Scenario Co-Evolution for Reinforcement Learning on a Grid World Smart Factory Domain

Thomas Gabor,
Andreas Sedlmeier,
Marie Kiermeier,
Thomy Phan
LMU Munich
thomas.gabor@ifi.lmu.de

Marcel Henrich,
Monika Pichlmair
University of Augsburg

Bernhard Kempter,
Cornel Klein,
Horst Sauer,
Reiner Schmid,
Jan Wieghardt
Siemens AG

ABSTRACT

Adversarial learning has been established as a successful paradigm in reinforcement learning. We propose a hybrid adversarial learner where a reinforcement learning agent tries to solve a problem while an evolutionary algorithm tries to find problem instances that are hard to solve for the current expertise of the agent, causing the intelligent agent to co-evolve with a set of test instances or *scenarios*. We apply this setup, called *scenario co-evolution*, to a simulated smart factory problem that combines task scheduling with navigation of a grid world. We show that the so trained agent outperforms conventional reinforcement learning. We also show that the scenarios evolved this way can provide useful test cases for the evaluation of any (however trained) agent.

CCS CONCEPTS

• **Computing methodologies → Adversarial learning**; **Neural networks**; **Genetic algorithms**; **Generative and developmental approaches**; *Robotic planning*; *Instance-based learning*; Mobile agents;

KEYWORDS

coevolution, reinforcement learning, evolutionary algorithms, automatic test generation, adversarial learning

ACM Reference Format:
Thomas Gabor, Andreas Sedlmeier, Marie Kiermeier, Thomy Phan, Marcel Henrich, Monika Pichlmair, and Bernhard Kempter, Cornel Klein, Horst Sauer, Reiner Schmid, Jan Wieghardt. 2019. Scenario Co-Evolution for Reinforcement Learning on a Grid World Smart Factory Domain. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321831

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '19, July 13–17, 2019, Prague, Czech Republic
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321831

1 INTRODUCTION

Reinforcement learning has been at the heart of most recent success stories in artificial intelligence [3, 24, 31, 45]. Naturally, many extensions of the basic concept have been proposed [23, 25, 28]. In this paper, we take a look at adversarial learning: Instead of one single agent, in adversarial learning, we train two agents with largely opposing goals [39]. Thus, while one agent tries to maximize the given reward function, the other agent is allowed to disturb the environment to a certain extent and thus work against the plan of the first agent. For instance, Pinto et al. [39] train an agent for the purpose of walking the maximal distance without falling down, given a virtual body set up in a particular way. At certain intervals in between that training process, they train another agent that has the goal to apply disturbances to the virtual environment (and in extent to the body controlled by the first agent) in such a way as to minimize the distance walked by the first agent. When set up right, this increases the difficulty for the first agent (without making the task impossible) but also the learning progress and eventually helps in training a better agent. This example can be seen as the standard instance of adversarial learning. Pinto et al. [39] have shown that an agent trained in this way is not only able to walk longer distances in the presence of disturbances (as they were present during its training) but also is a better agent for walking even when there are no disturbances at all. In a similar setting, Florensa et al. [16] have shown that starting with simple challenges and then increasing the difficulty can lead to better overall success during training.

A similar concept has been observed in biology and transferred to evolutionary algorithms: The phenomenon of two evolutionary processes influencing each other's fitness evaluations is called co-evolution. In biology, common observations include the flowers of pollinating plants and the beaks and mandibles of birds and insects that feed on their nectar. In this case, the involved species of plant and bird both benefit from a common and matching solution. However, there are also examples of competitive co-evolution: Prey and predator constantly adapt to each other's changes in a way quite similar to the scenario of the walking agents described above [37]: While the prey tries to maximize its chance of getting away or self-defend, the predator tries to impede that exact objective. When set up right, competitive co-evolution results in an arms race where both antagonistic populations try to outperform the other, possibly resulting in rapid progress and increased genetic robustness [8].

This concept has been used in the most recently released work of Wang et al. [50], who co-evolve a set of walking agents and a set of challenging environments. For both, they use evolutionary strategies, which are related to some techniques of reinforcement learning [54] but are now rather seen as a method of gradient-based black-box optimization in contrast to classical reinforcement learning [43, 53]. In their approach, called POET (which stands for Paired Open-Ended Trailblazer), they pair up instances of agents and environments to form single individuals in an overarching evolutionary process. By contrast, we suggest (among other differences, cf. Section 3) a simpler interaction model for agents and environments and train a reinforcement learning agent using standard back-propagation. Still, we are able to produce similar results for the discrete smart factory domain we introduce in this paper.

In all three examples introduced so far, the antagonistic agents or species only interact via the mutually shared results of fitness evaluations: A good result for one side is bad for the the other. (We will formalize this setup in Section 4.) This allows us to combine various techniques here, i.e., we propose to use a reinforcement learning agent to train for certain behavior (that is well encoded using neural networks) and an evolutionary algorithm to evolve challenging environments to operate in (that are easily encoded using discrete vectors without an obvious gradient to them). Thus, we can use adequate data structures on both sides of the co-evolution. Motivated by the industry origin of the domain, we evaluate our results not only against the fitness score, but also against a criterium of solved/failed instances, showing greater robustness for the co-evolutionary approach.

We consider this approach an instance of the general architectural model we described in [22], hence we also call it *scenario co-evolution*. There, the co-evolutionary principle is applied to software engineering, where productive code and software tests co-evolve: A test is good when it finds bad behavior in the productive code, while productive code is good when no test finds bad behavior. As suggested in both [50] and [22], we show in this paper that the co-evolved environments (also called *scenarios*) can be used efficiently for classical software testing, outperforming random testing in their prowess to challenge an agent.

We now continue to formally introduce the basic concepts mentioned so far (cf. Section 2) and then give an overview of related work (cf. Section 3). We formally describe our approach at scenario co-evolution in Section 4 and provide an empirical evaluation in Section 5. We conclude with Section 6.

