Learning to Evolve

Jan Schuchardt, Vladimir Golkov, Daniel Cremers

Abstract—Evolution and learning are two of the fundamental mechanisms by which life adapts in order to survive and to transcend limitations. These biological phenomena inspired successful computational methods such as evolutionary algorithms and deep learning. Evolution relies on random mutations and on random genetic recombination. Here we show that *learning to evolve*, i.e. learning to mutate and recombine better than at random, improves the result of evolution in terms of fitness increase per generation and even in terms of attainable fitness. We use deep reinforcement learning to learn to dynamically adjust the strategy of evolutionary algorithms to varying circumstances. Our methods outperform classical evolutionary algorithms on combinatorial and continuous optimization problems.

CONTENTS

Authors are with the Department of Informatics, Technical University of Munich, Germany. e-mail: jan.schuchardt@tum.de, vladimir.golkov@tum.de, cremers@tum.de

1 INTRODUCTION

MOST problems in engineering and the natural sciences can be formulated as optimization problems. Evolutionary computation is inspired by the powerful mechanisms of natural evolution. While other methods might get easily stuck when optimizing rugged objective functions, evolutionary algorithms can escape local optima, explore the solution space through random mutation and combine favorable features of different solutions through crossover, all while being simple to implement and parallelize.

Consequently, evolutionary algorithms have been applied to a range of engineering problems, from designing steel-beams [1] and antennas for space-missions [2], to more large-scale problems, like the design of wind parks [3], water supply networks, or smart energy grids.

Evolution and learning are two optimization frameworks that in living systems work at different scales with different advantages. Appropriate combinations of the two provide complementarity, and are a crucial part of the success of living systems. Here we propose new combinations of these optimization principles.

We propose using deep reinforcement learning to dynamically control the parameters of evolutionary algorithms. The goal is finding better solutions to hard optimization problems and facilitating the application of evolutionary algorithms.

Our *deep learning for evolutionary algorithms* is not to be confused with *evolutionary algorithms for deep learning*, such as neuroevolution [4] or population-based training [5].

This section provides a brief explanation of the used terms and definitions. Section 2 provides an overview of previous work dedicated to enhancing evolutionary algorithms through reinforcement learning. Section 3 explains how we aim to do away with the shortcomings of previous work, as well as the experimental setup used to provide initial evidence of the feasibility of our approach. Section 4 is dedicated to the results of our experiments and their discussion. Section 5 provides high-level conclusions to our experimental results.

1.1 Evolutionary Computation

Evolutionary computation is an umbrella term for optimization methods inspired by Darwinian evolutionary theory. In natural evolution, individuals strive for survival and reproduction in a competitive environment. Those with more favorable traits, acquired through inheritance or mutation, have a higher chance of succeeding.

Evolutionary algorithms are specific realizations of the concept of evolutionary computation. Evolutionary algorithms solve computational problems by managing a set (*population*) of *individuals*. Each individual encodes a candidate solution to the computational problem in its *genome*. To explore the solution space, *offspring* is generated from the *parent population* through *recombination operators* that combine properties of the parents. Additionally, *mutation operators* are applied to introduce random variations with the goal of enhancing exploration and preventing premature convergence. A new population is created by selecting a set of individuals from the parent population and from the offspring. This process of *recombination*, *mutation* and *survivor selection* comprises one *generation* and is repeated multiple times throughout a run of the algorithm. To guide the process towards better solutions, evolutionary pressure is applied through a *fitness function*. Fitter individuals are given a higher chance of reproducing, surviving, or both. Due to its well parallelizable design and their suitability for solving high-dimensional problems with a complex fitness landscape, evolutionary computation is a valuable tool in engineering applications or other domains where classical optimization methods fail or no efficient exact solver is available. Fig. 1 shows the data flow of evolutionary algorithms, and the role we propose therein for reinforcement learning.

1.2 Adaptation in Evolutionary Computation

A key problem in the application of evolutionary algorithms is selecting evolution parameters. Even simple implementations have a considerable number of parameters.

The choice of parameter values has a considerable impact on the performance of an evolutionary algorithm for different problems and even different problem instances. Furthermore, utilizing fixed parameters over the course of all generations can be sub-optimal, as different stages of the search process might have different requirements.

To account for this, it is desirable for evolutionary algorithms to be adaptive. In this context, *adaptation* refers to dynamic control of evolution parameters (not to be confused with the biological term *adaptation*).

The following taxonomy, taken from Ref. [6], describes the different levels on which adaptation can be used in evolutionary algorithms.

1) *Environment-Level Adaptation* changes the way in which individuals are evaluated by the environment, for example by altering the fitness function.
2) *Population-Level Adaptation* modifies parameters that affect the entirety or some subset of the population, for example by changing the population size.
3) *Individual-Level Adaptation* makes parameter choices for specific individuals in the population, for example by increasing the mutation probability of individuals with a low fitness value.
4) *Component-Level Adaptation* changes parameters that are specific to a certain part of an individual's genome, for example by managing per-gene mutation probabilites.

In Section 3.4, we propose adaptation methods for each of these levels of adaptation.

1.3 Deep Reinforcement Learning

Artificial neural networks are a popular machine learning technique and connectionist model, inspired by neurobiology.

A single neuron is parameterized by weights that model how its inputs relate to its output. By combining multiple neural layers that perform (nonlinear) data transformations, highly complicated functions can be approximated. In order to model a mapping with desired properties, the network weights are modified to minimize a loss function. This minimization is usually implemented using some form of gradient descent.

Artificial neural networks have seen a surge in popularity in the past few years and have been successfully applied in a variety of fields, like computer vision, natural language processing, biology, medicine, finance, marketing and others.

Reinforcement learning is an area of artificial intelligence concerned with training a learning agent by providing rewards for the actions it takes in different states of its environment. The ultimate goal is for the agent to follow a policy that maximizes these rewards.

The key limiting factor in the application of older reinforcement learning methods is the complexity of representing policies for large state and action spaces. Deep reinforcement learning is the idea of using artificial neural networks as function approximators that replace tabular or other representations used in classic reinforcement learning algorithms. One additional benefit of deep reinforcement learning is that it made the use of reinforcement learning for continuous control feasible (see, for example [7]).

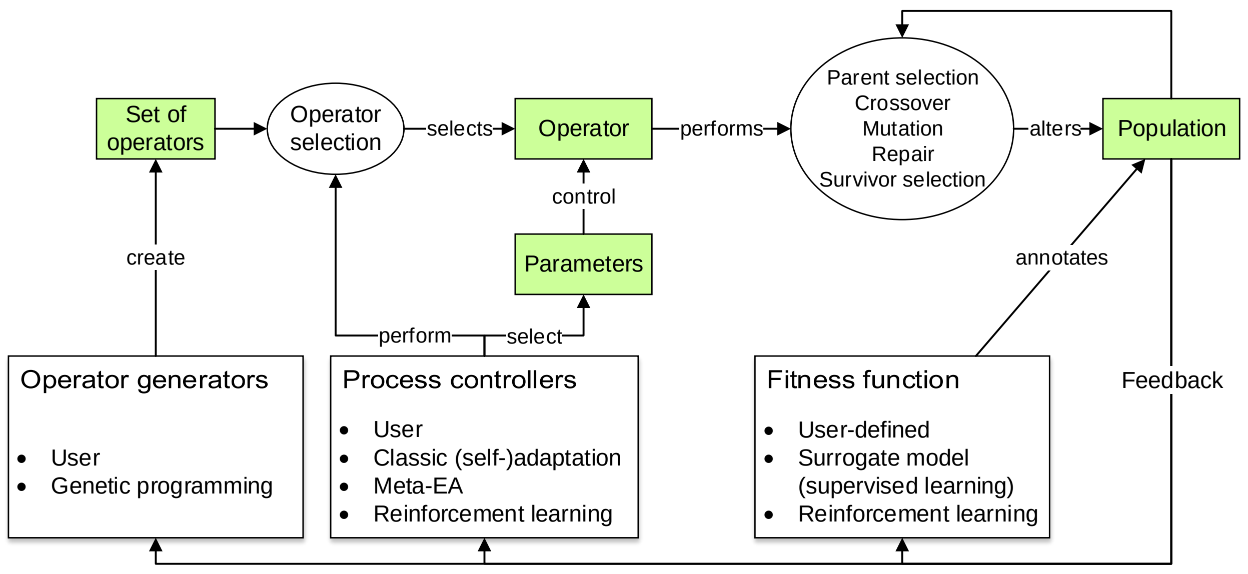


Fig. 1: Data flow in evolutionary algorithms. Operator generation, process control and fitness estimation can either be entirely predefined by the user, or follow some algorithmic approach and receive feedback from the population. We propose using deep reinforcement learning instead of classical process controllers or fitness estimators.

2 RELATED WORK

To our knowledge, there has been no previous work on the application of deep reinforcement learning to evolutionary computation. However, there have been several publications on using classic reinforcement learning techniques for adaptation in evolutionary algorithms.

Most previous work has been concerned with population-level adaptation. In 2002, Müller et al. enhanced a $(1+1)$ evolution strategy (i.e. an evolutionary algorithm for continuous optimization with population size 1) by controlling the step-size (i.e. standard deviation) through reinforcement learning [8]. Later work [9], [10] extended the use of reinforcement learning to the simultaneous control of multiple numerical evoluton parameters. Aside from this, reinforcement learning has also been used to dynamically select from a set of available evolutionary operators [11], [12]. Techniques for multi-armed bandits (i.e. reinforcement learning with a single state) have also been utilized for this purpose, both in single-objective [13] and multi-objective optimization [14].

Reinforcement learning has also been successfully applied to environment-level adaptation. In their 2011 paper [15], Afanasyeva and Buzdalov used reinforcement learning to select from a set of handcrafted auxiliary fitness functions that can be added to the main objective function, in order to reshape the fitness landscape. This approach was later expanded on to deal with non-stationary problems, in which the objective function changes over time [16].

As individual- and component-level adaptation requires larger action spaces, with which classic reinforcement learning algorithms struggle, there has been little research into learning such strategies through reinforcement learning. The only paper we were able to find used reinforcement learning to control two numerical evolution parameters per individual in the local search strategy of a memetic algorithm [17] (i.e. a combination of an evolutionary algorithm with a local search strategy). To our knowledge, there has been no previous work on reinforcement-learning based component-level adaptation.

A limiting factor in all of these approaches is that they employ reinforcement learning methods like Q-Learning [18] that represent the policy of the learning agent in a discretized fashion. Practical application of these older reinforcement learning methods is limited to learning low-dimensional mappings with a small number of state-action pairs.

Consequently, only a small, coarsely discretized subset of the potentially useful information about the optimization problem and the state of the evolutionary algorithms is used. Likewise, the action space is discretized coarsely, even though many of the controlled parameters are continuous in nature. There has been an attempt to address the problem of action space discretization by dynamically adapting the discretization bins [19], but this does resolve the problem that the underlying reinforcement learning algorithm is ill-suited to continuous control.

It should also be noted that all aforementioned work is concerned with learning on the fly (i.e. during the execution of an evolutionary algorithm) for a specific problem instance, whereas our method is designed to learn over the course of multiple runs of the evolutionary algorithm, as explained in the next section.

3 METHODS

To do away with the limitations of older approaches (see the end of Section 2), we propose using deep reinforcement learning to learn adaptation strategies for evolutionary algorithms.

