Proceedings Paper:
Sharma, Mudita, Komninos, Alexandros, López-Ibáñez, Manuel et al. (1 more author) (2019) Deep Reinforcement Learning Based Parameter Control in Differential Evolution. In: GECCO '19:Proceedings of the Genetic and Evolutionary Computation Conference. ACM Proceedings. ACM, pp. 709-717.

https://doi.org/10.1145/3321707.3321813

Deep Reinforcement Learning Based Parameter Control in Differential Evolution

Mudita Sharma
University of York
York, U.K.
ms1938@york.ac.uk

Alexandros Komninos
University of York
York, U.K.
alexandros.komninos@york.ac.uk

Manuel López-Ibáñez
University of Manchester
Manchester, U.K.
manuel.lopez-ibanez@manchester.ac.uk

Dimitar Kazakov
University of York
York, U.K.
dimitar.kazakov@york.ac.uk

ABSTRACT

Adaptive Operator Selection (AOS) is an approach that controls discrete parameters of an Evolutionary Algorithm (EA) during the run. In this paper, we propose an AOS method based on Double Deep Q-Learning (DDQN), a Deep Reinforcement Learning method, to control the mutation strategies of Differential Evolution (DE). The application of DDQN to DE requires two phases. First, a neural network is trained offline by collecting data about the DE state and the benefit (reward) of applying each mutation strategy during multiple runs of DE tackling benchmark functions. We define the DE state as the combination of 99 different features and we analyze three alternative reward functions. Second, when DDQN is applied as a parameter controller within DE to a different test set of benchmark functions, DDQN uses the trained neural network to predict which mutation strategy should be applied to each parent at each generation according to the DE state. Benchmark functions for training and testing are taken from the CEC2005 benchmark with dimensions 10 and 30. We compare the results of the proposed DE-DDQN algorithm to several baseline DE algorithms using no online selection, random selection and other AOS methods, and also to the two winners of the CEC2005 competition. The results show that DE-DDQN outperforms the non-adaptive methods for all functions in the test set; while its results are comparable with the last two algorithms.

CCS CONCEPTS

• **Computing methodologies → Bio-inspired approaches**; **Reinforcement learning**;

KEYWORDS

Parameter Control, Reinforcement Learning, Differential Evolution

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321813

ACM Reference Format:
Mudita Sharma, Alexandros Komninos, Manuel López-Ibáñez, and Dimitar Kazakov. 2019. Deep Reinforcement Learning Based Parameter Control in Differential Evolution. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321813

1 INTRODUCTION

Evolutionary algorithms for numerical optimization come in many variants involving different operators, such as mutation strategies and types of crossover. In the case of differential evolution (DE) [25], experimental analysis has shown that different mutation strategies perform better for specific optimization problems [17] and that choosing the right mutation strategy at specific stages of an optimization process can further improve the performance of DE [7]. As a result, there has been great interest in methods for controlling or selecting the value of discrete parameters while solving a problem, also called *adaptive operator selection (AOS)*.

In the context of DE, there is a finite number of mutation strategies (operators) that can be applied at each generation to produce new solutions from existing (parent) solutions. An AOS method will decide, at each generation, which operator should be applied, measure the effect of this application and adapt future choices according to some reward function. An inherent difficulty is that we do not know which operator is the most useful at each generation to solve a previously unseen problem. Moreover, different operators may be useful at different stages of an algorithm's run.

There are multiple AOS methods proposed in the literature [1, 9, 12] and several of them are based on *reinforcement learning (RL)* techniques such as probability matching [8, 23], multi-arm bandits [9], $Q(\lambda)$ learning [20] and SARSA [4, 5, 22], among others [10]. These RL methods use one or few features to capture the state of the algorithm at each generation, select an operator to be applied and calculate a reward from this application. Typical state features are fitness standard deviation, fitness improvement from parent to offspring, best fitness, and mean fitness [5, 10]. Typical reward functions measure improvement achieved over the previous generation [10]. Other parameter control methods use an offline training phase to collect more data about the algorithm than what is available within a single run. For example, Kee et al. [14] uses two types of learning: table-based and rule-based. The learning is performed during an offline training phase that is followed by an

online execution phase where the learned tables or rules are used for choosing parameter values. More recently, Karafotias et al. [13] trains offline a feed-forward neural network with no hidden layers to control the numerical parameter values of an evolution strategy. To the best of our knowledge, none of the AOS methods that use offline training are based on reinforcement learning.

In this paper, we adapt *Double Deep Q-Network (DDQN)* [28], a deep reinforcement learning technique that uses a deep neural network as a prediction model, as an AOS method for DE. The main differences between DDQN and other RL methods are the possibility of training DDQN offline on large amounts of data and of using a larger number of features to define the current state. When applied as an AOS method within DE, we first run the proposed DE-DDQN algorithm many times on training benchmark problems by collecting data on 99 features, such as the relative fitness of the current generation, mean and standard deviation of the population fitness, dimension of the problem, number of function evaluations, stagnation, distance among solutions in decision space, etc. After this training phase, the DE-DDQN algorithm can be applied to unseen problems. It will observe the run time value of these features and predict which mutation strategy should be used at each generation. DE-DDQN also requires the choice of a suitable reward definition to facilitate learning of a prediction model. Some RL-based AOS methods calculate rewards per individual [4, 20], while others calculate it per generation [22]. Moreover, reward functions can be designed in different ways depending on the problem at hand. For example, Karafotias et al. [11] defines and compares four per-generation reward definitions for RL-based AOS methods. Here, we also find that the reward definition has a strong effect on the performance of DE-DDQN and, hence, we analyze three alternative reward definitions that assign reward for each application of a mutation strategy.

As an experimental benchmark, we use functions from the CEC2005 special session on real-parameter optimization [26]. In particular, the proposed DE-DDQN method is first trained on 16 functions for both dimensions 10 and 30, i.e., a total of 32 training functions. Then, we run the trained DE-DDQN on a different set of 5 functions, also for dimensions 10 and 30, i.e., a total of 10 test functions. We also run on these 10 test functions the following algorithms for comparison: four DE variants, each using a single specific mutation strategy, DE with a random selection among mutation strategies at each generation, DE using various AOS methods (PM-AdapSS [8], F-AUC [9], and RecPM-AOS [23]), and the two winners of CEC2005 [26] competition, which are both variants of CMAES: LR-CMAES (LR) [2] and IPOP-CMAES (IPOP) [3].

