Universiteit Leiden

Master Computer Science

Online Mutation Strategy Selection in Differential Evolution through Deep Reinforcement Learning

Name: Marc Boel
Student ID: s2342456

Date: April 23, 2024

Specialisation: Artificial Intelligence

1st supervisor: Anna Kononova
2nd supervisor: Thomas Moerland
Daily supervisor: Diederick Vermetten
Daily supervisor: Jacob de Nobel

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Metaheuristics are essential tools to find (near-)optimal solutions to complex problems in a reasonable amount of time. Research has shown that many of such metaheuristics are highly sensitive to their parameter settings, making parameter tuning a critical but difficult task. This led to the development of adaptive parameter selection strategies to automatically tune the parameters. This thesis proposes a new variant of Differential Evolution, a widely used and successful metaheuristic, which uses an artificial neural network to dynamically select the optimal mutation operator and scaling factor for every individual in each generation in an online fashion. Moreover, this variant allows for parallelisation, which can significantly reduce computing time. The use of artificial neural networks in adaptive parameter selection is motivated by their ability to learn complex patterns and relationships from data, making this technique well-suited for interpreting complex state spaces. The neural network models are trained using deep reinforcement learning techniques. The DE state is defined by 112 different features derived from the fitness landscape and the success of the actions in previous generations. From these state features, the model chooses one out of four mutation strategies and one out of two scaling factors, giving a total of eight possible discrete actions. To evaluate the proposed algorithm, three models were trained from scratch on the 24 continuous functions of the BBOB benchmark. These models were then tested on the same functions and compared with a set of baselines: a random policy and, for every mutation strategy, a static policy that always selects the same strategy for every individual. On average all three models manage to outperform all baselines, but further analysis reveals significant variations in performance across different optimisation problems. The average rewards indicate that the models learn to find better solutions for a given problem every generation, but in some cases tend to make only small improvements. A recommendation for future research is given to improve the robustness and performance of this adaptive tuning method, making it more reliable and applicable to a wider range of optimisation problems.

Contents

1 Introduction

In the context of optimisation, metaheuristics are a class of optimisation algorithms that are designed to efficiently explore and find (near-)optimal solutions in complex search spaces. These algorithms are designed to 'solve' difficult problems where little is known about the fitness landscape and the characteristics of the problem may vary wildly. These properties make these algorithms useful in a wide range of applications, e.g. finance, transportation, engineering, biology and medical research. For such problems often nothing or little is known about the fitness landscape and with large search spaces and computationally expensive problems it is infeasible to go over every single possible setting. Metaheuristics provide an approach to find good solutions in an efficient and effective way by sampling a subset of solutions in the search space.

One class of metaheuristics is that of Evolutionary Algorithms (EAs), which, as its name suggests, is inspired by biological evolution. Its settings, e.g. population size or selection strategy, can greatly change the behaviour of the algorithm and thus the performance. Parameters control the overall behaviour and settings of the DE algorithm, while operators govern the specific mathematical operations used to evolve candidate solutions during the optimisation process. This thesis focuses specifically on choosing the correct mutation strategy for Differential Evolution (DE). In the original variant of DE, as introduced by Storn and Price [26], the settings are set beforehand and not changed throughout the operation of the algorithm. Despite DE being designed to be a robust method with easy and intuitive hyperparameters, further research has shown that depending on the characteristic of the problem and the settings of the algorithm, the results can vary wildly [23].

This thesis focuses on finding the optimal hyperparameters of a DE in an *online* fashion. Online hyperparameter tuning changes the settings of the DE *during* the optimisation process. This can either be done with a rule-based method or a more data-driven approach. Adaptive Operator Selection (AOS) aims to select the settings in an online fashion by figuring out good settings for the problem at hand. If this extra layer on top of the metaheuristic is able to find the optimal settings for the given problem with the information it gained in all previous generations, it should reduce the complexity for the researchers while also optimising the performance [16]. Such a model would (partially) circumvent the need for researchers to find good settings themselves, and also take advantage of potential performance gains that are lost when only having one setting for the entire run.

In this thesis we propose a method to adaptively select the mutation operator and mutation rate of each individual in every generation in DE using an artificial neural network, which is trained using Reinforcement Learning, which has shown promise [25]. Deep reinforcement learning can, unlike tabular methods, approximate the expected rewards and generalise beyond the trained behaviour [21] . This makes it capable of approximating continuous high-dimensional environments. The goal is to create an algorithm that matches or outperforms variants of DE without AOS while also reducing the complexity of manually setting the hyperparameters. Furthermore, we place particular focus on on making the DE able to be parallelised, which was the original intent of DE and is core in reducing computing time. We work exclusively on optimisation of the 24 single-objective continuous functions of the BBOB benchmark.

This thesis is partially based off the work of Sharma et al. [25], where a neural network is used to select the mutation operators based on a list of 99 state features. Three reward functions were tested, of which we use the most successful one. This thesis extends on this by also tuning the mutation factor F and by tweaking the state features such that the DE can be parallelised.

The new variant of DE proposed in this thesis manages to outperform all variants were the mutation strategy is kept static on average. Further analysis showed that there are sometimes large performance differences between the 24 BBOB functions. This algorithm shows promise,

but further research is needed to improve stability and performance to keep up with the state-of-the-art.

This thesis is divided into sections as follows: Section 2 gives an explanation on the individual components of both Differential Evolution and Reinforcement Learning. A summary of the current state of research in DE along with some relevant related work is presented in Section 3. A description of how DE and RL work together to form this algorithm, together with the methods and setup for training the neural network are outlined in Section 4. The results from our testing are presented and analysed in Section 5. In Section 6 these results are interpreted further and the main findings are summarised. The conclusion includes a list of directions for future work.

2 Background

In this section an overview is given on Differential Evolution (Section 2.1) and Deep Reinforcement Learning (Section 2.2).

2.1 Differential Evolution

DE [26] is a subclass of Evolutionary Algorithms (EAs), which are itself a subset from the field of evolutionary computation. EAs are population-based optimisation algorithms that are inspired by biological evolution. Before the functioning of DE is laid out, a short overview of optimisation is given to give a better understanding of what we aim to achieve.

DEs are used to find (near-)optimal solutions for a given function, but there is no guarantee that near-optimal solutions, or even solutions that could be considered good, are found. Typically, these functions for which we want to find the optimum are *black box* functions, meaning that nothing is known about the function and thus the fitness landscape beforehand.

Given a real-valued D-dimensional black-box function f:

$$f : \mathbb{R}^D \to \mathbb{R}\,, \tag{1}$$

the optimisation problem can be described as, assuming minimisation:

$$\text{Find } \vec{x}^* \mid f(\vec{x}^*) \leq f(\vec{x}) \; \forall \; \vec{x} \in \mathbb{R}^D\,. \tag{2}$$

Sometimes, typically in benchmarks, $f(\vec{x}^*)$ is known, while $\vec{x}^*$ is not. To avoid confusion we can write $f(\vec{x}^*)$ as simply f^*.

A DE, as proposed by Storn and Price [26], works as follows. First, a population P of NP random vectors $\vec{x}$ of size D is created:

$$P = \{\vec{x}_1, \vec{x}_2, ..., \vec{x}_{NP}\} \in [lb, ub]^D\,. \tag{3}$$

Since for these functions nothing is known about the fitness landscape beforehand, the vectors in the initial population are uniformly distributed over the entire search space $[lb, ub]^D$, where lb and ub are the lower and upper bounds of the search space respectively.

After the initial vector population is created, the algorithm loops over the following four steps until either the optimum is reached (if the optimum value is known) or the evaluation budget is depleted.

1. Mutation
2. Crossover
3. Evaluation
4. Selection

To create new vectors, parent vectors are selected from the parent population. The weighted difference between these vectors are used to create mutant vectors. There are many different *mutation operators* that select and combine the parent vectors in various ways. After generating NP mutant vectors, one for each individual in the population, the second step is crossover, where trial vectors are created from the target vectors and the mutant vectors. The resulting trial vectors are then evaluated and the population of the next generation is selected using paired elitist selection between every pair of trial vector and corresponding target vector. Mutation and crossover are explained in more detail in the next two sections.

2.1.1 Mutation Strategies

For the mutation step one mutant vector $\vec{v}_i$ is created for every individual $\vec{x}_i$. This results in a set of mutant vectors which we note with the bold $\vec{\mathbf{v}}$. Traditionally, one mutation strategy is chosen beforehand and not changed throughout the operation of the algorithm. For this thesis we look at only the following four mutation operators:

$$\text{“rand/1”} : \vec{v}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{r_2} - \vec{x}_{r_3}) \,, \tag{4}$$

$$\text{“rand/2”} : \vec{v}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{r_2} - \vec{x}_{r_3} + \vec{x}_{r_4} - \vec{x}_{r_5}) \,, \tag{5}$$

$$\text{“rand-to-best/2”} : \vec{v}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{\text{best}} - \vec{x}_{r_1} + \vec{x}_{r_2} - \vec{x}_{r_3} + \vec{x}_{r_4} - \vec{x}_{r_5}) \,, \tag{6}$$

$$\text{“curr-to-rand/2”} : \vec{v}_i = \vec{x}_i + F \cdot (\vec{x}_{r_1} - \vec{x}_i + \vec{x}_{r_2} - \vec{x}_{r_3}) \tag{7}$$

where mutually different random indices $r_1, r_2, ..., r_j \in \{1, 2, ..., NP\} \setminus i$ are selected uniformly and $\vec{x}_{\text{best}}$ is the member with the lowest objective value in the population. The difference vector is scaled by the mutation rate control parameter $F > 0$, but values of $F > 1$ are rarely deemed effective [22].

Because there are numerous possible combinations of difference vectors and base vectors for the mutation strategy, each with their own advantages and disadvantages, it is difficult to find the best strategy for a given function, even when more information about the fitness landscape is known [17, 23].

2.1.2 Crossover Strategies

Crossover is a necessary step to increase the diversity of the mutant vectors. By mixing the components of the mutant vectors $\vec{\mathbf{v}} = \vec{v}_1, \vec{v}_2, ..., \vec{v}_{NP}$ with the components of the target vectors $\vec{\mathbf{x}} = \vec{x}_1, \vec{x}_2, ..., \vec{x}_{NP}$ a new population of trial vectors $\vec{\mathbf{u}} = \vec{u}_1, \vec{u}_2, ..., \vec{u}_{NP}$ is created. While a range of mutation strategies are used in state-of-the-art implementations, two crossover strategies can be considered the default, namely binomial crossover and exponential crossover [17, 20]. The crossover rate control parameter $CR \in [0, 1]$ determines the fraction of components that are transferred from the mutant to the trial vector.

- **Binomial crossover**: For every vector $\vec{u}_i$ the binomial crossover algorithm loops over every component $u_{i,j}$ $(j = 1, 2, ..., D)$. The probability of every component to be transferred to the trial vector is set by CR. Also, for every vector a random index j_{rand} is chosen which is used to ensure that always at least one component of every mutant vector will be transferred to the trial population, even when $CR = 0$. Pseudocode of this crossover operator is provided in Algorithm 1.

- **Exponential crossover**: Exponential crossover is more akin to 1-point crossover in genetic algorithms. First, an index j is randomly chosen. This represents the starting point of the string of components that is transferred to the trial vector. CR sets the probability of every next component to also be transferred. The first component is always transferred and if the entire vector has been transferred the loop terminates. $CR = 0$ would result in just one random component from each mutant vector being transferred to the trial population, while $CR = 1$ would mean that the entire mutant population is transferred with all of its components. Pseudocode of this crossover operator is provided in Algorithm 2.

Exponential crossover has the property that components that are far apart (in the order that they are in the vector) are more likely to be disrupted, while the once that are adjacent are more likely to stay together. This can either be an advantage or a disadvantage [31] and knowing beforehand which is best to pick is not possible for black box functions. When benchmarking DEs with exponential crossover it is important to keep this property in mind.

