency as a function of number of processors*

A3C and ES, the average episodic rewards (smoothed) and for Sp-GA and simple GA the population average rewards are reported for comparative analysis.

For the games frostbite and spaceinvader, Sp-GA is relatively more sample efficient in comparison to other methods, especially after experiencing about 1.5M training frames. For a fixed sample size of 5M frames, Sp-GA outperform all other algorithms in 2 out 3 games. DQN proved to be the most efficient for qbert. It must be noted that a simple GA had impressive results and was a close second in all games. This strengthens the fact the gradient-free methods can be sample efficient in comparison to gradient-based methods in some situations.

These results suggest that EC-based methods can be sample efficient in comparison to gradient-based methods despite training multiple neural networks on the same amount of experience.

Experiment 5 : Performance evaluation on RET optimization

Figure 30 shows the reward for DQN and the elite model of Sp-GA at each iteration during training. This reward is a function of proprietary metrics. The performance of Sp-GA is better than DQN during the initial stages of the training, however, DQN seems to perform better later on. The results of this experiment were reported from a single run which might have resulted in a bias. In Sp-GA, each worker creates a copy of the environment it trains on; in this experiment, the memory requirements of the environment and the interaction time became a bottleneck. To compensate for it, the size of the population was reduced significantly, which might have resulted in a slight worse performance.

It is interesting to see that although DQN outperformed Sp-GA overall, there was a good improvement in all the metrics and KPIs of the environment. The experiment was conducted for a fixed amount of training steps, and not for a fixed duration of training time. It is possible that if Sp-GA was let to train for the same duration as DQN (which was significantly more), we would have seen better results. This can be investigated in future work.The results are reported in Table 7. The plots for all the metrics for both algorithms are available in Appendix A.1

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

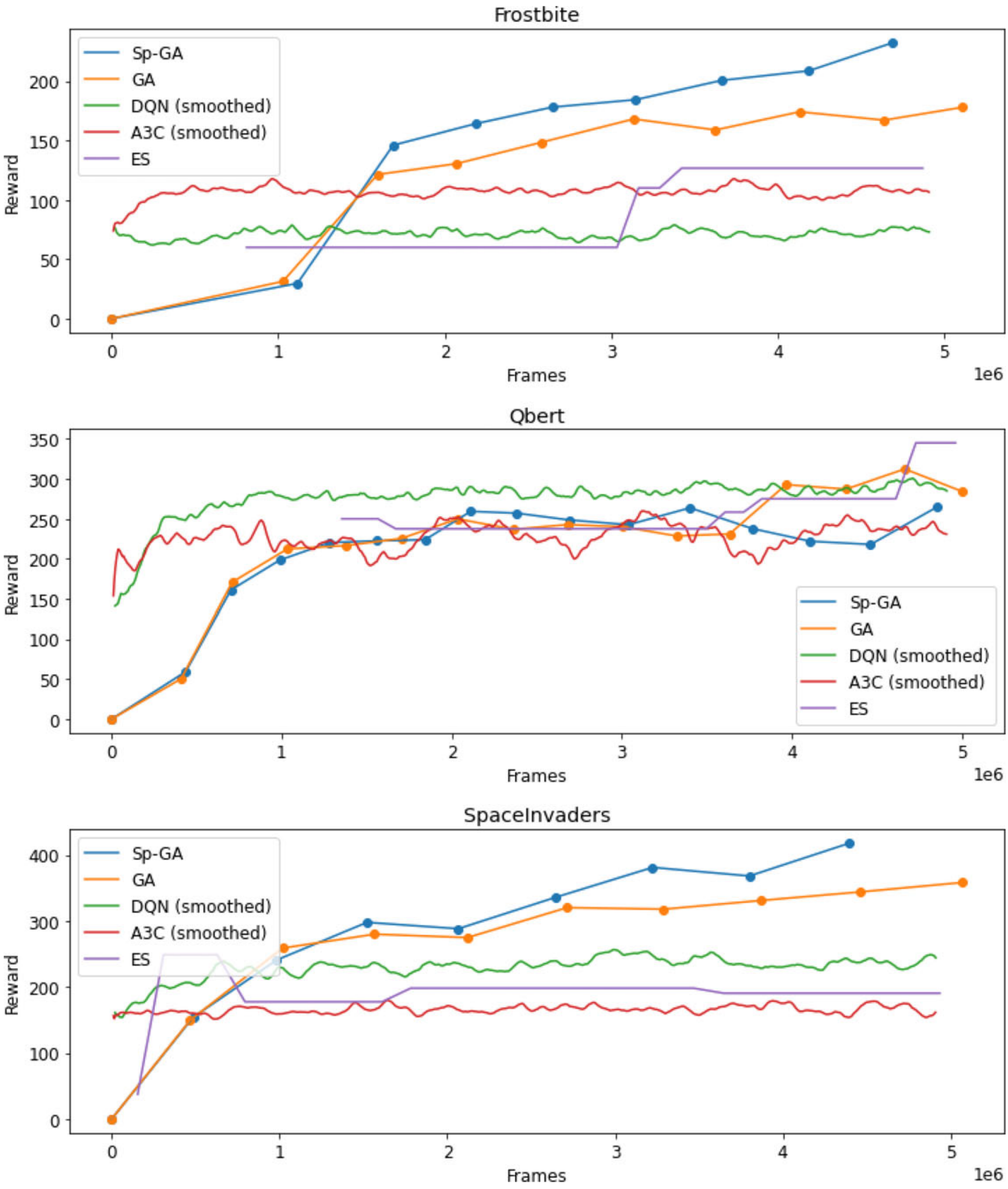


Figure 29. *Comparison of sample efficiency on Atari 2600 games*

These results suggest that GAs are also effective when applied to RL problems in different domains, including complex optimization tasks in telecommunication.