Our experimental results show that the DE variants using AOS completely outperform the DE variants using a fixed mutation strategy or a random selection. Although a non-parametric post-hoc test does not find that the differences between the CMAES algorithms and the AOS-enabled DE algorithms (including DE-DDQN) are statistically significant, DE-DDQN is the second best approach, behind IPOP-CMAES, in terms of mean rank.

The paper is structured as follows. First, we give a brief introduction to DE, mutation strategies and deep reinforcement learning. In Sect. 3, we introduce our proposed DE-DDQN algorithm, and explain its training and online (deployment) phases. Section 4 introduces the state features and reward functions used in the experiments, which are described in Sect. 5. We summarise our conclusions in Sect. 6.

2 BACKGROUND

2.1 Differential Evolution

Differential Evolution (DE) [21] is a population-based algorithm that uses a mutation strategy to create an offspring solution $\vec{u}$. A mutation strategy is a linear combination of three or more parent solutions $\vec{x}_i$, where i is the index of a solution in the current population. Some mutation strategies are good at exploration and others at exploitation, and it is well-known that no single strategy performs best for all problems and for all stages of a single run. In this paper, we consider these frequently used mutation strategies:

"rand/1": $\vec{u}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{r_2} - \vec{x}_{r_3})$
"rand/2": $\vec{u}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{r_2} - \vec{x}_{r_3} + \vec{x}_{r_4} - \vec{x}_{r_5})$
"rand-to-best/2": $\vec{u}_i = \vec{x}_{r_1} + F \cdot (\vec{x}_{\text{best}} - \vec{x}_{r_1} + \vec{x}_{r_2} - \vec{x}_{r_3} + \vec{x}_{r_4} - \vec{x}_{r_5})$
"curr-to-rand/1": $\vec{u}_i = \vec{x}_i + F \cdot (\vec{x}_{r_1} - \vec{x}_i + \vec{x}_{r_2} - \vec{x}_{r_3})$

where F is a scaling factor, $\vec{u}_i$ and $\vec{x}_i$ are the i-th offspring and parent solution vectors in the population, respectively, $\vec{x}_{\text{best}}$ is the best parent in the population, and r_1, r_2, r_3, r_4, and r_5 are randomly generated indexes within $[1, NP]$, where NP is the population size. An additional numerical parameter, the crossover rate ($CR \in [0, 1]$), determines whether the mutation strategy is applied to each dimension of $\vec{x}_i$ to generate $\vec{u}_i$. At least one dimension of each $\vec{x}_i$ vector is mutated.

2.2 Deep Reinforcement Learning

In RL [27], an agent takes actions in an environment that returns the reward and the next state. The goal is to maximize the cumulative reward at each step. RL estimates the value of an action given a state called *Q-value* to learn a *policy* that returns an action given a state. A variety of different techniques are used in RL to learn this policy and some of them are applicable only when the set of actions is finite.

When the features that define a state are continuous or the set of states is very large, the policy becomes a function that implicitly maps between state features and actions, as opposed to keeping an explicit map in the form of a lookup table. In *deep reinforcement learning*, this function is approximated by a deep neural network and the weights of the network are optimized to maximize the cumulative reward.

Deep Q-network (DQN) [18] is a deep RL technique that extends Q-learning to continuous features by approximating a non-linear Q-value function of the state features using a neural network (NN). The classical DQN algorithm sometimes overestimates the Q-values of the actions, which leads to poor policies. Double DQN (DDQN) [28] was proposed as a way to overcome this limitation and enhance the stability of the Q-values. DDQN employs two neural networks: a primary network selects an action and a target network generates a target Q-value for that action. The target-Q values are used to compute the loss function for every action during training. The weights of the target network are fixed, and only periodically or slowly updated to the primary Q-networks values.

In this work, we integrate DDQN into DE as an AOS method that selects a mutation strategy at each generation.

3 DE-DDQN

When integrated with DE as an AOS method, DDQN is adapted as follows. The environment of DDQN becomes the DE algorithm performing an optimization run for a maximum of $FE^{\max}$ function evaluations. A *state* s_t is a collection of features that measure static or run time features of the problem being solved or of DE at step t (function evaluation or generation counter). The actions that DDQN may take are the set of mutation strategies available (Sect. 2.1), and a_t is the strategy selected and applied at step t. Once a mutation strategy is applied, a reward function returns the estimated benefit (reward) r_t of applying action a_t, and the DE run reaches a new state, s_{t+1}. We refer to the tuple $\langle s_t, a_t, r_t, s_{t+1}\rangle$ as an *observation*.

Our proposed DE-DDQN algorithm operates in two phases. In the first *training* phase, the two deep neural networks of DDQN are trained on observations by running the DE-DDQN algorithm multiple times on several benchmark functions. In a second online (or deployment) phase, the trained DDQN is used to select which mutation strategy should be applied at each generation of DE when tackling unseen (or test) problems not considered during the training phase. We describe these two phases in detail next.

3.1 Training phase

In the training phase, DDQN uses two deep neural networks (NNs), namely primary NN and target NN. The primary NN predicts the Q-values $Q(s_t, a; \theta)$ that are used to select an action a given state s_t at step t, while the target NN estimates the target Q-values $\hat{Q}(s_t, a; \hat{\theta})$ after the action a has been applied, where θ and $\hat{\theta}$ are the weights of the primary and target NNs, respectively, s_t is the state vector of DE, and a is a mutation strategy.

The goal of the training phase is to train the primary NN of DDQN so that it learns to approximate the target $\hat{Q}$ function. The training data is a memory of observations that is collected by running DE-DDQN several times on training benchmark functions. Training the primary NN involves finding its weights θ through gradient optimization.

The training process of DE-DDQN is shown in Algorithm 1. Training starts by running DE with random selection of mutation strategy for a fixed number of steps (*warm-up size*) that generates observations to populate a memory of capacity N, which can be different from the warm-up size (line 2). This memory stores a fixed number of N recent observations, old ones are removed as new ones are added. Once the warm-up phase is over, DE is executed M times, and each run is stopped after $FE^{\max}$ function evaluations or the known optimum of the training problem is reached (line 7). For each solution in the population, the ϵ-greedy policy is used to select mutation strategy, i.e., with ϵ probability a random mutation is selected, otherwise the mutation strategy with maximum Q-value is selected. Using the current DE state s_t, the primary NN is responsible for generating a Q-value per possible mutation strategy (line 12). The use of a ϵ-greedy policy forces the primary NN to explore mutation strategies that may be currently predicted less optimal. The selected mutation strategy is applied (line 13) and a new state s_{t+1} is achieved (line 14). A reward value r_t is computed by measuring the performance progress made at this step.