The novelties of our approach include:

- Learning adaptation strategies for an entire problem class, instead of optimizing for a specific problem instance
- Using more information about problem instances and the state of the evolutionary algorithm
- Utilizing modern deep reinforcement learning techniques in order to
  operate in large, continuous state and action spaces.
  This allows us to:
  - Learn complex adaptation strategies
  - Entirely replace hand-crafted components of an evolutionary algorithm (e.g. parent selection) with learned strategies

While many other use cases are possible (see Section 5), we limit ourselves to learning adaptation that generalizes to previously unseen problem instances, using only a limited number of instances of the same problem class for training, and always running the evolutionary algorithms for a fixed number of generations. Within these constraints, we consider two distinct use cases:

1) The time/resources for training are large. In this case, the user can account for possible instabilities of the training process by selecting the best out of multiple trained agents.
2) The time/resources for training are limited, only allowing for the training of one or very few agents. In this case, it is important that the average performance of trained agents is high and the variance in performance among them is low, so that the user is likely to arrive at a good solution within the limitations of this use case.

The rest of the Methods section is structured as follows. We first explain our used reinforcement learning approach (Section 3.1) and reward function (Section 3.1.3). We then define three benchmark problem sets (Section 3.2) and basic evolutionary algorithms that can be used to optimize these problems (Section 3.3). Next, we propose different trainable adaptation methods to enhance these evolutionary algorithms (Section 3.4) and specify the neural network architecture used for performing the underlying calculations (Section 3.5). Finally, we define the experimental setup and performance metrics used for evaluating the different proposed adaptation methods (Section 3.6).

We will release the code at https://github.com/jan-schuchardt/learning-to-evolve.

3.1 Choice of Reinforcement Learning Algorithm

To allow us to perform both discrete and continuous actions – depending on the application – we propose the use of so-called stochastic policy gradient methods, which take actions by sampling from a probability distribution, parameterized by a neural network. Most state-of-the-art deep reinforcement learning algorithms fall into this category.

In this section, we first provide a more formal definition of reinforcement learning (Section 3.1.1), before explaining the specifics of our used reinforcement learning algorithm (Section 3.1.2).

3.1.1 Basics of Reinforcement Learning

Reinforcement learning is a field of study concerned with training intelligent agents through rewards or penalties, based on actions taken in an environment.

Reinforcement learning problems are typically specified as a Markov decision process, defined by:

- a set S of states,
- an initial state s_0 or a probability distribution $p(S_0)$ over a set S_0 of initial states,
- a set A of actions,
- the transition function $\mathcal{P}_a(s'|s)$, which describes the probability of reaching state s' from state s by taking action a,
- the reward function $\mathcal{R}_a(s', s)$ which assigns scalar rewards (larger is better) to a state transition,
- the Markov property $P(s_{t+1}|s_t, s_{t-1}, \dots, s_0) = P(s_{t+1}|s_t)$, meaning that the state transitions at time t are independent of the prior sequence of states.

The goal of reinforcement learning is to learn a policy $\pi : S, A \to [0, 1]$ that describes a probability distribution over actions, given a state. The learning has to be achieved solely based on the observed rewards and state transitions, without prior knowledge of the environment. The policy should maximize the expected value of some reward-based return function R. A typical choice is the discounted sum of accumulated rewards:

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,$$

with r being the sequence of received rewards, and $\gamma \in [0, 1)$ being a discount factor that ensures convergence of the series. A smaller γ means that short-term rewards are favored over long-term rewards.

3.1.2 Proximal Policy Optimization

Proximal policy optimization [20] is a stochastic policy gradient method that aims to enhance training stability by using trust-region optimization of the policy.

In stochastic policy gradient methods, the gradient of expected future rewards with respect to the parameters θ of a stochastic policy π_θ is used for learning. Each state is mapped to a probability distribution over actions. An action is selected by sampling from this probability distribution.

Proximal policy optimization uses the following clipped loss function for training:

$$\mathcal{L}_{\text{clip}} = -\mathbb{E}[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), [1-\varepsilon, 1+\varepsilon])\hat{A}_t)], \quad (1)$$

$$\text{with } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \quad (2)$$

where $\mathbb{E}$ is the average over a set of training samples, $\pi_\theta(a_t|s_t)$ is the probability of performing action a_t in state s_t under the probability distribution described by the policy π_θ, θ_{old} are the parameters of the policy during collection of the training samples and θ are the parameters of the policy as it undergoes optimization. The advantage estimator $\hat{A}_t$ describes how much higher than expected the reward for following action a_t at time t was. The clipping function $\text{clip}(r_t(\theta), [1-\varepsilon, 1+\varepsilon])$ maps $r_t(\theta)$ to the interval $[1-\varepsilon, 1+\varepsilon]$, i.e. $\text{clip}(r_t(\theta), [1-\varepsilon, 1+\varepsilon]) := \min\{\max\{r_t(\theta), 1-\varepsilon\}, 1+\varepsilon\}$. The benefit of this formulation is that it does not encourage increasing the probability of an advantageous a_t ($\hat{A}_t > 0$) or decreasing the probability of a worse-than-expected a_t ($\hat{A}_t < 0$) by more than ε, thus stabilizing the learning process and allowing for training samples to be reused without perturbing the policy, which increases sample efficiency.

Algorithm 1 Proximal policy optimization for multiple problem instances

```
for iteration=1,2, ... ,#iterations do
    for problem_instance=1,2, ... , $K$ do
        for actor=1,2, ... ,$N$ do
            Run policy $\pi_{\theta_{\text{old}}}$ for $T$ timesteps
            Calculate $\hat{A}_1, \hat{A}_2 \dots, \hat{A}_T$
            Store training samples
        end for
    end for
    for epoch=1,2, ... ,#epochs do
        Optimize clipped loss on samples w.r.t. $\theta$, using minibatch size $M \leq KNT$
    end for
    $\pi_{\theta_{old}} \leftarrow \pi_\theta$
    Discard training samples
end for
```

Algorithm 1 specifies how we use proximal policy optimization to optimize a policy for multiple problem instances. In each training iteration, the evolutionary algorithm is applied to each problem instances for a fixed number of times, in order to gather samples for subsequent training.

3.1.2.1 Advantage Estimation: We use generalized advantage estimation (see [21]) for calculating the advantage estimate $\hat{A}$ while ensuring a good trade-off between variance and bias.

Assuming an estimator $\hat{V}(s_t)$ (*value function*) of the discounted future rewards $V(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$, the advantage for a *trajectory* (i.e. sequence of state transitions, actions and rewards) of length T is calculated as an exponential moving average over temporal differences:

$$\hat{A}_t = \sum_{i=0}^{T-t+1} (\gamma\lambda)^i \delta_{t+i}, \tag{3}$$

$$\text{with } \delta_t = \gamma \hat{V}(s_{t+1}) + r_t - \hat{V}_t(s_t), \tag{4}$$

where the parameter $\lambda \in [0, 1]$ controls the trade-off between variance and bias of the advantage estimate. With higher λ, the sequence of rewards is given a higher weight, thus reducing the bias caused by the estimate $\hat{V}(s_t)$. However, the variance of the estimate increases with T, due to the randomness of the underlying Markov decision process.

3.1.2.2 Time-Awareness: In our approach, the evolutionary algorithm is run for a limited number of generations. Consequently, learned adaptation methods should try to maximize the fitness within this given time frame.

For simplicity, we treat the entire run of the evolutionary algorithm as a single episode of length T. To account for the time-limited nature of the environment, we make the following adjustments, based on [22]:

1) We enforce $\hat{V}(s_T) = 0$ in the generalized advantage estimation, as no further rewards can be gathered after the episode has ended.
2) We add a relative encoding of the remaining number of generations, $(T - t)/T$, to the state. This way, the policy can account for optimization problems in which the potential for gathering rewards might be considerably higher at the earlier state and adapt its behavior accordingly. By using a relative encoding, we scale $T-t$ into the $[0, 1]$ range, with the goal of better generalization when applying the evolutionary algorithm under varying T.

3.1.2.3 Actor-Critic Framework: There are a variety of ways to calculate the value approximator $\hat{V}(s_t)$, one of which is approximating V through a neural network with parameters θ_c. We optimize $\hat{V}_{\theta_c}$ by minimizing

$$\mathcal{L}_V = \mathbb{E}\left[\left\|\hat{V}_{\theta_c}(s_t) - \sum_{i=0}^{T-t} \gamma^i r_{t+i}\right\|^2\right]. \tag{5}$$

This neural network is typically referred to as a *critic*, rating the actions taken by the *actor* π_θ. Since both operate in the same environment, it is common practice to merge them into one network and only keep two separate output layers, so that they can operate on shared lower-level features. In this case, the loss for the entire network is $\mathcal{L}_{\text{clip}} + \alpha_v \mathcal{L}_V$, with the hyper-parameter α_v controlling the ratio between the actor and critic losses.

3.1.2.4 Entropy-Based Exploration: In order to learn a good policy and avoid bad local optima, it is vital to explore a variety of actions and states. To this end, the negative information-theoretical entropy $S[\pi_\theta]$ of π_θ can be added to the loss function [23]. By minimizing this term (maximizing the entropy), actions are taken with less certainty, thus discouraging premature convergence to local optima. This leads to the complete loss function $\mathcal{L} = \mathcal{L}_{\text{clip}} + \alpha_v \mathcal{L}_V + \alpha_e S[\pi_\theta]$, with the exploration-controlling coefficient α_e.

3.1.3 Reward Calculation

The goal of a reinforcement learning algorithm is to maximize rewards over the course of multiple state transitions (see Section 3.1.1), while the goal of an evolutionary algorithm is to find a solution of maximum fitness over the course of multiple generations (see Section 1.1). To unify these two goals, we associate the state s_t with the population of the evolutionary algorithm in generation t.

Let $f_{\max}(s_t)$ be a function that returns the fitness value of the fittest individual in the population associated with s_t. On a generation-to-generation level, the goal of finding a solution of maximum fitness then translates to maximizing the ratio $f_{\max}(s_{t+1})/f_{\max}(s_t)$.

Multiple smaller improvements should have the same effect as one large improvement that leads to the same final solution. We therefore define the reward function as

$$\mathcal{R}_a(s_t, s_{t+1}) = \alpha_r \log_{10} \frac{f_{\max}(s_{t+1})}{f_{\max}(s_t)}, \tag{6}$$

assuming positive fitness functions. We use the coefficient α_r to scale rewards approximately into the range $[-1, 1]$. The logarithm is taken, so that the sum of rewards over one run of the evolutionary algorithm equals the logarithm of the ratio between the initial and terminal fitness, $\alpha_r \log_{10} \frac{f_{\max}(s_T)}{f_{\max}(s_0)}$.

3.2 Benchmark Problems

Three different problem classes are used to investigate the usefulness of different reinforcement learning adaptation mechanisms. We use the 0-1 knapsack problem and the traveling salesman problem as examples for combinatorial optimization, and a set of two-dimensional objective functions as examples for continuous optimization. As explained in the beginning of Section 3, we only use a limited number of problem instances for training.

This section gives a brief explanation of the different optimization problems and how we define their respective fitness functions.

3.2.1 0-1 Knapsack Problem

An instance of the 0-1 Knapsack Problem is defined by a weight limit $w_{\max}$ and a set I of n items: $I = \{(w_i, v_i) \mid w, v \in \mathbb{R}, i \in [n]\}$, with weights w and values v. The optimization objective is

$$\max_{S \subseteq I} \sum_{(w,v) \in S} v \quad \text{subject to} \sum_{(w,v) \in S} w < w_{\max}. \tag{7}$$

For training, we generated 20 training instances with $w_{\max} = 10$, with weights and values uniformly sampled from $[0, 1]$, and ten more instances for validation.

3.2.2 Traveling Salesman Problem

The traveling salesman problem is another type of combinatorial optimization problem. We consider the case of finding a Hamiltonian cycle of maximum weight within a fully connected, weighted, undirected graph. We formulate it as a maximization problem (this is equivalent via a transformation of edge weights to the common formulation as a minimization problem).

For training, we use 40 different graphs with weights uniformly sampled from $[0, 1]$. For evaluation, another 10 problem graphs are used.