6. Discussion

The findings from this research outline the success of GAs and other Evolutionary Computation (EC)-based techniques as a scalable alternative to gradient-based algorithms for complex RL problems. Not

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table 7. *Average improvement of some KPI's provided by the environment with respect to a set baseline. A positive value is an indicator of a good policy*

Metric	Average Improvement (last 10 episodes)	
	DQN (%)	Sp-GA (%)
Reward KPI	11.52	8.08
Good traffic	26.01	17.98
Bad traffic	21.88	20.27
RRC congestion rate	3.04	3.57

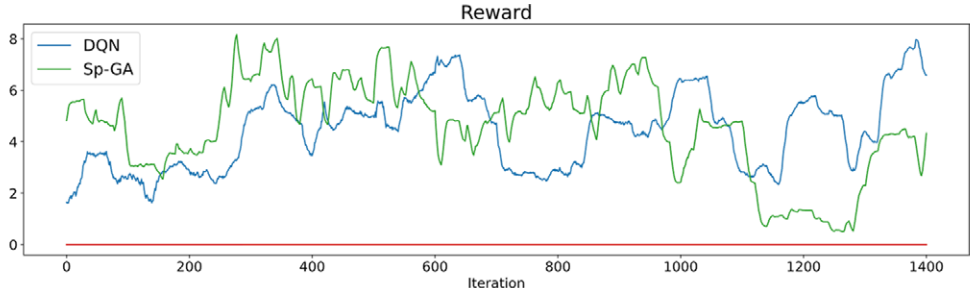


Figure 30. *Reward KPI from the RET environment at each episode of training for DQN and Sp-GA. The red line highlights zero improvement*

only did Sp-GA outperform DQN and A3C on several established benchmarks, it also reduced the overall training time by a big margin. A big advantage of the proposed method is its ability to constantly explore and learn from its experience irrespective of the density of rewards.

The proposed model encoding proved to be highly efficient and yielded significant improvements in memory utilization and overall training time. The intuitive application of genetic operators in the encoding space did not hinder the parallel implementation of the algorithm. Sp-GA and simple GA even outperformed ES, another EC-based method, on multiple benchmarks.

The speedup of the parallel implementation increased consistently upto 32 CPUs. Increasing the number to 40 somehow resulted in worse performance. Apart from the communication time being a bottleneck, a possible explanation could be a bug in the libraries used in the implementation. The topology of the workers and the communication patterns play an important role in the efficiency of any parallel algorithm especially in ML (Neglia *et al*., 2019). The work relied on ray for the set up of clusters and communication and this might have been a limiting factor.

It is well known that RL algorithms are sample inefficient when compared to supervised techniques (Yu, 2018), Sp-GA was sample efficient in a few cases, requiring fewer training samples to provide improved results. The successful implementation of Sp-GA for RET optimization and improvement in KPIs highlights that such gradient-free methods can also be applied to RL problems from other domains.

From these experiments, it can be concluded that repeated sampling around the region of *good solutions* can help discover *better solutions* without the need for estimating gradients. This hypothesis was also found out to be true by authors in Conti *et al*. (2018), Salimans *et al*. (2017). Although the results from our benchmarks are not directly comparable to the findings in Salimans *et al*. (2017), Conti *et al*. (2018), it was anticipated since the training was limited to only 25M frames on a subset of games due to computational constraints.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

6.1. Opportunities for improvement

For a few benchmarks, the performance did not improve as much as the others. The could be attributed to the decision of using the same parameters for all the experiments. Tuning these parameters for each benchmark by doing a grid search should result in an improved performance.

There were a few situations where introducing random perturbations had unexpected effects, this was seen in the game venture where all of the gradient-free algorithms scored 0. In the experiments, a constant amount of mutation was applied to all the networks; it would be interesting to see what would happen if this mutation is annealed, similar to annealing of a learning rate. Another possible improvement could be adapting the mutation inversely to the fitness of a solution, that is, making small perturbations to good solutions, whereas big perturbations to low-scoring ones. Utilizing the same parameters across experiments seems to have limited the performance in the experiments.

An unexpected result from the experiments was a simple GA without crossover outperforming Sp-GA on a few benchmarks. Crossover, which is well-known strategy to improve the performance of GA (Holland, 1992b) did not work as expected. A possible explanation for this might be its application in the space of encoding. Applying crossover, in such a way is a novel contribution which needs to investigated further as a part of future work.

7. Conclusions

In this work, we were successfully able to apply an evolutionary-based technique to train deep neural networks for several RL tasks from different domains. We have also demonstrated how different variants of GAs can be scaled and operators be applied in the encoding space.

The results from the experiments confirmed the findings from Salimans *et al*. (2017), Conti *et al*. (2018), that simple algorithms can perform surprisingly well on quite complex tasks. The final conclusions and insights to the research questions are summarized below:

1. EC-based RL algorithms like Sp-GA and simple GA can be applied to complex problems with a large state space. It is possible to outperform gradient-based as well gradient-free methods in terms of overall rewards and training time.
2. The scalability of the algorithm relies heavily on the efficacy of the model encoding. The proposed encoding worked pretty well which improved the parallel speedup and reduced the memory requirement significantly. A bottleneck, when the communication time between workers shadowed the evaluation time was observed when a large number of processors were utilized.
3. The improvements in sample efficiency were not significant and requires further investigation. A fair comparison of all the algorithms is difficult since the training procedures are inherently different. Gradient-based methods train a single NN where as population-based methods train multiple NN on the same amount of experience. Each NN from the population only gets to train on a fraction of the experience. Surprisingly, even with this issue, Sp-GA improves sample efficiency in a few benchmarks.
4. The algorithm is also successful in other domains. It must be noted, EC-based methods work on copies of environments in parallel. If these environments are very big, the algorithm cannot be scaled to its true potential. The concept of shared environments can be investigated to overcome this issue.

8. Future work

Future efforts will be devoted toward an investigative study on the effect of genetic operators in the encoding space. Our work was limited to Gaussian operator for mutation and single-point crossover. A plethora of strategies that improve the performance of GAs are available (Holland, 1992a) which can directly be applied to Sp-GA. Also, the training can be extended to the complete set of Atari 2600

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

benchmarks on 1B training frames for benchmarking the results against (Salimans *et al.*, 2017; Conti *et al.*, 2018).

The performance of Deep Neural Networks (DNNs) can be improved by utilizing techniques like dropout (Srivastava *et al.*, 2014), LSTM and GRU (Hochreiter & Schmidhuber, 1997; Cho *et al.*, 2014), residual networks (He *et al.*, 2015a), and scheduled annealing of parameters (Robbins & Monro, 1951). Such techniques can also be applied to the proposed algorithm. Lastly, a hybrid approach combing both evolutionary and MDP-based methods can be investigated.

References

Abbeel, P., Coates, A., Quigley, M. & Ng, A. 2006. An application of reinforcement learning to aerobatic helicopter flight. *In Advances in Neural Information Processing Systems*, 1–8.

Amdahl, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20,1967, Spring Joint Computer Conference*. AFIPS'67 (Spring). Atlantic City, New Jersey. Association for Computing Machinery, 483–485. ISBN:9781450378956. doi: 10.1145/1465482.1465560.

Baluja, S. & Caruana, R. 1995. Removing the genetics from the standard genetic algorithm. In *Proceedings of ICML'95*. Morgan Kaufmann Publishers, 38–46.