2 BASICS

2.1 Markov Decision Processes

We base our problem formulation on the notion of a Markov decision process (MDP) [41], which is given via the tuple: $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$. $\mathcal{S}$ is a (finite) set of states; $s_t \in \mathcal{S}$ is the state of the MDP at time step t. $\mathcal{A}$ is the (finite) set of actions; $a_t \in \mathcal{A}$ is the action the MDP takes at time step t. $\mathcal{P}(s_{t+1}|s_t, a_t)$ is the transition probability function; a state transition occurs by executing an action a_t in a state s_t. The resulting next state s_{t+1} is then determined according to $\mathcal{P}$. Note that in this paper we focus on a deterministic domain represented by a deterministic MDP, so $\mathcal{P}(s_{t+1}|s_t, a_t) \in \{0, 1\}$. Finally, $\mathcal{R}(s_t, a_t)$ is the reward awarded when the MDP takes action a_t when in state s_t; for this paper we assume that $\mathcal{R}(s_t, a_t) \in \mathbb{R}$.

The goal is to find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ in the space of all possible policies Π, which maximizes the (discounted) return G_t at state s_t over a potentially infinite horizon, given via

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot \mathcal{R}(s_{t+k}, a_{t+k}) \tag{1}$$

where $\gamma \in [0, 1]$ is the discount factor.

2.2 Reinforcement Learning

In order to search the policy space Π, we consider model-free reinforcement learning (RL), in which an agent interacts with an environment given as an MDP $\mathcal{M}$ by executing a sequence of actions $a_t \in \mathcal{A}, t = 0, 1, \ldots$ [48]. In the fully observable case of reinforcement learning, the agent knows its current state s_t and the action space $\mathcal{A}$, but not the effect of executing a_t in s_t, i.e., $\mathcal{P}(s_{t+1}|s_t, a_t)$ and $\mathcal{R}(s_t, a_t)$. In order to find the optimal policy π^* a commonly used value-based approach is Q-Learning [51], named for the action-value function $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}, \pi \in \Pi$, which describes the expected accumulated reward $Q^\pi(s_t, a_t)$ when taking action a_t when in state s_t and then following the policy π for all states $s_{t+1}, s_{t+2}, \ldots$ afterwards.

The optimal action-value function Q^* is any action-value function that yields higher accumulated rewards than all other action-value functions, i.e., $Q^*(s_t, a_t) \geq Q^\pi(s_t, a_t)\ \forall \pi \in \Pi$. Q-Learning aims to approximate Q^* by starting from an initial guess for Q, which is then updated via

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2}$$

by making use of experience samples $e_t = (s_t, a_t, s_{t+1}, r_t)$, where r_t is the reward earned at time step t, i.e., by executing action a_t when in state s_t. The learning rate α is a usually setup-specific parameter.

The learned action-value function Q converges to the optimal action-value function Q^*, which then implies an optimal policy $\pi^*(s_t) = \arg\max_a Q(s_t, a)$.

It is common to use a parameterized function approximator (like a neural network), to approximate the action-value function: $Q(s_t, a_t; \theta) \approx Q^*(s_t, a_t)$ with θ specifying the weights of the neural network. When a deep neural network is used as the function approximator, this approach is called deep reinforcement learning. Mnih et al. [31] showed that combining this approach with deep convolutional networks allows for successful learning from high-dimensional input features like raw image data.

2.3 Evolutionary Algorithms

For this paper, we assume an evolutionary process (EP) to be defined as follows: Given a fitness function $f : \mathcal{X} \rightarrow \mathbb{R}$ for an arbitrary set $\mathcal{X}$ called the search space, we want to find an individual $x \in \mathcal{X}$ with the best fitness, i.e., $f(x) \leq f(x')\ \forall x' \in \mathcal{X}$. Note that for consistency with the later application, we assume that the best fitness has the lowest values, i.e., that we try to minimize the fitness values. Usually, the search space $\mathcal{X}$ is too large or too complicated to guarantee that we can find the exact best individual(s) using standard computing models (and physically realistic time). Thus, we take discrete subsets of the search space $\mathcal{X}$ via sampling and iteratively improve their fitness. An evolutionary process $\mathcal{E}$ over

g generations, $g \in \mathbb{N}$, is defined as $\mathcal{E} = \langle \mathcal{X}, o, f, (X_i)_{i<g} \rangle$. $\mathcal{X}$ is the search space. $o : \mathfrak{P}(\mathcal{X}) \to \mathfrak{P}(\mathcal{X})$ is the evolutionary step function so that $X_{i+1} = o(X_i)\ \forall i \geq 0$. As defined above, $f : \mathcal{X} \to \mathbb{R}$ is the fitness function. $(X_i)_{i<g}$ is a series of populations so that $X_i \subseteq \mathcal{X}\ \forall i$.

For this work, we use the following evolutionary operators:

- The recombination operator $\mathrm{rec} : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ generates a new individual from two given individuals.
- The mutation operator $\mathrm{mut} : \mathcal{X} \to \mathcal{X}$ alters a given individual slightly to return a new one.
- The migration operator $\mathrm{mig} : \mathcal{X}$, also called hyper-mutation, generates a random individual $\mathrm{mig}() \in \mathcal{X}$.
- The selection operator $\mathrm{sel} : \mathfrak{P}(\mathcal{X}) \times \mathbb{N} \to \mathfrak{P}(\mathcal{X})$ returns a new population $X' = \mathrm{sel}(X, n)$ given a population $X \subseteq \mathcal{X}$, so that $|X'| \leq n$.

The operators $\mathrm{rec}, \mathrm{mut}, \mathrm{mig}$ can be applied to a population X by choosing individuals from X to fill their parameters (if any) according to some selection scheme σ and adding their return to the population. For example, we allow to write $\mathrm{mut}_\sigma(X) = X \cup \{\mathrm{mut}(\sigma(X))\}$.