Algorithm 1 Binomial crossover

for $i = 1$ to NP **do**
  for $j = 1$ to D **do**
    if $\mathcal{U}(0,1) \leq CR \vee j = j_{rand}$ **then**
      $u_{i,j} \leftarrow v_{i,j}$ ▷ Component is taken from mutant vector
    else
      $u_{i,j} \leftarrow x_{i,j}$ ▷ Component is kept from previous generation
    end if
  end for
end for

Algorithm 2 Exponential crossover

$\vec{u}_i \leftarrow \vec{x}_i$
$j \leftarrow \mathcal{U}\{1, D\}$ ▷ A random component index is chosen
$u_{i,j} \leftarrow v_{i,j}$ ▷ The first component is always transferred
$j \leftarrow j + 1 \mod D$
$L \leftarrow 1$ ▷ Length of transferred string is 1
while $\mathcal{U}(0,1) \leq CR \wedge L < D$ **do** ▷ Stop once all components are transferred
  $u_{i,j} \leftarrow v_{i,j}$
  $j \leftarrow j + 1 \mod D$
  $L \leftarrow L + 1$
end while

2.1.3 Boundary constraint handling

For this project we are dealing with function with box constraints, which means there are lower and upper limits for the components of the vectors. Vectors that lie outside the box do not have valid solutions. This is called a *violation*. Kononova et al. [12] show that during an optimisation run of DE, a large number of violations can occur. A Boundary Constrain Handling Method (BCHM) has the task of correcting this violation in some way. The way a BCHM handles a violation can have a large impact on the performance [2]. For this project we use the projection method, which was used by the original version of DE [3]:

$$v_{i,j} = \begin{cases} lb & \text{if } v_{i,j} < lb \\ ub & \text{if } v_{i,j} > ub \\ v_{i,j} & \text{otherwise} \end{cases} \tag{8}$$

2.2 Reinforcement Learning

In the field of Reinforcement Learning (RL) we can model sequential decision problems as Markov Decision Processes (MDPs). An MDP has the Markov property, meaning that the next state depends solely on the current state and the actions that can be taken. It is defined as a 5-tuple (S, A, T_a, R_a, γ) [21]:

- S is the finite set of all legal states of the environment.
- A is the finite set of all actions in the environment.
- $T_a(s, s')$ is the transition function. It gives the probability that action a on current state s will transition to next state s'. In our case, the environment has access to this function, but the model does not.
- $R_a(s, s')$ the reward the model receives after the model takes action a to make current state s transition to next state s'
- γ is the discount factor $\gamma \in [0, 1]$, used to scale between favouring only immediate rewards ($\gamma = 0$) or to treat all future rewards equally ($\gamma = 1$).

The transition from the current state s and an action a to the next state s' can either be deterministic or stochastic. For a deterministic environment it is possible, with enough experience and if the state space S is not too large, to know for any give state which action will result in the most favourable next state and thus how to maximise the reward. Since we will use RL for DE, which is stochastic, we will assume a stochastic environment for all examples.

An *agent* observes and interacts with an *environment*. After observing the *state* s the agent will choose an *action* a from the action space A, which results in a change of the state to s': $s \to a \to s'$. The perceived 'quality' of the new state affects *reward* r the agent receives. Many observation-action-reward-observation quadruples (s, a, r, s') after each other form a *trace* τ_t^n, where t denotes the timestep of the start of the trace and n denotes the length of the trace:

$$\tau_t^n = \{s_t, a_t, r_t, s_{t+1}, ..., s_{t+n}, a_{t+n}, r_{t+n}, s_{t+n+1}\}\,. \tag{9}$$

The cumulative reward of a trace is known as the *return* $R(\tau_t)$, not to be confused with the reward function, which uses the same notation. From now on R refers to the return, unless explicitly stated otherwise. The return is calculated using a discount factor $\gamma \in [0, 1]$ [21]:

$$\begin{aligned} R(\tau_t) &= r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + ... \\ &= \sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i}\,. \end{aligned} \tag{10}$$

The function that picks the next action for a given state is the *policy* function $\pi(a|s)$, which maps the state space S to a probability distribution over the action space $p(A)$:

$$\pi : S \to p(A)\,. \tag{11}$$

The goal is to find a policy that maximises the return from the initial state s_0. We call such a policy the optimal policy π^*.

2.2.1 Tabular Q-Learning

One way to achieve this is through tabular Q-learning. To understand Q-learning, we must understand what Q represents, which can be derived from the state value V.

Because we are dealing with a stochastic environment, we want to optimise the policy π such that the *expected* return for the start state is maximal. We define $V^\pi(s)$ as:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{p(\tau_t)}[R(\tau_t)|s_t = s] \\ &= \mathbb{E}_{p(\tau_t)}[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i}|s_t = s]\,, \end{aligned} \tag{12}$$

where $\mathbb{E}_{p(\tau_t)}$ denotes the expected value over all possible traces, weighted by the probability that each trace occurs.

The state-action value Q is quite similar to V, but the condition also includes the action:

$$Q^\pi(s,a) = \mathbb{E}_{p(\tau_t)}[R(\tau_t)|s_t = s, a_t = a]\,. \tag{13}$$

The relationship between V and Q is given by:

$$V^\pi(s) = \sum_{a \in A} \pi(a|s) Q^\pi(s,a)\,. \tag{14}$$

We want to find a policy where the state-action value is maximal for any given state:

$$\pi^* = \arg\max_\pi V^\pi(s) = \arg\max_\pi Q^\pi(s,a)\,. \tag{15}$$

This shows that we can use the value functions to find π^*.

Tabular Q-learning works by creating a table of all possible state-action combination of size $|S| \times |A|$. Actually calculating Q requires observing the reward of all possible traces. This adds up very quickly, especially if you remember that both the transition to the next state and the picking of the next action is not deterministic. This is why we estimate Q using the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{16}$$

If all state-action pairs are visited infinitely often, this will converge to the optimal Q-values [27], and thus the optimal policy. Just like tabular Q-learning, because Q is updated using the greedy action $\max_a Q(s_{t+1}, a)$ instead of the behaviour policy's action $Q(s_{t+1}, a_{t+1})$, we call this training method *off-policy*.

To avoid getting stuck in local optima the policy must not always choose the *greedy* option, which is to always choose the action with the highest Q for a given state. ϵ-greedy is an approach to make sure to explore a given fraction of the time. Instead of always choosing the action with the highest Q, the policy will choose a random action $\epsilon \in [0, 1]$ of the time. A variant of ϵ-greedy action selection starts with $\epsilon = 1$ and slowly explores less every step until ϵ is (near) 0. This way, the policy will only explore at first, and slowly move to exploiting more from the knowledge that was gathered before.

2.2.2 Deep Q-Learning

For continuous state spaces it is not possible to have a table of the Q-values, since it would need to be infinitely large. A solution is to divide the continuous state space into discrete bins, but this either results in very many bins, or large bins. Instead, we can use a neural network which acts

similarly to how a table would. If we create a network with $\dim(S)$ input nodes and $|A|$ output nodes (for a discrete action space) we can map the state space to a probability distribution over the action space, just as the policy does in Equation 11.

Instead of updating the Q-values directly, we train the Q-network by minimising a sequence of loss functions. This loss function is the squared difference between predicted Q-value and the target Y:

$$Y_t = r_{r+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)\,, \tag{17}$$

where θ are the parameters of the Q-network.

Notice how this is the same target as seen in tabular Q-learning (Equation 16). This target Y (our best guess of what the Q-value *should* be) together with the calculated Q-value are used to calculate the loss. This can then be used to update the model with backpropagation [7].

Challenges and solutions

Double DQN This target *moves*, which is one of the challenges with Deep Q-learning Networks (DQNs). Each time the network parameters θ are updated, the next target Y_{t+1} will be different. This moving target may cause the DQN to become unstable [33].

One of the solutions to this instability problem is to have the target move less frequently using a Double DQN (DDQN) [33]. The introduces a second network with a second set of parameters θ^- for the target network. Every τ steps the online network parameters θ are copied to the target network θ^-. The difference between the two target functions are minimal but can make a large difference in performance and stability. If we first expand Equation 17 into Equation 18, we can clearly see how minimal the changes are from DQN to DDQN:

$$Y_t^{\text{DQN}} = r_{r+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t)\,, \tag{18}$$

$$Y_t^{\text{DDQN}} = r_{r+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t^-)\,. \tag{19}$$

This target Y_t^{DDQN} is then used to update the online network. Because the target network is not updated for every step, the target itself also does not move as much.

Experience replay Another technique to make deep Q-learning more stable is break the correlation between subsequent states. Since subsequent samples are strongly correlated, the algorithm will tend to under-explore the state space. Also, after a while the network may forget previous behaviour since it has not seen certain parts of the state space for a while.

Experience replay is a technique that aims to circumvent these problems and thus prevent getting stuck in local minima by introducing a *replay buffer*. This buffer is filled with the last N *experiences* (s_t, a_t, r_t, s_{t+1}). Every training step, the algorithm randomly selects from the replay buffer instead of using the last sample. Because this ensures that the states are seen in random order, the correlation between each step is reduced, and the older samples are still visited after some time such that they are not 'forgotten', which improves coverage [21]. This effectively adds a form of supervised learning.

3 Related Work

While DE was designed with ease of use in mind [26], research has shown that its hyperparameters can greatly influence its performance [23]. By know, it is well known that different variants of DE can perform very differently depending on the characteristics of the problem that is being solved and that its settings are highly interdependent. Mezura-Montes et al. [17] test eight variants on DE to find that while "best/1/bin" performed best overall, being the only variant to solve the unimodal and nonseparable problem in their testing, "rand/2/dir" performed slightly better on multimodal and nonseparable problems.

The 'No Free Lunch' theorem states that "if an algorithm does particularly well on average for one class of problems then it must do worse on average over the remaining problems." [35]. In other words, averaged across all possible problems, any two algorithms will perform identically. This however goes both ways, so it also means that finding the best performing algorithm for each problem can potentially greatly increase the overall performance over the average. On black-box optimisation, this is by definition not possible beforehand. Any algorithm that attempts to tweak its internal setting online to increase performance, will still inevitably 'fail' on some problems. It is however important to keep in mind that the subset of problems we come across in practice is of course infinitely smaller than all problems.

While the original paper for DE considers the algorithm easy to set up [26], this algorithm had a large amount of research into its settings and parameters. Pant et al. [20] cover 283 papers to give an extended summary of DE. They show how improvements on the original version of DE have been made by researching aspects like population generation, mutation schemes, crossover schemes, variation in parameters and hybridised variants. Hybridisation in metaheuristic is a practice in which a metaheuristic is combined with one or more other optimisation techniques.

At first, research into parameter settings was about finding settings that work well in general [6, 14, 24, 37]. Adaptively tuning parameters of DE during a run has been an active research topic for since its suggestion in 2005 by Teo [32], where the population size is changed throughout the run. This idea has since been developed further resulting in state-of-the-art methods that improve over the original version of DE [36, 30, 34].

The field of deep reinforcement learning is very active and relatively new. In 2013 DeepMind developed a deep learning model that could successfully learn to play Atari 2600 games from high-dimensional sensory input using reinforcement learning [18]. Van Hasselt et al. [33] then show that DQNs have a tendency to substantially overestimate Q-values, even in best-case scenarios. Tabular Double Q-Learning was introduced a few years prior to combat these issues for the tabular Q-learning. Van Hasselt et al. [33] showed that these principles can be adapted to DQN, creating Double DQN, or DDQN. This led to more stable and reliable learning.

Combining RL and DEs has been an active research topic for the last five years. Li et al. [13] use a Q-table to select the mutation strategy to develop a version of DE suited for multi-objective optimisation. With only 9 states and 3 actions, there were only 27 Q-values. The states were encoded from the ranking based on the two conditions, and individuals get rewards based on if they move to a better state. In [10] a small Q-table with only 2 states and 3 actions is used. The offspring is either better than its parent, in which case the reward is 1, or not, in which case the reward is 0. The three actions determined if the mutation factor F is either decreased or increased by 0.1, or if it should stay the same. Tan et al. [29] use a Q-table to select one of 4 mutation strategies based of 4 states.

In [4] a dataset was generated by optimising functions from the BBOB with different parameters of DE and then using a neural network to find patterns between the DE parameters and the performance. It was shown that there is strong interdependence between the parameters.

Tan et al. [28] implement a Deep Q-Network to pick one out of three mutation strategies

based of a state vector with four fitness landscape features. These features are calculated from e.g. correlation between fitness values and distances between individuals and the distance to the best individual in the population. Rewards are given based on the amount of individuals that improve over their parents.