To prevent the primary NN from only learning about the immediate state of this DE run, randomly draw mini batches of observations (line 16) from memory to perform a step of gradient optimization. Training the primary NN with the randomly drawn observations helps to robustly learn to perform well in the task.

Algorithm 1 DE-DDQN training algorithm

1: Initialise parameter values of DE (F, NP, CR)
2: Run DE with random selection of mutation strategy to initialise memory to capacity N
3: Initialise Q-value for each action by setting random weights θ of primary NN
4: Initialise target Q-value $\hat{Q}$ for each action by setting weights $\hat{\theta} = \theta$ of target NN
5: **for** run 1, . . . M **do**
6: $t = 0$
7: **while** $t < FE^{\max}$ or optimum is reached **do**
8: **for** $i = 1, \ldots, NP$ **do**
9: **if** rand(0, 1) $< \epsilon$ **then**
10: Randomly select a mutation strategy a_t
11: **else**
12: Select $a_t = \arg\max_a Q(s_t, a; \theta)$
13: Generate trial vector $\vec{u}_i$ for parent $\vec{x}_i$ using mutation a_t
14: Evaluate trial vector and keep the best among $\vec{x}_i$ and $\vec{u}_i$
15: Store observation (s_t, a_t, r_t, s_{t+1}) in memory
16: Sample random mini batch of observations from memory
17: **if** run terminates **then**
18: $r^{\text{target}} = r_t$
19: **else**
20: $\hat{a}_{t+1} = \arg\max_a Q(s_{t+1}, a; \theta)$
21: $r^{\text{target}} = r_t + \gamma \hat{Q}(s_{t+1}, \hat{a}_{t+1}; \hat{\theta})$
22: Perform a gradient descent step on $(r^{\text{target}} - Q(s_j, a_j; \theta))^2$ with respect to θ
23: Every C steps set $\hat{\theta} = \theta$
24: $t = t + 1$
25: **return** θ (weights of primary NN)

The primary NN is used to predict the next mutation strategy $\hat{a}_{t+1}$ (line 20) and its reward (line 21), without actually applying the mutation. A target reward value r^{target} is used to train the primary NN, i.e., finding the weights θ that minimise the loss function $(r^{\text{target}} - Q(s_j, a_j; \theta))^2$ (line 22). If the run terminates, i.e., if the budget assigned to the problem is finished, r^{target} is the same as the reward r_t. Otherwise, r^{target} is estimated (line 21) as a linear combination of the current reward r_t and the predicted future reward $\gamma \hat{Q}(s_{t+1}, \hat{a}_{t+1})$, where $\hat{Q}$ is the (predicted) target Q-value and γ is the discount factor that makes the training focus more on immediate results compared to future rewards.

Finally, the primary and target NNs are synchronised periodically by copying the weights θ from the primary NN to the $\hat{\theta}$ of the target NN every fixed number of C training steps (line 23). That is, the target NN uses an older set of weights to compute the target Q-value, which keeps the target value r^{target} from changing too quickly. At every step of training (line 22), the Q-values generated by the primary NN shift. If we are using a constantly shifting set of values to calculate r^{target} (line 21) and adjust the NN weights (line 22), then the target value estimations can easily become unstable by falling into feedback loops between r^{target} and the (target) Q-values used to calculate r^{target}. In order to mitigate that risk, the target NN is used to generate target Q-values ($\hat{Q}$) that are used to compute r^{target}, which is used in the loss function for training the primary NN. While the primary NN is trained, the weights of the target NN are fixed.

Algorithm 2 DE-DDQN testing algorithm

1: Initialise parameter values of DE (F, NP, CR)
2: Initialise and evaluate fitness of each $\vec{x}_i$ in the population
3: Initialise $Q(\cdot)$ for each mutation strategy with fixed weights θ
4: $t = 0$
5: **while** $t < FE^{\max}$ **do**
6: **for** $i = 1, \ldots, NP$ **do**
7: Select $a_t = \arg\max_a Q(s_t, a; \theta)$
8: Generate trial vector $\vec{u}_i$ for parent $\vec{x}_i$ using operator a_t
9: Evaluate trial vector $\vec{u}_i$
10: Replace $\vec{x}_i$ with the best among parent and trial vector
11: $t = t + 1$
12: **return** best solution found

3.2 Online phase

Once the learning is finished, the weights of the primary NN are frozen. In the testing phase, the mutation strategy is selected online during an optimization run on an unseen function. The online AOS with DE is shown in Algorithm 2. Since the weights of the NN are not updated in this phase, we do not maintain a memory of observations or compute rewards. As a new state is observed s_t, the Q-values per mutation strategy are calculated and a new mutation strategy is chosen according to the greedy policy (line 7).

4 STATE FEATURES AND REWARD

In this section we describe the new state features and reward definitions explored for the proposed DE-DDQN method.

4.1 State representation

The state representation needs to provide sufficient information so that the NN can decide which action is more suitable at the current step. We propose a state vector consisting of various features capturing properties of the landscape and the history of operator performance. Each feature is normalised to the range $[0, 1]$ by design in order to abstract absolute values specific to particular problems and help generalisation. Features are summarised in Table 1.

Our state needs to encode information about how the current solutions in the population are distributed in the decision space and their differences in fitness values. The fitness of current parent $f(\vec{x}_i)$ is given to the NN as a first state feature. The next feature is the mean of the fitness of the current population. The first two features in the state are normalised by the difference of worst and best seen so far solution. The third feature calculates the standard deviation of the population fitness values. Feature 4 measures the remaining budget of function evaluations. Feature 5 is the dimension of the function being solved. The training set includes benchmark functions with different dimensions in the hope that the NN are able to generalise to functions of any dimension within the training range. Feature 6, stagnation count, calculates the number of function evaluations since the last improvement of the best fitness found for this run (normalised by $FE^{\max}$).

The next set of feature values describe the relation between the current parent and the six solutions used by the various mutation strategies, i.e., the five random indexes (r_1, r_2, r_3, r_4, r_5) and the best parent in the population ($\vec{x}_{\text{best}}$). Features 7–12 measure the Euclidean distance in decision space between the current parent $\vec{x}_i$ and the six solutions. These six euclidean distances help the NN learn to select the strategy that best combines these solutions. Features 13–18 use the same six solutions to calculate the fitness difference w.r.t. $f(\vec{x}_i)$. Feature 19 measures the normalised Euclidean distance in decision space between $\vec{x}_i$ and the best solution seen so far. We use distances instead of positions to make the state representation independent of the dimensionality of the solution space.