For any evolutionary process $\mathcal{E} = \langle \mathcal{X}, o, f, (X_i)_{i<g} \rangle$ and selection schemes $\sigma_1, \sigma_2, \sigma_3$ we assume that

$$X_{i+1} = o(X_i) = \mathrm{sel}(\mathrm{mig}_{\sigma_3}(\mathrm{mut}_{\sigma_2}(\mathrm{rec}_{\sigma_1}(X_i))), |X_i|). \quad (3)$$

Roughly, we assume that an evolutionary process fulfills its purpose if the best fitness of the population tends to better over time, i.e., $\min_{x \in X_i} f(x) \geq \min_{x \in X_{i+k}} f(x)$ for sufficiently large k.

3 RELATED WORK

3.1 Adversarial Learning

Adversarial Learning is a powerful paradigm towards robust reinforcement learning and has been widely used to train agents on zero-sum games and continuous tasks [3, 5, 39, 44–46, 49].

Self-play reinforcement learning is a popular way to train agents on complex zero-sum games like checkers, backgammon, or Go by training a single agent on data generated by playing games against itself [3, 39, 44–46, 49]. Since the agent always plays against itself, the opponent always has an adequate difficulty level for the agent to improve steadily. This can lead to complex behavioral strategies emerging from simple game rules. Self-play reinforcement learning can be regarded as the most simple way of adversarial learning, where only the self-playing agent adapts, while the environment remains static.

As mentioned in Section 1, agents can also be trained on single-agent problems by evolving the environment adversarially like via adding noise, disturbances, or extra forces to ensure robust behavior [39]. The adversarial environment itself can be modeled by a reinforcement learning agent, which tries to minimize the outcome of the actual agent to be trained. This model results in a zero-sum game between the original agent and the environment itself [39]. Another way is to provide adversarial input samples to fool the reinforcement learning agent into making suboptimal decisions [38].

Beyond pure reinforcement learning, co-evolution has also been used in many systems based on neuro-evolution, i.e., finding the right weights for neural networks using some kind of evolutionary process [32, 36]. *Paired Open-Ended Trailblazer (POET)* uses a regularized variant of co-evolution, which maintains a pool of environment-agent pairs, where only environments having an adequate difficulty level for the current agent pool are kept in the population [50]. The agents are trained with evolutionary strategies [43] (allowing for a distribution of computational effort across multiple machines) and attempted to be transferred from one environment to another to escape local optima.

3.2 Test Evolution

The environments generated by an approach like POET (described above) can also function as basis for software testing, as we argue in this paper. However, the generation of test cases for software products has been a widely-researched topic on its own [2, 15]. While classical approaches incorporate and often combine domain knowledge and random sampling [10, 34], search-based software testing aims a stochastic process towards more difficult test cases specifically [7, 30]. Many of those approaches, most prominently EvoSuite [17–19], also employ evolutionary algorithms to search through the space of possible test cases [12, 29, 42, 52].

In contrast to most state-of-the-art approaches, we consider as a system-under-test not a classical, fixed piece of software but a self-adaptive, learning system. Since these can change their own behavior over time, they require a dynamic testing method as well and are considered very hard to control for classical methods of software testing [6, 11, 13]. Especially reinforcement learning poses several challenges as the learning progress is usually hard to keep track off and the resulting behavior is hidden behind intransparent policy encodings like neural networks [1]. While techniques like adversarial learning usually use neural networks on both sides, we argue that a collection of test cases as can be derived from the population of scenarios is more transparent to human inspection than the test encodings generated by previous approaches. This argument has already been made for the process of software engineering as whole but not verified at a component level [22, 26].

Aside from generating software tests in a narrow sense, co-evolution has also played a role in augmenting evolutionary search towards more robust or more diverse results [4, 40]. It should be noted that while the field of cooperative co-evolution (c.f. [33] for a mathematical model and taxonomy) has interesting applications, especially to learning agents [55], we focus entirely on a case of competitive co-evolution.

4 APPROACH

We propose to combine an agent using reinforcement learning with an evolutionary process evolving hard test cases. Assume we have a family of MDPs $\mathcal{M}_x = (\mathcal{S}, \mathcal{A}, \mathcal{P}_x, \mathcal{R}_x)$ for an arbitrary parameter $x \in \mathcal{X}$, with $\mathcal{X}$ being an arbitrary parameter space to the MDP. A specific setting for x is also called a *scenario*. We limit the difference between two differently parametrized MDPs to the transition probability function and the reward function with the state and action space remaining constant. Note that changing the transition probability function may render some areas of the state space unreachable.

Typically, when we want our agent to perform well against any instance of the family $\mathcal{M}_x$, we need to provide it with experience samples $e_t = (s_t, a_t, s_{t+1}, r_t)$ that were generated for all (or as many as possible) different scenarios $x \in \mathcal{X}$. Note that the setting of x directly affects the values of s_{t+1} via $\mathcal{P}_x$ and r_t via $\mathcal{R}_x$. Usually, some effect of a specific setting of x may also be visible in s_t and thus exposed to the agent. This is the case for the smart factory domain we introduce later.

The acknowledgement of certain non-user-controllable parameters within the environment is crucial to realistic applications. In most cases we have but a rough model of how the environment may behave but no way to pinpoint the specifics unless we try out all possibilities. So the agent needs to be trained against all of them, ideally. Of course, for sufficiently large or complex $\mathcal{X}$ this becomes infeasible. A standard approach is to take random samples from $\mathcal{X}$ instead. This causes the agent to specialize on the average scenario $x \in \mathcal{X}$ after a while of training, which may be a good choice per se. However, in most real-world scenarios, good average case performance is largely outweighed by bad worst case performance, i.e., a navigation software that (even rarely) provokes incidents is bad for the job, even if on average it finds the way quicker than its competition.

So instead of taking random samples from the scenario space $\mathcal{X}$, we may want to focus on the hard settings for x, i.e., those values x for which the agent's performance deteriorates. In order to do so, we have to find the respective values for x first, though. We propose to do so using an evolutionary algorithm (as described in Section 2.3) that optimizes for hard settings for x. This evolutionary algorithm constructs an evolutionary process with search space $\mathcal{X}$ (rendering our variable naming scheme consistent). The resulting population after a few generations of optimizing for hard x is then used to generate experience samples for the reinforcement learning agent. The best reinforcement learning agent so far is in turn used to evaluate the hardness of the settings for x for the next few generations of evolution.

Figure 1 shows a schematic representation of the combined process called *scenario co-evolution (SCoE)*: The interaction points between the evolutionary process and the reinforcement learning agent are:

- The experience samples necessary to train the agent are drawn using settings for $x \in \mathcal{X}$ that are included in the current population X of the evolutionary process. When all $x \in X$ have been used, the evolutionary process evolves further for a few generations.
- The fitness $f(x)$ assigned to each $x \in X$ is computed using the accumulated reward of running the current agent policy π on the MDP $\mathcal{M}_x$, i.e.,

$$f(x) = \sum_{t=0}^{h} \mathcal{R}_x(s_t, \pi(s_t)) \tag{4}$$

  where h is the end of the current episode, i.e., $\mathcal{P}_x(s|s_h, a) = 0 \ \forall s \in \mathcal{S}, a \in \mathcal{A}$. Note that we defined the reinforcement learning agent to maximize its reward while the evolutionary process tries to minimize the fitness.

These interactions suffice to give rise to competitive co-evolution between a supposedly robust agent and a set of hard scenarios. However, our evaluation shows that the so-trained reinforcement learning agent not only performs better in the hard scenarios it was trained for, but also in randomly selected average scenarios. We call this the "exam effect": When we confront the agent with hard scenarios (and it can solve those), we can also assume it can solve easy scenarios. Thus, there is no additional use to confront it with easy scenarios during training. Effectively, this is why we can talk about "easy" and "hard" scenarios in the first place: The agent does not simply specialize on a specific subset of scenarios and gets worse on other scenarios in return, but it gets better in all scenarios by training on some scenarios we thus call "hard". This implies a hierarchy or order among scenarios. The scenarios that can be learned alongside training on hard scenarios can then be called "easy".

5 EVALUATION

5.1 Smart Factory Domain

For the evaluation of our approach, we implemented a smart factory domain, in which a number of *items* have to be processed at *workstations* of different types, while avoiding collisions with dynamically placed *obstacles*. Thus, the main focus of the task lies in navigation through the smart factory. However, at certain times the agent also needs to decide which workstation to visit next.

The environment is implemented as a discrete grid of size 7×8 as can be seen in Figure 2. Five workstations of three different types are placed at fixed positions. Five items are placed at these workstations that need to be processed at various other workstations according to an item-specific, fixed sequence of length $1 - 3$. While the agent always starts each episode at the fixed position $(1, 1)$, 4 obstacles are placed at varying positions on the grid. The positions of these obstacles are the only free variables in the environment and are either determined randomly, or according to the SCoE method's evolution, i.e., the SCoE approach optimizes for the most impeding position of obstacles to the agent, $\mathcal{X} \subset (\{0, ..., 6\} \times \{0, ..., 7\})^4$. However, note that we check for unsolvable instances (when a single workstation is completely blocked, e.g.) and exclude these from both random sampling and evolution.

For the purpose of passing the factory state as an input to the reinforcement learning agent (and in extent its neural network), the factory state is encoded as a stack of 7×8 feature planes, where each plane represents the spatial positions of workstations or the agent w.r.t. to some attribute. See Figure 3 for an informal description of these feature planes.

At each timestep t, the agent can execute a single action a_t from the action space $\mathcal{A}$: move north, south, west, east, pick-up, place. Valid movements, i.e., movements onto free grid fields cause a reward of -1, while collisions with the grid boundary or an obstacle keep the agent's position unchanged and are punished with a reward of -100. A valid pick-up action can only be executed if the agent is not already carrying an item and is standing on a field adjacent to a workstation where an item is available. If the agent is carrying an item, it can execute a valid place action if it is positioned on a field adjacent to a workstation with a type matching the item's next step in the processing sequence. A place action at any other state is considered invalid. The current implementation contains no stochasticity, i.e., the state transitions and rewards are deterministic. A valid pick-up or place action is rewarded with 100, while the reward of an invalid one is -50. An episode is completed if all items were processed correctly.

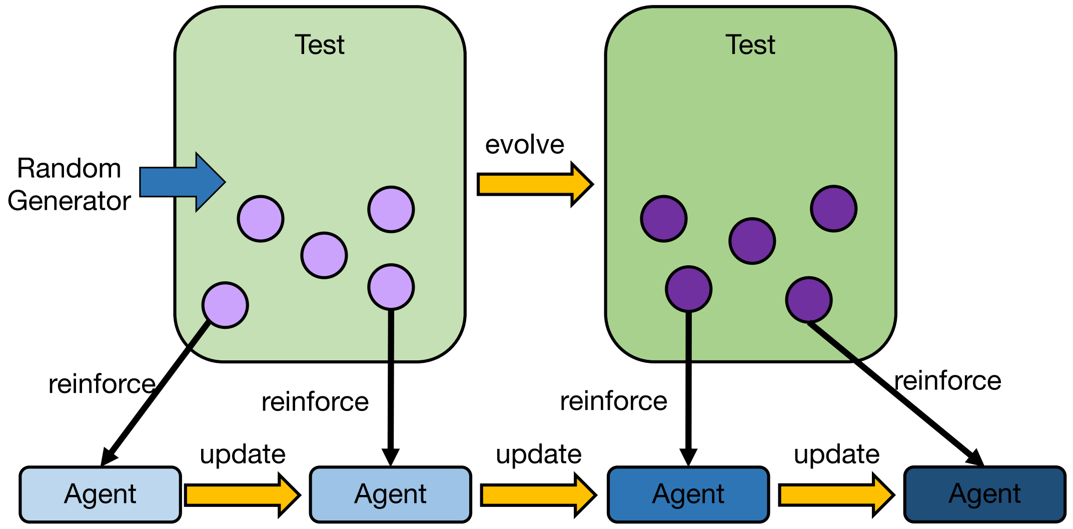

Figure 1: Schematic representation of a SCoE process. A population of test scenarios is first generated at random and then improved via evolution. Between evolutions, the test scenario population is fully utilized as training data for the reinforcement learning agent, which causes the agent to improve in parallel to the test scenario population.

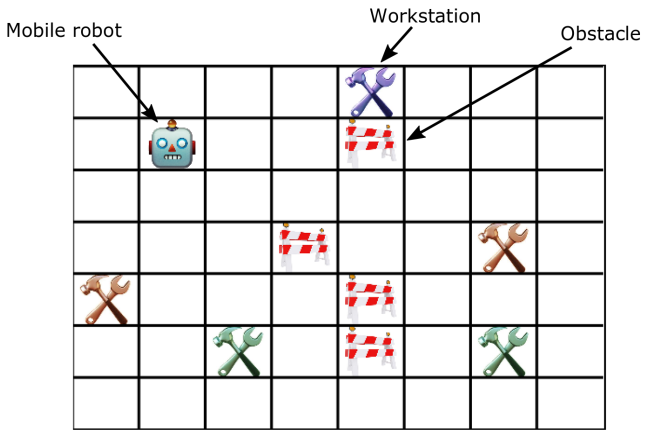

Figure 2: Visualization of the smart factory domain. A mobile robot can travel north, east, south and west on the grid. It needs to visit workstations in order to retrieve items and then needs to visit other workstations in order to process these items. Attempting to walk out of the grid, into a workstation or into an obstacle is penalized. Obstacle positions vary according to the setting of the scenario x.

5.2 Setup

A neural network is used as the function approximator for Q^*; it is composed of 3 convolutional layers with 64 neurons each, a kernel size of 3 and a stride length of 1, followed by a dense layer with 128 neurons and a dense output layer with 6 neurons, matching the size of the action space $\mathcal{A}$. All neurons use ReLU nonlinearity [35] as the activation function, while Adam [27] is used to minimize the mean squared error loss. In order to discover new actions to take, the agent uses ϵ-greedy exploration, starting with $\epsilon = 1$ and exponentially decaying to $\epsilon = 0.1$ after 40000 actions. We use the learning rate $\alpha = 0.01$ and the discount factor $\gamma = 0.95$.

Scenarios encode the position of the 4 obstacles in the domain, so $\mathcal{X} = (\{0, ..., 6\} \times \{0, ..., 7\})^4 \ \setminus \ \mathcal{Y}$ where $\mathcal{Y}$ are unsolvable or non-sensical setups (placing obstacles directly on workstations, encapsulating workstations or the agent and so on). While the state-of-the-art agent (called "random" in the plots) selects its scenarios to use for training episodes using random sampling, the SCoE agent draws them from an evolutionary process with population size 500. As SCoE uses all individuals for training exactly once, the evolution has to be continued every 500 episodes. When evolution resumes, it runs for another 500 generations. The SCoE evolutionary process simply selects the best 500 individuals from parents and children combined as survivors and uses tournaments of size 250 for parent selection. Parents are recombined via uniform crossover on a per-obstacle basis. A migration (i.e., hyper-mutation) rate of 3% balances that strong convergence. Mutation rate is 1% for a mutation operator that moves a single obstacle by one grid cell (if possible).

5.3 Training

In order to compare the overall performance of the state-of-the-art "random" agent and a SCoE-trained agent, we first need to define a fair evaluation function. The scores/fitnesses (see Equation 4) returned during training cannot be compared directly, since SCoE trains against deliberately harder scenarios and is thus expected to return lower scores. So instead, we defined a test set of 1000 randomly generated scenarios that (most probably) neither agent got to see during training. We evaluate the agents' scores on these scenarios and plot the results for a direct comparison.

Figure 4 shows the direct comparison based on the number of episodes the respective agents where trained on. Note for this plot

Plane	Feature	Description
1	Agent position	The agent's position on the grid
2	Obstacle positions	The positions of the obstacles on the grid
3	Workstation positions	The positions of the workstations on the grid
4	Items for pick-up	The amount of items that can be picked up at the respective positions
5	Item place positions	If the agent carries an item, the positions where this item can be placed at

Figure 3: Description of all feature planes contained in the state input s_t for Q_θ.

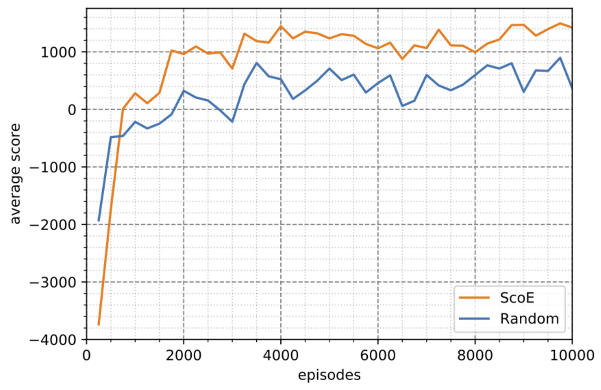

Figure 4: Scores achieved by SCoE and standard "random" reinforcement learning during training over 10000 episodes. Scores are averages of running the current agent against 1000 randomly generated test scenarios.

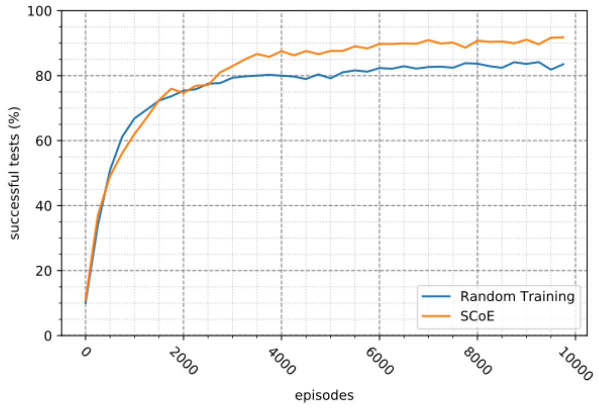

Figure 6: Percentage of successfully solved test scenarios by SCoE and standard "random" reinforcement learning. The values are calculated from a randomly generated set of 1000 scenarios.

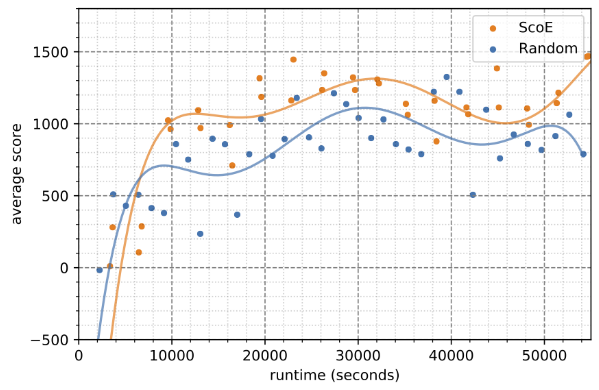

Figure 5: Scores achieved by a SCoE and standard "random" reinforcement learning during training for ≈ 50000 seconds of runtime. Scores are averages of running the current agent against 1000 randomly generated test scenarios. The plot shows single runs with an added trend line. Over the same amount of training time, SCoE generally achieves slighty higher average scores.

we took a snapshot of the agent every 250 episodes, resulting in the horizontal resolution of the plot. While we can see clearly that SCoE outperforms the "random" agent even on randomly generated scenarios, it does have a bit of an unfair advantage: Sampling scenarios randomly obviously takes less computational effort than running several hundred generations of evolution to get the hardest scenarios.

For this reason, we plotted the same data according to runtime in Figure 5 as measured in physical seconds running on a standard computer. The trend line still shows a net benefit of using SCoE with respect to the time-quality trade-off. This means that implementing SCoE and running the elaborate evolutionary process in contrast to just using random sampling for training scenarios actually pays off in performance.

5.4 Test

As stated in the introduction, improving scores is of course a nice benefit, but especially in real-world applications we are often more interested in the agent avoiding complete failures rather than getting the last bits of performance in already good scenarios. The smart factory domain was constructed in such a way that it has a clear overall goal: A sequence of actions is successful iff in the end all items have been fully processed, i.e., have been transported to all the workstations they needed to visit.

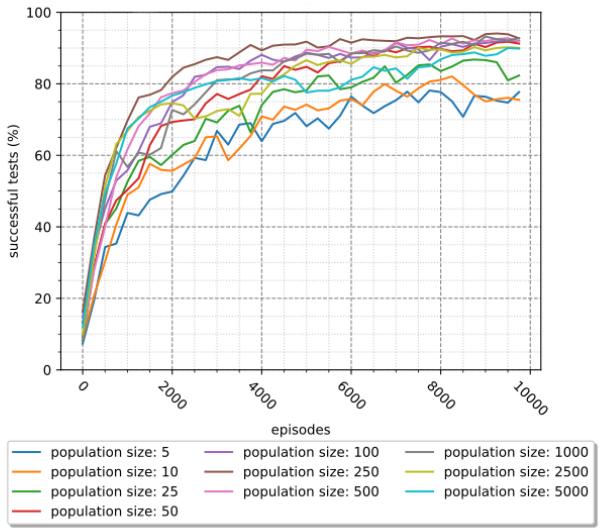

Figure 7: Test success achieved by SCoE for various population sizes. Note that population size equals the batch size for the reinforcement learning agent.

Figure 6 shows the percentage of successful tests among (again) 1000 randomly generated scenarios. These results first verify our choice of a score/fitness function as it apparently aids in learning successful behavior. Note that while reinforcement learning would allow us to simply award the agent a score of +1 or −1 at the end of each episode, depending on whether we consider it to be solved successfully or not, this makes for a very hard reinforcement learning problem. The discipline of constructing a score/fitness function so that it helps to optimize for a different overall objective but still is easy to learn is often known as reward engineering and beyond the scope of this work [14].

On this setup, we also performed an evaluation of the evolutionary parameters population size and generation size, i.e., the amount of generations evolved each time the evolutionary process of SCoE is resumed. Figure 7 shows the test success achieved for various population sizes. While very small population sizes result in considerably lower performance, the difference diminishes beyond 100 and even sizes much higher than 500 do not seem to provide substantial benefit.

A similar picture can be seen in Figure 8 for the evaluation of generation sizes. Note that smaller generation sizes, i.e., less generations of evolution happening between reinforcement learning, not only hinder the optimization for hard scenarios, leading to results quite comparable to the non-SCoE approach in Figure 6 for generations sizes of 5 and 10. Also, they cause the population of scenarios to not change very much during evolution, which means that the reinforcement learning agent continues to train on very similar episodes most of the time, wasting training resources. Again, it is interesting to note that even relatively small generation sizes (like 25 or 50) already result in an advantage over the state-of-art "random" approach (again cf. Figure 6).

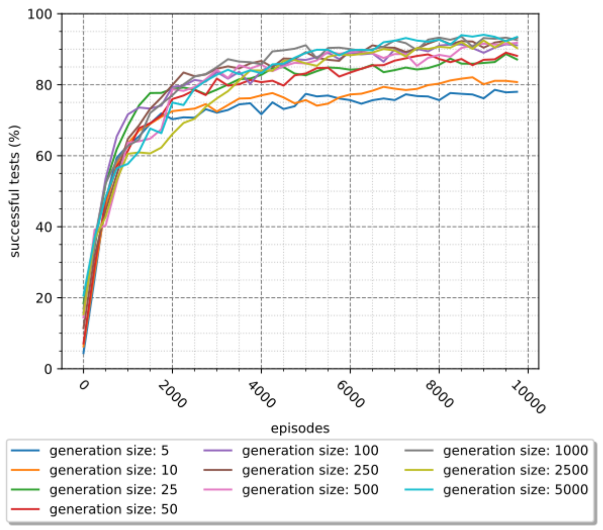

Figure 8: Test success achieved by SCoE for various generation sizes, i.e., the amount of generations computed each time scenarios are evolved.

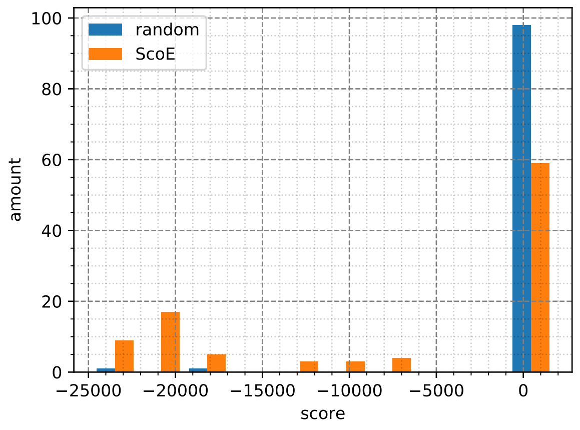

Figure 9: Histogram of scores of a standard "random" reinforcement learning agent on 100 scenarios generated via SCoE compared to 100 scenarios generated at random.

Lastly, we want to verify our claim that the SCoE approach results not only in a better-trained agent but also returns test scenarios that can be used for the testing of any agents, i.e., scenarios that are hard not only for the agent they were evolved against but for agents solving the same domain in general. To this end, we took a set of the 100 hardest scenarios that came out of a SCoE-based training run actually as a by-product and evaluated a state-of-the-art "random" agent's performance on these scenarios. Figure 9 shows the results compared to the results of 100 scenarios generated randomly. As we can see in the histogram, the "random" reinforcement learning agent has a hard time solving the SCoE-generated scenarios, resulting in comparatively many (and radically bad) negative scores.

Note that none of the runs yielding negative scores can be considered successful w.r.t. to the above definition. This shows that the SCoE-generated scenarios are indeed more challenging in general and not overly specialized on the shortcomings of the single agent they were evolved against.

6 CONCLUSION

We augmented reinforcement learning by adding a process of scenario co-evolution (SCoE), a technique that uses evolution to generate hard training scenarios for the reinforcement learning agent instead of using random sampling as it is common practice. While it has been known that biased sampling may aid reinforcement learning and competitive co-evolution for evolutionary processes has been well-known and studied, the specific combination of reinforcement learning with a genetic algorithm with the exact opposite objective function is novel to the authors' knowledge. We found that SCoE not only aids in finding better solutions (i.e., policies) but also aids in finding better solutions per runtime, thus bringing a general benefit for our application. Furthermore, we tested how our approach performs not only measured against the objective function given to it but also against the intended goal of the system designer before translation into an easily learnable objective function. SCoE showed superior performance in both regards.

Finally, we tested the expressiveness of the test scenarios generated as a by-product when applying the SCoE approach. We showed that the scenarios generated during a SCoE-based training of a reinforcement learning agent are not necessarily specialized on that same agent but are much harder than random scenarios for an independently trained agent as well, thus suggesting that SCoE's scenarios can afterwards be used for software testing on the domain in general.

For the evaluation of our approach we introduced and implemented a small grid world domain inspired by the vision of the smart factory. We focused on a single domain as our goal was to show all intricacies and the variety of parameters that need to be minded on the reinforcement learning and the evolutionary algorithm side of the applications. Interestingly, the approach as well as the results can be compared to the also very recent findings of [50] for a co-evolutionary setting without common back-propagation-based reinforcement learning. Naturally, we recognize that the approach calls for a much broader evaluation on a variety of domains. However, the generality of the concepts involved, i.e., both reinforcement learning and evolutionary algorithms being known for their broad applicability (at least each on their own), leads us to suspect similar results can be achieved for other domains.

We would like to point out the following limitations of our current implementation of the SCoE approach and suggest them to be tackled in future work:

- Domain variety: As discussed, transferability of the results needs to be shown. While most intuitive domains lend themselves to parametrized versions (having free parameters for SCoE to optimize), it is still unclear how multiple sources of free parameters should be handled. For example, if our smart factory domain not only had obstacles but also faulty items, should these be optimized by separate evolutionary processes or should we build a single process for a more complex, combined search space?
- Stochasticity: We only showed results for a deterministic domain. While the framework easily allows for stochasticity and preliminary experiments have suggested to us that the approach is robust w.r.t. to domains with non-deterministic transition probability functions, we still require a thorough evaluation if and how SCoE needs to adapted to stochastic domains (which are the common case in real-world applications). The known robustness of evolutionary processes to random effects may be exploitable for SCoE [9].
- Efficiency: At present, the SCoE approach uses independent evaluations when computing the score for training in reinforcement learning and when computing the fitness functions for the individual scenarios. We showed that (at least when scenario evaluations are not all too expensive) SCoE still manages to slightly outperform standard reinforcement learning regarding runtime. However, we suspect that the evaluations could be shared to some extent, further improving the performance of SCoE.
- Diversity: Usually, within a parametrized domain there exist several different archetypes of hard scenarios. Even for our relatively simple smart factory domain with obstacles, we could place obstacles to block off the agent, to block off a workstation or in the middle of an area where most pathways cross. It may be beneficial for the evolution to represent this diversity within each single population as well. Diversity in evolutionary algorithms has been shown to be beneficial in principle for many different domains [20, 47]. The presence of a dynamic fitness function may suggest that SCoE already favors diversity to some extent [21]. However, the exact impact diversity has and could have on the SCoE results still needs to be understood more explicitly.

Of course, this selection of open questions and problems is far from complete. We also suggest that further connections to software engineering processes and software test design as sketched in [22, 50] could be made, for example.

Our results show that the hybridization of different search methods and the deliberate construction of co-evolutionary systems can be a promising endeavor. While these complex, intertwined systems seem hard to control at first, we suggest that approaches like SCoE, bringing part of the control (i.e., testing) into the system, can actually aid the transparency and manageability of traditionally "black box" methods (like reinforcement learning). Eventually, one may hope for a better theoretic and practical understanding of complex systems in the future.

REFERENCES

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016).

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[3] Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*. 5360–5370.

[4] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on.* IEEE, 162–168.
[5] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. 2018. Emergent Complexity via Multi-Agent Competition. In *ICLR.*
[6] Lenz Belzner, Michael Till Beck, Thomas Gabor, Harald Roelle, and Horst Sauer. 2016. Software engineering for distributed autonomous real-time systems. In *Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems.* ACM, 54–57.
[7] Lenz Belzner and Thomas Gabor. 2017. Bayesian verification under model uncertainty. In *Software Engineering for Smart Cyber-Physical Systems (SEsCPS), 2017 IEEE/ACM 3rd International Workshop on.* IEEE, 10–13.
[8] Camillo Bérénos, K Mathias Wegner, and Paul Schmid-Hempel. 2010. Antagonistic coevolution with parasites maintains host genetic diversity: an experimental test. *Proceedings of the Royal Society of London B: Biological Sciences* (2010).
[9] Hans-Georg Beyer. 2000. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Computer methods in applied mechanics and engineering* 186, 2-4 (2000), 239–267.
[10] Joshua Brown, Zhi Quan Zhou, and Yang-Wai Chow. 2018. Metamorphic Testing of Navigation Software: A Pilot Study with Google Maps. In *Proceedings of the 51st Hawaii International Conference on System Sciences.*
[11] Tomas Bures, Danny Weyns, Christian Berger, Stefan Biffl, Marian Daun, Thomas Gabor, David Garlan, Ilias Gerostathopoulos, Christine Julien, Filip Krikava, et al. 2015. Software Engineering for Smart Cyber-Physical Systems–Towards a Research Agenda: Report on the First International Workshop on Software Engineering for Smart CPS. *ACM SIGSOFT Software Engineering Notes* 40, 6 (2015), 28–32.
[12] Fulvio Corno, Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. 2004. Automatic test program generation: a case study. *IEEE Design & Test of Computers* 21, 2 (2004), 102–109.
[13] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II.* Springer, 1–32.
[14] Daniel Dewey. 2014. Reinforcement learning and the reward engineering principle. In *2014 AAAI Spring Symposium Series.*
[15] Jon Edvardsson. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering.* 21–28.
[16] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. 2017. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300* (2017).
[17] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software.* IEEE, 31–40.
[18] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* ACM, 416–419.
[19] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.
[20] Thomas Gabor, Lenz Belzner, and Claudia Linnhoff-Popien. 2018. Inheritance-based diversity measures for explicit convergence control in evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM, 841–848.
[21] Thomas Gabor, Lenz Belzner, Thomy Phan, and Kyrill Schmid. 2018. Preparing for the Unexpected: Diversity Improves Planning Resilience in Evolutionary Algorithms. In *2018 IEEE International Conference on Autonomic Computing (ICAC).* IEEE, 131–140.
[22] Thomas Gabor, Marie Kiermeier, Andreas Sedlmeier, Bernhard Kempter, Cornel Klein, Horst Sauer, Reiner Schmid, and Jan Wieghardt. 2018. Adapting quality assurance to adaptive systems: the scenario coevolution paradigm. In *International Symposium on Leveraging Applications of Formal Methods.* Springer, 137–154.
[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems.* 2672–2680.
[24] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence.*
[25] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence.*
[26] Matthias Hölzl, Nora Koch, Mariachiara Puviani, Martin Wirsing, and Franco Zambonelli. 2015. The ensemble development life cycle and best practices for collective autonomic systems. In *Software Engineering for Collective Autonomic Systems.* Springer, 325–354.
[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[28] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).
[29] Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation.* ACM, 1098–1105.
[30] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.
[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.
[32] Nick Moran and Jordan Pollack. 2018. Coevolutionary Neural Population Models. *arXiv preprint arXiv:1804.04187* (2018).
[33] Jason Morrison and Franz Oppacher. 1999. A general model of co-evolution for genetic algorithms. In *Artificial Neural Nets and Genetic Algorithms.* Springer, 262–268.
[34] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing.* John Wiley & Sons.
[35] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10).* 807–814.
[36] Geoff S Nitschke, AE Eiben, and Martijn C Schut. 2012. Evolving team behaviors with specialization. *Genetic Programming and Evolvable Machines* 13, 4 (2012), 493–536.
[37] Randal S Olson, David B Knoester, and Christoph Adami. 2016. Evolution of swarming behavior is shaped by how predators attack. *Artificial life* 22, 3 (2016), 299–318.
[38] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. 2018. Robust deep reinforcement learning with adversarial attacks. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems.* International Foundation for Autonomous Agents and Multiagent Systems, 2040–2042.
[39] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. 2017. Robust adversarial reinforcement learning. *arXiv preprint arXiv:1703.02702* (2017).
[40] Jordan B Pollack and Alan D Blair. 1998. Co-evolution in the successful learning of backgammon strategy. *Machine learning* 32, 3 (1998), 225–240.
[41] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons.
[42] André Reichstaller, Thomas Gabor, and Alexander Knapp. 2018. Mutation-based test suite evolution for self-organizing systems. In *International Symposium on Leveraging Applications of Formal Methods.* Springer, 118–136.
[43] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).
[44] Arthur L Samuel. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* 3, 3 (1959), 210–229.
[45] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.
[46] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354.
[47] Giovanni Squillero and Alberto Tonda. 2016. Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. *Information Sciences* 329 (2016), 782–799.
[48] Richard S Sutton and Andrew G Barto. 1998. *Introduction to reinforcement learning.* Vol. 135. MIT press Cambridge.
[49] Gerald Tesauro. 1995. Temporal difference learning and TD-Gammon. *Commun. ACM* 38, 3 (1995), 58–69.
[50] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. 2019. Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions. *arXiv preprint arXiv:1901.01753* (2019).
[51] Christopher John Cornish Hellaby Watkins. 1989. *Learning from delayed rewards.* Ph.D. Dissertation. King's College, Cambridge.
[52] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. 2002. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation.* Morgan Kaufmann Publishers Inc., 1233–1240.
[53] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. 2008. Natural evolution strategies. In *IEEE World Congress on Computational Intelligence.* IEEE, 3381–3387.
[54] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.
[55] Chern Han Yong and Risto Miikkulainen. 2001. Cooperative coevolution of multi-agent systems. *University of Texas at Austin, Austin, TX* (2001).

View publication stats