This thesis is most similar to the work of Sharma et al. [25], where a larger network is used to select the mutation operators based on a list of 99 state features. Three reward functions were tested, of which we use the most successful one. This thesis extends on this by also tuning the mutation factor F and by tweaking the state features such that the DE can be parallelised.

4 Methods

This section describes the methods used for the experiments. First, an explanation on how DE and RL are combined to create a new algorithm (Section 4.1). Sections 4.2 and 4.3 describe the state representation the reward function respectively, which are heavily based on the work by Sharma et al. [25]. Section 4.4 gives a detailed description of the experiments.

All source code is written in Python and is available on GitHub[1], along with checkpoints of the trained models, additional plots and code for recreating the figures.

4.1 DE-DDQN

Figure 1 gives us an overview of the algorithm. Mutation takes the latest generation $\vec{\mathbf{x}}$ and creates a population of mutant vectors $\vec{\mathbf{v}}$. $\vec{\mathbf{x}}$ and $\vec{\mathbf{v}}$ are then combined in the Crossover phase which generates the population of trial vectors $\vec{\mathbf{u}}$. This is then evaluated to produce $f(\vec{\mathbf{u}})$. The Selection phase compares $f(\vec{\mathbf{u}})$ with the previous generation to produce the next generation. If either the evaluation budget is depleted or the optimum is found, the algorithm terminates, else it will pass the function values to the mutation strategy selector.

The function values and positions of the individuals in the past generations are used to create a state vector $\vec{s}$ which gives the Neural Network the information it needs to compute the vector of Q-values, $\vec{Q}$, which represents the expected cumulative reward for every mutation strategy. This is a two-dimensional table of the Q-value for every action, for every individual in the generation. For every individual the action with the maximum Q-value is given to the Mutation phase.

From the 'point-of-view' of the model, the DE is the environment from which states are observed. This means that for every step in this environment (a generation) we need to make multiple decisions (choose NP mutation strategies based on NP state vectors). In other words, for every step in the environment the model takes NP steps training the model. Because of this, we need to use a slightly tweaked version of DDQN which supports this situation.

4.2 State representation

The state function must produce a state feature vector for every individual in the population that gives the neural network the information it needs to pick the optimal mutation strategy. These feature can be computed only from the information that is known: the positions of the individuals, their fitness values and information from previous generations. Ideally the state feature encompasses everything that is needed to find the optimal action.

The exact fitness landscape is of course unknown, but we can give the model statistics that were gathered from previous generations, e.g. the normalised relative fitness or standard deviation of the fitness. The relative positions of the individuals is also taken into account, since the model will decide the mutation strategy for each individual separately. For example, the distance between randomly selected individuals or the distance to the best individual so far could give the model insight about whether that individual is exploring of exploiting.

The performance of the mutation strategies in previous generations of a run can also be used as hints as to which will perform well later on.

[1] `https://github.com/Marcaroni8/MSS-DE-DRL`

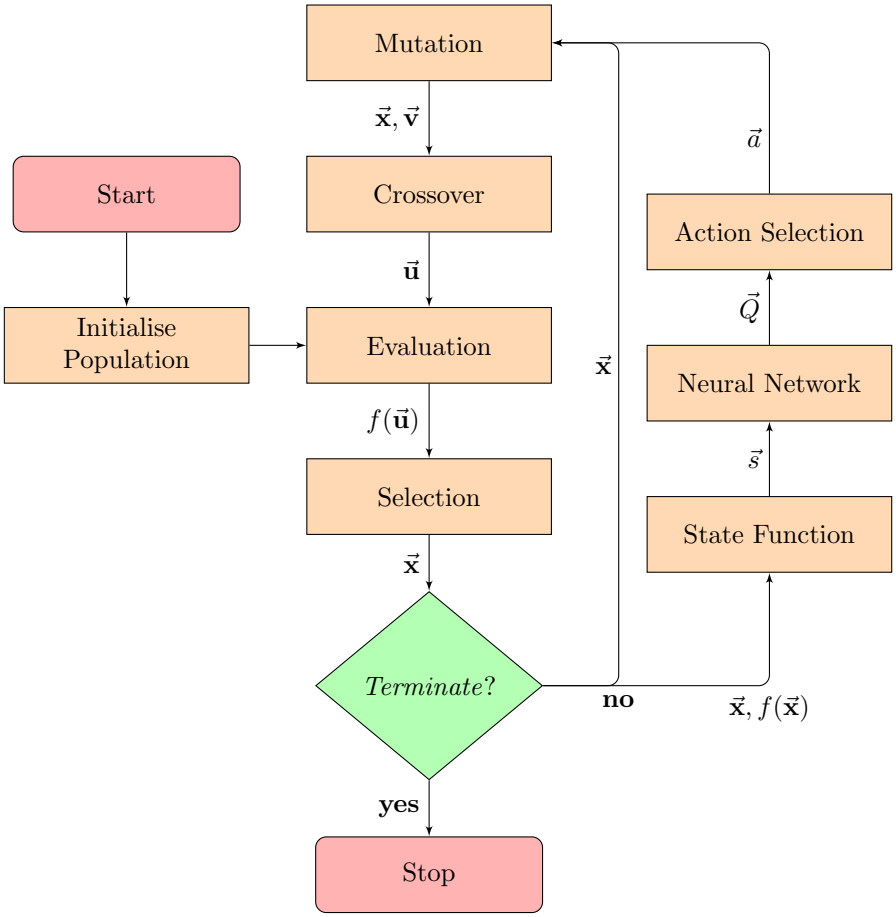

Figure 1: Diagram of the proposed algorithm. When the algorithm starts, the population $\vec{\mathbf{x}}$ is initialised and evaluated $f(\vec{\mathbf{x}})$. For the first generation, all individuals go through the selection phase. If the termination requirements are met, the algorithm stops. The rightmost three boxes represent the added functionality to a traditional DE. The current population is used together with an memory of previous generation to calculate the state features $\vec{s}$. The model uses the state features to produce a vector of Q-values for every action for every individual $\vec{Q}$. The action with the highest Q-value is selected for every action. The parent population $\vec{\mathbf{x}}$ and the actions $\vec{a}$ are used to produce the mutation vectors $\vec{\mathbf{v}}$, after which crossover produces the trial vectors $\vec{\mathbf{u}}$. This loop continues until the termination requirements are met.

Every generation the following values are stored for every individual: the action (mutation strategy) that was used, if and how much the new individual improved over its parent, if and how much the new individual improved over the best individual from the parent generation and if it's better than the median fitness of the parent generation:

1. $OM_1 = f(\vec{x}_i) - f(\vec{u}_i)$
2. $OM_2 = f(\vec{x}_{\text{best}}) - f(\vec{u}_i)$
3. $OM_3 = \text{median}(f(\vec{x})) - f(\vec{u}_i)\,,$

where $f(\vec{x}_i)$ is the fitness of the parent, $f(\vec{u}_i)$ is the fitness of the offspring, $f(\vec{x}_{\text{best}})$ is the fitness of the best individual in the parent population and $\text{median}(f(\vec{x}))$ is the median fitness of the parent population. $OM_m(g, k, st)$ gives the value of metric m for generation g, for individual k and mutation strategy st. Since the information from the most recent generations of the run are most useful, we only store the last $gen = 10$ generations. No information of previous runs is stored.

With these values we can compute the number of successful application of each mutation strategy, just as the sum of fitness improvements for each of the four fitness improvements that were stored. Lastly, we can also look at just the current and previous generations and note the difference in improvement.

Sharma et al. [25] use an extra set of state features based on a window. In their implementation of DE every individual goes through the mutation, crossover and selection step one at a time. This does not alter how the DE finds solutions in any way, but it does make the code much slower to run. Computing the individuals one by one makes it possible to choose the mutation operator one at a time, and thus also make decisions based on previously handled individuals from the *current* generation. For this project we chose to compute the whole generation at once to allow parallelisation, which makes the code run much more efficiently, especially for larger populations, but at the drawback of making it impossible to put this potentially useful information into the state features.

Index	Feature	Notes
1	$\frac{\sum_{j=1}^{NP} \frac{f(\vec{x}_j)}{NP} - f_{\text{bsf}}}{f_{\text{wsf}} - f_{\text{bsf}}}$	The best-so-far individual is the best individual in the population $f_{bsf} = f(x_{best}) = \min_x(f(x))$. f_{wsf} is the worst-so-far fitness up to this step within a single run.
2	$\frac{\sigma_{j=1,\ldots,NP}(f(\vec{x}_j))}{\sigma^{\max}}$	$\sigma(\cdot)$ calculates the standard deviation and $\sigma^{\max}$ is the standard deviation when half of the population has fitness f_{wsf} and the other half has f_{bsf}
3	$\frac{FE^{\max} - t}{FE^{\max}}$	$FE^{\max}$ is the maximum number of function evaluations per run, and $FE^{\max} - t$ gives the remaining number of evaluations at step t.
4	$\frac{stagcount}{FE^{\max}}$	*stagcount* is the *stagnation counter*, i.e. the number of function evaluations (steps) without improving f_{bsf}
5-9	$\frac{dist(\vec{x}_i, \vec{x}_j)}{dist^{\max}}, \forall j \in r_1, r_2, r_3, r_4, r_5$	$dist(\cdot)$ is the Euclidean distance between two solutions; $dist^{\max}$ is the maximum distance possible, calculated between the lower and upper bounds of the decision space; r_1, r_2, r_3, r_4, r_5 are random indexes.
10	$\frac{dist(\vec{x}_i, \vec{x}_{\text{best}})}{dist^{\max}}$	
11-15	$\frac{f(\vec{x}_i) - f(\vec{x}_j)}{f_{\text{wsf}} - f_{\text{bsf}}}, \forall j \in r_1, r_2, r_3, r_4, r_5$	
16	$\frac{f(\vec{x}_i) - f(\vec{x}_{\text{best}})}{f_{\text{wsf}} - f_{\text{bsf}}}$	
17-40	$\sum_{g=1}^{gen} \frac{N_m^{\text{succ}}(g,st)}{N^{\text{tot}}(g,st)}$	For each st and $m \in 1, 2, 3$ and normalised over all operators; gen is the maximum number of recent generations recorded; $N_m^{\text{succ}}(g, st)$ and $N^{\text{tot}}(g, st)$ are successful and total applications of st according to OM_m at generation g.
41-64	$\frac{\sum_{g=1}^{gen} \sum_{k=1}^{N_m^{\text{succ}}(g,st)} OM_m(g,k,st)}{\sum_{g=1}^{gen} N^{\text{tot}}(g,st)}$	
65-88	$\frac{OM_m^{\text{best}}(gen,st) - OM_m^{\text{best}}(gen-1,st)}{OM_m^{\text{best}}(gen-1,st) \cdot \lvert N^{\text{tot}}(gen,st) - N^{\text{tot}}(gen-1,st) \rvert}$	$OM_m^{\text{best}}(g, st)$ is the maximum value of $OM_m(g, k, st)$
89-112	$\sum_{g=1}^{gen} OM_m^{\text{best}}(g, st)$	

Table 1: List of the state features. Based on DE-DDQN [25].

4.3 Reward function

The reward is used during training by the neural network to gauge how well previous actions did. Generally, the reward must be high when an action did what we wanted (make large steps towards the optimum), and low if it did not (no or very small improvement). We want the neural network to select the mutation strategies that lead to optimum the quickest, and the way it gets there does not matter as much. This is why rare but big steps towards the optimum can be preferable over many small steps. It is thus important to set up the reward function such that it encourages picking the mutation strategies that lead to finding the global optimum the quickest. The reward is calculated for every individual in the population in every generation, since actions are chosen for every individual every generation. This is then stored together with the state, action and next state.