Describing the current population is not sufficient to select the best strategy. Reinforcement learning requires the state to be Markov, i.e., to include all necessary information for selecting an action. To this end, we enhance the state with features about the run time history. Using historical information has shown to be useful in our previous work [23]. In addition to the remaining budget and the stagnation counter described above, we also store four metric values $OM_m(g, k, op)$ after the application of op at generation g:

(1) $OM_1(g, k, op) = f(\vec{x}_i) - f(\vec{u}_i)$, that is, the k-th fitness improvement of offspring $\vec{u}_i$ over parent $\vec{x}_i$;
(2) $OM_2(g, k, op)$, the k-th fitness improvement of offspring over $\vec{x}_{\text{best}}$, the best parent in the current population;
(3) $OM_3(g, k, op)$, the k-th fitness improvement of offspring over $\vec{x}_{\text{bsf}}$, the best so far solution; and
(4) $OM_4(g, k, op)$, the k-th fitness improvement of offspring over the median fitness of the parent population.

For each OM_m, the total number of fitness improvements (*successes*) is given by $N_m^{\text{succ}}(g, op)$, that is, the index k is always $1 \leq k \leq N_m^{\text{succ}}(g, op)$. The counter $N^{\text{tot}}(g, op)$ gives the total number of applications of op at generation g. We store this historical information for the last gen number of generations.

With the information above, we compute the sum of success rates over the last gen generations, where each success rate is the number of successful applications of operator op, i.e., mutation strategy, in generation g that improve metric OM_m divided by the total number of applications of op in the same generation. For each metric OM_m, the values for an operator are normalised by the sum of all values of all operators. A different success rate is calculated for each combination of OM_m ($m \in \{1, 2, 3, 4\}$) and op (four mutation strategies) resulting in features 20–35.

We also compute the sum of fitness improvements for each OM_m divided by the total number of applications of op over the last gen generations (features 36–51). Features 52–67 are defined in terms of best fitness improvement of a mutation strategy op according to metric OM_m over a given generation g, that is, $OM_m^{\text{best}}(g, op) = \max_k^{N_m^{\text{succ}}(g,op)} OM_m(g, k, op)$. In this case, we calculate the relative difference in best improvement of the last generation with respect to the previous one, divided by the difference in number of applications between the last two generations (gen and $gen - 1$). Any zero value in the denominator is ignored. The sum of best improvement seen for combination of operator and metric is given as features 68–83.

Features 84-99 are calculated by maintaining a fixed size window W where each element is a tuple of the four metric values OM_m, $m \in \{1, 2, 3, 4\}$ and $f(\vec{u}_i)$ resulting from the application of a mutation strategy to $\vec{x}_i$ that generates $\vec{u}_i$. Initially the window is filled with OM_m values as new improved offsprings are produced. Once it is full, new elements replace existing ones generated by that mutation

Table 1: State features

Index	Feature	Notes
1	$\frac{f(\vec{x}_i) - f_{\text{bsf}}}{f_{\text{wsf}} - f_{\text{bsf}}}$	$\vec{x}_i$ denotes the i-th solution of the population and $f(\vec{x}_i)$ denotes its fitness; f_{bsf} and f_{wsf} denote the best-so-far and worst-so-far fitness values found up to this step within a single run
2	$\frac{\sum_{j=1}^{NP} \frac{f(\vec{x}_j)}{NP} - f_{\text{bsf}}}{f_{\text{wsf}} - f_{\text{bsf}}}$	NP is the population size
3	$\frac{\text{std}_{j=1,\ldots,NP}(f(\vec{x}_j))}{\text{std}^{\max}}$	std$(\cdot)$ calculates the standard deviation and std$^{\max}$ is the value when $NP/2$ solutions have fitness f_{wsf} and the other half have fitness f_{bsf}
4	$\frac{FE^{\max} - t}{FE^{\max}}$	$FE^{\max}$ is the maximum number of function evaluations per run, and $FE^{\max} - t$ gives the remaining number of evaluations at step t
5	$\frac{dim_f}{dim^{\max}}$	dim_f is the dimension of the benchmark function f being optimised, and $dim^{\max}$ is the maximum dimension among all training functions
6	$\frac{stagcount}{FE^{\max}}$	*stagcount* is the *stagnation counter*, i.e., the number of function evaluations (steps) without improving f_{bsf}
7-11	$\frac{dist(\vec{x}_i - \vec{x}_j)}{dist^{\max}}, \forall j \in \{r_1, r_2, r_3, r_4, r_5\}$	$dist(\cdot)$ 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
12	$\frac{dist(\vec{x}_i - \vec{x}_{\text{best}})}{dist^{\max}}$	$\vec{x}_{\text{best}}$ is the best parent in the current population
13-17	$\frac{f(\vec{x}_i) - f(\vec{x}_j)}{f_{\text{wsf}} - f_{\text{bsf}}}, \forall j \in \{r_1, r_2, r_3, r_4, r_5\}$	
18	$\frac{f(\vec{x}_i) - f(\vec{x}_{\text{best}})}{f_{\text{wsf}} - f_{\text{bsf}}}$	
19	$\frac{dist(\vec{x}_i - \vec{x}_{\text{bsf}})}{dist^{\max}}$	$\vec{x}_{\text{bsf}}$ denotes the solution with fitness f_{bsf}
20-35	$\sum_{g=1}^{gen} \frac{N_m^{\text{succ}}(g, op)}{N^{\text{tot}}(g, op)}$	For each op and $m \in \{1, 2, 3, 4\}$ and normalised over all operators; gen is the number of recent generations recorded; $N_m^{\text{succ}}(g, op)$ and $N^{\text{tot}}(g, op)$ are successful and total applications of op according to OM_m at generation g
36-51	$\frac{\sum_{g=1}^{gen} \sum_{k=1}^{N_m^{\text{succ}}(g,op)} OM_m(g, k, op)}{\sum_{g=1}^{gen} N^{\text{tot}}(g, op)}$	
52-67	$\frac{OM_m^{\text{best}}(gen, op) - OM_m^{\text{best}}(gen-1, op)}{OM_m^{\text{best}}(gen-1, op) \cdot \lvert N^{\text{tot}}(gen, op) - N^{\text{tot}}(gen-1, op)\rvert}$	For each op and $m \in \{1, 2, 3, 4\}$ and normalised over all operators; $OM_m^{\text{best}}(g, op)$ is the maximum value of $OM_m(g, k, op)$
68-83	$\sum_{g=1}^{gen} OM_m^{\text{best}}(g, op)$	For each op and $m \in \{1, 2, 3, 4\}$ and normalised over all operators
84-99	$\sum_{w=1}^{W} OM_m(w, op)$	For each op and $m \in \{1, 2, 3, 4\}$ and normalised over all operators; $OM_m(w, op)$ is the w-th value in the window generated by op

strategy according to the First-In First-Out (FIFO) rule. If there is no element produced by that operator in the window, the element with the worst (highest) $f(\vec{u}_i)$ is replaced. Each feature is the sum of OM_m values within the window for each m and each operator. The difference between features extracted from recent generations (68-83) and from the fixed-size window (84-99) is that the window captures the best solutions for each operator, and the number of solutions present per operator vary. In a sense, solutions compete to be part of the window. Whereas when computing features from the last gen generations, all successful improvements per generation are captured and there is no competition among elements. As the most recent history is the most useful, we use small values for last $gen = 10$ generations and window size $W = 50$.