3.2.3 Continuous Function Optimization

For continuous optimization, nineteen standard benchmark $\mathbb{R}^2 \mapsto \mathbb{R}$ objective functions, as defined in Ref. [24], are used. The goal in each case is to find the global minimum. The Ackley function, Beale function and Levy function #13 are used for validation. The following functions are used for training: Rastrigin, Rosenbrock, Goldstein–Price, Bukin #6, Matyas, Cross-in-Tray, Eggholder, Holder, McCormick, Schaffer #2, Schaffer #4, Styblinski-Tang, Sphere, Himmelblau, Booth, Three-Hump Camel. While the Beale function is plateau-shaped, except for its steep borders, the Ackley function and the Levy function #13 are highly rugged with a considerable number of local optima.

For data normalization purposes, we rescale and translate the functions so that their domain is $[-1, 1] \times [-1, 1]$ and subtract their minimum value. Obviously, this normalization is only possible because the minimum value is already known. While this is not representative of real-world problems, it is still sufficient for investigating whether evolutionary algorithms with deep reinforcement learning can be applied in continuous problem domains at all.

3.3 Baseline Evolutionary Algorithms

To solve the three types of benchmark problems, we use baseline evolutionary algorithms, which we shall later enhance through deep reinforcement learning.

The following paragraphs give a brief explanation of the specifics of these baseline algorithms, their configurable parameters, and how the fitness of individuals is defined.

3.3.1 Baseline Algorithm for the 0-1 Knapsack Problem

In the evolutionary algorithm used for the knapsack problem, solutions are encoded as binary vectors. Fitness is defined as the sum of weights of the selected items.

The initial population is created by randomly generating binary vectors with equal probability for 0 and 1. To ensure that the weight limit is not exceeded, items are randomly removed from invalid candidate solutions until the constraint is fulfilled.

Parent selection is performed through *tournament selection* with tournament size 2. In a tournament, two individuals are randomly taken from the population and the fitter one is selected as parent. Two tournaments are performed for each pair of parents. The winner of the first tournament does not participate in the second one, but can again be selected in any future pair of tournaments.

Recombination is performed through uniform crossover. With a probability of $1 -$ `crossover_rate` the parents are directly copied into the offspring generation. Else, two children are created by combining the parent genomes. For each gene (i.e. entry of the binary vector), there is a 50% chance of child 1 inheriting from parent 1 and child 2 inhering from parent 2. Else, child 1 inherits from parent 2 and child 2 inherits from parent 1.

All children then undergo mutation. Each bit is flipped with a probability of `mutation_rate`.

Survivor selection is performed using an elitism mechanism, which ensures that the fitness of the best individual in a population never degrades. The `elite_size` fittest individuals from the parent population and the `population_size` $-$ `elite_size` fittest offspring individuals are selected for survival into the next generation.

3.3.2 Baseline Algorithm for the Traveling Salesman Problem

In the evolutionary algorithm used for the traveling salesman problem, integer-valued genes are used. For a graph with n nodes, a solution is encoded as a permutation $(a_0, a_1, \ldots, a_{n-1})$ of $(0, 1, \ldots, n-1)$. The fitness of a solution is calculated as

$$\sum_{i=0}^{n-1} w_{a_i, a_{i+1 \bmod n}}, \tag{8}$$

where $w_{i,j}$ is the weight of the edge between nodes i and j. The initial population is a set of random permutations.

Like in the evolutionary algorithm for the knapsack problem, parents are chosen via tournament selection (see Section 3.3.1). The different crossover operators (described below) only generate one child for each pair of parents, so twice the number of tournaments have to be performed for the same population size.

We use the traveling salesman problem to evaluate the ability of a reinforcement learning agent to select from a set of different operators. To this end, we employ the following seven crossover operators: *one-point crossover*, *two-point crossover*, *linear-order crossover*, *cycle crossover*, *position-based crossover*, *order-based crossover*, and *partially mapped crossover*, as explained in Ref. [25]. Depending on the crossover operator, children inherit sub-paths, the relative order of nodes, the position of nodes, or a combination thereof, from their parents. The probability of performing crossover, instead of directly copying the parents into the offspring population, is defined by the `crossover_rate` parameter.

Mutation is performed through inversion, as follows. Each child is mutated with a probability defined by `mutation_rate`. If the child is mutated, two positions in its genome are randomly chosen. The order of all the genes between these two positions is then inverted.

Survivor selection is performed with the same elitism mechanism used for the 0-1 knapsack problem.

3.3.3 Baseline Algorithm for Continuous Function Minimization

The evolutionary algorithm for minimization of $\mathbb{R}^2 \to \mathbb{R}$ functions represents candidate solutions as real-valued vectors. For self-adaptive mutation, each genome also encodes a positive, real-valued step-size υ. The fitness of a solution (x_1, x_2) evaluated on a function g is defined as

$$1/\max(g(x_1, x_2), 10^{-20}). \tag{9}$$

Taking the reciprocal value turns the minimization into a maximization problem, so that our definitions from previous sections are consistent across all problem classes. The max-operator prevents problems with floating point calculations.

The initial population is generated by uniformly sampling from the function domain. The step-size of each individual is first set to `initial_step_size`.

Evolutionary pressure is induced by only selecting the `parent_percentage`×`population_size` fittest individuals as a set of parents for mutation. No crossover operator is used.

For each offspring individual, a parent from the parent set is randomly selected and then mutated through one-step self-adaptive mutation, as follows: First, the step-size υ_i of individual i is multiplied with $\max(\upsilon_i, \texttt{min\_step\_size})$, where υ_i a sample from the log-normal distribution $e^{\mathcal{N}(0,\tau)}$, with self-adaptation strategy parameter τ. Then, a sample from $\mathcal{N}(0, \upsilon_i)$ is taken for each gene and added onto the current value. If mutation leads an individual to leave the function's domain, it is re-initialized at a uniformly sampled random coordinate, and υ is reset to `initial_step_size`.

The same elitism mechanism as in the other baseline algorithms is used for survivor selection.

3.4 Adaptation Methods

Now that we have established the baseline algorithms, we propose different ways of enhancing them through reinforcement learning. Each of the proposed adaptation methods replaces or enhances one component of the evolutionary algorithm (parent selection, crossover, mutation, or survivor selection, as explained in Section 1.1). To show the range of possibilities for applying deep reinforcement learning to evolutionary algorithms, we propose methods for all levels of adaptation explained in Section 1.2).

Recall that our reinforcement learning algorithm learns a stochastic policy (see Section 3.1), meaning that actions are taken by sampling from a probability distribution conditioned on the neural network's parameters θ and its input. We use the following probability distributions, which have different definition domains and are therefore useful for taking different types of actions:

- Bernoulli trials are used for discrete binary actions, as sampling from them returns either 0 or 1. The neural networks outputs a probability $p_\theta \in [0, 1]$ to parameterize the distribution. We use Bernoulli trials to:
  - select subsets of the population as parents (Section 3.4.3.4),
  - decide which bits should be mutated in the evolutionary algorithm with binary encoding. (Section 3.4.4.1)
- Beta distributions can be used for real-valued, constrained policies, as proposed in Ref. [26]. Sampling from them yields a number between 0 and 1. For a unimodal beta distribution, the neural network has to output two scalars $\alpha_\theta, \beta_\theta \in (1, \infty)$. We use beta distributions to:
  - control the mutation rate (explained in Section 3.3.1) for the entire population (Section 3.4.2.1),
  - control the mutation rate of each individual separately (Section 3.4.3.1).
- Categorical distributions are useful for selecting a single action from a finite set of k discrete actions. The distribution is parametrized by probabilities $(p_\theta)_i$ for each element i to be selected. We use a categorical distribution to:
  - select from a set of different crossover operators (listed in Section 3.3.2) on the fly (Section 3.4.2.3).
- Normal distributions are utilized for real-valued, unbounded actions. Normal distributions are parameterized by a mean $\mu_\theta \in \mathbb{R}$ and a standard deviation $\sigma_\theta \in \mathbb{R}_+$. We use normal distributions to:
  - alter (multiplicatively) the fitness of individuals to influence the selection of parents (Section 3.4.1.1),
  - alter (overwrite) the strategy parameter τ (explained in Section 3.3.3) for the entire population (Section 3.4.2.2),
  - alter (overwrite) the strategy parameter τ separately for each individual (Section 3.4.3.2),
  - alter (overwrite) the step-size υ (explained in Section 3.3.3) of each individual in the population (Section 3.4.3.3),
  - alter (overwrite) step-sizes for each gene of each individual (Section 3.4.4.2)
  - alter (overwrite) the fitness value of individuals to influence survivor selection, in order to select a fixed number of survivors (Section 3.4.1.2).

In some cases, we apply a function (exp or softplus) to the samples from a normal distribution. Note that in the context of the proximal policy optimization algorithm, we treat the sample from the normal distribution as the action. The subsequent transformations are part of executing the action and are not considered in the gradient calculation.

The following sections explain the details of how these probability distributions are used by the different adaptation methods, on an implementation-independent level. In Section 3.5 we then define how the neural network that controls the distributions operates, how its inputs are encoded and how the constraints on its output domains are enforced.

Note that sampling from a random distribution parameterized by the neural network means that the network uses information about the current situation (see Section 3.5), i.e. the randomness is intelligently constrained rather than arbitrary.

3.4.1 Environment-Level Adaptation

On the environment level, we let an agent alter or replace the fitness function without using handcrafted auxiliary functions. Altering the fitness landscape could allow for more diverse populations, which could help in exploring more of the solution space.

3.4.1.1 Fitness Shaping: In fitness shaping, we sample a vector $\varepsilon \in \mathbb{R}^{\texttt{population\_size}}$ from a set of `population_size` normal distributions parameterized by the neural network, and multiply it elementwise with the population's fitness values, before applying the parent selection mechanism of the baseline algorithm. On the continuous problem set, we multiply fitness values with $\exp(\varepsilon)$, as the difference in fitness values is typically much larger.

3.4.1.2 Survivor Selection: In survivor selection, we assign each individual from the parent and offspring population a fitness value by sampling from $2 \cdot$ `population_size` independent normal distributions parameterized by the neural network. We then select the `population_size` individuals with the highest fitness value for survival. Unlike in fitness shaping, the learned fitness function does not merely alter the objective function, but replaces it entirely.

3.4.2 Population-Level Adaptation

On the population level, we propose two methods that dynamically control the mutation rate / strategy parameter of the baseline evolutionary algorithms. This could – for example – enable a coarse-to-fine approach to optimization, in which the amount of mutation decreases over time. We also propose a method for selecting from the set of crossover operators for the traveling salesman problem, which could allow the evolutionary algorithm to explore along better trajectories in the solution space, as different operators let children inherit different features from their parents. These methods work as follows:

3.4.2.1 Mutation Rate Control: To control the mutation rate on the population level, we sample a value $\in [0, 1]$ from a single beta distribution parameterized by the neural network.

3.4.2.2 Strategy Parameter Control: To control the strategy parameter of the evolutionary algorithm for continuous optimization on the population level, we sample a value $\tau' \in \mathbb{R}$ from a single normal distribution parameterized by the neural network, and use the $\mathrm{softplus}$ nonlinearity to calculate the positive-valued strategy parameter $\tau = \mathrm{softplus}(\tau') = \log(1 + e^{\tau'})$.

3.4.2.3 Operator Selection: To select from the set of available crossover operators for the traveling salesman problem, we sample from a categorical distribution parameterized by the neural network (where each category corresponds to a crossover operator).