Sharma et al. [25] test three different reward functions. These functions use the parent $\vec{x}_i$ and offspring $\vec{u}_i$ and the best fitness so far f_{bsf} and optimal fitness f^*:

$$\text{R1} = \max\{f(\vec{x}_i) - f(\vec{u}_i), 0\} \tag{20}$$

$$\text{R2} = \begin{cases} 10 & \text{if } f(\vec{u}_i) < f_{\text{bsf}} \\ 1 & \text{else if } f(\vec{u}_i) < f(\vec{x}_i) \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

$$\text{R3} = \max\{\frac{f(\vec{x}_i) - f(\vec{u}_i)}{f(\vec{u}_i) - f^*}, 0\} \tag{22}$$

It was concluded that the model learned to pick the mutation operator best with reward function R2 (Equation 21). For this reason we will use reward function R2 for this project. This function is generalised for all fitness function, i.e. the offset and scale of the function does not affect the reward function. The function returns 0 if the individual did not improve, 1 if it improved over its parent and 10 if it improved over the best so far solution. This way improvements are rewarded more if they find a better solution than has been found so far.

4.4 Experiments

For the experiments we use 4 mutation operators (Equation 4–7) and 2 mutation rates $[0.3, 0.8]$. This gives 8 possible mutation strategies. Table 2 lists the settings of the DE. The values that are adaptive are noted within [brackets].

To assess the performance we use the benchmark suite BBOB (Black-Box Optimisation Benchmarking) [5], from the COCO (COmparing Continuous Optimizer) platform and which contains 24 noiseless functions. BBOB has the feature to create multiple different variations (instances) of its functions by applying transformation methods to the base forms of the 24 functions, which acts as a form of data augmentation. These transformations rotate and translate the function, while keeping the 'shape' of the fitness landscape intact. For the models to be able to generalise, it is important to train and test on multiple instances [15].

The evaluation budget is 10^4 and for simplicity we only use functions with dimensionality $D = 10$. We train the models on random instances of this set of 24 functions and set the target fitness 10^{-8} above the optimal fitness:

$$f_{\text{target}} = f^* + 10^{-8} \tag{23}$$

On difficult multimodal functions DEs can converge prematurely on a local optimum and stagnate. If the DE does not manage to escape the local optimum a large fraction of the budget can be wasted. COCO encourages restarts [9] to improve the chances of the algorithm finding a better solution. A restart resets the internal state of the algorithm, possibly using information from the previous attempt to tweak internal parameters, like population size, to improve performance. For this project we use independent restarts that do not change any parameters [1, 8].

The criteria for the algorithm to restart must make sure the algorithm does not restart prematurely and prevent the algorithm from converging on the actual global optimum, while also not too late and waste resources. In [38], a range of stopping criteria were examined. We adopt the strategy named *Diff*:

$$\textbf{converged} = \begin{cases} \texttt{true} & \text{if } f(x_{worst}) - f_{best} < \mathcal{E} \\ \texttt{false} & \text{otherwise} \end{cases}. \tag{24}$$

$\mathcal{E}$ is recommended to be set one order of magnitude smaller than the target precision (Equation 4.4) [38], so we use $\mathcal{E} = 10^{-9}$.

DE parameters	**Parameter value**
Population size (NP)	100
Evaluation budget	10^4
Mutation operator	[rand/1, rand/2, rand-to-best/2, curr-to-rand/2]
Mutation factor (F)	$[0.3, 0.8]$
Crossover operator	Binomial
Crossover rate (CR)	0.9
Selection strategy	Paired elitist
BCHM	Projection (Equation 8)

Table 2: The DE settings. The adaptive values are noted within [brackets].

4.4.1 Training the models

To assess the performance we train three separate models from scratch. Each model is trained for 10^4 episodes (entire runs of DE until termination), which means there are ~10^6 generations, depending on the average length of an episode. Before training starts, the buffer is filled with experiences by picking random actions. Then the ϵ-greedy policy takes over, where the exploration factor starts off at $\epsilon = 1.0$ and decays to 0.075. The replay buffer fits 10^6 experiences and is filled using FIFO. This means that with $NP = 100$ the entire buffer is refreshed every 10^4 generations or ~10^2 episodes. The target network is refreshed every $\tau = 10$ episodes. All settings for the model are listed in Table 3. For this project, minimal tweaking of the model's hyperparameters was done since it was deemed mostly out of scope, and most settings we based on the work by Sharma et al. [25]. Most notably the nodes per layer is increased to make the network more capable of understanding the complex state space and τ was increased to make the target more stable.

General model parameters	**Parameter value**
Model type	Double Deep Q-Network
Hidden layers	4
Nodes per layer	1024
Activation function	Rectified Linear Unit (ReLU) [19]
State features history (*gen*)	10
Training parameters	**Parameter value**
Training algorithm	Adam [11]
Learning Rate	0.001
Batch size	512
Target network update τ	Every 10 episodes
Training policy	ϵ-greedy with ϵ decay
Exploration factor ϵ	$1.0 \rightarrow 0.075$
Decay factor	0.9995 per individual ≈ 0.95 per generation
Discount rate γ	0.95
Replay buffer size	10^6 experiences $\approx 10^2$ episodes
Train time	10^4 episodes
Testing parameters	**Parameter value**
Online policy	Greedy

Table 3: Hyperparameters of DDQN models.

5 Results

The three trained models are tested for 20 runs on 5 instances of each function, resulting in 100 runs per function, and 2400 runs in total. These models are simply named *model 1*, *model 2* and *model 3*. We first analyse the performance of the mutation strategy selection models by comparing how fast they reach a set of targets in Section 5.1. To give more insight on what decision the models are making, the actions were analysed in Section 5.2. Lastly, to give a bit more insight into why the performance differs per function and per model the average reward is analysed in Section 5.3.

To improve readability and avoid confusion, when a mutation strategy is *italicised*, we are talking about a constant policy using that mutation strategy. If they are **bold**, we are talking about the strategy or action itself.

5.1 Performance analysis

Since the time in which the DE finds the final target or the error when the budget is depleted tell only parts of the complete story, we set a range of targets. By setting 51 targets spaced uniformly on the logarithmic scale $[10^{-8}, 10^2]$ from the global minimum, we can note for each run *if*, and *when*, a target is hit. We can than plot for every generation how many of these targets have been hit. Averaging these plots for the total of 2400 runs shows us how many targets are hit at each point of the optimisation progress. We can do this for the 8 constant policies, a random policy and the three models.

Figure 2 shows the plot of the Empirical Cumulative Distribution Function (ECDF) of the average over all 24 BBOB functions. The three dotted lines show the performance when each of the three trained models were selecting the mutation strategies respectively. The coloured solid lines show the performance without any online mutation selection, and the black line shows the performance when the mutation strategies are selected at random. Data from before 100

evaluations can not be collected, since that is the amount of function evaluations used for a single generation, i.e. $NP = 100$. Keep in mind that for all ECDF plots, the y-axis scales with the lines to improve readability.

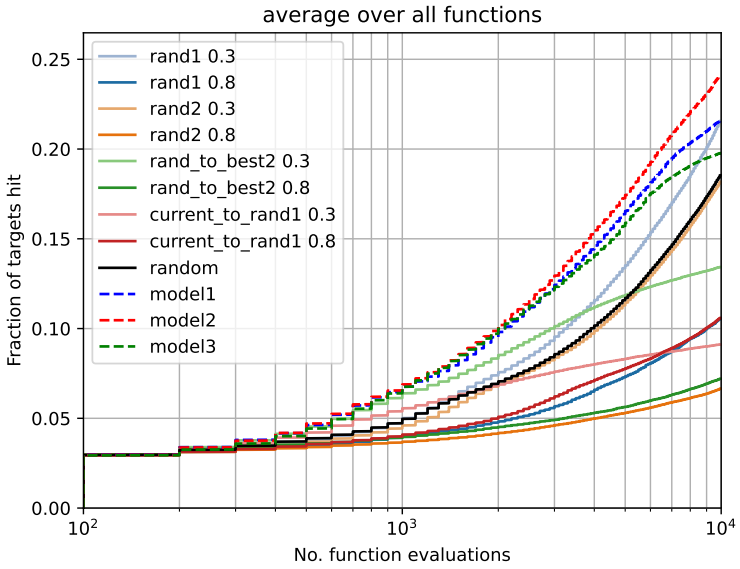

Figure 2: ECDF of average fraction of targets hit of 100 runs over 5 instances, averaged over all 24 functions. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

To give better insight into what is happening for each function separately we can look at Figures 3-5, which show ECDF plots for all 24 functions. If we for instance look at the black line in the plot for $f1$, the Sphere function, we can see that when selecting mutation strategies at random will cause the DE to hit on average ~10% of the targets after 10 generations (10^3 function evaluations for $NP = 100$) and ~60% when the budget is depleted. The average fraction of targets each policy hits at the end of all runs can be found in Table 4.

We can use the trapezoidal rule to calculate the Area Under the Curve (AUC) of the ECDF plots, giving us a single figure to measure the performance for each algorithm, which we see in Table 5. If the DE hits most targets quickly, the AUC of the ECDF will be high. If few targets are found, or if the targets are found only at the end, the value will be low. AUC is calculated with $\log_{10}$ for the x-axis.

In Figure 2 we can see that on average, all three models perform better than when choosing the mutation strategies randomly. The mutation strategy *rand1 0.3* manages to overtake *model 3* before the budget is depleted, because it seems to plateau about halfway through the runs on average.

If we then look at Figures 3-5 we see that the performance differences between the three models is much bigger for different functions. On $f1$, which is the easiest problem in the BBOB benchmark. *Model 1* and *2* almost always find the optimum before the budget is depleted, while all other variants struggle. This does mean that the neural networks are capable of improving over the constant or random variants given the state features.

For $f2$ we see again that the differences in performance is large. Note that the variants for which we do not see the lines in the plots, none of the 51 targets are ever hit before evaluation budget is depleted. The reason why much fewer targets are hit in this function than on $f1$ is the much higher *conditioning*[2]. With fixed value targets but higher function values much fewer targets are hit.

While for most functions at least one of the models is, sometimes minimally, better than the variants with constant or random mutation strategies, for others the models all fail completely. $f5$, the linear slope, has all three models performing significantly worse the *rand1 0.3*, with even a random strategy beating out two of the models.

On some functions, e.g. $f6$, $f8$ and $f14$, the models seem to find reach targets very quickly, but at some point struggle to find any more. The is also why sometimes the models have a higher AUC (Table 5), while reaching a smaller fraction of the targets when the budget is depleted (Table 4).

On both $f16$ and $f23$ all variants seem to perform almost identically. These two functions are highly rugged, repetitive and multimodal. Because of the little global structure it is difficult to find the basin of the global optimum, so with many local optima with nearly identical fitness value, most runs get stuck at the same error.

[2]If the ratio of steepest slope to flattest slope is high, the function is said to have high conditioning.