4.2 Reward definitions

While we only know the true reward of a sequence of actions after a full run of DE is completed, i.e., the best fitness found, such sparse rewards provide a very weak signal and can slow down training. Instead, we calculate rewards after every action has been taken, i.e., a new offspring $\vec{u}_i$ is produced from parent $\vec{x}_i$. In this paper, we explore three reward definitions, each one using different information related to fitness improvement:

$$\text{R1} = \max\{f(\vec{x}_i) - f(\vec{u}_i), 0\} \quad \text{R2} = \begin{cases} 10 & \text{if } f(\vec{u}_i) < f_{\text{bsf}} \\ 1 & \text{else if } f(\vec{u}_i) < f(\vec{x}_i) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{R3} = \max\{\frac{f(\vec{x}_i) - f(\vec{u}_i)}{f(\vec{u}_i) - f_{\text{optimum}}}, 0\}$$

R1 is the fitness difference of offspring from parent when an improvement is seen. This definition has been used commonly in literature for parameter control [4, 20, 22]. R2 assigns a higher reward to an improvement over the best so far solution than to an improvement over the parent. Finally, R3 is a variant of R1 relative to the difference between the offspring fitness and the optimal fitness, i.e., maximise the fitness difference between parent and

Table 2: Hyperparameter values of DE-DDQN

Training and online parameters	**Parameter value**
Scaling factor (F)	0.5
Crossover rate (CR)	1.0
Population size (NP)	100
$FE^{\max}$ per function	10^4 function evaluations
Max. generations (gen)	10
Window size (W)	50
Type of neural network	Multi layer perceptron
Hidden layers	4
Hidden nodes	100 per hidden layer
Activation function	Rectified linear (Relu) [19]
Batch size	64
Training only parameters	**Parameter value**
Training policy	ϵ-greedy ($\epsilon = 0.1$)
Discount factor (γ)	0.99
Target network synchronised (C)	every $1e3$ steps
Observation memory capacity	10^5
Warm-up size	10^4
NN training algorithm	Adam (learning rate: 10^{-4})
Online phase parameters	**Parameter value**
Online policy	Greedy

offspring and minimise fitness difference between offspring and optimal solution. This definition can only be used when the optimum values of the functions used for training are known in advance.

5 EXPERIMENTAL DESIGN

In our implementation of DE-DDQN, the primary and target NNs are multi-layer perceptrons. We integrate the three reward definitions R1, R2 and R3 into DE-DDQN and the resulting methods are denoted DE-DDQN1, DE-DDQN2 and DE-DDQN3, respectively. For each of these methods, we trained four NNs using batch sizes 64 or 128 and 3 or 4 hidden layers, and we picked the best combination of batch size and number of hidden layers according to the total accumulated reward during the training phase. In all cases, the most successful configuration was batch size 64 with 4 hidden layers. Results of other configurations are not shown in the paper.

The rest of the parameters are not tuned but set to typical values. In the training phase, we applied ϵ-greedy policy with $\epsilon = 10\%$ of the actions selected randomly and the rest according to the highest Q-value. In the warm-up phase during training, we set the capacity of the memory of observations larger than the warm-up size so that 90% of the memory is filled up with observations from random actions and the rest with actions selected by the NN. The gradient descent algorithm used to update the weights of the NN during training is Adam [15]. Table 2 shows all hyperparameter values.

We compared the three proposed DE-DDQN variants with ten baselines: random selection of mutation strategies (Random), four different fixed-strategy DEs (DE1-DE4), PM-AdapSS (AdapSS) [8], F-AUC (FAUC) [9], RecPM-AOS (RecPM) [23] and the two winners of CEC2005 competition, which are both variants of CMAES: LR-CMAES (LR) [2] and IPOP-CMAES (IPOP) [3]. Among all these alternatives, AdapSS, FAUC, RecPM are AOS methods that were proposed to adaptively select mutation strategies. The parameters of these AOS methods were previously tuned with the help of an offline configurator IRACE [23] and the tuned hyperparameter values (parameters of AOS and not DE) have been used in the experiments. The first eight baselines involve the DE algorithm with the following parameter values: population size ($NP = 100$), scaling factor ($F = 0.5$) and crossover rate ($CR = 1.0$). This choice for parameter F has shown good results [6]. CR as 1.0 has been chosen to see the full potential of mutation strategies to evolve each dimension of each parent. The results of LR and IPOP are taken from their original papers from the CEC2005 competition for the comparison.

5.1 Training and testing

In order to force the NN to learn a general policy, we train on different classes of functions. From the 25 functions of the CEC2005 benchmark suite [26], we excluded non-deterministic functions and functions without bounds (functions $F4$, $F7$, $F17$ and $F25$). The remaining 21 functions can be divided into four classes: unimodal functions $F1 - F5$; basic multimodal functions $F6 - F12$; expanded multimodal functions $F13 - F14$; and hybrid composition functions $F15 - F24$. We split these 21 functions into roughly 75% training and 25% testing sets, that is, 16 functions ($F1$, $F2$, $F5$, $F6$, $F8$, $F10$–$F15$, $F19$–$F22$ and $F24$) are assigned to the training set and the rest ($F3$, $F9$, $F16$, $F18$ and $F23$) are assigned to the test set. According to the above classification, the training set contains at least two functions from each class and the test set contains at least one function from each class except for expanded multimodal functions, as both functions of this class are included in the training set. For each function, we consider both dimensions 10 and 30, giving a total of 32 problems for training and 10 problems for testing.

During training, we cycle through the 32 training problems multiple times and keep track of the mean reward achieved in each cycle. We overwrite the weights of the NN if the mean reward is better than what we have observed in previous cycles. We found this measure of progress was better than comparing rewards after individual runs, because different problems vary in difficulty making rewards incomparable. After each cycle, the 32 problems are shuffled before being used again. The mean reward stopped improving after 1890 cycles (60480 problems, 6048×10^5 FEs) which indicated the convergence of the learning process.