3.4.3 Individual-Level Adaptation

The first two methods for individual-level adaptation use the same continuous mutation parameters as on the population level, but control them separately for each individual. Next, we propose an alternative way of controlling self-adaptation in evolution strategies. Controlling mutation per individual could increase the capability of the evolutionary algorithm to deal with diverse populations, for example by mutating low-fitness individuals more. Finally, we introduce a way of letting a learning agent directly perform the parent selection processes of an evolutionary algorithm. This could allow us to guide the population through the fitness landscape more deliberately than the baseline methods do. These methods work as follows:

3.4.3.1 Mutation Rate Control: To control the mutation rate per individual, we sample from `population_size` independent beta distributions parameterized by the neural network.

3.4.3.2 Strategy Parameter Control: To control the strategy parameter per individual, we sample values $(\tau'_1, \ldots, \tau'_{\texttt{population\_size}})$ from a set of independent normal distributions parameterized by the neural network, and then calculate the strategy parameter for individual i as $\tau_i = \mathrm{softplus}(\tau'_i)$.

3.4.3.3 Step-Size Control: Instead of controlling strategy parameters to indirectly influence the evolution of step-sizes,

the step-size control method lets the neural network output multipliers for the step-sizes more directly. To do so, the step-size υ_i of individual i is changed multiplicatively via $\upsilon_i \leftarrow \mathrm{softplus}(\xi_i)\upsilon_i$, where ξ_i is a sample from a normal distribution parameterized by the neural network. The step-sizes are then used to mutate the genes of the individuals, as in the baseline algorithm.

3.4.3.4 Parent Selection: To select a subset of parents, we sample a binary vector $x \in \{0, 1\}^{\texttt{population\_size}}$ from a set of independent Bernoulli distributions parameterized by the neural network. Iff $x_i = 1$, individual i becomes a parent candidate for the offspring population.In the evolutionary algorithm for the knapsack problem, we let the agent perform a pre-selection of parent candidates, and then apply the baseline parent selection method to create pairings. In the evolutionary algorithm for continuous optimization there is no parent-pairing step, so this adaptation method directly controls which parents produce offspring.

3.4.4 Component-Level Adaptation

The last class of proposed adaptation methods is component-level adaptation. We propose a method for mutating binary genes and a method for mutating real-valued genes. Component-level mutation could allow the agent to

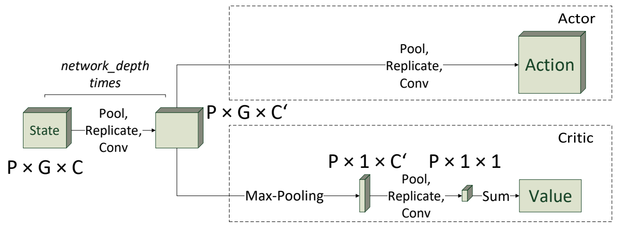


Fig. 2: The overall neural network architecture. P, G and C are the population size, genome size and the number of channels, respectively. The dimensionality of the output action can be further reduced through max-pooling, depending on the adaptation method. Actor and critic operate on the same low-level features extracted by the "Pool, Replicate, Conv" substructure visualized in Fig. 3.

more directly control the direction in which individuals move through the solution space.

3.4.4.1 Binary Mutation: To directly control binary mutation, we sample a matrix $\in \{0,1\}^{\texttt{population\_size} \times \texttt{genome\_size}}$ from independent Bernoulli distributions parameterized by the neural network. Each element corresponds to a gene in one specific individual of the population. If an entry of the matrix is 1, the gene value is inverted.

3.4.4.2 Step-Size Control: For component-level adaptation in the evolutionary algorithm for continuous optimization, we assign each individual i a vector $(v_{i,1}, \ldots, v_{i,\texttt{genome\_size}})$ of step-sizes. These step-sizes are multiplicatively mutated via $v_{i,j} \leftarrow \text{softplus}(\xi_{i,j})v$, where $\xi_{i,j}$ is a sample from a normal distribution parameterized by the neural network. Each solution-encoding gene k of individual i is then mutated by adding a value sampled from $\mathcal{N}(0, v_{i,k})$, similarly to the baseline algorithm. Through this mechanism, offspring is sampled from a multivariate Gaussian distribution with a diagonal covariance matrix. Alternatively, this can be interpreted as a trainable diagonal preconditioner, learning to rescale the fitness landscape around each parent individual to facilitate optimization. This could allow the evolutionary algorithm to make more deliberate decisions regarding the direction of mutation, compared to using the same step size along all problem dimensions or altering step sizes through a random process with static parameters (as in the baseline algorithm).

3.5 Network Architecture

To perform the calculations for our adaptation methods, we propose the use of a 2D convolutional neural network (see Figs. 2 and 3). This section describes the requirements that a neural network architecture should fulfill in our application as well as a specific network architecture that fulfills these requirements.

3.5.1 Requirements

Instead of relying on hand-crafted features, the neural network should be offered as much information as possible about the state of the evolutionary algorithm and the problem instance, so that it can then extract the relevant features itself.

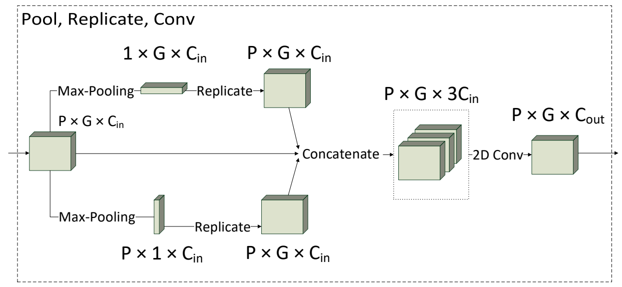


Fig. 3: The "Pool, Replicate, Conv"-substructure of the neural network. P and G are the population and genome size, respectively. $C_{\texttt{in}}$ and $C_{\texttt{out}}$ are the numbers of input and output channels, respectively. Global features are extracted by pooling along either of the two "spatial" dimensions. They are then replicated along the same dimension, combined with local features through concatenation, and processed by a 2D convolutional layer with kernel size 1×1. This architecture "broadcasts" parts of global information to each individual and each gene, and is equivariant under permutations of individuals and of genes, i.e. treats them equally.

The order in which individuals are stored in the computer should not affect the results. More specifically, there is no fixed index i reserved for individuals that across all problem instances have a specific special role; in other words, the meaning of the order of individuals is not persistent across problem instances. Hence, treatment of individuals should depend only on their features and not their order; in other words, the operations performed by the network should be equivariant under permutation of this order.

Since *in the case of our problem classes* the same holds for the order of genes, the network should also be equivariant under permutation of the gene order.

A learned adaptation strategy should be useable with varying values of `population_size` and (in the case of our problem classes, where no chromosome index has a special role across all problem instances) varying values of `genome_size`.

Information about the entire population might be relevant for taking good actions. Hence, the network should extract and use features of the entire population. These features should be permutation-invariant, for the same reasons as the permutation-equivariance explained above. Similarly, information about all genes might be relevant, so that the network should also extract and use features of entire genomes (permutation-invariant features *in the case of our problem classes*; see above).

As described in Section 3.1.2.3, there are likely features that are relevant to both the actor and critic element. To avoid redundancy and facilitate training, the actor and critic should operate on the same low-level features.

We use different types of probability distributions for the different adaptation methods introduced in Section 3.4. Each distribution type has different parameters and therefore requires different output nonlinearities. For example, a categorical distribution requires probabilities in $[0, 1]$.

Depending on the level of adaptation that a method operates on, the neural network has to output the parameters of a probability distribution for each gene, each individual or for the entire population. The dimensionality of the neural network's output is chosen accordingly.

3.5.2 Realization

The following input representation and architecture fulfill the aforementioned requirements:

The network input is defined as a 4D array of size `population_size` × `genome_size` × `num_channels` (and a `batch_size` dimension). Population-wide information (e.g. the number of remaining generations) is replicated across the two trailing dimensions. Information about individuals (e.g. their fitness) is replicated along the `genome_size` dimension. We use the following feature channels for the knapsack problem and continuous optimization:

- Knapsack problem: Individual genomes, fitness values, the remaining number of generations, the weight limit, the weight of each item, and the value of each item.
- Continuous optimization: Individual genomes, logarithm of fitness values, the remaining number of generations, and individual step-sizes.

On the traveling salesman problem, we control the selection of crossover operators. Therefore, the input is based on pairs of parent individuals. Parents of a pair are assigned an arbitrary order. The input channels for a traveling salesman problem instance with N nodes are:

- Individual genomes of first parents,
  individual genomes of second parents, fitness of first parents, fitness of second parents, the remaining number of generations, N distance information channels for first parents, N distance information channels for second parents. In each block of N distance information channels, entry (i, j) of channel k contains the distance from node $g_{i,j}$ to node k, with $g_{i,j}$ being the the value of gene j (i.e. the j^{th} visited node) of individual i. We plan a different representation for a future version of this work.

We then extract shared hidden features for the actor and critic through 2D convolutional layers with kernel size 1×1. To propagate global information, we perform the following operation before each convolutional layer separately along the `population_size` and `genome_size` dimension: For each channel, the maxima along the respective dimension are calculated. The resulting vector is then replicated along the same dimension, yielding a new matrix of size `population_size` × `genome_size`. The global features extracted through successive pooling and replication can then be processed together with local features by the next convolution filter. This process of pooling, replication and convolution is illustrated in Fig. 3.

For the critic's output, we eliminate the `genome_size` dimension through max-pooling. We then add population-wise features through max-pooling and replication along the `population_size` dimension and apply one more convolutional layer with one 1×1 filter. The resulting scalars are summed up to calculate the value estimate.

For the actor, we simply apply one more step of global pooling, replication and convolution to the shared hidden features. This is followed by max-pooling along the `genome_size` dimension or both the `genome_size` and `population_size` dimension, if a vector or scalar output is required. To fulfill the constraints on the output domain for the different adaptation methods, we use the following output nonlinearities

- Bernoulli distribution: A single channel for $p \in [0, 1]$ with the nonlinearity $\operatorname{sigmoid}(z) = \frac{1}{1+e^{-z}}$.
- Normal distribution: One channel for $\mu \in \mathbb{R}$ without any nonlinearity. One channel for $\sigma \in \mathbb{R}_+$ with the nonlinearity $\operatorname{softplus}(z) = \ln(1 + e^z)$.
- Beta distribution: Two channels for $\alpha, \beta \in [1, \infty)$, with the nonlinearity $\operatorname{softplus}(z) + 1$.
- Categorical distribution: One channel for each of the k category probabilities $p_i \in [0, 1] : \sum_{i=0}^{k-1} p_i = 1$, using the softmax nonlinearity

$$p_i = \frac{e^{z_i}}{\sum\limits_{j=0}^{k-1} e^{z_j}}, \tag{10}$$

where $(z_0, \ldots, z_{k-1})$ are the network activations before the nonlinearity.

3.6 Evaluation Methods

To evaluate the usefulness of the different proposed adaptation methods, they are compared to the baseline algorithms. To do so, we use the performance metrics and the evaluation procedure defined in the following sections.

3.6.1 Performance Metrics

We use two performance metrics to evaluate our evolutionary algorithms: mean best fitness (MBF) and mean best function value (MBFv):

- Mean best fitness is the fitness value of the fittest individual in the population, per generation, averaged over multiple runs of the evolutionary algorithm.
- Mean best function value is the lowest objective function value found by an individual in the population, per generation, averaged over multiple runs of the evolutionary algorithm.

We use MBF to assess the performance in combinatorial optimization and MBFv to assess the performance in continuous optimization. We use the mean best function value because we want to assess the quality w.r.t. to the objective function, not the clipped fitness function from Eq. (9).

We refer to the average fitness / function value achieved in the final episode as terminal mean best fitness (tMBF) / terminal mean best function value (tMBFv).

3.6.2 Evaluation Procedure

Each experiment is about optimizing one evolution parameter for one of the three problem classes (knapsack, traveling salesman, continuous optimization). A fine-grained search for an optimal (but *static*) value of that evolution parameter within the baseline algorithm is compared to our methods that learn to (*dynamically*) control that evolution parameter.

During each experiment on one evolution parameter, all other evolution parameters are held fixed at their default values. These default values are determined in advance by a coarse grid-search with the baseline algorithms. The coarseness of the search allowed a reasonable runtime (about 2 days). The coarseness of the search also means that the thereby determined default evolution parameters are not perfect. However, this is okay, because our algorithms and the baseline algorithms work with the same set of fixed values for the evolution parameters (except the parameter on which *static* vs. *dynamic* fine-tuning is compared). We deliberately chose to set the default elite size to 0 (i.e. the entire population is replaced in each generation), as we found this to magnify the impact of the remaining parameters, allowing for a better assessment of the quality of different adaptation methods. The default evolution parameter values are listed in Table S2. The fine-tuned evolution parameter values are listed in Table S3.

When fine-tuning the discrete parameters (elite size, number of parents, crossover operators), we tested all possible values.

For the mutation rate parameter (values in $[0,1]$, we searched the best-performing range of parameters $[0.005, 0.013]$ with a step size of 0.0001. For the mutation parameter for continuous optimization (values in $\mathbb{R}_+$), we searched in the best-performing range $[0,1]$ with an accuracy of 2 decimal digits. This accuracy appears sufficient, as we observed little to no difference in MBF(v) around the discovered optima.

For each adaptation method, we use a separate training set to train 21 agents using the same evolution parameters, which allows us to assess how reliably good policies can be learned. Each agent is trained for 500 iterations.

The deep learning hyperparameter values we use are summarized in Table S1:

After training, we evaluate the mean best fitness / function value achieved by each agent on a separate validation set and compare it to that of the baseline algorithm.

Mean best fitness is calculated over 100 runs of the evolutionary algorithm. Mean best function value is calculated over 500 runs of the evolutionary algorithm.

When using beta or normal distributions, actions are not taken by random sampling during validation. Instead, the mean of the distribution is taken deterministically. We found that this improves performance after the limited number of training iterations, as one does not have to wait for the loss function to decrease the distribution entropy after convergence of the policy and value estimate (see Section 3.1.2.4).

4 Results and Discussion

Following the evaluation procedure defined in Section 3.6.2, we first tuned the evolution parameters of the baseline evolutionary algorithms, before benchmarking our proposed adaptation methods against them.

In general, we found that agents could learn behavior with properties that compare favorably to the baseline algorithms: Achieving a better MBF(v) in fewer generations, not stagnating in fitness prematurely, or at least matching the performance of hand-crafted heuristics. We were successful in training agents both for discrete (e.g. parent selection) and continuous (e.g. mutation probability) action spaces, as well as for discrete and continuous optimization problems.

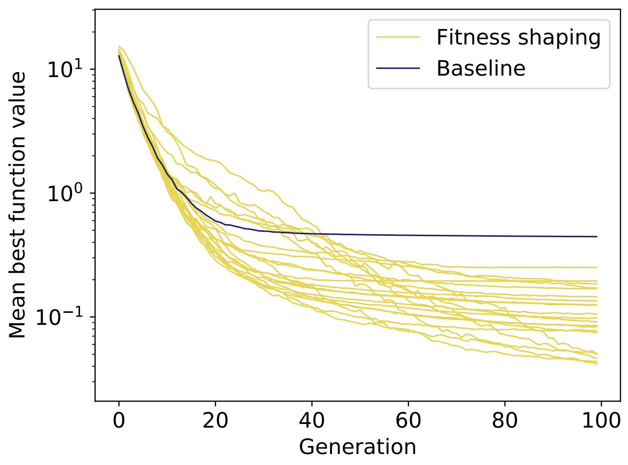


Fig. 4: MBFv (smaller is better) of 21 agents trained for fitness shaping, evaluated on the Levy #13 function, compared to the baseline algorithm. All trained agents achieved better-than-baseline performance up to a factor of 12. This shows that *learning to evolve* improves the results evolutionary algorithms. After a single training run, the user can expect above-baseline performance, but choosing the best out of multiple agents is likely to yield even better results.

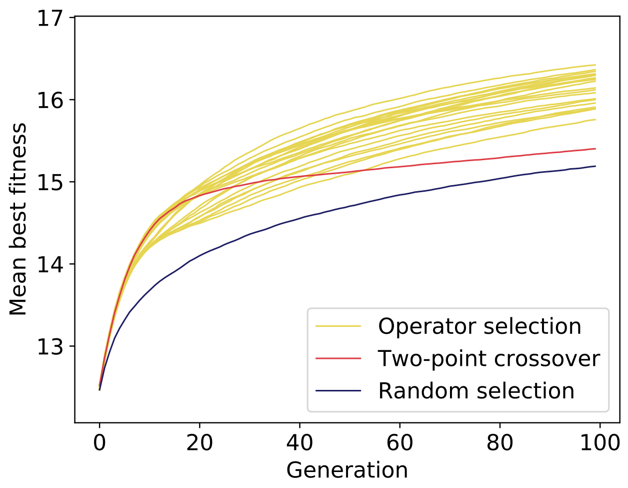


Fig. 5: MBF (larger is better) of 21 agents trained for crossover operator selection on the traveling salesman problem, compared to the best-performing single operator, as well as random operator selection with uniform probability. All trained agents performed better than baseline from generation 55 onward. The user can thus expect better-than-baseline performance even after training only one agent. tMBF varied between 16.422 and 15.757, i.e. selecting the best out of multiple trained agents is likely to yield even better results.

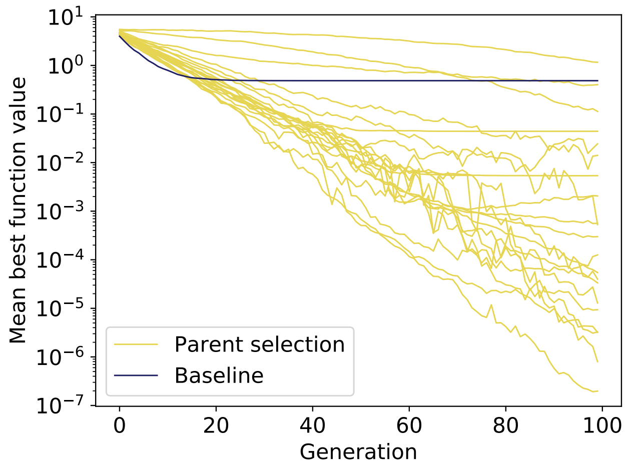


Fig. 6: MBFv (smaller is better) of 21 agents trained for parent selection, evaluated on the Ackley function, compared to the baseline algorithm. All but one of the trained agents outperform the baseline algorithm by factors of up to $2.5 \cdot 10^6$ but tMBFv varies by multiple orders of magnitude among agents. The best agent exhibits a nearly exponential improvement in fitness across all generations. After a single training run, the user can expect above-baseline performance, but choosing the best out of multiple agents is likely to yield even better results.

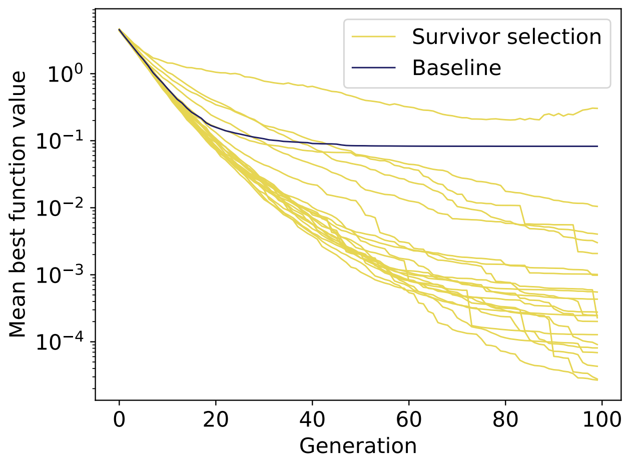


Fig. 7: MBFv (smaller is better) of 21 agents trained for survivor selection, evaluated on the Ackley function, compared to the baseline algorithm. The baseline algorithm was outperformed by up to four orders of magnitude, but the variance in tMBFv values among agents was large. After a single training run, the user can expect above-baseline performance, but choosing the best out of multiple agents is likely to yield even better results.

Furthermore, methods from all different levels of adaptation were able to outperform the baseline algorithms.

However, the adaptation methods differed considerably in their performance. In some cases, there were also large performance differences between agents belonging to the same method. We can distinguish between the following four cases, which relate to the suitability of the adaptation methods for the two considered use cases: Training with limited time/resources and training with much time/resources (see the beginning of Section 3):

Case 1) All agents achieve similar or better performance than the baseline algorithm and the variance among agents is small. This is favorable for the use case with limited training time/resources, as one can expect to achieve good performance after training a single agent. The following adaptation methods belong to this case:

- Population-level mutation rate control (Fig. S1a): All trained agents matched the performance of the baseline algorithm with an optimized mutation rate. This is remarkable, as the mutation rate only yields good results for a small range of parameter values (around 2% of the valid interval $[0, 1]$), as determined experimentally). Despite this difficulty, our reinforcement learning algorithm was able to learn a well-performing policy.
- Survivor selection, knapsack problem (Fig. S2): While all agents ended with slightly below-baseline tMBF values (average of 15.81, compared to 15.86 of the baseline algorithm), they exhibited slightly higher MBF values during the first 40 generations. Most importantly, they consistently learned a meaningful survivor selection mechanism that performed much better than replacing the population in each generation (tMBF of 15.43).

Case 2) (Nearly) all trained agents achieve similar or better performance than the baseline algorithm, but the variance in performance among well-performing agents is large. One can expect to achieve good performance after training a single agent. But if more time/resources are available for training, selecting the best-performing agent out of several trained agents is likely to lead to even better results. The following adaptation methods pertain to this case:

- Fitness shaping, knapsack problem set (Fig. S3a): Most of the 21 trained agents matched the performance of the baseline algorithm, but two out of 21 trained agents achieved noticeably higher tMBF values.
- Fitness shaping, continuous problem set: Nearly all agents outperformed the baseline algorithm. The best agents were better by factors of up to approximately 10^3, 2, and 10 on the Ackley (Fig. S4a), Beale (Fig. S4b) and Levy #13 (Fig. 4) function, respectively.
- Operator selection (Fig. 5): From generation 55 onward, all trained agents achieved higher MBF values than the deterministic application of the best crossover operator (two-point crossover) and than random operator selection with uniform probability.
- Parent selection, knapsack problem set (Fig. S3b): Except for one outlier, all agents reached baseline or above-baseline performance, with the highest tMBF being 15.732, compared to 15.432 for the baseline algorithm.
- Parent selection, continuous problem set: On the Ackley function (Fig. 6), 19 out of 21 trained agents reached

tMBFv values that were better than the baseline algorithm's by a factor of up to 10^6. The majority of agents improve their MBF near-exponentially across all generations, while the baseline algorithm stagnated after generation 20. On the Levy #13 function (Fig. S5b), 17 out of 21 trained agents performed better than the baseline algorithm by up to one order of magnitude. On the Beale (Fig. S5a) function, the impact of the method was smaller, but many trained agents reached near- or better-than-baseline performance.

- Survivor selection, continuous problem set: Nearly all agents reached better MBFv than the baseline algorithm. On the Ackley (Fig. 7) and Levy #13 (Fig. S6b) function, the baseline algorithm was in many cases outperformed by several orders of magnitude. On the Beale function (Fig. S6a), the best agent reached tMBFv that were smaller by a factor of 2.