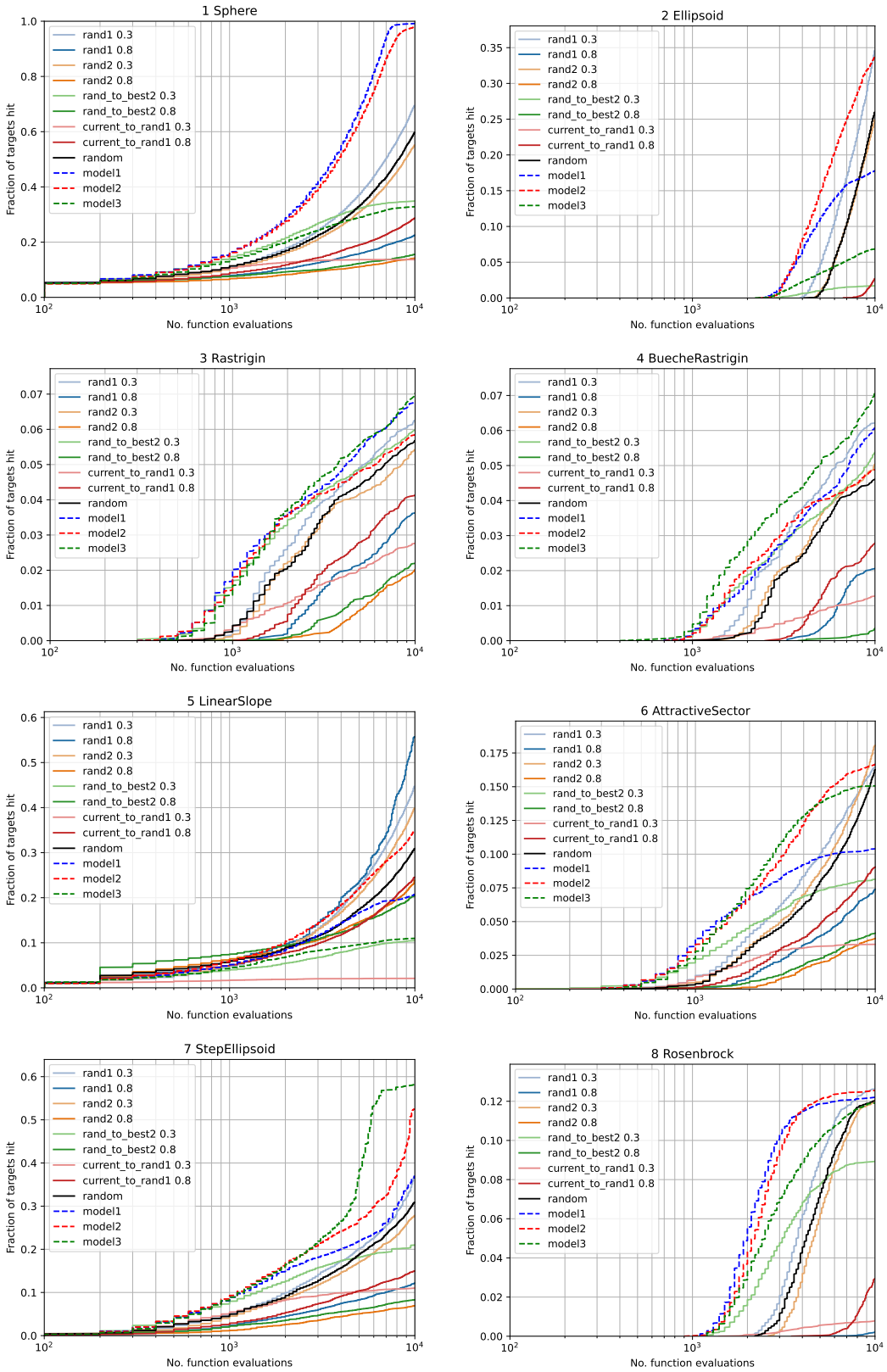

Figure 3: ECDF of average fraction of targets hit of 100 runs over 5 instances for $f1 - f8$. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

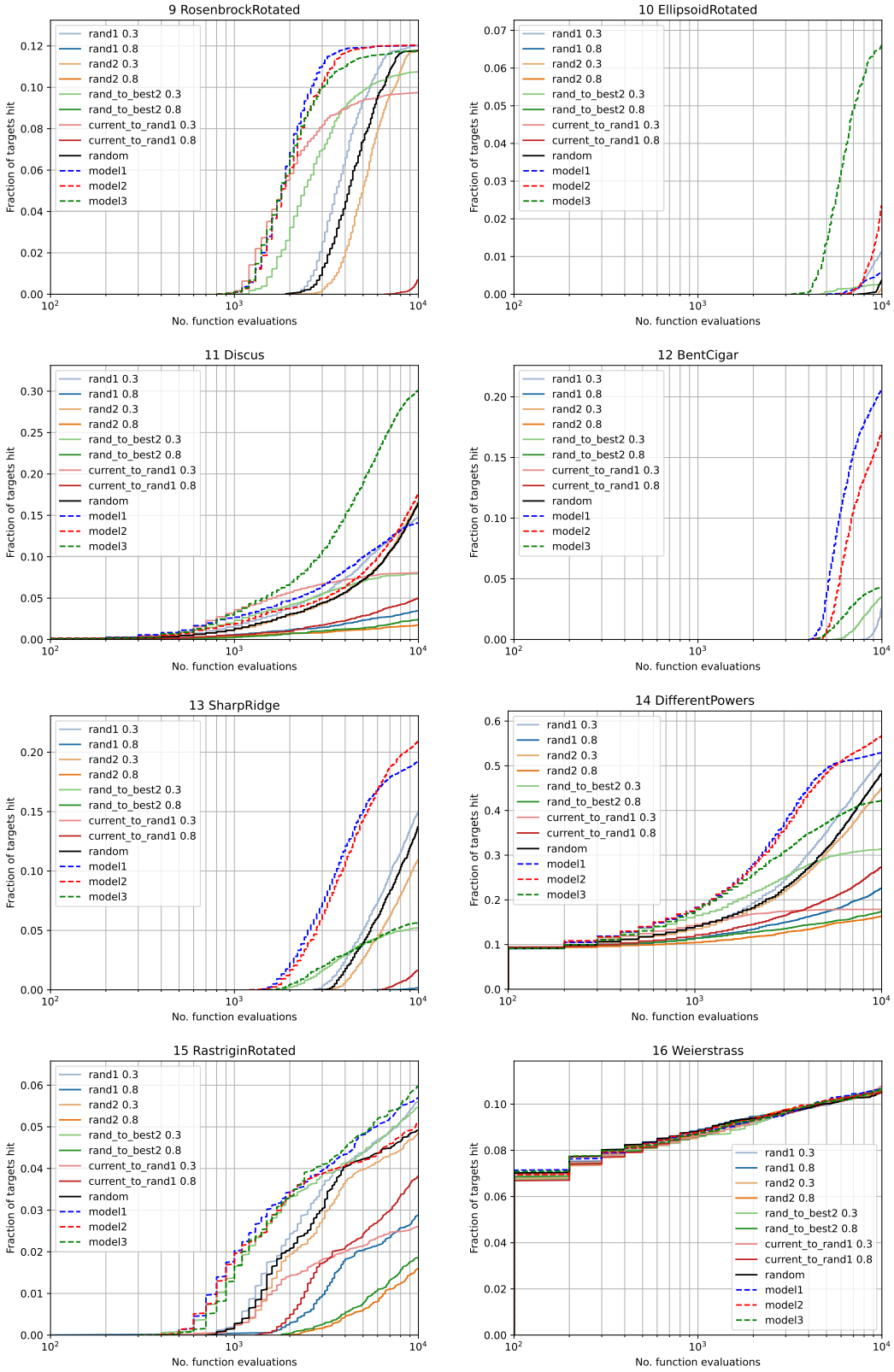

Figure 4: ECDF of average fraction of targets hit of 100 runs over 5 instances for $f9 - f16$. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

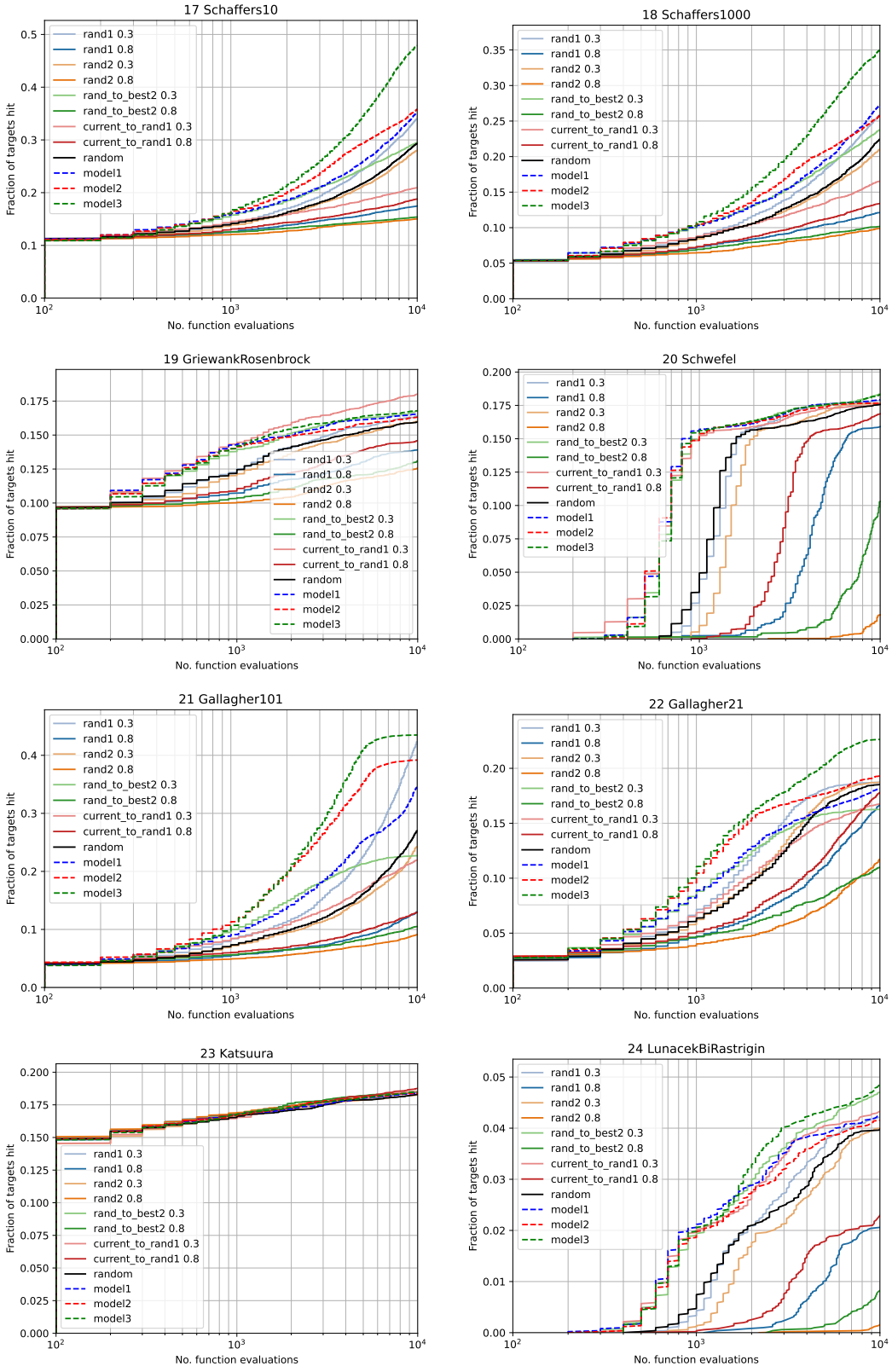

Figure 5: ECDF of average fraction of targets hit of 100 runs over 5 instances for $f17 - f24$. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	avg
rand1 0.3	0.694	**0.346**	0.063	0.062	0.447	0.164	0.366	**0.126**	0.120	0.011	0.148	0.026	0.150	0.514	0.056	0.105	0.340	0.258	0.164	0.179	0.423	0.187	0.185	0.042	0.216
rand1 0.8	0.225	0.000	0.036	0.021	**0.557**	0.074	0.122	0.002	0.000	0.000	0.035	0.000	0.002	0.226	0.029	0.105	0.175	0.121	0.139	0.159	0.130	0.168	0.185	0.021	0.105
rand2 0.3	0.551	0.249	0.054	0.050	0.399	**0.181**	0.278	0.121	0.118	0.002	0.166	0.000	0.110	0.450	0.049	0.105	0.280	0.210	0.160	0.177	0.243	0.188	0.183	0.040	0.182
rand2 0.8	0.142	0.000	0.020	0.000	0.239	0.038	0.069	0.000	0.000	0.000	0.018	0.000	0.000	0.164	0.016	0.105	0.150	0.099	0.126	0.018	0.091	0.117	0.185	0.002	0.067
rand-to-best2 0.3	0.349	0.017	0.060	0.054	0.105	0.081	0.210	0.089	0.108	0.003	0.079	0.035	0.052	0.314	0.055	0.106	0.295	0.238	0.168	**0.184**	0.227	0.163	0.185	0.047	0.134
rand-to-best2 0.8	0.156	0.000	0.022	0.004	0.204	0.041	0.083	0.000	0.000	0.000	0.024	0.000	0.000	0.174	0.019	0.106	0.154	0.102	0.131	0.103	0.105	0.110	0.185	0.008	0.072
current-to-rand1 0.3	0.137	0.000	0.028	0.013	0.021	0.033	0.110	0.008	0.098	0.000	0.081	0.000	0.000	0.179	0.026	**0.108**	0.209	0.166	**0.180**	0.176	0.220	0.168	0.185	0.043	0.091
current-to-rand1 0.8	0.287	0.027	0.041	0.028	0.245	0.090	0.150	0.029	0.007	0.000	0.050	0.000	0.017	0.274	0.038	0.105	0.188	0.134	0.146	0.169	0.129	0.178	**0.188**	0.023	0.106
random	0.597	0.260	0.057	0.046	0.309	0.163	0.309	0.120	0.118	0.004	0.165	0.000	0.137	0.482	0.049	0.105	0.293	0.224	0.160	0.176	0.270	0.185	0.183	0.040	0.185
model1	**0.992**	0.178	0.068	0.061	0.207	0.104	0.369	0.122	0.120	0.006	0.141	**0.205**	0.192	0.529	0.057	0.107	0.352	0.273	0.166	0.179	0.345	0.182	0.184	0.043	0.216
model2	0.979	0.338	0.058	0.049	0.349	0.167	0.525	0.126	**0.120**	0.024	0.175	0.171	**0.210**	**0.566**	0.051	0.106	0.358	0.258	0.163	0.177	0.392	0.193	0.184	0.042	**0.241**
model3	0.328	0.069	**0.070**	**0.071**	0.110	0.151	**0.582**	0.119	0.118	**0.066**	**0.301**	0.043	0.056	0.422	**0.060**	0.107	**0.479**	**0.350**	0.168	0.183	**0.435**	**0.226**	0.185	**0.049**	0.198