Although the computational cost of the training phase is significant compared to a single run of DE, this cost is incurred offline, i.e., one time on known benchmark functions before solving any unseen function, and it can be significantly reduced by means of parallelisation and GPUs. On the other hand, we conjecture that training on even more data from different classes of functions should allow the application of DE-DDQN to a larger range of unknown functions.

After training, the NN weights were saved and used for the testing (online) phase.[1] For testing, each DE-DDQN variant was independently run 25 times on each test problem and each run was stopped when either absolute error difference from the optimum is smaller than 10^{-8} or 10^4 function evaluations are exhausted. Mean

[1]The weights obtained after training are available on Github [24] together with the source code, and can be used for testing on similar functions including expanded multimodal. The code may be adapted to train or test using other benchmark suites such as BBOB with functions of up to dimension 50.

Table 3: Mean (and standard deviation in parenthesis) of function error values obtained by 25 runs for each function on test set. Former five are dimension 10 and last five are dimension 30. We refer DE-DDQN as DDQN. Bold entry is the minimum mean error found by any method for each function.

Function	Random	DE1	DE2	DE3	DE4	AdapSS	FAUC	RecPM	LR	IPOP	DDQN1	DDQN2	DDQN3
$F3$-10	2.34e+8 (1.06e+8)	2.78e+8 (1.30e+8)	2.26e+8 (1.10e+8)	2.38e+8 (1.23e+8)	2.63e+8 (1.42e+8)	3.37e+4 (3.62e+5)	3.53e+5 (1.65e+4)	3.08e+4 (2.64e+4)	**4.94e-9** (1.45e-9)	5.60e-9 (1.93e-9)	3.98e+3 (1.91e+3)	7.38 e+0 (3.59e0)	2.12e+1 (1.14e+1)
$F9$-10	1.20e+2 (1.32e+1)	1.18e+2 (1.20e+1)	1.22e+2 (1.88e+1)	1.16e+2 (1.44e+1)	1.22e+2 (1.71e+1)	4.10e+1 (6.36e+0)	4.36e+1 (5.99e+0)	3.79e+1 (6.33e+0)	8.60e+1 (3.84e+1)	**6.21e+0** (2.10e+0)	4.19e+1 (6.21e+0)	3.68e+1 (4.64e+0)	3.86e+1 (7.66e+0)
$F16$-10	6.46e+2 (1.02e+2)	6.50e+2 (9.65e+1)	6.31e+2 (1.15e+2)	5.91e+2 (1.07e+2)	6.33e+2 (9.97e+1)	1.90e+2 (2.21e+1)	2.05e+2 (1.41e+1)	1.89e+2 (1.25e+1)	1.49e+2 (8.01e+1)	**1.11e+2** (1.66e+1)	1.93e+2 (1.24e+1)	1.79e+2 (2.05e+1)	1.88e+2 (1.41e+1)
$F18$-10	1.33e+3 (1.16e+2)	1.36e+3 (8.81e+1)	1.39e+3 (1.11e+2)	1.36e+3 (1.09e+2)	1.36e+3 (9.67e+1)	6.13e+2 (1.67e+2)	6.94e+2 (1.93e+2)	6.48e+2 (1.82e+2)	8.40e+2 (2.17e+2)	6.02e+2 (2.76e+2)	**5.20e+2** (1.93e+2)	5.81e+2 (2.47e+2)	5.98e+2 (2.61e+2)
$F23$-10	1.49e+3 (5.16e+1)	1.51e+3 (6.71e+1)	1.51e+3 (6.03e+1)	1.51e+3 (5.58e+1)	1.49e+3 (4.97e+1)	6.66e+2 (1.99e+2)	7.73e+2 (2.05e+2)	6.37e+2 (1.23e+2)	1.22e+3 (5.16e+2)	9.49e+2 (3.52e+2)	**6.18e+2** (1.40e+2)	6.56e+2 (1.57e+2)	6.90e+2 (1.35e+2)
$F3$-30	2.48e+9 (6.60e+8)	2.68e+9 (7.84e+8)	2.50e+9 (9.04e+8)	2.65e+9 (6.69e+8)	2.51e+9 (8.22e+8)	1.52e+7 (5.50e+7)	6.44e+7 (5.88e+6)	1.31e+7 (6.84e+6)	**1.28e+6** (7.13e+5)	6.11e+6 (3.79e+6)	1.52e+7 (9.07e+6)	3.06e+6 (2.54e+6)	5.72e+6 (1.30e+7)
$F9$-30	5.33e+2 (3.09e+1)	5.27e+2 (3.40e+1)	5.42e+2 (3.73e+1)	5.19e+2 (4.53e+1)	5.41e+2 (3.43e+1)	2.54e+2 (2.69e+1)	2.88e+2 (1.72e+1)	2.53e+2 (1.26e+1)	4.19e+2 (1.02e+2)	**4.78e+1** (1.15e+1)	2.73e+2 (1.97e+1)	2.39e+2 (1.52e+1)	2.73e+2 (2.24e+1)
$F16$-30	1.19e+3 (1.36e+2)	1.18e+3 (1.72e+2)	1.18e+3 (1.16e+2)	1.21e+3 (1.35e+2)	1.20e+3 (1.63e+2)	3.11e+2 (6.26e+1)	3.48e+2 (5.27e+1)	2.97e+2 (3.00e+1)	2.52e+2 (2.08e+2)	**1.96e+2** (1.45e+2)	3.18e+2 (4.22e+1)	3.74e+2 (9.03e+1)	3.39e+2 (8.41e+1)
$F18$-30	1.41e+3 (5.70e+1)	1.43e+3 (4.70e+1)	1.41e+3 (6.47e+1)	1.42e+3 (4.59e+1)	1.42e+3 (5.54e+1)	9.65e+2 (5.59e+1)	1.02e+3 (2.37e+1)	9.71e+2 (2.31e+1)	9.64e+2 (1.46e+2)	**9.08e+2** (2.76e+0)	1.04e+3 (2.27e+1)	9.45e+2 (1.42e+1)	9.48e+2 (3.25e+1)
$F23$-30	1.58e+3 (4.64e+1)	1.57e+3 (4.05e+1)	1.55e+3 (4.51e+1)	1.57e+3 (4.14e+1)	1.57e+3 (5.15e+1)	9.43e+2 (1.40e+2)	1.10e+3 (1.01e+2)	9.67e+2 (1.30e+2)	7.51e+2 (3.30e+2)	**6.92e+2** (2.38e+2)	1.17e+3 (6.30e+1)	9.74e+2 (1.69e+2)	9.64e+2 (1.70e+2)

Table 4: Average ranking of all methods.