Case 3) A minority of the trained agents outperform the baseline algorithm and the variance in performance is large. In the use case with much training time/resources, these methods are still valuable, as one can select the best-performing out of several trained agents. The following adaptation methods pertain to this case:

- Population-level strategy parameter control: Most trained agents performed worse than the baseline algorithm. Nevertheless, on the Ackley (Fig. S7a) function, a single trained agent achieved a tMBFv that is approximately 10^5 times better than that of the baseline algorithm. On the Beale (Fig. S7b) function, two out of 21 trained agents outperformed the baseline algorithm.
- Individual-level step-size control: This method performed better than the individual-level strategy parameter control method, confirming our idea that eliminating one level of stochasticity by directly controlling step-sizes facilitates the learning of useful policies. On the Ackley (Fig. S8a) and Beale (Fig. S8b) function, three out of 21 trained agents outperformed the baseline algorithm by more than one order of magnitude. On the Levy #13 function (Fig. S8c), three agents were able to match its performance.
- Component-level step-size control (Fig. S9): On all objective functions, approximately one third of the trained agents outperformed the baseline algorithms, in some cases by multiple orders of magnitude. This is better than individual-level step-size control, where only one seventh of the trained agents exhibited good performance. These better results are likely due to the method's ability to control mutation along both problem dimensions separately, thus being able to better adapt to the fitness landscape.
- Component-level binary mutation (Fig. S1c): Despite the increase in action-space dimensionality, four out of 21 agents noticeably outperformed the baseline algorithm. The best agent was able to reach a tMBF of 15.931, compared to the 15.488 of the baseline algorithm. However, the majority of trained agents were unable to perform any optimization whatsoever so careful selection of the best agents out of many is particularly important.

Case 4) Only few trained agents match the performance of the baseline algorithm and the variance in performance is large. In this case, static tuning of the parameters of the baseline algorithm is likely more sensible than training many agents just to achieve the same level of performance. The following adaptation methods pertain to this case:

- Individual-level mutation rate control (Fig. S1b): Only a single trained agent out of 21 was able to slightly outperform the baseline algorithm (tMBF of 15.538 compared to 15.488). A possible explanation is that learning to keep multiple parameters (one per individual) in a very narrow range of feasible values is considerably harder than doing so with a single parameter, as in the population-level method.
- Individual-level strategy parameter control: On the Ackley (Fig. S10a) and Beale (Fig. S10b) function, only a single trained agent reached a tMBF close to that of the baseline algorithm, exhibiting a faster convergence in the beginning of the optimization process. On the Levy #13 function (Fig. S10c), all trained agents were outperformed by the baseline algorithm.

5 Conclusions

The goal of this paper was to investigate whether deep reinforcement learning can be used to improve the effectiveness of evolutionary algorithms and facilitate their application. To this end, we developed an approach for learning optimization strategies off-line through deep reinforcement learning.

For experimental evaluation of our approach, we considered use cases in which strategies for previously unseen problem instances have to be learned from a limited set of training instances.

Adaptation methods trained using our approach were in many cases able to outperform classical evolutionary algorithms on combinatorial and continuous optimization tasks. We also showed that the use of reinforcement learning for evolutionary algorithms is not limited to controlling single numerical parameters of an evolutionary algorithm, but can also be used for both continuous and discrete multi-dimensional control. Furthermore, we achieved promising results with methods that do not merely control existing parameters of evolutionary algorithms, but learn entirely new dynamic fitness functions or selection operators that intelligently guide evolutionary pressure.

However, we noticed that for some of the investigated methods, training was more unstable and results varied more heavily. A more thorough experimental evaluation is required to discern whether this has to be attributed to ill-chosen hyperparameters, the limited size of the used training sets, or the design of the methods. Nevertheless, we demonstrated that deep reinforcement learning can be used to improve the effectiveness of evolutionary algorithms.

Further investigation of evolutionary algorithms enhanced by deep reinforcement learning could lead to better population-based optimization algorithms that can more easily be applied to a wide range of problems. To explore the suitability of reinforcement-learning-based adaptation methods to different application domains, future work could consider a wider range of use cases than we did in our experiments, for example:

- unlimited training set (e.g. problem instances can be randomly generated),

- various degrees of availability of training time/resources,
- training to optimize performance on [not necessarily finite] problem instances known at training time (as opposed to generalization to unseen problem instances),
- training for multiple problem classes at once (to learn problem-class-independent meta-optimization behavior),
- optimization for a variable number of generations.

Future work should also benchmark against a wider range of methods, and especially combine our approach with a wider range of evolutionary algorithms.

Acknowledgments

The authors would like to thank Paolo Notaro for valuable discussions.

References

[1] E. Kameshki and M. Saka, "Optimum design of nonlinear steel frames with semi-rigid connections using a genetic algorithm," *Computers & Structures*, vol. 79, no. 17, pp. 1593–1604, 2001. DOI: 10.1016/s0045-7949(01)00035-9.

[2] J. D. Lohn, G. S. Hornby, and D. S. Linden, "An evolved antenna for deployment on NASA's Space Technology 5 mission," in *Genetic Programming Theory and Practice II*, Springer-Verlag, pp. 301–315. DOI: 10.1007/0-387-23254-0_18.

[3] G. Mosetti, C. Poloni, and B. Diviacco, "Optimization of wind turbine positioning in large windfarms by means of a genetic algorithm," *Journal of Wind Engineering and Industrial Aerodynamics*, vol. 51, no. 1, pp. 105–116, 1994. DOI: 10.1016/0167-6105(94)90080-9.

[4] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019. DOI: 10.1038/s42256-018-0006-z.

[5] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*, https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[6] R. Hinterding, Z. Michalewicz, and A. Eiben, "Adaptation in evolutionary computation: A survey," in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, IEEE. DOI: 10.1109/icec.1997.592270.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *ArXiv e-prints*, 2015. arXiv: 1509.02971v5 [cs.LG].

[8] S. Müller, N. Schraudolph, and P. Koumoutsakos, "Step size adaptation in evolution strategies using reinforcement learning," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, IEEE. DOI: 10.1109/cec.2002.1006225.

[9] A. E. Eiben, M. Horvath, W. Kowalczyk, and M. C. Schut, "Reinforcement learning for online control of evolutionary algorithms," in *Engineering Self-Organising Systems*, Springer Berlin Heidelberg, pp. 151–160. DOI: 10.1007/978-3-540-69868-5_10.

[10] G. Karafotias, A. E. Eiben, and M. Hoogendoorn, "Generic parameter control with reinforcement learning," in *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*, ACM Press, 2014. DOI: 10.1145/2576768.2598360.

[11] J. E. Pettinger and R. M. Everson, "Controlling genetic algorithms with reinforcement learning," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '02, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 692–, ISBN: 1-55860-878-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=646205.682951.

[12] A. Buzdalova, V. Kononov, and M. Buzdalov, "Selecting evolutionary operators using reinforcement learning," in *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion - GECCO Comp '14*, ACM Press, 2014. DOI: 10.1145/2598394.2605681.

[13] L. DaCosta, A. Fialho, M. Schoenauer, and M. Sebag, "Adaptive operator selection with dynamic multi-armed bandits," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, ACM Press, 2008. DOI: 10.1145/1389095.1389272.

[14] K. Li, A. Fialho, S. Kwong, and Q. Zhang, "Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 1, pp. 114–130, 2014. DOI: 10.1109/tevc.2013.2239648.

[15] A. Afanasyeva and M. Buzdalov, "Choosing best fitness function with reinforcement learning," in *2011 10th International Conference on Machine Learning and Applications and Workshops*, IEEE, 2011. DOI: 10.1109/icmla.2011.163.

[16] I. Petrova, A. Buzdalova, and M. Buzdalov, "Improved selection of auxiliary objectives using reinforcement learning in non-stationary environment," in *2014 13th International Conference on Machine Learning and Applications*, IEEE, 2014. DOI: 10.1109/icmla.2014.99.

[17] P. Bhowmik, P. Rakshit, A. Konar, E. Kim, and A. K. Nagar, "DE-TDQL: An adaptive memetic algorithm," in *2012 IEEE Congress on Evolutionary Computation*, IEEE, 2012. DOI: 10.1109/cec.2012.6256573.

[18] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992. DOI: 10.1007/bf00992698.

[19] A. Rost, I. Petrova, and A. Buzdalova, "Adaptive parameter selection in evolutionary algorithms by reinforcement learning with dynamic discretization of parameter range," in *Proceedings of the 2016 on Genetic*

and Evolutionary Computation Conference Companion - GECCO '16 Companion, ACM Press, 2016. DOI: 10.1145/2908961.2908998.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *ArXiv e-prints*, 2017. arXiv: 1707.06347v2 [cs.LG].

[21] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *ArXiv e-prints*, 2015. arXiv: 1506.02438v5 [cs.LG].

[22] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, “Time limits in reinforcement learning,” *ArXiv e-prints*, 2017. arXiv: 1712.00378v2 [cs.LG].

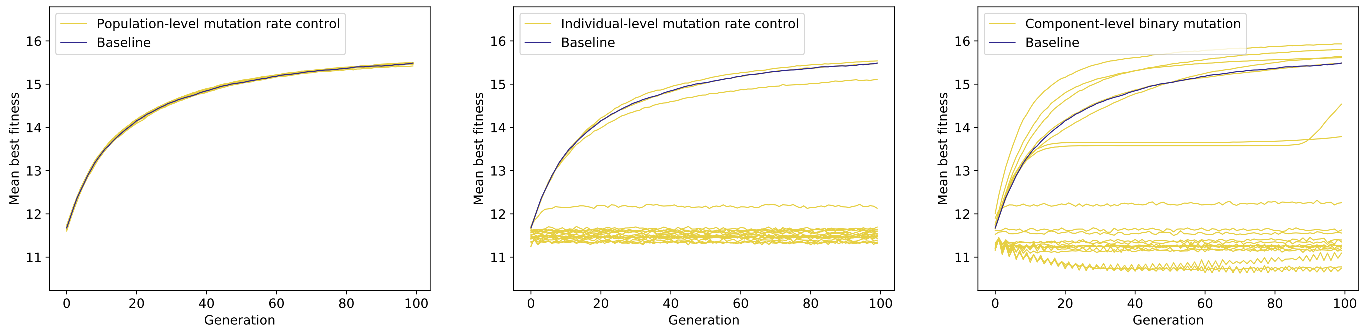
[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16, New York, NY, USA, 2016, pp. 1928–1937. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045594.

[24] S. Surjanovic and D. Bingham, *Virtual library of simulation experiments: Test functions and datasets*, Retrieved May 8, 2019, from http://www.sfu.ca/~ssurjano.

[25] E. Anand and R. Panneerselvam, “A study of crossover operators for genetic algorithm and proposal of a new crossover operator to solve open shop scheduling problem,” *American Journal of Industrial and Business Management*, vol. 06, no. 06, pp. 774–789, 2016. DOI: 10.4236/ajibm.2016.66071.

[26] P.-W. Chou, D. Maturana, and S. Scherer, “Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution,” in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 834–843. [Online]. Available: http://proceedings.mlr.press/v70/chou17a.html.

Appendix
Supplementary Figures and Tables



(a) Population-level mutation rate control (b) Individual-level mutation rate control (c) Component-level binary mutation

Fig. S1: MBF (larger is better) of 21 agents trained for the different mutation-controlling adaptation methods (namely on the population level, individual level, and component level) for the knapsack problem, compared to the baseline algorithm. On the population level, all agents matched the performance of the baseline algorithm with an optimized mutation rate. On the individual level, the majority of learned policies lead to a stagnation in fitness. Only one out of 21 agents was slightly better than the baseline algorithm. On the component level, four trained agents outperformed the baseline algorithm, but the majority of learned policies lead to a stagnation in fitness. Given much training time/resources, the user can select the best-performing component-level agent to outperform the baseline algorithm. Component-level methods have outputs with more degrees of freedom, and thus achieve better solutions but are also more difficult to train.

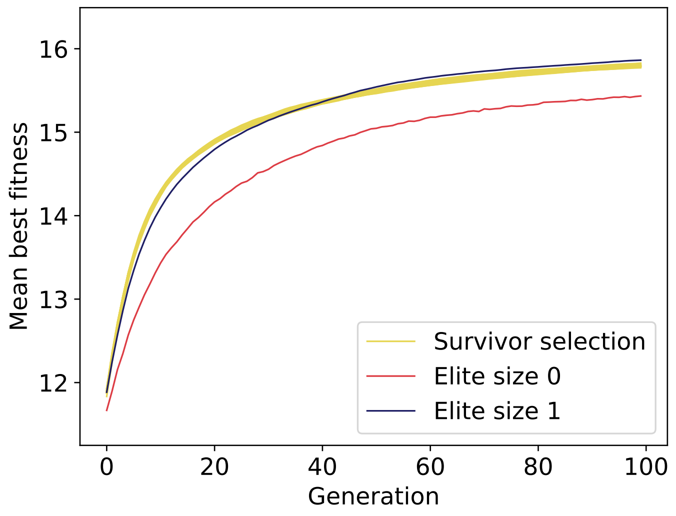
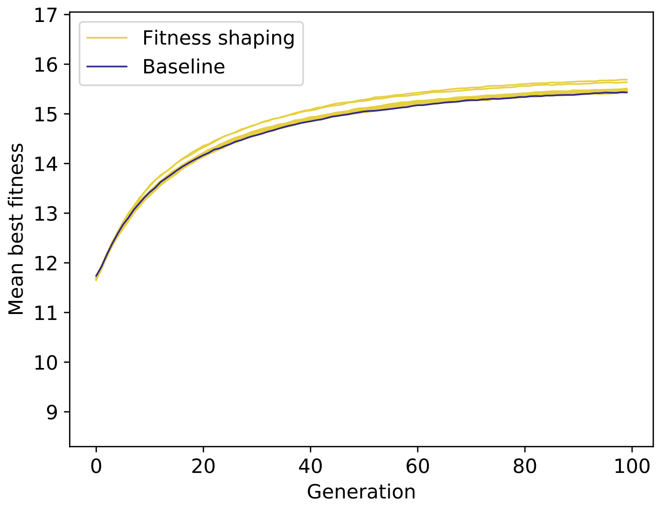
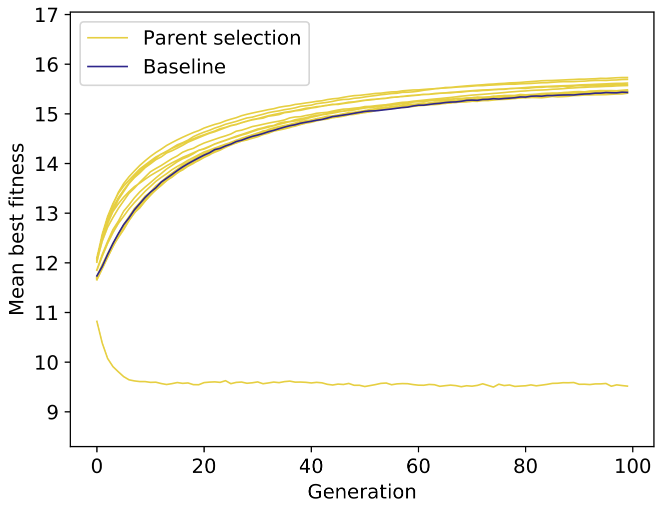


Fig. S2: MBF (larger is better) of 21 agents trained for survivor selection on the knapsack problem, compared to the baseline with an optimal elite size of 1 and an elite size of 0. The learned policy performed much better than replacement of the population in each generation (i.e. elite size 0). Given limited time/resources for training, the user can expect good performance after training a single agent.

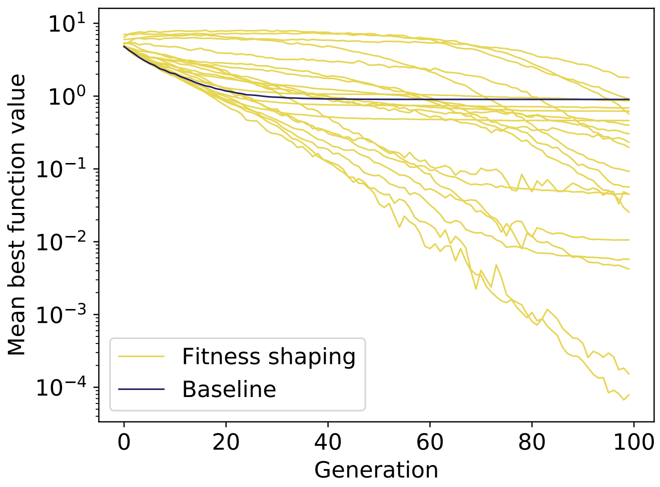


(a) Fitness shaping

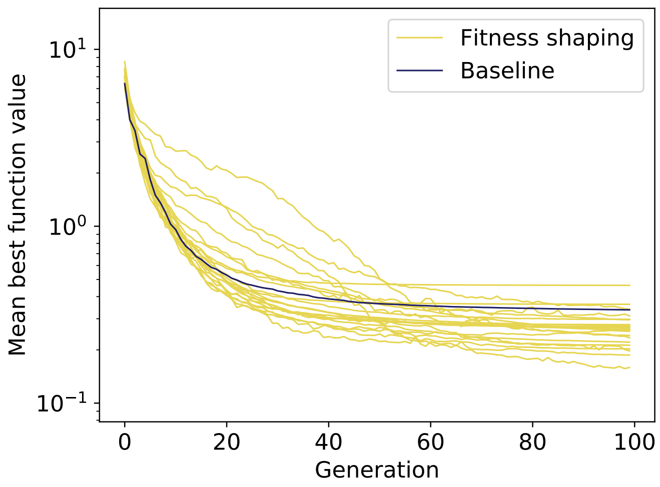


(b) Parent selection

Fig. S3: MBF (larger is better) of 21 agents trained for parent selection and fitness shaping on the knapsack problem, compared to the baseline algorithm. Except for one outlier, all trained agents of both methods matched the MBF of the baseline algorithm or exceeded it. However, the impact of parent selection is larger, with more agents outperforming the baseline algorithm, most noticeably during the first 20 generations. With both methods, the user can expect above-baseline performance after a single training run, but choosing the best out of multiple agents is likely to yield even better results.

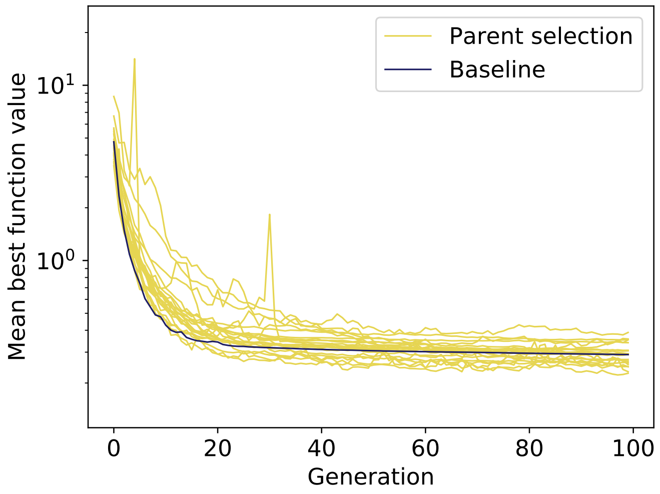


(a) Optimization of the Ackley function

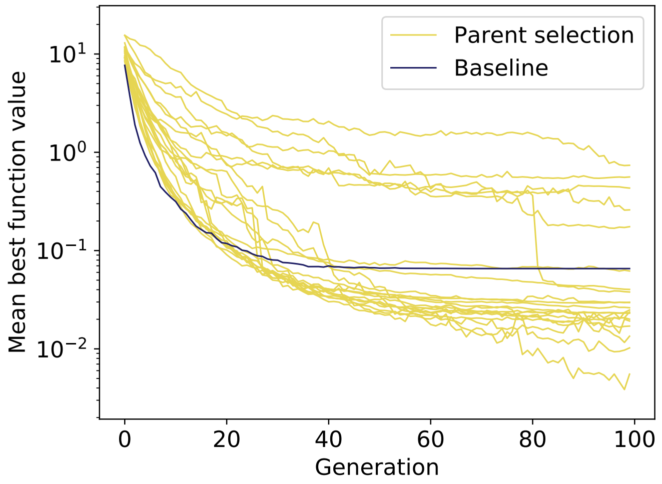


(b) Optimization of the Beale function

Fig. S4: . MBFv (smaller is better) of 21 agents trained for fitness shaping, evaluated on the Ackley and Beale functions, compared to the baseline algorithm (see also Fig. 4 for the Levy #13 function). Nearly all trained agents achieved better-than-baseline performance – especially on the Ackley function, where the tMBFv of the best trained agent is lower by a factor of more than 10^3. After a single training run, the user can expect above-baseline performance, but choosing the best out of multiple agents is likely to yield even better results.

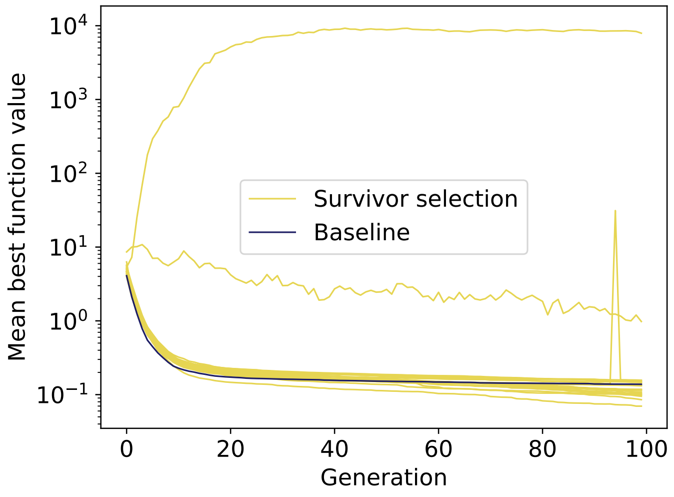


(a) Optimization of the Beale function

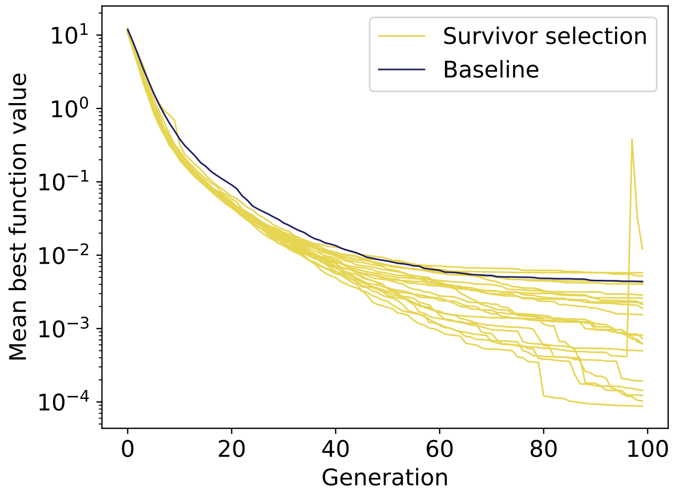


(b) Optimization of the Levy #13 function

Fig. S5: MBFv (smaller is better) of 21 agents trained for parent selection, evaluated on the Beale and Levy #13 functions, compared to the baseline algorithm (see also Fig. 6 for the Ackley function). Many of the trained agents outperformed the baseline algorithm on the Levy #13 function, but tMBFv varies by multiple orders of magnitude among agents. In both methods, the user can expect above-baseline performance after a single training run, but choosing the best out of multiple agents is likely to yield even better results.