Barr, K. 2007. *ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits*. McGraw-Hill Education. ISBN:9780071481618. https://www.accessengineeringlibrary.com/content/book/9780071481618.

Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. 2013. The arcade learning environment: an evaluation platform for general agents. *Journal of Artificial Intelligence Research* **47**(1), 253–279. ISSN:1076-9757. doi: 10.1613/jair.3912.

Bellman, R. 1958. Dynamic programming and stochastic control processes. *Information and Control* **1**(3), 228–239. ISSN:0019-9958. https://doi.org/10.1016/S0019-9958(58)80003-0. https://www.sciencedirect.com/science/article/pii/S0019995858800030.

Bellman, R. E. 1954. *The Theory of Dynamic Programming*. RAND Corporation.

Bottou, L. 1998. *Online Learning and Stochastic Approximations*.

Boutilier, C., Dean, T. & Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *The Journal of Artificial Intelligence Research (JAIR)* **11**. doi: 10.1613/jair.575.

Brockman, B., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. 2016. *OpenAI Gym*. arXiv:1606.01540 [cs.LG].

Cho, K., et al. 2014. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. arXiv:1409.1259 [cs.CL].

Conti, E., Madhavan, V., Such F. P., Lehman, J., Stanley, K. O. & Clune, J. 2018. Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. In Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada (pp. 5032–5043).

Cully, A., et al. 2015. Robots that can adapt like animals. *Nature* **521**, 503–507. doi: 10.1038/nature14422.

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* **2**, 303–314.

Da Ronco, C. C. & Benini, E. 2014. A simplex-crossover-based multi-objective evolutionary algorithm. In Kim, H. K. et al. (eds), 583–598. doi: 10.1007/978-94-007-6818-5_41.

Darwin, C. 1859. *On the Origin of Species by Means of Natural Selection*. or the Preservation of Favored Races in the Struggle for Life. Murray.

Dong, Y. & Zou, X. 2020. Mobile robot path planning based on improved DDPG reinforcement learning algorithm. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 52–56. doi: 10.1109/ICSESS49938.2020.9237641.

Fitch, F. B. 1944. Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biophysics*, vol. 5 (1943), pp. 115–133. *Journal of Symbolic Logic* **9**(2), 49–50. doi: 10.2307/2268029.

Gangwani, T. & Peng, J. 2018. *Policy Optimization by Genetic Distillation*. arXiv:1711.01012 [stat.ML].

Glorot, X. & Bengio, Y. 2010. Understanding the difficulty of training deep feedforwardneural networks. *Journal of Machine Learning Research-Proceedings Track* **9**, 249–256.

Goodfellow, I., Bengio, Y. & Courville, A. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Hansen, N. & Auger, A. 2011. CMA-ES: evolution strategies and covariance matrix adaptation. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO'11, Dublin, Ireland. Association for Computing Machinery, 991–1010. ISBN:9781450306904. doi: 10.1145/2001858.2002123.

Hasselt, H. 2010. Double Q-learning. In *Advances in Neural Information Processing Systems*, Lafferty, J., et al., **23**. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

Haupt, S. & Haupt, R. 2003. Genetic algorithms and their applications in Environmental Sciences. 3rd Conference on Artificial Intelligence Applications to the Environmental Science. vol. 23. pp. 49–62.

He, K., et al. 2015a. *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 [cs.CV].

He, K., et al. 2015b. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *IEEE International Conference on Computer Vision (ICCV2015)*, 1502. doi: 10.1109/ICCV.2015.123.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Herculano-Houzel, S. 2009. The human brain in numbers: a linearly scaled-up primate brain. eng. In *Frontiers in Human Neuroscience* 3.PMC2776484[pmcid], 31–31. ISSN:16625161. doi: 10.3389/neuro.09.031.2009.

Hochreiter, S. & Schmidhuber, J. 1997. Long short-term memory. *Neural Computation* **9**(8), 1735–1780. ISSN:0899-7667. doi: 10.1162/neco.1997.9.8.1735.

Holland, J. H. 1992a. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* The MIT Press. ISBN:9780262275552. doi: 10.7551/mitpress/1090.001.0001.

Holland, J. H. 1992b. Genetic algorithms. *Scientific American* **267**(1). Full publication date: July 1992, 66–73. http://www.jstor.org/stable/24939139.

Jaderberg, M., et al. 2017. Population Based Training of Neural Networks. arXiv:1711.09846 [cs.LG].

Jouppi, N., et al. 2017. In-datacenter performance analysis of a tensor processing unit. *ACM SIGARCH Computer Architecture News* **45**, 1–12. doi: 10.1145/3140659.3080246.

Kalashnikov, D., et al. 2018. *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. arXiv:1806.10293 [cs.LG].

Kavalerov, M., Likhacheva, Y. & Shilova, Y. 2017. A reinforcement learning approach to network routing based on adaptive learning rates and route memory. In *SoutheastCon 2017,* 1–6. doi: 10.1109/SECON.2017.7925316.

Khadka, S., Majumdar, S., et al. 2019. Collaborative evolutionary reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning*, Chaudhuri, K. & Salakhutdinov, R. (eds), **97**. Proceedings of Machine Learning Research. PMLR, 3341–3350. https://proceedings.mlr.press/v97/khadka19a.html.

Khadka, S. & Tumer, K. 2018. *Evolution-Guided Policy Gradient in Reinforcement Learning*. arXiv:1805.07917 [cs.LG].

Kullback, S. & Leibler, R. A. 1951. On information and sufficiency. *Annals of Mathematical Statistics* **22**(1), 79–86.

Larranaga, P., et al. 1996. Learning Bayesian network structures by searching for the best ordering with genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* **26**(4), 487–493. doi: 10.1109/3468.508827.

Larranaga, P., et al. 1999. Genetic algorithms for the travelling salesman problem: a review of representations and operators. *Artificial Intelligence Review* **13**, 129–170. doi: 10.1023/A:1006529012972.

Lehman, J. & Stanley, K. 2008. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE.*

Lillicrap, T., et al. 2015. Continuous control with deep reinforcement learning. CoRR.

Liu, H., et al. 2017. Hierarchical representations for efficient architecture search. arXiv e-prints, arXiv:1711.00436 [cs.LG].

Liu, H., et al. 2018. *Hierarchical Representations for Efficient Architecture Search*. arXiv:1711.00436 [cs.LG].

Luong, N. C., et al. (2019). Applications of deep reinforcement learning in communications and networking: a survey. *IEEE Communications Surveys Tutorials* **21**(4), 3133–3174. doi: 10.1109/COMST.2019.2916583.

Ma, S., et al. 2020. Image and video compression with neural networks: a review. *IEEE Transactions on Circuits and Systems for Video Technology* **30**(6), 1683–1698. ISSN:15582205. doi: 10.1109/tcsvt.2019.2910119.

Mania, H., Guy, A. & Recht, B. 2018. *Simple random search provides a competitive approach to reinforcement learning*. arXiv:1803.07055 [cs.LG].

Markov, A. A. 1906. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. In *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete* **15**, 135156, 18.

Markov, A. A. 2006. An example of statistical investigation of the text Eugene Onegin concerning the connection of samples in chains. *Science in Context* **19**(4), 591–600. doi: 10.1017/S0269889706001074.

Melo, F. S., Meyn, S. P. & Ribeiro, M. I. 2008. An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th International Conference on Machine Learning.* ICML'08. Helsinki, Finland. Association for Computing Machinery, 664–671. ISBN:9781605582054. doi: 10.1145/1390156.1390240.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press. ISBN:0262133164.

Mnih, V., Badia, A. P., et al. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, Balcan, M. F. & Weinberger, K. Q. (eds), **48**. Proceedings of Machine Learning Research. New York, New York, USA, PMLR, 1928–1937. http://proceedings.mlr.press/v48/mniha16.html.

Mnih, V., Kavukcuoglu, K., et al. 2015. Human-level control through deep reinforcement learning. *Nature* **518**, 529–33. doi: 10.1038/nature14236.

Munemasa, I., et al. 2018. Deep reinforcement learning for recommender systems. In *2018 International Conference on Information and Communications Technology (ICOIACT)*, 226–233. doi: 10.1109/ICOIACT.2018.8350761.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S. & Legg, S., 2015. Massively parallel methods for deep reinforcement learning. arXiv:1507.04296.

Neglia, G., et al. 2019. The role of network topology for distributed machine learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, 2350–2358. doi: 10.1109/INFOCOM.2019.8737602.

Nikou, A., et al. to appear. Symbolic reinforcement laming for safe RAN control. In *International Conference of Autonomous Agents and Multi Agent Systems (AAMAS).*

Proshansky, H. & Murphy, G. 1942. The effects of reward and punishment on perception. *The Journal of Psychology: Interdisciplinary and Applied* **13**, 295–305. doi: 10.1080/00223980.1942.9917097.

Pugh, J., Soros, L. & Stanley, K. 2016. Quality diversity: a new frontier for evolutionary computation. *Frontiers in Robotics and AI* **3**. doi: 10.3389/frobt.2016.00040.

Purwins, H., et al. 2019. Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing* **13**(2), 206–219. ISSN:1941-0484. doi: 10.1109/jstsp.2019.2908700.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st edition. John Wiley & Sons, Inc. ISBN:0471619779.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. & Sutskever, I. (2019). Language models are unsupervised multitask learners. *In OpenAI blog* **1**(8), 9.

Rall, L. B. 1981. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science.

Robbins, H. & Monro, S. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* **22**(3), 400–407. doi: 10.1214/aoms/1177729586.

Rumelhart, D., Hinton, G. & Mcclelland, J. 1986. A general framework for parallel distributed processing. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1**.

Salimans, T., et al. 2017. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. arXiv:1703.03864 [stat.ML].

Schaul, T., Quan, J., Antonoglou, I. & Silver, D., 2015. Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Silver, D., Huang, A., et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489. doi: 10.1038/nature16961.

Silver, D., Hubert, T., et al. 2017. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. arXiv:1712.01815 [cs.AI].

Sousa, C., 2016. An overview on weight initialization methods for feedforward neural networks. doi: 10.1109/IJCNN.2016.7727180.

Srivastava, N., et al. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* **15**(56), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html.

Stanley, K. 2007. Compositional pattern producing networks: a novel abstraction of development. *In Genetic Programming and Evolvable Machines* **8**, 131–162. doi: 10.1007/s10710-007-9028-8.

Stanley, K., D'Ambrosio, D. & Gauci, J. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life* **15**, 185–212. doi: 10.1162/artl.2009.15.2.15202.

Stanley, K. O. & Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* **10**(2), 99–127. ISSN:1063-6560. doi: 10.1162/106365602320169811.

Strubell, E., Ganesh, A. & Mccallum, A. 2019. Energy and policy considerations for deep learning in NLP, 3645–3650. doi: 10.18653/v1/P19-1355.

Such, F. P., et al. 2018. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. arXiv:1712.06567 [cs.NE].

Sutton, R. S. & Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book. ISBN:0262039249.

Tassa, Y., Erez, T. & Todorov, E. 2012. Synthesis and stabilization of complex behaviors through online trajectory optimization, 4906–4913. ISBN:978-1-4673-1737-5. doi: 10.1109/IROS.2012.6386025.

Van Hasselt, H. 2013. Reinforcement Learning in Continuous State and Action Spaces. doi: 10.1007/978-3-642-27645-3_7.

van Hasselt, H., Guez, A. & Silver, D. 2016. Deep reinforcement learning with double Q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona. AAAI Press, 2094–2100.

Vannella, F., et al. 2021. *Remote Electrical Tilt Optimization via Safe Reinforcement Learning*. arXiv:2010.0584 2 [cs.LG].

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. & Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume* 48. ICML'16. New York, NY, USA. JMLR.org, 1995–2003.

Whiteson, S. 2012. Evolutionary computation for reinforcement learning. In *Reinforcement Learning: State of the Art*. doi: 10.1007/978-3-642-27645-3_10.

Wierstra, D., et al. 2008. Natural evolution strategies. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence),* 3381–3387. doi: 10.1109/CEC.2008.4631255.

Xu, K., et al. 2016. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. arXiv:1502.03044 [cs.LG].

Yajnanarayana, V., Ryden, H. & Hevizi, L. 2020. 5G handover using reinforcement learning. In *2020 IEEE 3rd 5G World Forum (5GWF)*. doi: 10.1109/5gwf49715.2020.9221072.

Yu, Y. 2018. Towards sample efficient reinforcement learning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. IJCAI'18. Stockholm, Sweden. AAAI Press, 5739–5743. ISBN:9780999241127.

Zheng, G., Zhang, F., Zheng, Z., Xiang, Y., Yuan, N.J., Xie, X. & Li, Z. 2018. DRN: a deep reinforcement learning framework for news recommendation. In *WWW'18: Proceedings of the 2018 World Wide Web Conference*, 167–176.