Algo	IPOP	DDQN2	DDQN3	RecPM	LR	AdapSS	DDQN1	FAUC	Random	DE3	DE2	DE4	DE1
Rank	2.3	3.3	4.1	4.4	4.4	4.9	5.4	7.2	10.5	10.8	10.8	11.4	11.5

and standard deviation of the final error values achieved by each of the 25 runs are reported in Table 3.

5.2 Discussion of results

The average rankings of each method among the 10 test problem instances are shown in Table 4. The differences among the 13 algorithms are significant ($p < .01$) according to the non-parametric Friedman test. We conducted a post-hoc analysis using the best performing method (DE-DDQN2) among the newly proposed ones as the control method for pairwise comparisons with the other methods. The p-values adjusted for multiple comparisons [16] are shown in Table 5. The differences between DE-DDQN2 and the five baselines, random selection of operators and single strategy DEs (DE1-DE4), are significant while differences with other methods are not. The analysis makes clear that the proposed method learns to adaptively select the strategy at different stages of a DE run.

While differences between the three reward definitions are not statistically significant, the rankings provide some evidence that R2 performs better than the other two definitions. R2 being a simple definition assigning fixed reward values does not get affected by the function range, whereas R1 and R3 involving raw functions values may mislead the NN when dealing with functions with different fitness ranges. R2 assigns ten times more reward when offspring improves over the best so far solution than when it improves over its parent. Thus, DE-DDQN2 may learn to generate offspring that not only tend to improve over the parent but also improve the best fitness seen so far. On the contrary, R1 considers the improvement of offspring over parent only and is less informative than R3, which considers improvement over parent and optimum value. The improvement can be small or large when function values with different ranges is considered. As a result, R1 and R3 become less informative about choosing operators that will solve the problem within the given number of function evaluations. Although R3 scales fitness improvement with distance from the optimum which partially mitigates the effect of different ranges among functions, inconsistent ranges are still problematic. The R2 definition encourages the generation of better offsprings than the best so far candidate and it is invariant to differences in function ranges. Comparing with other methods proposed in the literature shows that DE variants with a suitable operator selection strategy can perform similarly to CMAES variants which are known to be the best performing methods for this class of problems.

To further analyze the difference between DE-DDQN and other AOS methods we provide boxplots of the results of 25 runs of DE-DDQN2, PM-AdapSS and RecPM-AOS on each function (Fig. 1). We observe that the overall minimum function value found across the 25 runs is lower for DE-DDQN2 in all problems except $F9$-10 and $F16$-30. As seen in box plots, for $F18$ and $F23$ with dimension 10, DE-DDQN2 often gets stuck at local optima, but manages to find a better overall solution compared to the other methods. Other methods find high variance solutions in these cases. At the same time, the median values of solutions found are better for six out of ten problems. This observation suggests that incorporating restart strategies similar to those used by IPOP-CMAES can be particularly

Table 5: Post-hoc (Li) using DE-DDQN2 as control method.

Comparison	Statistic	Adjusted p-value	Result
DDQN2 vs DE1	4.70819	0.00001	H0 is rejected
DDQN2 vs DE4	4.65077	0.00008	H0 is rejected
DDQN2 vs DE2	4.30627	0.00005	H0 is rejected
DDQN2 vs DE3	4.30627	0.00005	H0 is rejected
DDQN2 vs Random	4.13402	0.00010	H0 is rejected
DDQN2 vs FAUC	2.23926	0.06630	H0 is not rejected
DDQN2 vs DDQN1	1.20576	0.39166	H0 is not rejected
DDQN2 vs AdapSS	0.91867	0.50299	H0 is not rejected
DDQN2 vs Rec-PM	0.63159	0.59848	H0 is not rejected
DDQN2 vs LR	0.63159	0.59848	H0 is not rejected
DDQN2 vs IPOP	0.57417	0.61515	H0 is not rejected
DDQN2 vs DDQN3	0.45934	0.64599	H0 is not rejected

useful for DE-DDQN and give us a direction for future work. DE-DDQN2 performs well consistently for the unimodal $F3$ with both 10 and 30 dimensions, while the other AOS methods find relatively higher error solutions with high variance. We interpret this as an indication that DE-DDQN can identify this type of problem and apply a more suitable AOS strategy than Rec-PM and PM-AdapSS. On the other hand, we see that for $F16$-30 and $F23$-30, DE-DDQN2 exhibits higher variance of solutions, which suggests that higher dimensional multimodal functions often confuse the NN, leading it to suboptimal behaviour.

6 CONCLUSION

We presented DE-DDQN, a Deep-RL-based operator selection method that learns to select online the mutation strategies of DE. DE-DDQN has two phases, offline training and online evaluation phase. During training we collected data from DE runs using a reward metric to assess the performance of the selected mutation action and 99 features to evaluate the state of the DE. Features and reward values are used to optimise the weights of a neural network to learn the most rewarding mutation given the DE state. The weights learned during training are then used during the online phase to predict the mutation strategy to use when solving a new problem. Experiments were run using 21 functions from CEC2005 benchmark suite, each function was evaluated with dimensions 10 and 30. A set of 32 functions was used for training and we run the online phase on a different test set of 10 functions.

All three proposed methods outperform all the non-AOS baselines based on mean error seen in 25 runs on test functions. This shows that the proposed methods can learn to select the right strategy at different stages of the algorithm. Our statistical analysis suggests that differences between the best proposed method and the AOS methods from the literature are not significant, but the best performing version of our model, DE-DDQN2, was ranked overall second after IPOP-CMAES. The R2 reward function, which assigns fixed reward values when better solutions are found, is more helpful for learning an AOS strategy.