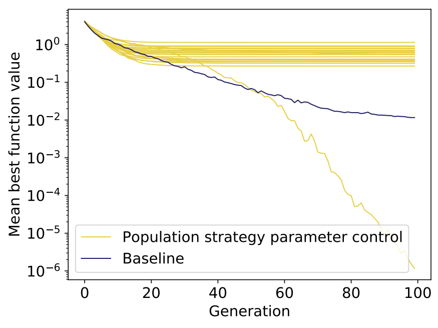


(a) Optimization of the Beale function

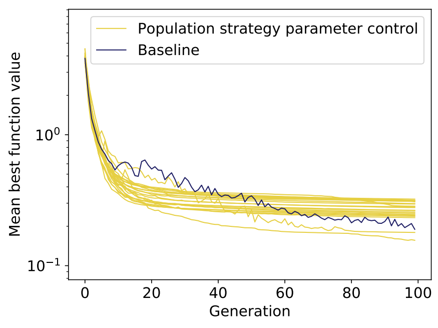


(b) Optimization of the Levy #13 function

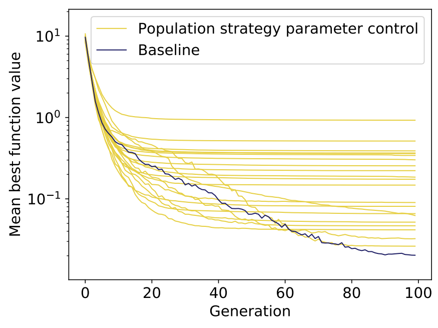
Fig. S6: MBFv (smaller is better) of 21 agents trained for survivor selection, evaluated on the Beale and Levy #13 functions, compared to the baseline algorithm (see also Fig. 7 for the Ackley function). The majority of trained agents performed better than the baseline algorithm, but the variance in tMBFv values among agents was large. After a single training run, the user can expect above-baseline performance, but choosing the best out of multiple agents is likely to yield even better results.



(a) Optimization of the Ackley function

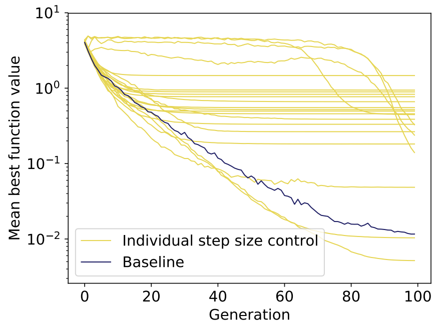


(b) Optimization of the Beale function

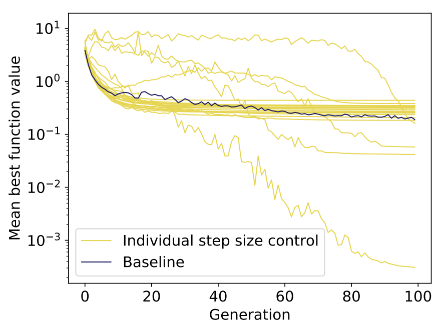


(c) Optimization of the Levy #13 function

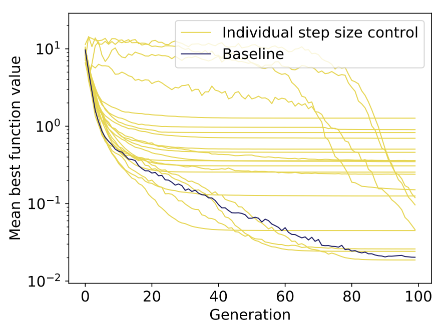
Fig. S7: MBFv (smaller is better) of 21 agents trained for population-level strategy parameter control for continuous optimization, evaluated on the validation set, compared to the baseline algorithm. On the Ackley- and Beale function, one and two agents, respectively, outperformed the baseline algorithm. On the Levy #13 function, two agents reached near-baseline tMBFv values. Although most agents performed worse on all three functions, the best ones performed either better or not much worse than baseline, so that the method could be useful for the use case with much time/resources for training several agents.



(a) Optimization of the Ackley function



(b) Optimization of the Beale function

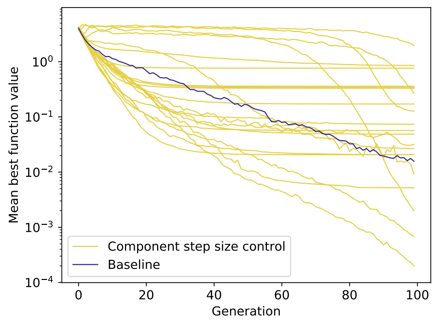


(c) Optimization of the Levy #13 function

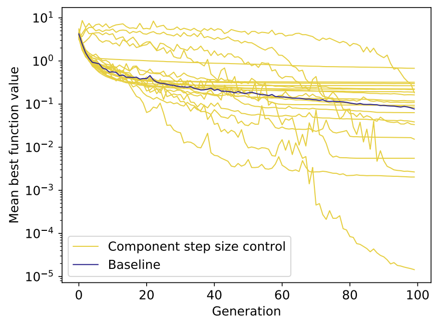
Fig. S8: MBFv (smaller is better) of 21 agents trained for individual-level step-size control for continuous optimization, evaluated on the validation set, compared to the baseline algorithm. On the Ackley and Beale function, three out of 21 trained agents outperformed the baseline algorithm. On the Levy #13 function, three trained agents achieved baseline levels of performance. This adaptation method also performed much better than strategy parameter control (see Fig. S10). While both methods control the mutation step-sizes, this method does so deterministically, whereas strategy-parameter control only alters parameters of a random process that changes step-sizes. Reducing this level of stochasticiy likely facilitates training. This method appears well suited for the use case with much training time/resources, in which the user can select the best out of multiple trained agents to improve upon the baseline algorithm.

Method	Problem	Learning rate	Batch size	#epochs	α_e	#actors
Fitness shaping	Knapsack	$1 \cdot 10^{-4}$	200	8	10^{-4}	4
Fitness shaping	Continuous	$5 \cdot 10^{-4}$	400	8	10^{-4}	4
Survivor selection	Knapsack	$1 \cdot 10^{-4}$	400	4	10^{-4}	4
Survivor selection	Continuous	$1 \cdot 10^{-4}$	800	8	10^{-4}	4
Population-level mutation rate control	Knapsack	$1 \cdot 10^{-4}$	800	4	10^{-4}	4
Population-level strategy parameter control	Continuous	$1 \cdot 10^{-4}$	400	4	10^{-4}	4
Operator selection	TSP	$1 \cdot 10^{-4}$	400	8	10^{-2}	2
Individual-level mutation rate control	Knapsack	$5 \cdot 10^{-4}$	400	4	10^{-4}	4
Individual-level strategy parameter control	Continuous	$1 \cdot 10^{-4}$	400	4	10^{-4}	4
Individual-level step-size control	Continuous	$1 \cdot 10^{-3}$	400	8	10^{-4}	4
Parent selection	Knapsack	$1 \cdot 10^{-4}$	400	8	10^{-3}	4
Parent selection	Continuous	$1 \cdot 10^{-4}$	800	8	10^{-3}	4
Component-level binary mutation	Knapsack	$5 \cdot 10^{-4}$	200	$4, 8$	10^{-4}	8
Component-level step-size control	Continuous	$5 \cdot 10^{-4}$	800	8	10^{-4}	8

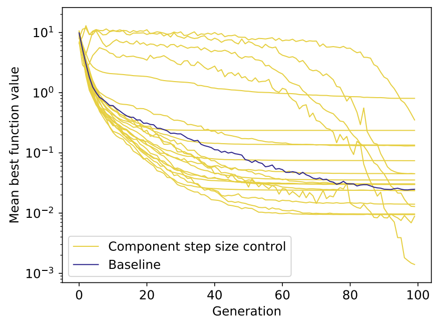
TABLE S1: Hyperparameter values used for training the different adaptation methods. Some additional hyperparameters were set to the same value across all experiments: $\lambda = \gamma = 0.99$, $\epsilon = 0.2$, $\alpha_v = 0.5$. The reward-scaling factor α_r was set to 100 for the knapsack and travelling salesman problem and to 1 for continuous optimization. We used a network depth of 3 with 64 filters per layer.



(a) Optimization of the Ackley function

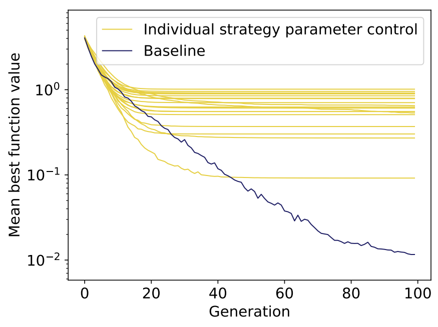


(b) Optimization of the Beale function

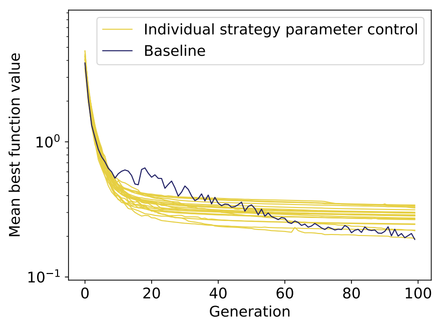


(c) Optimization of the Levy #13 function

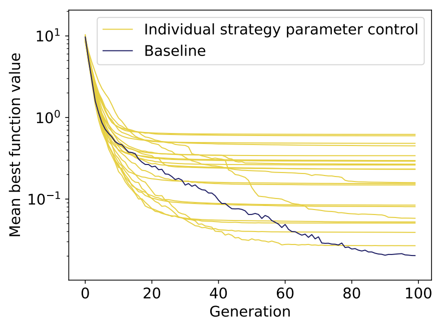
Fig. S9: MBFv (smaller is better) of 21 agents trained for component-level step-size control for continuous optimization, evaluated on the validation set, compared to the baseline algorithm. Approximately one third of the agents outperformed the baseline algorithm, in many cases by multiple orders of magnitude. This method appears well suited for the use case with much training time/resources, in which the user can select the best out of multiple trained agents to improve upon the baseline algorithm.



(a) Optimization of the Ackley function



(b) Optimization of the Beale function



(c) Optimization of the Levy #13 function

Fig. S10: MBFv (smaller is better) of 21 agents trained for individual-level strategy parameter control for continuous optimization, evaluated on the validation set, compared to the baseline algorithm. Some agents outperformed the baseline algorithm in early generations, but none in late generations.

Problem / Hyperparameter	Continuous	Knapsack	TSP
population_size	10	10	10
parent_percentage	20%, (50%) [1]	–	–
elite_size	0	0	1
crossover_rate	–	0.9	1
mutation_rate	–	0.01	0.01
strategy_parameter	0.5	–	–
initial_step_size	0.1	–	–
min_step_size	$1 \cdot 10^{-8}$	–	–

TABLE S2: Default evolutionary parameter values used in benchmarking our adaptation methods on the different problem classes. When evaluating the impact of elite size, the survivor selection adaptation method or the fitness shaping adaptation method, $parent\_percentage$ is set to 50%

Problem / Hyper-parameter	Ackley	Beale	Levy #13	Knapsack	TSP
parent_percentage	20%	20%	10%	–	–
elite_size	6	2	9	1	–
mutation_rate	–	–	–	0.0082	–
strategy_parameter	0.19	0.11	0.22	–	–
crossover_operator	–	–	–	–	Two-point

TABLE S3: Optimized evolutionary parameter values used in benchmarking our adaptation methods on the different validaton problem sets. Evolutionary algorithms run with these optimized parameter values were compared to corresponding adaptation methods.