Table 4: The average fraction of targets hit when the budget is depleted for each function, including the overall average in the last column **avg**. For each column the highest fraction of targets hit is **bold and coloured in grey**. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	avg
rand1 0.3	0.368	0.056	0.037	0.028	0.196	0.075	0.175	0.048	0.048	0.001	0.072	0.001	0.035	0.389	0.033	0.173	0.332	0.217	0.254	0.155	0.239	0.182	0.334	0.028	0.145
rand1 0.8	0.196	0.000	0.014	0.004	**0.224**	0.027	0.073	0.000	0.000	0.000	0.019	0.000	0.000	0.255	0.012	0.174	0.264	0.153	0.224	0.062	0.124	0.124	0.335	0.007	0.096
rand2 0.3	0.319	0.034	0.031	0.021	0.186	0.066	0.146	0.036	0.032	0.000	0.061	0.000	0.021	0.357	0.028	0.172	0.309	0.195	0.247	0.140	0.177	0.169	0.330	0.023	0.129
rand2 0.8	0.154	0.000	0.006	0.000	0.155	0.013	0.043	0.000	0.000	0.000	0.011	0.000	0.000	0.226	0.004	0.174	0.249	0.138	0.210	0.002	0.108	0.098	**0.336**	0.000	0.080
rand-to-best2 0.3	0.344	0.006	0.043	0.027	0.094	0.062	0.178	0.047	0.062	0.001	0.061	0.004	0.022	0.366	0.039	0.172	0.342	0.232	0.268	0.200	0.234	0.182	0.334	0.039	0.140
rand-to-best2 0.8	0.168	0.000	0.008	0.000	0.165	0.016	0.058	0.000	0.000	0.000	0.012	0.000	0.000	0.239	0.005	0.173	0.256	0.145	0.215	0.018	0.122	0.107	0.335	0.001	0.085
current-to-rand1 0.3	0.199	0.000	0.016	0.006	0.032	0.026	0.107	0.003	0.069	0.000	0.072	0.000	0.000	0.281	0.017	0.173	0.296	0.187	**0.280**	0.201	0.195	0.164	0.331	0.037	0.112
current-to-rand1 0.8	0.220	0.001	0.019	0.007	0.142	0.035	0.088	0.003	0.000	0.000	0.023	0.000	0.001	0.278	0.016	0.173	0.271	0.160	0.231	0.089	0.134	0.136	0.334	0.010	0.099
random	0.341	0.035	0.033	0.019	0.171	0.060	0.163	0.041	0.041	0.000	0.062	0.000	0.029	0.368	0.030	**0.176**	0.317	0.200	0.252	0.155	0.186	0.165	0.331	0.027	0.133
model1	**0.597**	0.060	**0.047**	0.029	0.150	0.085	0.218	**0.080**	**0.085**	0.001	0.083	**0.044**	**0.087**	**0.494**	**0.043**	0.175	0.357	0.239	0.271	**0.203**	0.251	0.187	0.333	0.038	0.173
model2	0.566	**0.091**	0.043	0.029	0.196	**0.108**	0.257	0.077	0.082	0.001	0.072	0.029	0.084	0.494	0.039	0.175	0.376	0.245	0.267	0.201	0.329	0.210	0.334	0.036	**0.181**
model3	0.318	0.021	0.047	**0.039**	0.104	0.107	**0.315**	0.064	0.081	**0.014**	**0.145**	0.008	0.023	0.425	0.042	0.175	**0.412**	**0.282**	0.271	0.200	**0.347**	**0.228**	0.333	**0.041**	0.168

Table 5: AUC of ECDF plots from Figures 3-5 of targets hit when the budget is depleted for each function, including the overall average in the last column **avg**. For each column the highest fraction value is **bold and coloured in grey**. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

5.2 Action analysis

To try and understand why the trained models perform differently on each function and from each other, we can analyse the actions that were taken for each model on each function. The actions taken by *models 1*, *2* and *3* are noted in Tables 6, 7 and 8 respectively.

rand2 0.8 and **rand-to-best2 0.8** are performing the worst of all eight mutation strategies (Tables 4-5), which explains why the first is chosen $< 0.0\%$ of the time by all three models, and why the latter is only chosen by *model 3* 1.9% of the time. Interestingly, while **rand1 0.3** is chosen often for *model 1* and *model 3*, *model 2* practically never chooses that mutation strategy. While all three models perform better *on average* than picking a constant model (Figure 2), there are big differences in the mutation strategies that they pick, which explains why the differences in performance are large on each function (Figures 3-5).

To give a better understanding on what is happening during a run, we can plot the actions that were taken together with the error $f_{best} - f^*$. If the error goes up, which normally can not happen for paired elitist selection, the population has converged and has restarted (see Section 4.4). Figures 6-8 show on the left for *single runs* on $f1$ the actions taken for each generation with the left y-axis showing the actions per generation, together with the error on the right y-axis. On the right the Q-value is plotted for the individual that had the lowest function value at the end of the run that is plotted on the left. Eight lines are visible; one per action. For Figures 9-11 two representative runs were picked for each model to give more insight. More figures can be found in the GitHub repository.

Although *model 1* and *2* perform similarly on $f1$ (Figure 3), Figures 6-8 show how their strategies are very different. *Model 1* shows very consistent behaviour and switches strategies just a few times at an almost stable interval, while *model 2* switches mutation strategies much more often. *Model 3* struggles much more, especially after the first 20 generations. *Models 1* and *2* are apparently both able to use the state features to pick mutation strategies that lead to finding the optimum quicker, while *model 3* does not. It is interesting to note that all three models make little use of mutation strategy **rand1 0.3**, even though it performs best out of all variants with constant mutation strategies.

In the plots on the right in Figures 6-8 we see that the Q-values are often very close together, meaning that the model at that point does not expect there to be much difference between the future rewards for the actions from that state. Secondly, the Q-values tend to oscillate up and down, meaning that the model constantly get more or less confident about how much reward it expects. The reward is analysed further in Section 5.3.

Figure 9 shows the plot for *model 1*'s actions on $f3$ and $f12$. Firstly, for $f3$ we see that the model outperforms all constant variants, even though the model picks almost only **rand1 0.3**. This shows that changing the strategy only a few times in the beginning can make an improvement over a constant strategy, albeit not much. For $f12$ *model 1* performed the best out of all variants, with 6 out of the 8 constant mutation strategies failing to hit any target on all 100 runs. BentCigar is a difficult function, because it is non-separable and very highly conditioned. There is only one single optimum, but it is difficult for a DE to get close to it. Again, by combining just three mutation strategies, *model 1* is able to hit on average over a fifth of the targets.

As we see in Figure 10, for $f6$ *model 2* seems to switch mutation strategy every few generations. While not ending with the highest fractions of targets hit, the AUC of the ECDF is the highest for *model 2* (Table 5). On $f14$ *model 2* was able to surpass *model 1* with the average fractions hit about half-way through the runs. In Figure 10 we see how the model decides to only pick **rand2 0.3** after 60 generations.

Model 3 managed to outperform all other variants on $f4$. In Figure 11 switches between just two actions; **rand1 0.3** and **rand-to-best2 0.3**. *rand1 0.3* outperformed all other variants (Figure 3), but *rand-to-best2 0.3* actually performed somewhat worse. By combining these two actions *model 3* is able to find a slight performance improvement. On $f7$ *model 3* vastly outperformed all other methods, especially when looking at AUC (Table 5). Again, we see the model only switching strategy occasionally. This function has lower conditioning, and plateaus with a gradient of zero, meaning it is easy for the DE to get stuck even though the function is unimodal.

These results show that the models each tends to stick with their subset of 3-4 actions, which explain why each model performs differently on the different functions. They each tend to prefer different actions, which happen to do well in some cases, but do worse in others.

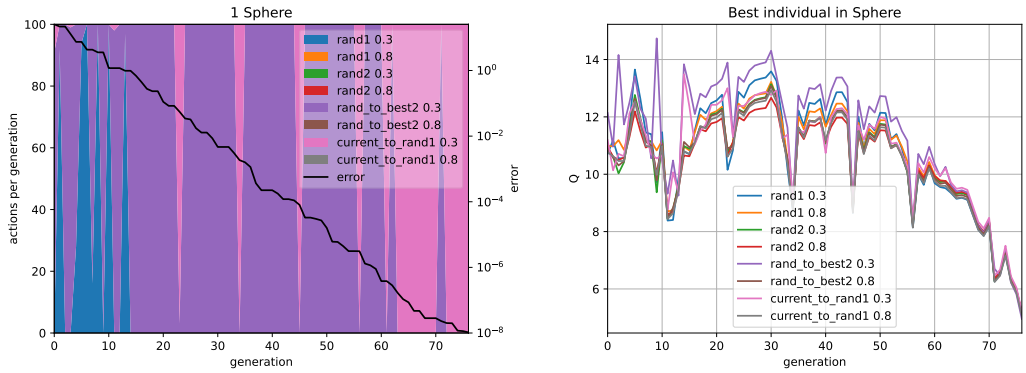

Figure 6: *Left*: Actions taken and error per generation for single runs on $f1$ for **model 1**. *Right*: Q-values of each action per generation for the individual with the lowest error after termination.

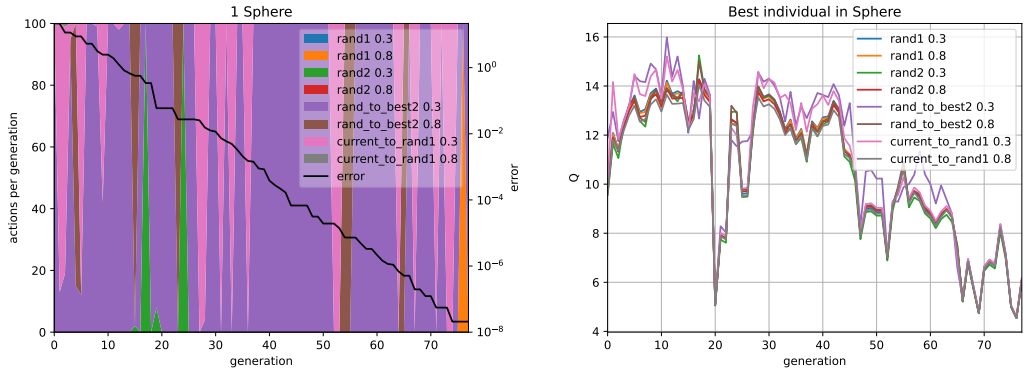

Figure 7: *Left*: Actions taken and error per generation for single runs on $f1$ for **model 2**. *Right*: Q-values of each action per generation for the individual with the lowest error after termination.