For future work, we want to explore applications of Deep RL for learning to control more parameters of evolutionary algorithms,

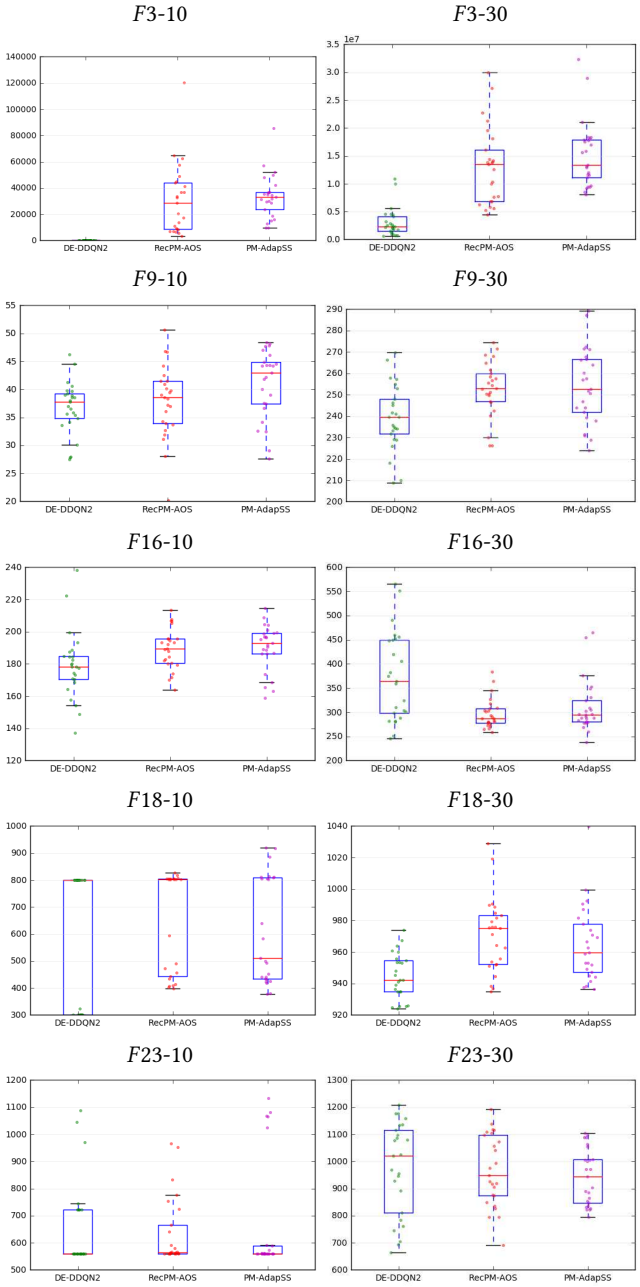

Figure 1: Function error values obtained by 25 runs of DE-DDQN2, RecPM-AOS and PM-AdapSS for each function on test set with dimension 10 and 30.

including combinations of discrete and continuous parameters. We also expect that an extensive tuning of state features and hyperparameter values will further improve performance of the method.

REFERENCES

[1] A. Aleti and I. Moser. 2016. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *Comput. Surveys* 49, 3, Article 56

(Oct. 2016), 35.

[2] A. Auger and N. Hansen. 2005. Performance evaluation of an advanced local search evolutionary algorithm. In *Proceedings of the 2005 Congress on Evolutionary Computation (CEC 2005)*. IEEE Press, Piscataway, NJ, 1777–1784.

[3] A. Auger and N. Hansen. 2005. A restart CMA evolution strategy with increasing population size. In *Proceedings of the 2005 Congress on Evolutionary Computation (CEC 2005)*. IEEE Press, Piscataway, NJ, 1769–1776.

[4] F. Chen, Y. Gao, Z.-q. Chen, and S.-f. Chen. 2005. SCGA: Controlling genetic algorithms with Sarsa(0). In *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, Vol. 1. IEEE, 1177–1183.

[5] A. E. Eiben, M. Horvath, W. Kowalczyk, and M. C. Schut. 2006. Reinforcement learning for online control of evolutionary algorithms. In *International Workshop on Engineering Self-Organising Applications*. Springer, 151–160.

[6] Á. Fialho. 2010. *Adaptive operator selection for optimization*. Ph.D. Dissertation. Université Paris Sud-Paris XI.

[7] Á. Fialho, R. Ros, M. Schoenauer, and M. Sebag. 2010. Comparison-based adaptive strategy selection with bandits in differential evolution. In *Parallel Problem Solving from Nature, PPSN XI*, R. Schaefer et al. (Eds.). Lecture Notes in Computer Science, Vol. 6238. Springer, Heidelberg, Germany, 194–203.

[8] Á. Fialho, M. Schoenauer, and M. Sebag. 2010. Toward comparison-based adaptive operator selection. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2010*, M. Pelikan and J. Branke (Eds.). ACM Press, New York, NY, 767–774.

[9] W. Gong, Á. Fialho, and Z. Cai. 2010. Adaptive strategy selection in differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2010*, M. Pelikan and J. Branke (Eds.). ACM Press, New York, NY, 409–416.

[10] G. Karafotias, A. E. Eiben, and M. Hoogendoorn. 2014. Generic parameter control with reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2014*, C. Igel and D. V. Arnold (Eds.). ACM Press, New York, NY, 1319–1326.

[11] G. Karafotias, M. Hoogendoorn, and A. E. Eiben. 2015. Evaluating reward definitions for parameter control. In *Applications of Evolutionary Computation, EvoApplications 2015*, A. M. Mora and G. Squillero (Eds.). Lecture Notes in Computer Science, Vol. 9028. Springer, Heidelberg, Germany, 667–680.

[12] G. Karafotias, M. Hoogendoorn, and A. E. Eiben. 2015. Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation* 19, 2 (2015), 167–187.

[13] G. Karafotias, S. K. Smit, and A. E. Eiben. 2012. A generic approach to parameter control. In *Applications of Evolutionary Computation, EvoApplications 2012*, D. C. C. et al. (Eds.). Lecture Notes in Computer Science, Vol. 7248. Springer, Heidelberg, Germany, 366–375.

[14] E. Kee, S. Airey, and W. Cyre. 2001. An adaptive genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2001*, E. D. Goodman (Ed.). Morgan Kaufmann Publishers, San Francisco, CA, 391–397.

[15] D. P. Kingma and J. Ba. 2014. Adam: A method for stochastic optimization. *Arxiv preprint arXiv:1412.6980 [cs.LG]* (2014). https://arxiv.org/abs/1412.6980

[16] J. D. Li. 2008. A two-step rejection procedure for testing multiple hypotheses. *Journal of Statistical Planning and Inference* 138, 6 (2008), 1521–1527.

[17] E. Mezura-Montes, J. Velázquez-Reyes, and C. A. Coello Coello. 2006. A comparative study of differential evolution variants for global optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006*, M. Cattolico et al. (Eds.). ACM Press, New York, NY, 485–492.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[19] V. Nair and G. E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. ACM Press, New York, NY, 807–814.

[20] J. E. Pettinger and R. M. Everson. 2002. Controlling genetic algorithms with reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, W. B. Langdon et al. (Eds.). Morgan Kaufmann Publishers, San Francisco, CA, 692–692.

[21] K. Price, R. M. Storn, and J. A. Lampinen. 2005. *Differential Evolution: A Practical Approach to Global Optimization*. Springer, New York, NY.

[22] