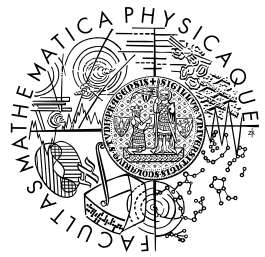

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Pospěch

Modern evolutionary strategies for reinforcement learning problems

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Roman Neruda, CSc.
Study programme:	Computer science
Study branch:	General computer science

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I'd like to thank my supervisor for his guidance and patience. Then I'd like to thank my friends for their support and advices. And last but not least I'd like to thank my family for their support and providing me with a roof over my head.

Title: Modern evolutionary strategies for reinforcement learning problems

Author: Michal Pospěch

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Evolutionary strategies are one of many approaches to solving reinforcement learning tasks. This thesis explores two modern approaches based on them, OpenAI-ES and NS-ES (and its extensions) which utilises novelty search. They are being studied on two environments, Cartpole-swingup and Slimevolley. On Cartpole-swingup they all have some success while the performance on Slimevolley is really sensitive to initial seed.

Keywords: evolutionary strategies reinforcement learning novelty search neuroevolution

Contents

Introduction

Reinforcement learning has been getting a lot of publicity in recent years, including competing with the best players in DotA 2 [1], playing Texas hold'em poker [2] or mastering Go, chess, shogi and other games [3]. Other than that reinforcement learning has been applied to many domains such as robot control [4], energy consupmtion optimisation [5], healthcare [6] or autonomous driving [7]. However most of the current research is focused on deep reinforcement learning while other potentially promising approaches are mostly ignored. This thesis focuses on exploring one of such approaches, evolutionary strategies, a subclass of evolutionary algorithms.

Evolutionary algorithms are an optimisation metaheuristic inspired by biological evolution. They utilise set of candidate solutions which is being periodically evaluated amd modified using genetic operators in order to improve results in the subsequent generation. They are applied to problems that are hard to solve using conventional methods, such as the traveling salesman problem. [8]

The method explored in this thesis is *OpenAI-ES* [9]. Contrary to classic reinforcement learning approaches it is easy to paralellise, scales very well and does not require differentiable policy. Other method explored is *NS-ES* and its variations *NSR-ES* and *NSRA-ES* [10] which utilise information about the agent's behaviour to direct the search to explore agents which behave in a different manner even at the cost of them potentially performing worse temporarily.

In the first part the task of reinforcement learning is formally introduced using Markov decision processes along with several methods for solving the task. After that, evolutionary algorithms are described and are subsequently extended with evolutionary strategies which are the basis of the methods explored in this thesis. Finally evolutionary strategies applied on reinforcement learning problems are explored along with novelty search. In the following chapter experiments are described along with descriptions of the test environments and their results are shown and discussed. In the penultimate chapter some details regarding technical implementation are shared and explained.

Chapter 1

Theoretical background

1.1 Reinforcement learning

The key problem that reinforcement learning is trying to solve is controlling an *agent* in an *environment*. The agent interacts with the environment by selecting some action and the environment responds by presenting the agent with a new situation. The environment also provides a numerical reward to the agent whose task is to maximise it over time. [11]

More formally, the agent interacts with the environment in discrete (if the problem has continuous time, it is discretised) time steps $t = 1, 2, 3, \ldots$. At each time step the agent recieves S_t from state-space $\mathcal{S}$, a representation of internal state of the environment. Based on that it selects an action A_t from action-space $mathcal A$. This results in the agent receiving a reward R_{t+1} from reward-space $\mathcal{R} \subset \mathbb{R}$ in addition to a new state S_{t+1} in the next step generating a sequence or *trajectory* beginning like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{1.1}$$

The random variables S_t and R_t depend only on the preceding state and action based on function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ characterising *dynamics* of the problem with following definition:

$$p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}. \tag{1.2}$$

This can, however, be viewed as an restriction on information contained in state rather than restriction on the environment. If the state does contain all information that influence the future then it has the *Markovian property* and the problem at hand is a *Markov decision process*.

To formally describe what is the goal of reinforcement learning *expected return* G_t needs to be defined first. It is a function of the reward sequence, in the

simplest case an ordinary sum of rewards

$$G_t = \sum_{i=t+1}^{T} R_i, \tag{1.3}$$

where T is the final time step.

This approach works only if the there is a final step in the interaction which would divide the interaction into subsequences called *episodes*. Each episodes ends when the environment reaches a *terminal state* and is reset into its initial state afterwards. Such tasks are called *episodic tasks*.

Contrary to that there are tasks which do not break into episodes, they continue indefinitely. They are called *continuing tasks* For these the definition 1.3 doesn't work as the sum could possibly be infinite. To calculate expected reward for such tasks a modified approach is needed. A *discount rate* $\gamma \in [0, 1]$ is introduced to define *discounted return*

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}. \tag{1.4}$$

The discount rate makes future reward worth less at the moment of decision. If $\gamma < 1$ and the rewards are bounded, then the infinite sum 1.4 is finite. For $\gamma = 0$ the agent is said to be "myopic" taking into account only the immediate reward and ignoring the future. Otherwise as γ approaches 1 the future rewards have bigger weight and influence the agent's decision more.

A common occurence in reinforcement learning is an estimating *value function* which estimates how beneficial it is for the agent to be in that state. This is defined by the expected return which is dependent on the actions that the agent will perform. Therefore a value function must be defined with respect to such way of acting called *policy*.

Policy is a function $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ returning probability of selecting each action in given state meaning that agent following policy π at time t would select action $A_t = a$ if in state $S_t = s$ with probability $\pi(a|s)$. The policy function is changed by reinforcement learning based on the agent's experience.

For state s and policy π the *state-value function* $v_\pi(s)$ gives the expected return when starting in state s and following policy π, formally (for MDPs)

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s], \tag{1.5}$$

where $\mathbb{E}_\pi[.]$ is expected value when following policy π in every timestep.

Action-value function $q_\pi(s, a)$ for policy π is defined similarly. It gives the expeted return of taking action a in state s while following policy π, formally

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{1.6}$$

1.1.1 Methods

Methods of reinforcement learning can be divided into 3 categories[12]:

- classical (tabular),
- policy gradient-based and
- evolutionary.

Classical methods rely on updating *Q-values* for each state-action pair therefore they require having discrete sets of actions and states. Based on the Q-values a policy is derived, such as an ε-greedy policy which (in training) chooses a random action with probability ε and the currently best action in all other cases. One of the most well known algorithms is *Q-learning*. In it the Q-values are updated each step via following formula:

$$q(s_t, a_t) = q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a q(s_{t-1}, a) - q(s_t, a_t) \right), \tag{1.7}$$

where α is the learning rate.

Algorithm 1 Q-Learning

parameters: learning rate $\alpha \in (0, 1]$
initialise: $q(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ arbitrarily, except for $q(terminal, \cdot) = 0$
for each episode **do**
    initialise $s \in \mathcal{S}$
    repeat
        choose a from s using policy derived from q
        take action a, observe s', r
        $q(s, a) = q(s, a) + \alpha \left(r + \gamma \max_a q(s', a) - q(s, a) \right)$
        $s = s'$
    until s is terminal state
end for

Policy-gradient based methods are methods that utilise the gradient in policy space. They are one of the few optimisation strategies able to handle reinforcement learning tasks that are high-dimensional and have continuous state and action space. One of the most known algorithms from this group is *REINFORCE*[11].

Another, not as known, class of policy-gradient based algorithms are PEPG (Parameter exploring policy gradient).[12] They use samples from parameter

Algorithm 2 REINFORCE

parameters: step size $\alpha > 0$
input: differentiable policy parametrisation $\pi(a|s, \theta)$
initialise: policy parameters θ atrbitrarily
for each episode **do**
    generate $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ by following $\pi(\cdot|\cdot, \theta)$
    for each timestep $t \in \{0, 1, \ldots, T-1\}$ **do**
        $G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\theta = \theta + \alpha \gamma^t G \nabla \ln \pi(A_t, S_t, \theta)$
    end for
end for

space to estimate the log-likelihood on parameter level. In traditional policy gradient methods the policy is probabilistic, it returns a distribution from which the next action is selected and final gradient is calculated by differentiating the policy with respect to parameters. This however causes high variance in samples over more episodes thus a noisy gradient estimate. PEPG circumvent this issue by having a probability distribution over policy parameters with a deterministic policy.

Evolutionary methods utilise only *fitness* describing the overall performance of the agent. Exploration is done via changing parameters that influence the agent's behaviour. However the basis of this class of algorithms is not derived from the mathematical principles of reinfocement learning. They are further described in following chapters.

1.2 Evolutionary algorithms

Evolutionary algorithms (EA) are a type of optimisation metaheuristics inspired by the process of bilogical evolution. [13] At first a number of possible solutions to the problem at hand is generated (*population*) and each solution (*individual*) is encoded (via a domain-specific encoding) and evaluated giving us the value of its *fitness*. Fitness is a function describing how good that particular individual is and it is then utilised in the process of selection. Then a new population is created using a *crossover* (combination) of 1 or more individuals which are selected using the operator of *parental selection*. Each of the newly created individuals has a chance to be mutated via the *mutation* operator. Finally a new population is selected from *offsprings* and possibly the parents based of fitness and enters the next iteration of the EA and the following generation is chosen using *environmental selection* operator. The algorithm repeats until the stop condition is

Algorithm 3 PGPE (Policy Gradients with Parameter-based Exploration) with symmetric sampling

parameters: step size $\alpha > 0$, number of histories n
initialise: μ, σ (both n dimensional) to preselected initial values, $m = 0$
repeat
    for $i \in 1, \ldots, n$ **do**
        sample $\epsilon^i \sim \mathcal{N}(0, I\sigma^2)$
        $\theta^+ = \mu + \epsilon^i$
        $\theta^- = \mu - \epsilon^i$
        evaluate policies $\pi(\cdot|\cdot, \theta^+)$ and $\pi(\cdot|\cdot, \theta^-)$ and get rewards r^{+i}, r^{-i}
    end for
    Set matrix T as $T_{ij} = \epsilon_i^j$
    Set matrix S as $S_{ij} = \frac{(\epsilon_i^j)^2 - \sigma_i^2}{\sigma_i}$
    $r_T = [(r^{+1} - r^{-1}), \ldots, (r^{+n} - r^{-n})]^T$
    $r_S = [\frac{(r^{+1} + r^{-1})}{2}, \ldots, \frac{(r^{+n} + r^{-n})}{2}]^T$
    Update $\mu = \mu + \alpha T r_T$
    Update $\sigma = \sigma + \alpha S r_S$
until stop criterion is fullfilled

met, usually a set number of iterations or small improvement of fitness between 2 generations.

There are many variands of EAs such as genetic algorithms (most common), genetic programming, evolutionary programming, neuroevolution or evolutionary strategies that are further described in following chapter.

1.3 Evolutionary strategies

Evolutionary strategies (ES) are a type of optimisation metaheuristic which further specialises EA and restricts their level of freedom. The selection for crossover is unbiased, mutation is parametrised and thus controllable, individuals which should be put to next generation are chosen ordinally based on fitness and individuals contain not only the problem solution but also control parameters.[14][15]

More formally ES $(\mu/\rho, \kappa, \lambda)$ has μ individuals in each generation, which produces λ offsprings, each created by crossover of ρ individuals and each individual is able to survive for up to κ generations as described in algorithm 5. This notation further generalizes the old (μ, λ) and $(\mu + \lambda)$ notations, where the "," notation means $\kappa = 1$ and "+" notation $\kappa = \infty$.

Algorithm 4 Evolutionary algorithm

initialize population P^0 with n individuals
set $t = 0$
repeat
    $Q^t = \{\}$
    for $i \in \{1 \dots m\}$ **do**
        $p_1, \dots, p_\rho = ParentalSelection(P^t)$
        $q = Crossover(p_1, \dots, p_\rho)$
        $q = Mutation(q)$ with chance p
        $Q^t = q \cup Q^t$
    end for
    $P^{t+1} = EnvironmentalSelection(Q^t \cup P^t)$
    increment t
until stop criterion fullfilled

To design an ES one must first select an appropriate representation for an individual and the most natural one is prefered in most cases, if all parameters are of one type (e.g. a real number) a simple vector will suffice, if the types are mixed, a tuple of vectors is required. This however causes an increased complexity of the variation operator.

As for design of the variation operator there are some guidelines that should be followed when designing it.

Reachability every solution should be reachable from any other solution in a finite number of applications of the variation operator with probability $p > 0$

Unbiasedness the operator should not favour any particular subset of solution unless provided with information about problem at hand

Control the operator should be parametrised in such way that the size of the distribution can be controlled (practice had shown that decreasing it as the optimal solution is being approached is necessary)

A big part of designing efficient evolutionary strategy algorithms is adapting the covariance matrix of the used multivariate normal distribution that is commonly used as variation operator. It is assumed that setting the covariance matrix Σ of the distribution proportional to the inverse Hessian matrix of Taylor expansion of the fitness function. This would align the hyperellipsoid of equal probabilities of the mutation distribution with the hyperellipsoid of equal fitness values.

Algorithm 5 $(\mu/\rho, \kappa, \lambda)$-ES

initialize population P^0 with μ individuals
set age for each $p \in P^0$ to 1
set $t = 0$
repeat
    $Q^t = \{\}$
    for $i \in \{1 \dots \lambda\}$ **do**
        select ρ parents $p_1, \dots, p_\rho \in P^t$ uniformly at random
        $q = variation(p_1, \dots, p_\rho)$ with age 0
        $Q^t = q \cup Q^t$
    end for
    P^{t+1} = select μ best (wrt. fitness) individuals from $Q^t \cup \{p \in P^t : age(p) < \kappa\}$
    increment age by 1 for each $p \in P^{t+1}$
    increment t
until stop criterion fullfilled

1.3.1 CMA-ES

The aforementioned idea is used in Covariance Matrix Adaptation Evolutionary Strategy algorithm.[16] In it the population of new individuals is generated by sampling a multivariate normal distribution. The individuals are sampled via following equation for generation $t \in \mathrm{N}_0$:

$$x_k^{t+1} \sim m^t + \sigma^t \mathcal{N}(0, C^t),\ k \in \{1, 2, \dots, \lambda\}, \tag{1.8}$$

where

x_k^{t+1} is k-th individual from generation $t + 1$,

m^t is mean value of the search distribution at generation t,

σ^t is the step size at generation t,

C^t is the covariance matrix at generation t and

λ is the population size.

At each step the mean is moved to the weighted average of μ selected individuals (parents) from the current generation. The individuals are selected based on their fitness and the weights are parameters of the algorithm. Then the covariance matrix C^t is updated using the covariance matrix from previous generation, the evolution steps and the new individuals. To control the step size σ^t only information from evolution steps is used.

Algorithm 6 CMA-ES

parameters: $\lambda, w_{i=1\ldots\lambda}, c_\sigma, d_\sigma, c_c, c_\mu$ (set according to Table 1 in [16])
initialise: $p_\sigma = 0, p_c = 0$, covariance matrix $C = I$, $g = 0$, distribution mean m and $\sigma > 0$ step-size depending on the problem
set $t = 0$
repeat
    increment t
    calculate matrices B, D such that $C = BDDB^T$ (spectral decomposition)
    for $k = 1, \ldots, \lambda$ **do** ▷ sample new individuals
        $z_k \sim \mathcal{N}(0, I)$
        $y_k = BDz_k$
        $x_k = m + \sigma y_k$
        evaluate x_k
    end for
    $\langle y_t \rangle_w = \sum_{i=1}^{\mu} w_i y_{y:\lambda}$ ▷ $y_{i:\lambda}$ is i-th best performing individual
    $m = m + c_m \sigma \langle y \rangle_w$
    $p_\sigma = (1 - c_\sigma) p_\sigma + \sqrt{c_\sigma (2 - c_c) \mu_{\text{eff}}} C^{-\frac{1}{2}} \langle y \rangle_w$ ▷ $\mu_{\text{eff}} = (\sum_{i=1}^{\lambda} w_i^2)^{-1}$
    $\sigma = \sigma \exp(\frac{c_\sigma}{d_\sigma}(\frac{\|p_\sigma\|}{\mathbb{E}\|\mathcal{N}(0,I)\|} - 1))$
    $w_i' = w_i(1 \text{ if } w_i > 0 \text{ else } n/\|C^{-\frac{1}{2}} y_{i:\lambda}\|^2)$
    $C = (1 + c_1 \delta(h_\sigma) - c_1 - c_\mu \sum_{i=1}^{\lambda} w_i) C + c_1 p_c p_c^T + c_\mu \sum_{i=1}^{\lambda} w_i' y_{i:\lambda} y_{i:\lambda}^T$
until stop criterion fullfilled

Chapter 2

Evolutionary strategies in reinforcement learning

Black-box optimisation is an alternative approach to solving RL tasks also known as Direct policy search or neurevolution when applied to neural networks. It has several attractive properties such as indifferenco to distribution of rewards, no need for backpropagation and tolerance of arbitrarily long episodes.[9]

Compared to reinfocement learning using evolutionary strategies has the advantage of not needing a gradient of the policy performance. Also as the state transition function is not known the gradient can't be computed using backpropagation-like algorithm. Thus some noise needs to be added to make the problem smooth and the gradient to be estimable. Here is where reinfocement learning and evolutionary strategies differ, reinfocement learning adds noise in the action space (actions are chosen from a distribution) while evolutionary strategies add noise in the parameter space (parameters perturbed while actions are deterministic).

Not requiring backpropagation has several advantages over other RL methods. First the amount of computation necessary for one episode of ES is much lower (about one third, potentially even less for memory usage). Not calculating gradient using analytic methods also protects these methods from suffering from *exploding gradient* which is a common issue with recurrent neural networks. And last, the network can contain elements that are not differentiable such as hard attention.

Also ES are easily paralellisable. First, contrary to RL, where value function is inherently linear procedure and has to be performed more times to improve a given policy. Furthermore it operates on whole whole episodes, therefore it does not require frequent commuincation between workers. And finally, as the only information received by each worker is the reward, it is possible to communicate only the reward inbetween workers. That, however, requires synchronising

seeds known to other workers beforehand and recreating perturbations based on that information. Thus the required bandwidth is extremely low comapred to communication of whole gradients which would be required for paralellisation of a policy gradient based algorithm.

2.1 OpenAI Evolutionary Strategy

These ideas are explored in OpenAI-ES algorithm. The agent acting in the environment is represented be a policy π_θ with parameter vector θ and $f(\cdot)$ is the reward returned by the environment. As we need to introduce some noise, the population is distribution p_θ is instantiated as an an isotropic multivariate Gaussian with mean ψ and covariance $\sigma^2 I$. Then $\mathbb{E}_{\theta \sim p_\psi} f(\theta) = \mathbb{E}_{\epsilon \sim N(0,I)} f(\theta + \sigma\epsilon)$. Thus the gradient approximation is calculated as follows:

$$\nabla_\theta \mathbb{E}_{\theta \sim p_\psi} f(\theta) = \nabla_\theta \mathbb{E}_{\epsilon \sim N(0,I)} f(\theta + \sigma\epsilon) \approx \frac{1}{n\sigma} \sum_{i=1}^{n} f(\theta_i)\epsilon_i, \tag{2.1}$$

where $\theta_i = \theta + \sigma\epsilon_i, \epsilon_i \sim N(0, I)$ and n is the population size.

The resulting algorithm uses SGD (or Stochastic Gradient Ascent - SGA in this case) or other gradient based optimisation technique for parameter vector θ update.

As the ES could be seen as method for computing a derivative estimate using finite differences in randomly chosen direction it would suggest that it would scale poorly with dimensions of parameters θ same as the finite differences method. In theory the number of necessary optimisation steps should scale linearly with the dimension. That however doesn't mean that larger networks optimised using ES will perform worse than smaller ones, that depends on the difficulty (intrinsic dimension) of the problem. The network will perform the same however it will take more optimisation steps to do so.

In practice ES performs slightly better on larger networks and it is hypothesised that it is for the same reason as why it is easier to optimise large networks using standard gradient based methods: larger networks have fewer local minima.

Due to perturbing the parameters and not the actions ES are invariant to the frequency at which the agent acts in the environment. Tradtional MDP-based reinforcement learning methods rely on *frameskip* as one their parameters that is crucial to get right for the optimisation to be successful. While this is solvable for problems that do not require long term planning and actions, long term strategic behaviour poses a challenge and reinfocement learning needs hiearchy to be succesful unlike evolutionary strategy.

Algorithm 7 OpenAI-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, n number of workers
initialize n workers with known random seeds, and initial parameters θ_0
set $t = 0$
repeat
    for $i \in \{1 \dots n\}$ **do**
        sample $\epsilon_i \sim \mathcal{N}(0, I)$
        compute returns $f_i = f(\theta_t + \sigma\epsilon_i)$
    end for
    Send all scalar returns f_i from each worker to every other worker
    for $i \in \{1 \dots n\}$ **do**
        reconstruct all perturbations ϵ_j for $j \in \{1, \dots, n\}$ using known random seeds
        set $\theta_{t+1} = \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} f_i \epsilon_i$
    end for
    increment t
until stop criterion fullfilled

2.2 Novelty search

While the main drive in of improvement in ES is the value of fitness (how "good" the result is), novelty search takes a different approach. Novelty search is focused on finding different solutions, as it is inspired by nature's drive towards diversity. Each policy has its novelty calculated with respect to previous policies and search is directed to parts of search space with high novelty. This approach makes it less succeptible to local optima created by deceptive rewards than reward-based method. [10]

Each policy π gets assigned its domain-dependent behavorial characteristics $b(\pi)$ (e.g. final position of the agent) and it is added to an archive set A of characteristics of previous policies. Then the novelty $N(b(\pi_\theta), A)$ is calculated as average distance from k nearest neighbours from the archive set A.

$$\begin{gathered} N(\theta, A) = N(b(\pi_\theta), A) = \frac{1}{|S|} \sum_{j \in S} \|(\pi_\theta) - b(\pi_j)\|_2 \\ S = kNN(b(\pi_\theta), A) \end{gathered} \tag{2.2}$$

2.2.1 Combination with evolution strategies

To find and follow the gradient of expected novelty with respected to θ^t we use the framework outlined in 2.1.

With archive A and sampled parameters $\theta_t^i = \theta_t + \sigma\epsilon_i$, the gradient estimate can be calculated via following formula:

$$\nabla_{\theta_t}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}[N(\theta_t + \sigma\epsilon)|A] \approx \frac{1}{n\sigma}\sum_{i=1}^{n} N(\theta_t^i, A)\epsilon_i \tag{2.3}$$

It is possible because archive A is fixed during one iteration and is updated only at the end. Only characteristics corresponding to each θ^t are added to A, as adding each sampled would cause the archive A to inflate too much increasing the complexity of calculation of nearest-neighbours.

To encourage additional diversity an initial meta-population of M, selection of M is domain dependent, agents is created. While it is possible to optimise the behaviour of a single agent and reward it for behaving differently than its ancestors, this way we get the benefits of population-based exploration (each agent can become expert at some particular subtask the environment presents amd can be further combined, more approaches can be explored and only the promising ones can be explored further). Each agent has parameters θ^m and is being rewarded for behaviour different from all prior agents, thus we get M differently behaving policies.

M random parameter vectors are initialised and in each iteration one is selected to be updated. The selection probability is proportional to its novelty.

$$P(\theta^m) = \frac{N(\theta^m, A)}{\sum_{i=1}^{M} N(\theta^i, A)} \tag{2.4}$$

To perform the update step, we need to calculate the gradient estimate of expected novelty with respect to θ_t^m using equation 2.3 where n is the number of perturbations. When we get the gradient estimate we use SGD (or in this case Stochastic Gradient Ascent) with learning rate α to update the parameters θ^m

$$\theta_{t+1}^m := \theta^m + \alpha\frac{1}{n\sigma}\sum_{i=1}^{n} N(\theta_t^{i,m}, A)\epsilon_i. \tag{2.5}$$

After updating the individual, a new behavorial characteristics $b(\pi_{\theta_{t+1}^m})$ is calculated and added to the archive A.

This process is repeated for a predetermined number of times as novelty search is not supposed to converge to a "best" solution and returns the best performing policy which is being preserved during the run of the algorithm.

Algorithm 8 NS-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, n number of workers, T number of iterations
initialize n workers with known random seeds, empty archive A, and initial parameters $\{\theta_0^1, \dots, \theta_0^M\}$
set $t = 0$
for $i \in \{1 \dots M\}$ **do**
    calculate $b(\pi_{\theta_0^i})$
    add $b(\pi_{\theta_0^i})$ to A
end for
for $t \in \{0 \dots T-1\}$ **do**
    sample θ_t^m from $\{\theta_t^1, \dots, \theta_t^M\}$ based on equation 2.4
    for $i \in \{1 \dots n\}$ **do**
        sample $\epsilon_i \sim \mathcal{N}(0, I)$
        compute characteristics $b(\theta_t^m + \sigma\epsilon_i)$
        compute $n_i = N(b(\theta_t^m + \sigma\epsilon_i), A)$
    end for
    Send all novelties n_i from each worker to every other worker
    for $i \in \{1 \dots n\}$ **do**
        reconstruct all perturbations ϵ_j for $j \in \{1, \dots, n\}$ using known random seeds
        set $\theta_{t+1}^m = \theta_t^m + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} n_i \epsilon_i$
        add $b(\theta_{t+1}^m)$ to A
    end for
end for

2.2.2 Combination with reward based exploration

While *NS-ES* helps agents avoid local optima and deceptive reward signals, it also completly discards reward which might cause the performance to suffer. Therefore NSR-ES, an improved version of NS-ES, uses both reward (fitness) and novelty for computation of the update step. NSR-ES is in many ways similar to NS-ES, it calculates both novelty and reward at once and it operates on entire episodes. The only difference is, that the calculation of the gradient estimate is based on the average of reward and novelty.

Specifically, for parameter vector $\theta_t^{m,i} = \theta_t^m + \sigma\epsilon_i$ we calculate the reward $f(\theta_t^{m,i})$ and novelty $N(\theta_t^{m,i}, A)$, rank-normalise both values independently (as both values usually have completely different scales), calculate the average and set it as weight for corresponding ϵ_i for gradient estimation. Then the estimated gradient is used to update the parameter vector via SGD (or other gradient based

optimisation method) simirally as in equation 2.5:

$$\theta_{t+1}^{m} \coloneqq \theta^{m} + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} \frac{N(\theta_t^{i,m}, A) + f(\theta_t^{i,m})}{2} \epsilon_i. \tag{2.6}$$

Intuitively, following the approximated gradient based on both novelty and reward directs the search areas of the parameter-space with both high novelty and reward. This can, however, be improved further.

Algorithm 9 NSR-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, n number of workers, T number of iterations
initialize n workers with known random seeds, empty archive A, and initial parameters $\{\theta_0^1, \ldots, \theta_0^M\}$
set $t = 0$
for $i \in \{1 \ldots M\}$ **do**
    calculate $b(\pi_{\theta_0^i})$
    add $b(\pi_{\theta_0^i})$ to A
end for
for $t \in \{0 \ldots T-1\}$ **do**
    sample θ_t^m from $\{\theta_t^1, \ldots, \theta_t^M\}$ based on equation 2.4
    for $i \in \{1 \ldots n\}$ **do**
        sample $\epsilon_i \sim \mathcal{N}(0, I)$
        compute characteristics $b(\theta_t^m + \sigma\epsilon_i)$
        compute $n_i = N(b(\theta_t^m + \sigma\epsilon_i), A)$
        compute $f_i = f(\theta_t^m + \sigma\epsilon_i)$
    end for
    Send all novelties and rewards n_i, f_i from each worker to every other worker
    for $i \in \{1 \ldots n\}$ **do**
        reconstruct all perturbations ϵ_j for $j \in \{1, \ldots, n\}$ using known random seeds
        set $\theta_{t+1}^m = \theta_t^m + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} \frac{n_i + f_i}{2} \epsilon_i$
        add $b(\theta_{t+1}^m)$ to A
    end for
end for

While NSR-ES uses a linear combination of reward and novelty to approximate that is static for the whole duration of the training. Contrary to that NSRAdapt-ES (NSRA-ES) dynamically changes the ratio of reward gradient $f(\theta_t^{i,m})$ and novelty gradient $N(\theta_t^{i,m}, A)$ based on how the training is currently

progressing. This way, it will give more weight to the reward (thus following the performance gradient) when making progress and more weight to the novelty (following novelty gradient) when stuck in a local optima to give more incentive to find different approaches.

Formally, a parameter w is used to control the ratio of reward and novelty used for calculation of gradient estimate. For a specific w at a given generation parameter vector θ_t^m is updated (SGD used) via following expression:

$$\theta_{t+1}^m \coloneqq \theta_t^m + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} (1-w) N(\theta_t^{i,m}, A)\epsilon_i + w f(\theta_t^{i,m})\epsilon_i. \tag{2.7}$$

Algorithm 10 NSRA-ES

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, n number of workers, T number of iterations
initialize n workers with known random seeds, empty archive A, and initial parameters $\{\theta_0^1, \dots, \theta_0^M\}$
set $t = 0, t_{best} = 0, f_{best} = -\infty$
for $i \in \{1 \dots M\}$ **do**
    calculate $b(\pi_{\theta_0^i})$
    add $b(\pi_{\theta_0^i})$ to A
end for
for $t \in \{0 \dots T-1\}$ **do**
    sample θ_t^m from $\{\theta_t^1, \dots, \theta_t^M\}$ based on equation 2.4
    for $i \in \{1 \dots n\}$ **do**
        sample $\epsilon_i \sim \mathcal{N}(0, I)$
        compute characteristics $b(\theta_t^m + \sigma\epsilon_i)$
        compute $n_i = N(b(\theta_t^m + \sigma\epsilon_i), A)$
        compute $f_i = f(\theta_t^m + \sigma\epsilon_i)$
    end for
    Send all novelties and rewards n_i, f_i from each worker to every other worker
    for $i \in \{1 \dots n\}$ **do**
        reconstruct all perturbations ϵ_j for $j \in \{1, \dots, n\}$ using known random seeds
        set $\theta_{t+1}^m = \theta_t^m + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} \frac{n_i + f_i}{2} \epsilon_i$
        add $b(\theta_{t+1}^m)$ to A
    end for
    if $f(\theta_{t+1}^m) > f_{best}$ **then**
        $w = max(1, w + \delta_w)$
        $f_{best} = f(\theta_{t+1}^m)$
        $t_{best} = 0$
    else
        $t_{best} += 1$
    end if
    if $t_{best} \geq t_w$ **then**
        $w = min(0, w - \delta_w)$
        $t_{best} = 0$
    else
    end if
end for

Chapter 3

Experiments and discussion

3.1 Environments

First environment used for evaluation is the *Slimevolley* environment. The Slimevolley environment [17] is based on a game called "Slime Volleyball" created by an unknown author. The agent's task in this environment is to get the ball to hit ground on the opponent's side to make the oponent lose a life. The opponent is controlled by a small, 120-parameter, pre-trained neural network. [18]

Each agent has 5 lives in the beginning and the episode ends after 3000 steps or when either agent loses or their lives, whichever comes first. The agent receives a reward of $+1$ point when the opponent loses a life and -1 if it loses the life. In addition to this, for each survived timestep, the agent receives $+0.01$ reward. State is represented as a vector with 12 entries, x and y positions and respective velocities of the agent, opponent and the ball. Actions are represented as a vector with 3 entries, one for each action that the agent can do, jump, go forward or go backward. The agent will perform an action when the appropriate value is higher than 0.

Second environment is a *Cartpole-swingup* environment. It is inspired by classic reinforcement learning benchmark task, pole-balancing. In this task one end of a pole is attached to a card which moves left and right and the goal is to keep the pole upright. In the original the episode ends when the pole is tilted too much off its neutral position. However in this version the pole is able to rotate $360°$ and the episode ends only if the cart goes out of bounds or 1000 timesteps goes past.

The physics of the pole is controlled by equations specified for "Pendullum Swing-up" in PILCO software package [19]. Reward is given based on how far is the cart from centre (closer is better) and what is the angle of the pole (more

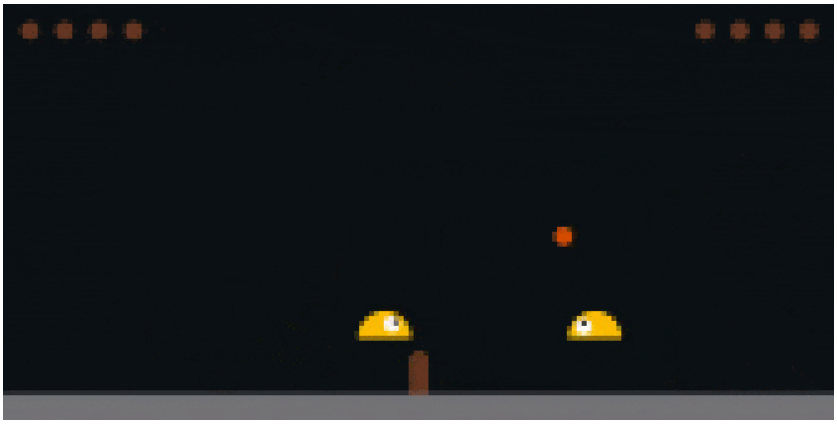

Figure 3.1 Screenshot of Slimevolley environment

upright is better). State is represented with a 4-entry vector containing the position of the cart, its velocity, sine and cosine of the pole's angle and its angular velocity while action is a number from -1 to 1 which represents force which is applied, in either direction, to the cart.

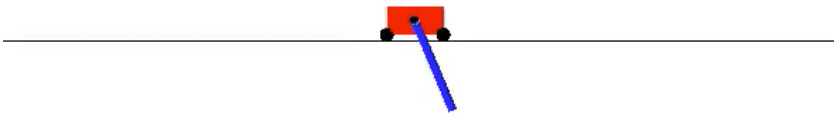

Figure 3.2 Screenshot of Cartpole-swingup environment

The behavioural characteristics is calculated similarly in both environments, inspired by approach used in [20] for classic Cartpole problem. For Slimevolley the x-coordinate of the controlled agent at timesteps 10, 30, 100, 300, 1000 and 3000 is taken and in Cartpole environment the x-coordinate of the cart at timesteps 5, 10, 50, 100, 500 and 100 is taken. If the environment doesn't reach

that state a 0 is filled in. This slightly differs from the implementation outlined in [20]. Then simple Euclidean norm is used to calculcate distance between two such characteristics.

Controlled agents's policies are represented with small feed-forward neural networks, for Cartpole 1-layer neural network with 5 inputs, 10 neurons in middle layer and one output neuron and $\tanh$ activation function applied to output of all neurons. Slimevolley has a bigger network, 12 input neurons, two layers with 20 neurons and 3 output neurons with $\tanh$ activation function.

3.2 Experiments

There were several experiments run in each environment. All experiments were run 10 times with different starting seeds for 1000 or 900 (in case of novelty based algorithms due to memory contraint on system where experiments were run) generations with population of 90 and 3 episodes per evaluation. The 5 methods that were tried are:

- CMA-ES,
- OpenAI ES,
- NS-ES,
- NSR-ES and
- NSRA-ES.

For CMA-ES, the only parameter was σ_{init} which controls the initial standard deviation and it was set to 0.1. OpenAI-ES started with σ 0.1, rate of decay 0.999 and limit 0.01. Novelty search based algorithm had fixed σ (because it doesn't make sense to change it since there are more individuals) of 0.1 and used 5 as k in k-nearest neighbours and metapopulation with size 5. In addition to that NSRA-ES waited 10 steps before increasing the ratio of novelty in the gradient by 0.05. All algorithms, except for CMA-ES, used the Adam [21] optimiser, with learning rate 0.1. In all presented graphs a median is shown along with the first and third quartile and the presented value is the current "mean" individual (or the best from metapopulation) evaluated on 270 runs. This evaluation was done every 25 generations.

3.2.1 Cartpole-swingup

This environment has proven to be significantly less challenging than Slimovolley. Both CMA-ES and OpenAI-ES have managed to solve (figures 3.3 and 3.4)

it but CMA-ES seems to have a significant performance dropoff in later generations (see figure 3.3). Closer inspection of data from traning has shown that in some cases the trainig gets stuck at values around 925 with parameters slowly growing in magnitude before completely exploding and performance dropping off. Bad performance from NS-ES was expected as it has no incetive to pursue better performance. NSR-ES (figure 3.6) slowly and steadily improves and the trajectory suggests that it would improve further given more time. This is most likely thanks to having 1:1 ratio of fitness and novelty therefore the the search for novelty slows down the search for improvement. In case of NSRA-ES (figure 3.7) a fast improvement can be seen in the beginning however as the weight of novelty slowly increases the algorithm starts to perform similarly to NS-ES (figure 3.5). This behaviour can be probably avoided with either better hyperparameter tuning or better novelty calculation.

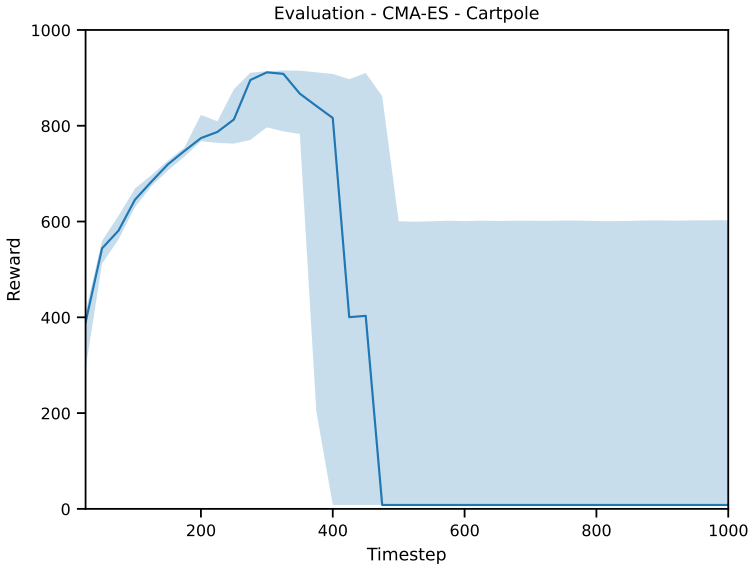

Figure 3.3 CMA-ES on Cartpole environment

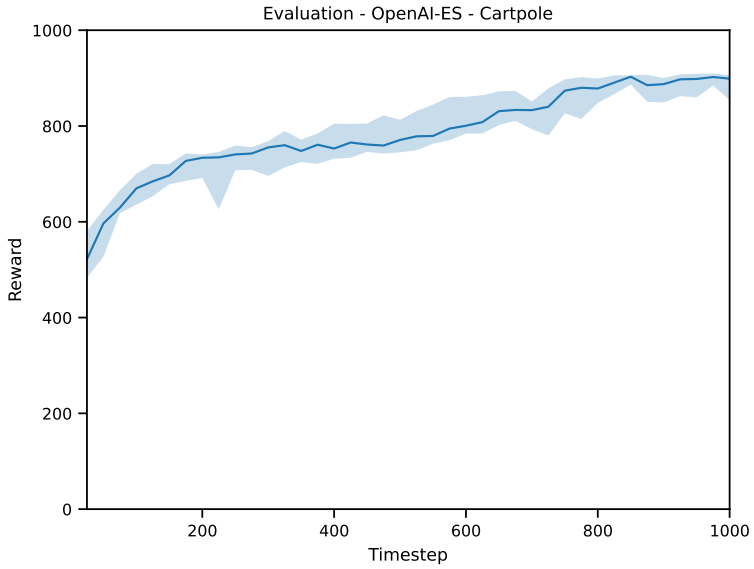

Figure 3.4 OpenAI-ES on Cartpole environment

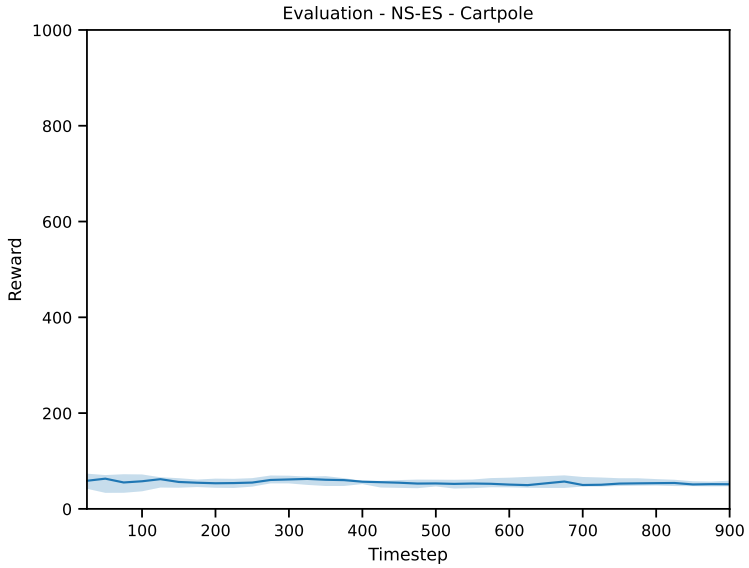

Figure 3.5 NS-ES on Cartpole environment

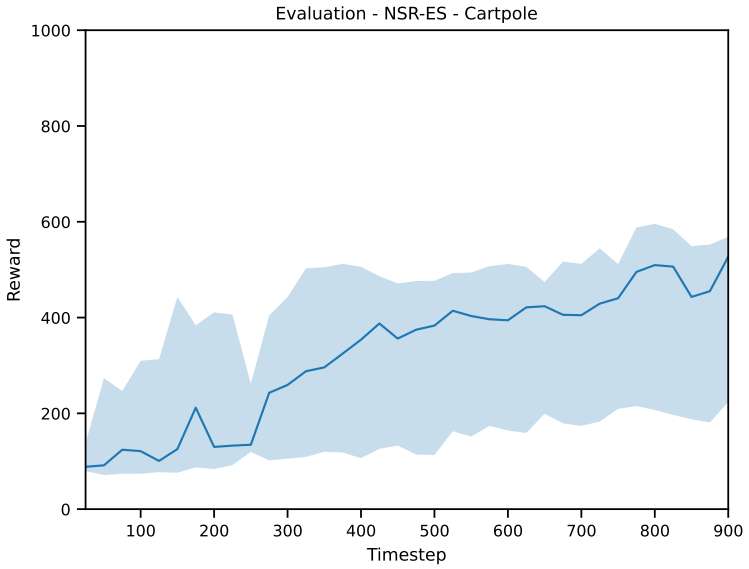

Figure 3.6 NSR-ES on Cartpole environment

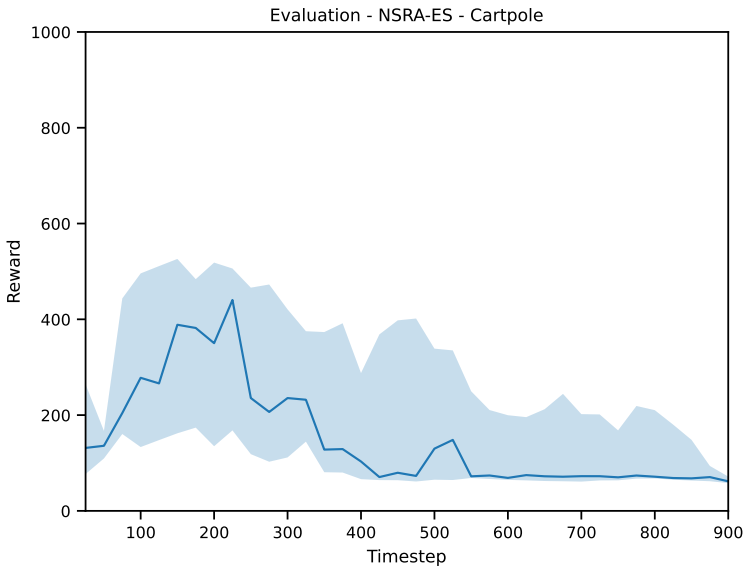

Figure 3.7 NSRA-ES on Cartpole environment

3.2.2 Slimevolley

The experiments have shown that this environment is quite challenging to solve. Only CMA-ES (figure 3.8) has managed to get good results in the 10 runs included in the experiment. Since the reward received by CMA-ES caps at 30 it would suggest that the agent didn't learn to beat the opponent but only to defend well enough. However during development OpenAI-ES (figure 3.9) has managed to get some good results as well but it was strongly sensitive to seed, equalling the performance of CMA-ES in some runs while not improving at all in others. Novelty search-based methods (figures 3.10, 3.11 and 3.12) did fail as well on this problem while intuitively they should have a higher chance of producing good results as they are able to explore in multiple directions at once thus have higher probability of going in the right direction. Bad performance by NS-ES was expected and in accordance with [10]. However both NSR-ES and NSRA-ES did not perform well either which points to either badly designed novelty or suboptimal hyperparameter setting.

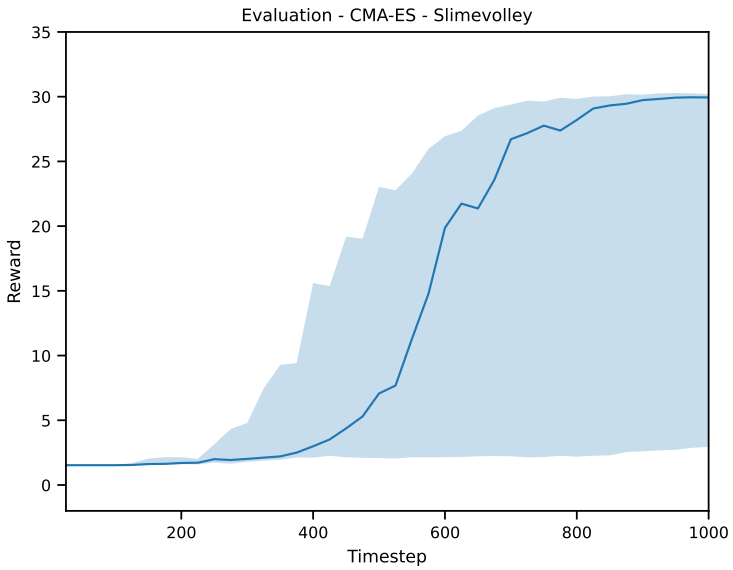

Figure 3.8 CMA-ES on Slimevolley environment

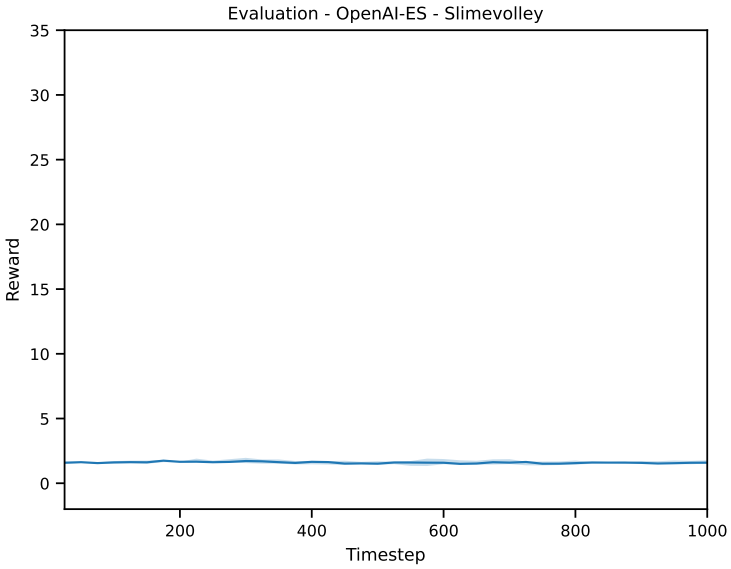

Figure 3.9 OpenAI-ES on Slimevolley environment

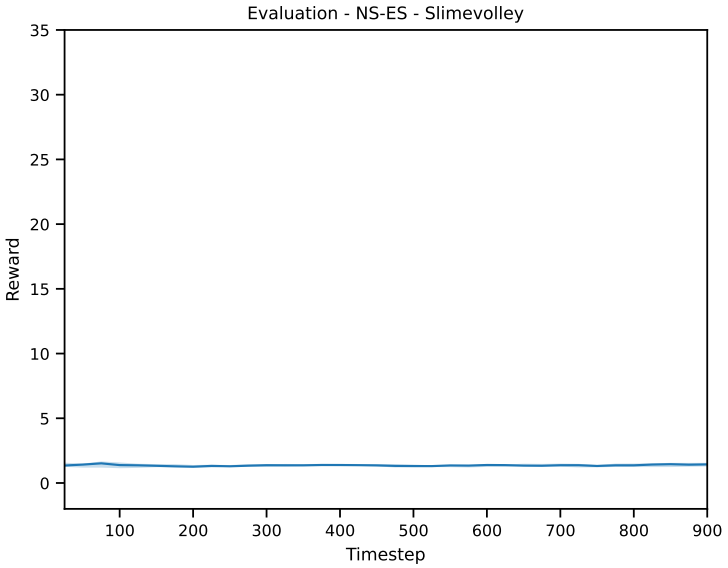

Figure 3.10 NS-ES on Slimevolley environment

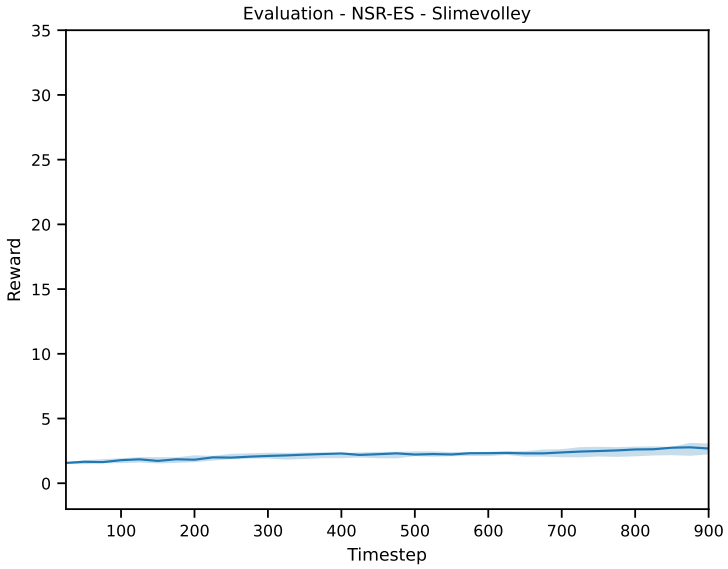

Figure 3.11 NSR-ES on Slimevolley environment

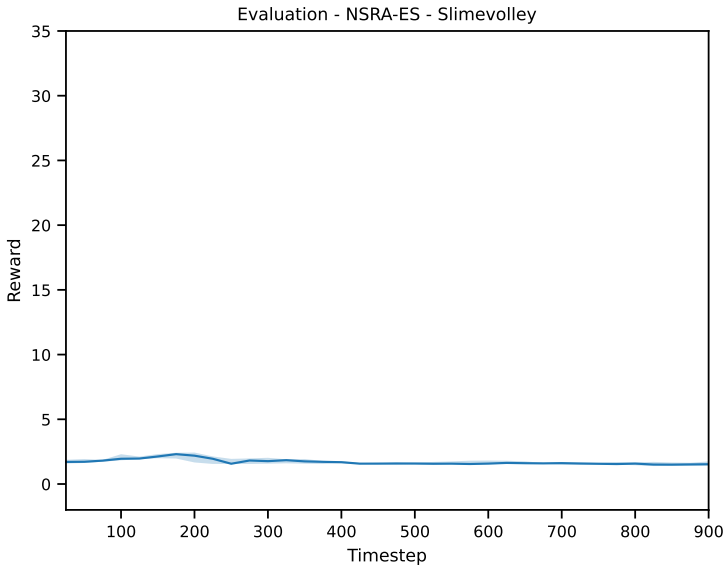

Figure 3.12 NSRA-ES on Slimevolley environment

Chapter 4

Technical implementation

All source code used in this thesis along with sample configuration files and instructions can be found at `https://github.com/MichalPospech/bc-project`.

The experiments were run using modified the `estool` tool, written in the Python programming language, which was originally used for an article about evolutionary strategies [22]. Another option was to use implementation by Uber available at `https://github.com/uber-research/deep-neuroevolution` however it is much more complex (uses redis for inter-worker communication) and thus would be harder to extend.

`estool` makes use of the master-slave paradigm with one thread controlling the evolution and slaves only evaluating the candidate solutions. Communication between workers is done using `MPI` framework. The slave workers accept 3 types of commands from the master worker:

- `eval` - tells the slave to evaluate the solutions that will be sent next and send the results back to the master,
- `archive` - tells the slave to add the behavioral characteristic that will be sent next to the archive used for novelty calculation,
- `kill` - tells the slave to stop.

All the implemented algorithms share interface with two methods crucial for their functionality which are being called alternatly. The `ask` method provides a new generation based on current state of the algorithm used. The `tell` method does the exact opposite. It takes results of evaluations and updates the internal state of the algorithm.

`estool` was extended to support novelty search by making a new class `NSAbstract` which encapsulates logic that all novelty search based algorithms

(NS-ES, NSR-ES and NSRA-ES) have in common and classes for specific algorithms inherit from it and implement the algorithm-specific behaviour. Another modification necessary to enable usage of novelty was modification of the communication protocol between master and slaves, before it could handle only sending solutions from master to slaves and sending rewards back. Thus the aforementioned commands were introduced.

During training solution is evaluated more times (configured with parameter `num_episode`) to get a better estimate of particular solution's performance because it varies based on selected seed. Similar approach is used for handling behavioural characteristics of solutions. One characteristic consists of `num_episode` vectors and to calculate distance between two characteristics mean of all pairwise distances is used to account for strong dependence on seed. During evaluation of current solution, not used in training, an equivalent of whole population with each individiual being the solution to evaluate is created and gets evaluated to receive a more robust estimate of current performance.

However there is a slight deviation from the algorithms outlined in chapter 2. There only the results of solution evalution (and characteristics of new solutions in case of an algorithm that utilises novelty) are communicated between workers and all parameter perturbations are done locally based on shared seeds while `estool` does all perturbations in the master worker and solutions are sent, along with seeds for the task, to slave workers only for evaluation. This impacts the algorithm performance as the communication overhead increases with larger population and larger solutions (parameter vectors for policies) while the quality of generated solutions is not affected at all because the way new parameter vectors θ are calculated doesn't change.

As the project uses MPI for multiprocessing, running it is slightly more complex than running an ordinary Python project. It needs to be run using `mpiexec` as follows.

```
mpiexec -n NUM_CPU python -m mpi4py train.py -f CONFIG
```

Parameter `NUM_CPU` must be at least 2 (due to the the master-slave paradigm) and at most the physical number of cores/nodes and `CONFIG` is a path to JSON configuration file for the experiment. Further details and instructions can be found in the `Readme.md` file in the project repository at `https://github.com/MichalPospech/bc-project`.

Conclusion

In this thesis reinforcement learning was formally introduced along with several methods such as Q-learning or REINFORCE. Then evolutionary algorithms were described and further expanded by evolutionary strategies including the introduction of CMA-ES. Following that OpenAI-ES was outlined, a evolutionary strategy that was designed for solving reinforcement learning tasks, high level of paralellisation and doesn't require differentiable policy as there is no differentiation needed unlike in traditional methods such as REINFORCE. This algorithm is further extended with novelty search which uses behavioral characteristics of evolved agents to evolve agents to behave in a different manner than their ancestors which should enable the agents to deal with falling into a local optima. Then the 2 test environment were outlined and data from experiments discussed. Finally some technical details about the implementation were explained.

Based on conducted experiments Slimevolley is a task that is hard to solve since only one method (CMA-ES) managed to do so and OpenAI-ES did so only few times and the performance varied heavily with seed. Cartpole has proven to be slightly easier with more methods being successful or would succeed given more time. Furthermore novelty search-based algorithm were found to be sensitive to quality of novelty calculation.

This work has shown that designing novelty search-based algorithms is mostly designing a good behavioral characteristic of an agent. Therefore this would be, along with more hyperparameter tuning, a possible direction to follow in research. Furthermore exploring performance of these methods on more environments would be beneficial as well. To get more accurate results on the two environments presented, spending more time with hyperparameter tuning and trying running longer experiments with a bigger population would help. Running more runs for each experiment would also be beneficial as it would create batter

Bibliography

[1] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[2] Noam Brown and Tuomas Sandholm. "Superhuman AI for multiplayer poker". In: *Science* 365.6456 (2019), pp. 885–890. ISSN: 0036-8075. DOI: 10.1126/science.aay2400. eprint: https://science.sciencemag.org/content/365/6456/885.full.pdf. URL: https://science.sciencemag.org/content/365/6456/885.

[3] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038/s41586-020-03051-4.

[4] OpenAI et al. *Solving Rubik's Cube with a Robot Hand*. 2019. arXiv: 1910.07113 [cs.LG].

[5] Paulo Lissa et al. "Deep reinforcement learning for home energy management system control". In: *Energy and AI* 3 (2021), p. 100043. ISSN: 2666-5468. DOI: https://doi.org/10.1016/j.egyai.2020.100043. URL: https://www.sciencedirect.com/science/article/pii/S2666546820300434.

[6] Chao Yu, Jiming Liu, and Shamim Nemati. *Reinforcement Learning in Healthcare: A Survey*. 2020. arXiv: 1908.08796 [cs.LG].

[7] B Ravi Kiran et al. *Deep Reinforcement Learning for Autonomous Driving: A Survey*. 2021. arXiv: 2002.00444 [cs.LG].

[8] Jean-Yves Potvin. "Genetic algorithms for the traveling salesman problem". In: *Annals of Operations Research* 63.3 (1996), pp. 337–370. ISSN: 1572-9338. DOI: 10.1007/BF02125403. URL: https://doi.org/10.1007/BF02125403.

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

[10] Edoardo Conti et al. *Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents*. 2018. arXiv: 1712.06560 [cs.AI].

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[12] Frank Sehnke. "Parameter Exploring Policy Gradients and their Implications". Dissertation. München: Technische Universität München, 2012.

[13] P. A. Vikhar. "Evolutionary algorithms: A critical review and its future prospects". In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 2016, pp. 261–265. DOI: 10.1109/ICGTSPICC.2016.7955308.

[14] Hans-Paul Schwefel and Günter Rudolph. "Contemporary evolution strategies". In: *Advances in Artificial Life*. Ed. by Federico Morán et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 891–907. ISBN: 978-3-540-49286-3.

[15] Günter Rudolph. "Evolutionary Strategies". In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 673–698. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_22. URL: https://doi.org/10.1007/978-3-540-92910-9_22.

[16] Nikolaus Hansen. *The CMA Evolution Strategy: A Tutorial*. 2016. arXiv: 1604.00772 [cs.LG].

[17] David Ha. *Slime Volleyball Gym Environment*. https://github.com/hardmaru/slimevolleygym. 2020.

[18] David Ha. "Neural Slime Volleyball". In: *blog.otoro.net* (2015). URL: https://blog.otoro.net/2015/03/28/neural-slime-volleyball/.

[19] Marc Peter Deisenroth et al. *PILCO Code Documentation v0.9*. 2013. URL: http://mlg.eng.cam.ac.uk/pilco/release/pilcodocV0.9.pdf.

[20] Benjamin Inden et al. "An examination of different fitness and novelty based selection methods for the evolution of neural networks". In: *Soft Computing* 17.5 (2013), pp. 753–767. ISSN: 1433-7479. DOI: 10.1007/s00500-012-0960-z. URL: https://doi.org/10.1007/s00500-012-0960-z.

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[22] David Ha. “Evolving Stable Strategies”. In: *blog.otoro.net* (2017). URL: http://blog.otoro.net/2017/11/12/evolving-stable-strategies/.