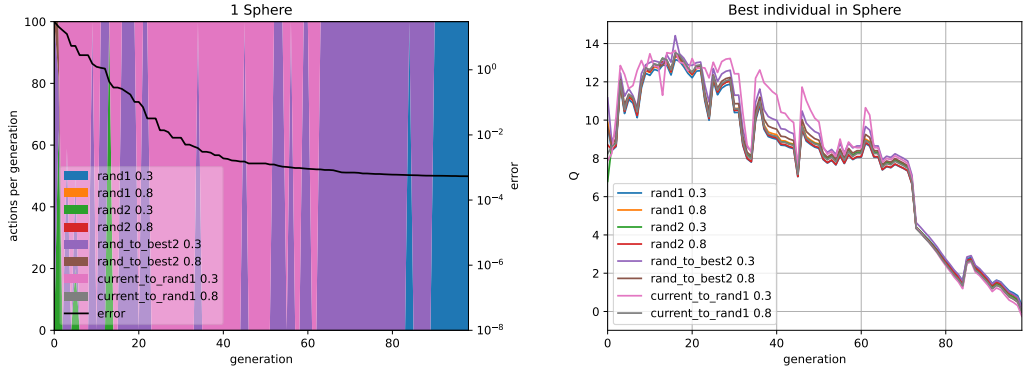

Figure 8: *Left*: Actions taken and error per generation for single runs on $f1$ for **model 3**. *Right*: Q-values of each action per generation for the individual with the lowest error after termination.

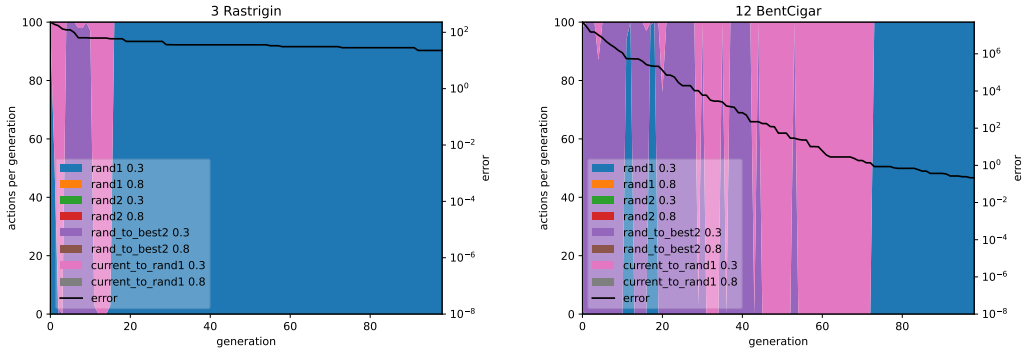

Figure 9: Actions taken and error per generation for single runs on $f3$ and $f12$ for *model 1*.

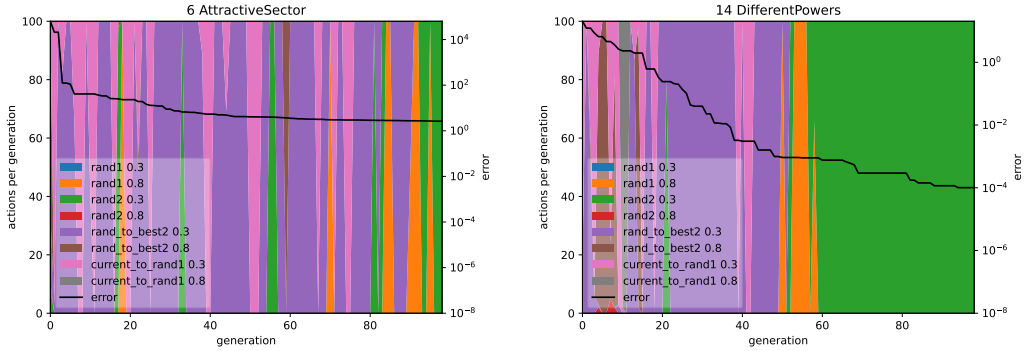

Figure 10: Actions taken and error per generation for single runs on $f6$ and $f14$ for *model 2*.

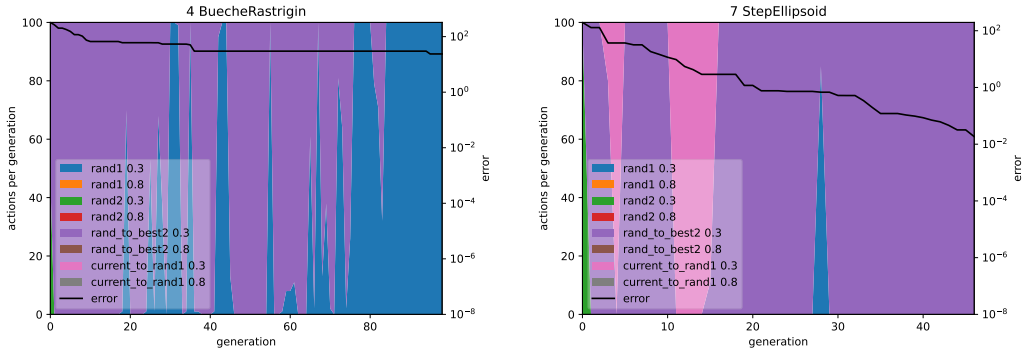

Figure 11: Actions taken and error per generation for single runs on $f4$ and $f7$ for *model 3*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Total
rand1 0.3	6.9%	22.9%	87.9%	86.7%	23.8%	22.7%	72.5%	24.7%	24.3%	58.2%	94.7%	26.0%	25.7%	31.4%	88.2%	96.2%	88.5%	89.9%	91.9%	85.7%	27.2%	22.7%	95.6%	89.3%	58.2%
rand1 0.8	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand2 0.3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand2 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand_to_best2 0.3	76.2%	53.8%	5.9%	4.2%	56.7%	29.3%	8.7%	51.8%	53.0%	3.3%	1.6%	33.7%	34.5%	30.1%	6.0%	1.1%	3.4%	3.2%	3.3%	7.4%	12.6%	31.5%	1.6%	4.8%	21.0%
rand_to_best2 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
current_to_rand1 0.3	16.9%	23.3%	6.1%	9.1%	19.5%	48.0%	18.8%	23.5%	22.7%	38.6%	3.7%	40.2%	39.7%	38.5%	5.8%	2.7%	8.1%	6.8%	4.8%	6.9%	60.2%	45.8%	2.8%	5.9%	20.7%
current_to_rand1 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 6: For each function the percentage each action (mutation strategy) was chosen by *model 1* plus the percentage over all 24 functions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Total
rand1 0.3	0.3%	0.2%	0.1%	0.1%	0.1%	0.1%	0.0%	0.2%	0.1%	0.0%	0.0%	0.2%	0.1%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	0.0%	0.0%	0.1%
rand1 0.8	2.0%	4.6%	5.0%	5.9%	4.7%	5.2%	9.1%	5.0%	4.8%	10.4%	6.6%	4.8%	4.9%	6.2%	5.3%	13.6%	11.3%	12.2%	4.0%	3.3%	3.8%	4.9%	14.2%	8.6%	6.7%
rand2 0.3	8.1%	12.7%	78.3%	74.5%	10.8%	11.5%	56.6%	11.5%	9.1%	71.8%	86.6%	32.1%	24.7%	31.9%	79.5%	80.9%	51.2%	59.7%	85.9%	79.5%	12.2%	13.9%	80.5%	77.5%	47.9%
rand2 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand_to_best2 0.3	64.0%	58.6%	10.0%	12.1%	71.1%	61.7%	22.9%	56.7%	59.6%	13.2%	4.5%	34.7%	41.7%	39.4%	9.6%	4.2%	28.4%	21.7%	7.3%	10.8%	55.8%	56.0%	3.9%	9.5%	31.3%
rand_to_best2 0.8	3.6%	3.5%	1.8%	1.5%	1.8%	3.0%	1.1%	4.4%	2.8%	0.4%	0.5%	4.5%	3.5%	2.4%	1.1%	0.3%	0.5%	0.4%	0.3%	1.2%	2.5%	2.3%	0.3%	0.8%	1.9%
current_to_rand1 0.3	21.8%	19.7%	4.2%	5.5%	11.2%	17.3%	9.7%	21.2%	22.9%	3.7%	1.5%	22.8%	24.7%	19.2%	4.0%	0.9%	8.3%	5.7%	2.3%	4.5%	25.0%	22.5%	1.0%	3.4%	11.7%
current_to_rand1 0.8	0.2%	0.7%	0.5%	0.4%	0.2%	1.0%	0.6%	0.9%	0.7%	0.4%	0.2%	0.8%	0.4%	0.6%	0.5%	0.1%	0.3%	0.3%	0.2%	0.6%	0.6%	0.2%	0.1%	0.1%	0.4%

Table 7: For each function the percentage each action (mutation strategy) was chosen by *model 2* plus the percentage over all 24 functions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Total
rand1 0.3	10.0%	9.5%	41.9%	43.6%	10.0%	10.2%	10.4%	10.2%	10.3%	21.3%	28.8%	11.4%	10.6%	13.7%	43.5%	23.6%	20.6%	26.2%	43.1%	44.7%	10.1%	10.6%	23.1%	28.9%	21.7%
rand1 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand2 0.3	1.5%	0.7%	0.9%	0.5%	0.2%	0.9%	1.2%	1.4%	1.3%	0.6%	0.6%	1.2%	1.6%	1.6%	0.8%	0.3%	0.9%	0.7%	0.8%	0.9%	0.2%	0.8%	0.4%	0.5%	0.9%
rand2 0.8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rand_to_best2 0.3	41.8%	53.7%	52.4%	52.4%	51.3%	68.5%	81.7%	54.4%	45.4%	77.0%	69.7%	46.5%	42.6%	49.6%	51.0%	76.0%	77.2%	71.6%	53.3%	47.0%	81.3%	64.5%	76.3%	67.5%	60.3%
rand_to_best2 0.8	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
current_to_rand1 0.3	46.2%	36.0%	4.7%	3.6%	38.1%	20.4%	6.6%	33.9%	43.0%	1.1%	0.8%	40.9%	44.8%	35.2%	4.6%	0.1%	1.3%	1.5%	2.8%	7.4%	8.4%	24.0%	0.2%	2.8%	17.1%
current_to_rand1 0.8	0.4%	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.3%	0.1%

Table 8: For each function the percentage each action (mutation strategy) was chosen by *model 3* plus the percentage over all 24 functions.

5.3 Reward analysis

To see why each model sticks with only a subset of actions, we can analyse the rewards that were given. The rewards were logged during testing for every individual so that we can see if and how much the models actually improved on what they were directly trained for. For constant and random policies that do not use the model, the rewards are also logged, just as if the model were to choose that action. In Table 9 we see per function the average reward for each model, together with the average reward for the constant and random strategies to compare. The last column is the average over all functions. All averages are given as the average *per individual*, not the average of the sum of rewards per generation. The reward function is explained in more detail in Section 4.3.

This table show some large differences with Tables 4 & 5. On $f1$ *model 1* and *2* perform much better than *model 3*, having a higher average fraction of targets hit and a higher AUC, while the differences in average reward are much smaller. This explains why *model 3* expects a large future reward for the first half of the run (Figure 8 *Right*). The reward function does not depend on the size of the improvement, so many small improvements are awarded just as much as many big improvements. As long as the line in Figure 8 *Left* is decreasing, the cumulative reward goes up.

This may also explain why for some functions the AUC of the ECDF plots is highest for most of the models (Table 5), while the other variants have a higher fraction of targets hit when the budget is depleted (Table 4). The reward function encourages improvements, no matter how small or big they are. This gives us a higher AUC.

The differences are just as big for the DE variants with constant and random mutation strategies. On $f10$, *current-to-rand1 0.3* fails to hit any targets, but it got the highest average reward out of all strategies. This again suggests that this variant makes many small improvements instead of trying to reach the global optimum as quickly as possible. It is however interesting to note that all models performed better than *current-to-rand1 0.3* on $f10$, despite achieving a lower average reward.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	avg
rand1 0.3	0.474	0.484	0.141	0.162	0.508	0.438	0.277	0.414	0.404	0.244	0.206	0.377	0.372	0.365	0.131	0.048	0.282	0.246	0.094	0.146	0.334	0.422	0.046	**0.111**	0.280
rand1 0.8	0.157	0.225	0.119	0.115	0.304	0.140	0.116	0.129	0.102	0.091	0.059	0.140	0.147	0.143	0.108	0.047	0.101	0.100	0.076	0.117	0.080	0.125	0.046	0.082	0.119
rand2 0.3	0.377	0.399	0.134	0.148	0.448	0.316	0.211	0.312	0.288	0.184	0.153	0.289	0.291	0.306	0.125	0.047	0.217	0.192	0.090	0.137	0.187	0.313	0.046	0.107	0.222
rand2 0.8	0.098	0.158	0.099	0.080	0.188	0.104	0.086	0.086	0.073	0.079	0.053	0.093	0.103	0.096	0.088	0.046	0.076	0.079	0.061	0.076	0.053	0.081	0.046	0.056	0.086
rand-to-best2 0.3	0.570	0.558	0.130	0.155	0.550	0.528	0.335	0.520	0.536	0.300	0.233	0.533	0.539	0.539	0.121	0.048	0.258	0.246	0.089	0.151	0.494	0.514	0.046	0.105	0.338
rand-to-best2 0.8	0.104	0.168	0.096	0.082	0.178	0.106	0.091	0.091	0.076	0.079	0.057	0.096	0.106	0.099	0.089	0.046	0.074	0.077	0.064	0.086	0.062	0.075	**0.047**	0.060	0.088
current-to-rand1 0.3	0.564	0.546	**0.154**	0.157	0.521	0.540	0.330	0.542	0.566	**0.492**	0.275	0.557	0.562	0.556	**0.144**	**0.048**	0.186	0.194	**0.096**	**0.173**	0.142	0.265	0.047	0.110	0.324
current-to-rand1 0.8	0.200	0.260	0.123	0.127	0.303	0.161	0.139	0.155	0.124	0.105	0.068	0.172	0.174	0.181	0.115	0.047	0.116	0.113	0.079	0.132	0.077	0.144	0.046	0.090	0.136
random	0.363	0.375	0.129	0.144	0.369	0.310	0.217	0.312	0.298	0.192	0.153	0.295	0.292	0.304	0.121	0.046	0.216	0.196	0.087	0.127	0.188	0.290	0.047	0.100	0.215
model1	**0.729**	0.698	0.135	**0.167**	0.704	0.617	0.325	0.652	0.663	0.350	0.223	0.525	0.528	0.496	0.123	0.048	0.292	0.258	0.087	0.135	0.302	0.502	0.047	0.104	0.359
model2	0.703	0.693	0.124	0.143	**0.722**	0.675	0.271	0.664	0.683	0.211	0.160	0.441	0.480	0.440	0.116	0.047	0.275	0.226	0.082	0.119	**0.604**	0.616	0.047	0.098	0.357
model3	0.720	**0.720**	0.145	0.165	0.689	**0.708**	**0.430**	**0.705**	**0.711**	0.343	**0.291**	**0.681**	**0.683**	**0.627**	0.131	0.048	**0.396**	**0.325**	0.090	0.155	0.599	**0.671**	0.046	0.107	**0.423**

Table 9: Average reward per individual for all functions, including the total average in the last column. For each function, the variant with the highest average reward is **bold and coloured in grey**. The three models refer to the variants of DE using a model to choose the mutation strategies. `random` is DE with a random policy (choosing a mutation strategy at random for every individual every generation), and the eight mutation strategies refer to the variants using constant policies.

6 Discussion & Conclusion

Analysis of the results reveals that, on average, all three models outperform both random and constant variants. Notably, significant performance variations are observed across different functions. While on some functions a model may do very well, on other functions it may sometimes even perform worse than a random policy.

These differences in performance can be explained when looking at the actions that are taken. Firstly, each model tend to mostly stick with a subset of three or four actions. We saw that some mutation strategies perform better than others, so it is to be expected that some actions are picked more. This subset of actions is different for each model. Secondly, where one model might choose to stick with one a single mutation strategy for most of the run, another model may switch strategy seemingly every generation. Lastly, while one may expect the models to pick one or more strategies that perform well when the DE settings are static, testing shows that sometimes the models find ways to combine strategies that perform bad on their own, but outperform all other variants when 'combined'.

The reason why each model tends to stick with its own subset of actions may be for of a number of reasons. The state features include a lot of information on how the mutation strategies performed on previous generations. If at the start of training one or two strategies happen to do very well, the model may learn to use those strategies at the start of every run. These adequate performing strategies are then only 'encouraged' by the state features. The model may make more informed decisions when given more information in the state vector about the fitness landscape. Secondly, we saw that slight changes in strategy may greatly alter the overall performance. This may make it difficult for the neural network to get out of a local optimum. Further research with separate sets of training functions and testing functions could show how well the model can generalise on unseen functions.

Lastly, we show that the performance when looking at the ECDF plots can differ greatly from the values that are rewarded to the model. This means that in some cases the model is encouraged to pick a mutation strategy that we in reality might find suboptimal, and vice versa. This could explain why some actions are chosen by the model that do not well in general. These mutation strategies may tend improve over the optimum, but only in minimal steps.

Despite these difficulties, we show the models are often capable of improving over the constant and random variants, sometimes by little, sometimes more. Hence, this implies that the models have the capacity to identify patterns within the state features, and use this to pick the mutation strategies that increase performance. A multitude of facets that could be improved have been identified, which are outlined in the next section.

6.1 Future work

Analysis of the results show a multitude of facets that could benefit from improvement. Future work can improve the stability of the models, help the models generalise better on a wider set of problems and increase the overall performance. This work can be divided into four subjects:

- **State** The state feature set for this thesis is based on the work of Sharma et al. [25]. Of our 112 state features, only 16 give information of the fitness landscape. As outlined in Section 3, other research use very different state features, often more focused on the fitness landscape. State features that use information from all fitness values from the entire run instead of just the previous generation could also benefit the performance.

- **Reward** As outlined in Section 4.3, it is key to use a reward function that encourages what we want the model to do. Our findings indicate that there is in some cases a dis-

parity between our objectives and the behaviours we incentivise. A reward function that encourages big improvements more than small ones could improve the performance, but it is important to have it generalise well, especially on function with high conditioning. Secondly, having more sparse rewards can encourage getting to big milestones quicker, instead of trying to get as much reward as possible every generation.

- **Model** Since we are working with a very large and complex state space, The model could benefit from a larger neural network. For this project, minimal tweaking was done since it was deemed out of scope, and most settings we based on the work by Sharma et al. [25]. We expect that a more comprehensive tuning of the hyperparameter values will result in a notable improvement in the model's performance. In addition, since deep RL is a very active research topic, a more modern reinforcement learning algorithm could show improved results.

- **Actions** For this thesis we have the model choose out of eight actions. For future research this number can be increased further by adding more operators and factors, and adding other parameters, e.g. crossover operator, crossover rate and/or population size. A redesign of the model would make it possible to use continuous actions for the scaling factor and crossover rate, giving the model even more control over the DE.

References

[1] Anne Auger and Nikolaus Hansen. A restart cma evolution strategy with increasing population size. In *2005 IEEE congress on evolutionary computation*, volume 2, pages 1769–1776. IEEE, 2005.

[2] Rick Boks, Anna V Kononova, and Hao Wang. Quantifying the impact of boundary constraint handling methods on differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1199–1207, 2021.

[3] Janez Brest, Sao Greiner, Borko Boskovic, Marjan Mernik, and Viljem Zumer. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE transactions on evolutionary computation*, 10(6):646–657, 2006.

[4] Manu Centeno-Telleria, Ekaitz Zulueta, Unai Fernandez-Gamiz, Daniel Teso-Fz-Betoño, and Adrián Teso-Fz-Betoño. Differential evolution optimal parameters tuning with artificial neural network. *Mathematics*, 9(4):427, 2021.

[5] Steffen Finck, Nikolaus Hansen, Raymond Ros, and Anne Auger. Real-parameter black-box optimization benchmarking 2009: Presentation of the noiseless functions. Technical report, Citeseer, 2010.

[6] Roger Gämperle, Sibylle D Müller, and Petros Koumoutsakos. A parameter study for differential evolution. *Advances in intelligent systems, fuzzy systems, evolutionary computation*, 10(10):293–298, 2002.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[8] Nikolaus Hansen. Benchmarking a bi-population cma-es on the bbob-2009 function testbed. In *Proceedings of the 11th annual conference companion on genetic and evolutionary computation conference: late breaking papers*, pages 2389–2396, 2009.

[9] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. Coco: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1):114–144, 2021.

[10] Zhenzhen Hu, Wenyin Gong, and Shuijia Li. Reinforcement learning-based differential evolution for parameters extraction of photovoltaic models. *Energy Reports*, 7:916–928, 2021.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Anna V Kononova, Fabio Caraffini, and Thomas Bäck. Differential evolution outside the box. *Information Sciences*, 581:587–604, 2021.

[13] Zhihui Li, Li Shi, Caitong Yue, Zhigang Shang, and Boyang Qu. Differential evolution based on reinforcement learning with fitness ranking for solving multimodal multiobjective problems. *Swarm and Evolutionary Computation*, 49:234–244, 2019.

[14] Junhong Liu. On setting the control parameter of the differential evolution method. In *Proceedings of the 8th international conference on soft computing (MENDEL 2002)*, pages 11–18, 2002.

[15] Fu Xing Long, Diederick Vermetten, Bas van Stein, and Anna V Kononova. Bbob instance analysis: Landscape properties and algorithm performance across problem instances. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 380–395. Springer, 2023.

[16] Jorge Maturana and Frédéric Saubion. A compass to guide genetic algorithms. In *International conference on parallel problem solving from nature*, pages 256–265. Springer, 2008.

[17] Efrñn Mezura-Montes, Jesús Velázquez-Reyes, and Carlos A Coello Coello. A comparative study of differential evolution variants for global optimization. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 485–492, 2006.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[19] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[20] Millie Pant, Hira Zaheer, Laura Garcia-Hernandez, Ajith Abraham, et al. Differential evolution: A review of more than two decades of research. *Engineering Applications of Artificial Intelligence*, 90:103479, 2020.

[21] Aske Plaat. *Deep reinforcement learning*, volume 10. Springer, 2022.

[22] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[23] Kanchan Rajwar, Kusum Deep, and Swagatam Das. An exhaustive review of the metaheuristic algorithms for search and optimization: Taxonomy, applications, and open challenges. *Artificial Intelligence Review*, 56(11):13187–13257, 2023.

[24] Jani Ronkkonen, Saku Kukkonen, and Kenneth V Price. Real-parameter optimization with differential evolution. In *2005 IEEE congress on evolutionary computation*, volume 1, pages 506–513. IEEE, 2005.

[25] Mudita Sharma, Alexandros Komninos, Manuel López-Ibáñez, and Dimitar Kazakov. Deep reinforcement learning based parameter control in differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 709–717, 2019.

[26] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11:341–359, 1997.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] Zhiping Tan and Kangshun Li. Differential evolution with mixed mutation strategy based on deep reinforcement learning. *Applied Soft Computing*, 111:107678, 2021.

[29] Zhiping Tan, Yu Tang, Kangshun Li, Huasheng Huang, and Shaoming Luo. Differential evolution with hybrid parameters and mutation strategies based on reinforcement learning. *Swarm and Evolutionary Computation*, 75:101194, 2022.

[30] Ryoji Tanabe and Alex Fukunaga. Success-history based parameter adaptation for differential evolution. In *2013 IEEE congress on evolutionary computation*, pages 71–78. IEEE, 2013.

[31] Ryoji Tanabe and Alex Fukunaga. Reevaluating exponential crossover in differential evolution. In *International Conference on Parallel Problem Solving from Nature*, pages 201–210. Springer, 2014.

[32] Jason Teo. Differential evolution with self-adaptive populations. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 1284–1290. Springer, 2005.

[33] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[34] Xinxin Wang, Chengjun Li, Jiarui Zhu, and Qinxue Meng. L-shade-e: Ensemble of two differential evolution algorithms originating from l-shade. *Information Sciences*, 552:201–219, 2021.

[35] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[36] Jingqiao Zhang and Arthur C Sanderson. Jade: adaptive differential evolution with optional external archive. *IEEE Transactions on evolutionary computation*, 13(5):945–958, 2009.

[37] Karin Zielinski and Rainer Laur. Constrained single-objective optimization using differential evolution. In *2006 IEEE International Conference on Evolutionary Computation*, pages 223–230. IEEE, 2006.

[38] Karin Zielinski and Rainer Laur. Stopping criteria for differential evolution in constrained single-objective optimization. In *Advances in differential evolution*, pages 111–138. Springer, 2008.