APPENDIX A Supplementary Information

Appendix A. Additional Technical Information

This section contains some additional technical information about the work. Table A.1 contains the technical specifications of the Atari 2600 games utilized in the work. Table A.2 provides the hyperparameter used for training DQN on RET simulator. Table A.3 gives the technical details of Experiment 4.

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Table A.1. *Specification of Atari 2600 games used for the experiments*

Game	Size per frame (state space)	Number of actions (action space)	Reward range
amidar	(250, 160, 3)	10	$(-\infty, \infty)$
assault	(250, 160, 3)	7	$(-\infty, \infty)$
asterix	(210, 160, 3)	9	$(-\infty, \infty)$
asteroid	(210, 160, 3)	14	$(-\infty, \infty)$
atlantis	(210, 160, 3)	4	$(-\infty, \infty)$
enduro	(210, 160, 3)	9	$(-\infty, \infty)$
frostbite	(210, 160, 3)	18	$(-\infty, \infty)$
gravitar	(210, 160, 3)	18	$(-\infty, \infty)$
kangaroo	(210, 160, 3)	18	$(-\infty, \infty)$
seaquest	(210, 160, 3)	18	$(-\infty, \infty)$
skiing	(250, 160, 3)	3	$(-\infty, \infty)$
venture	(210, 160, 3)	18	$(-\infty, \infty)$
zaxxon	(210, 160, 3)	18	$(-\infty, \infty)$
beamrider	(210, 160, 3)	9	$(-\infty, \infty)$
qbert	(210, 160, 3)	6	$(-\infty, \infty)$
spaceinvaders	(210, 160, 3)	6	$(-\infty, \infty)$

Table A.2. *Hyperparameters used to train DQN on RET environment*

Parameter	Value
ϵ	1
ϵ decay	0.997
Learning rate α	0.001
Episodes	1500
Memory size	50
Batch size	5

Table A.3. *Total training time (including communication between workers) and average time taken per generation as a function of number of CPU's*

Num of processors	Avg. generation time (in min)	Total time (in min)	Speedup	Efficiency
1	23.65	94.61	1	1
2	12.48	49.93	1.89	0.95
4	6.49	25.96	3.64	0.91
8	3.42	13.68	6.92	0.87
10	2.88	11.53	8.21	0.82
16	1.95	7.83	12.08	0.76
20	1.69	6.78	13.95	0.7
32	1.51	6.06	15.61	0.49
40	1.58	6.33	14.95	0.37

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

Appendix A. Additional Results

This section presents some additional results from the experiments.

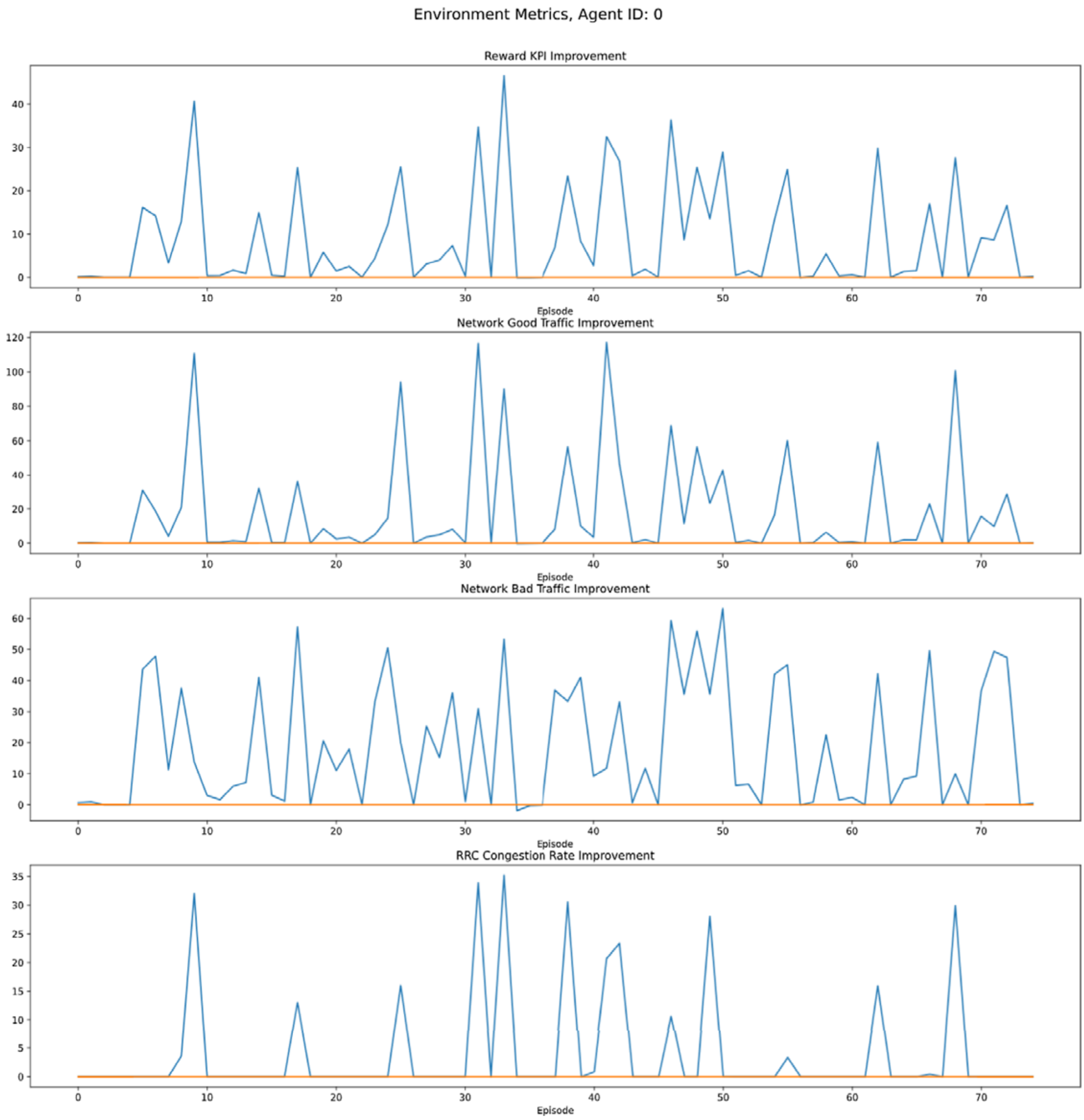


Figure A.1. *Metrics returned by RET environment at each episode during training for Sp-GA*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

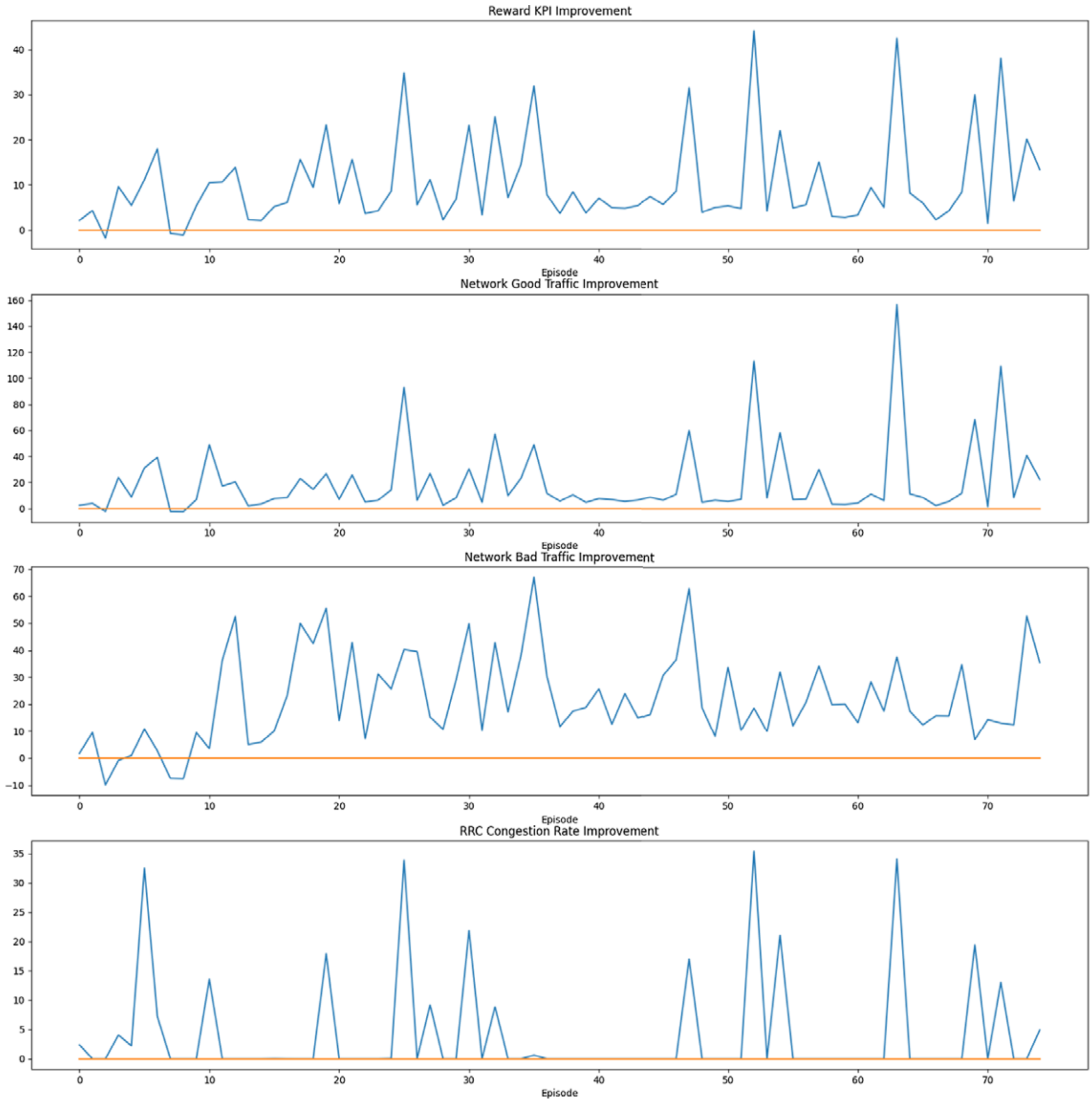


Figure A.2. *Metrics returned by RET environment at each episode during training for DQN*

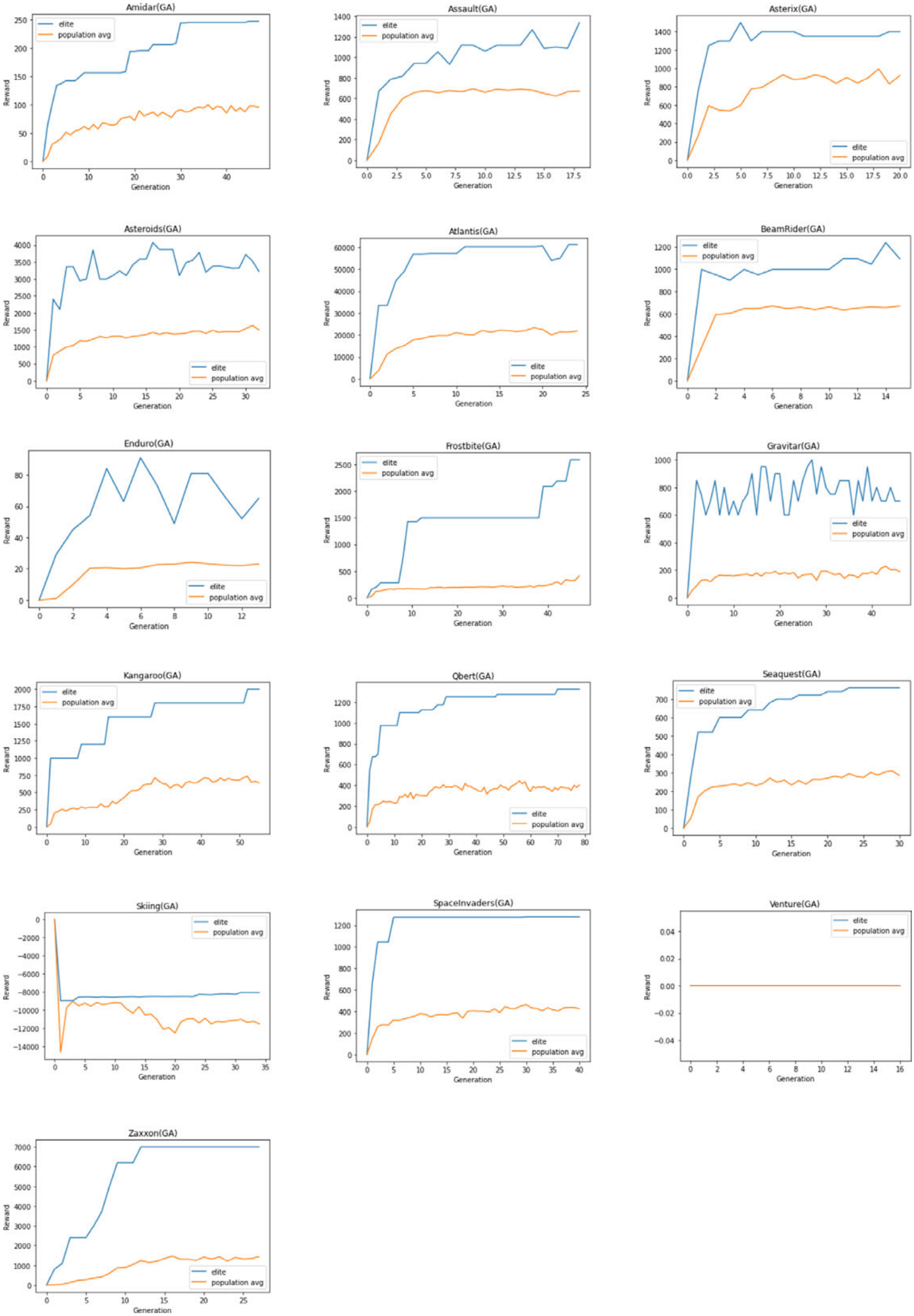


Figure A.3. *Elite model's score and population average achieved by GA on Atari 2600 games*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

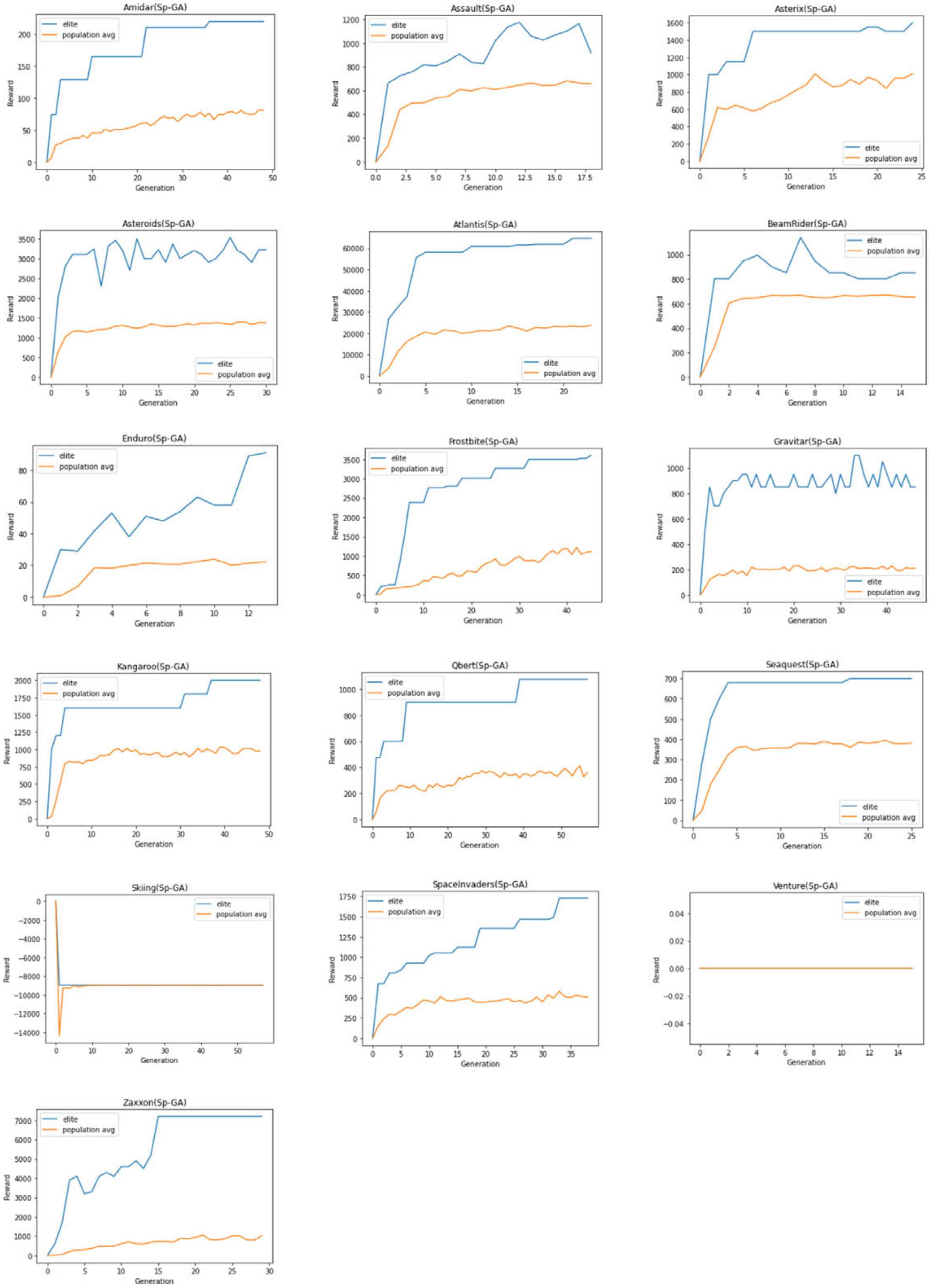


Figure A.4. *Elite model's score and population average achieved by Sp-GA on Atari 2600 games*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

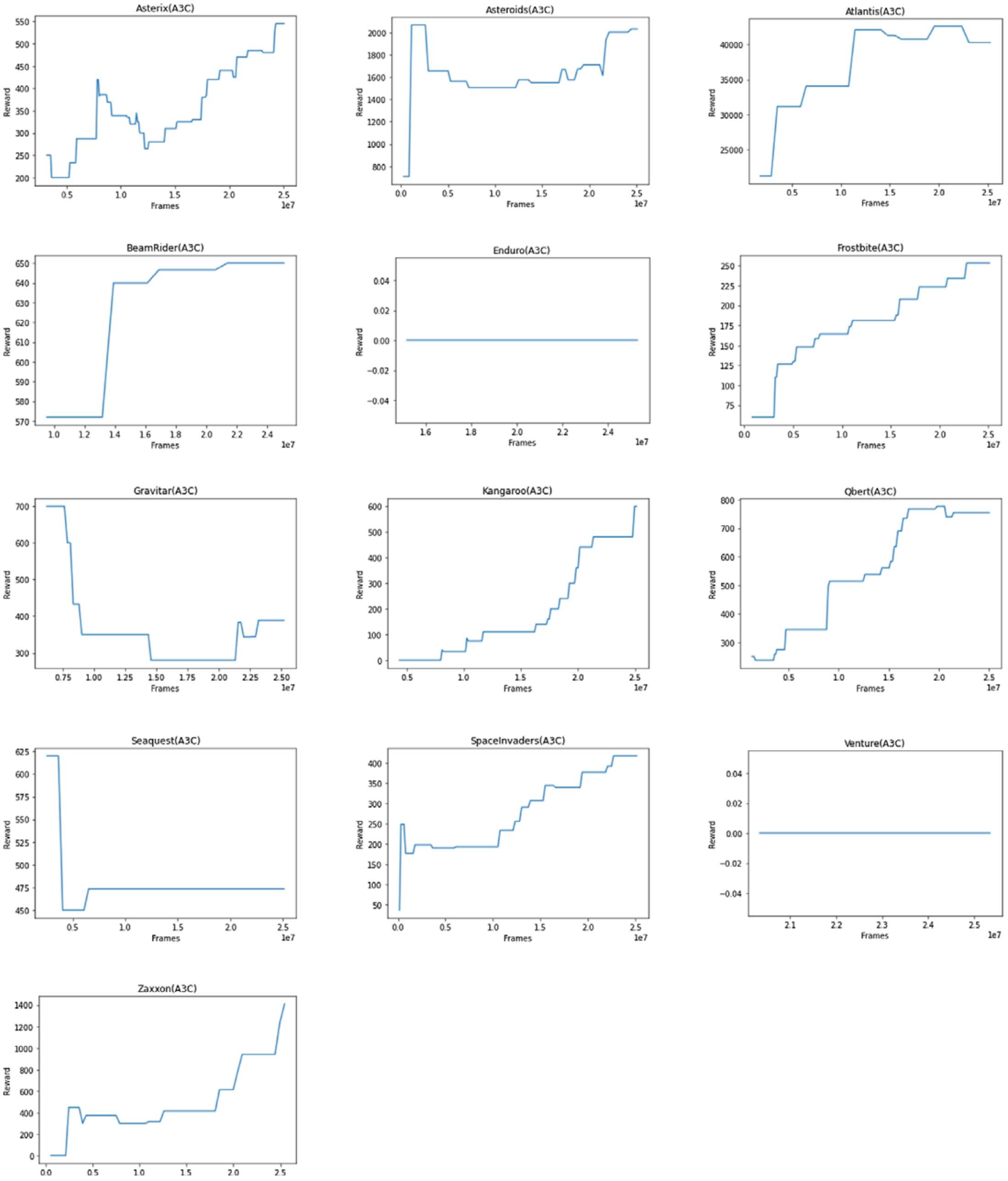


Figure A.5. *Average episodic reward achieved by ES on Atari 2600 games*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press

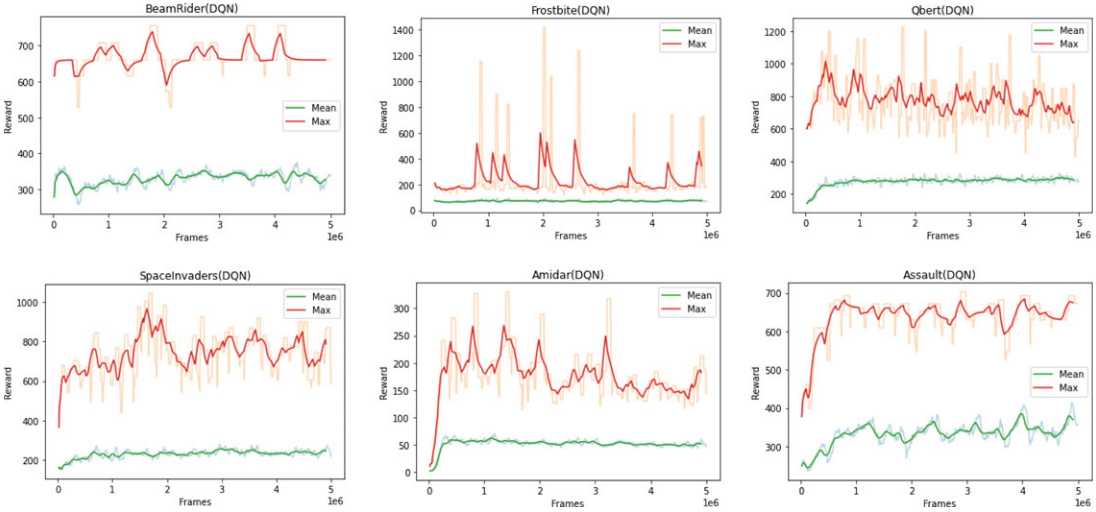


Figure A.6. *Maximum and average episodic reward achieved by DQN on Atari 2600 games*

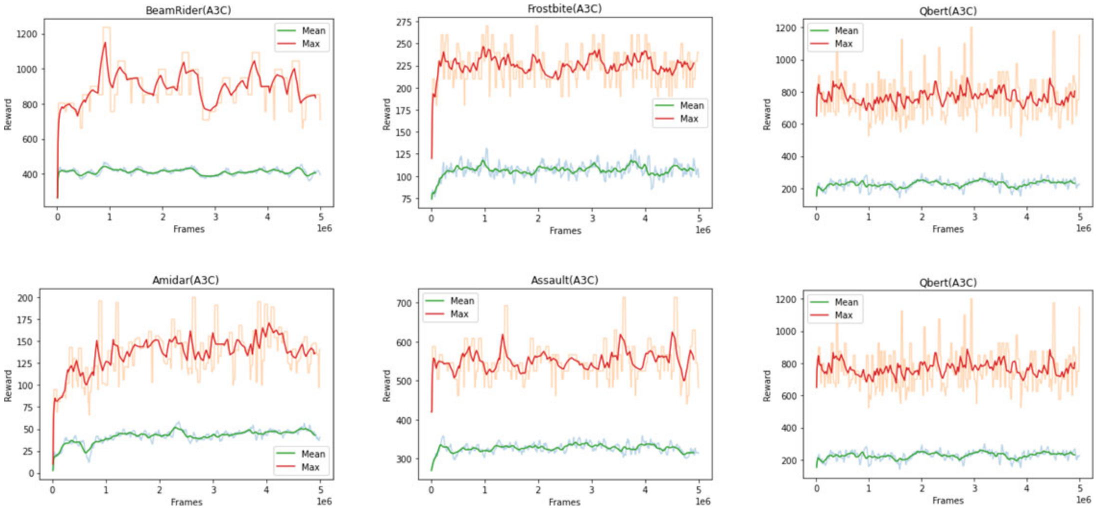


Figure A.7. *Maximum and average episodic reward achieved by A3C on Atari 2600 games*

https://doi.org/10.1017/S0269888922000042 Published online by Cambridge University Press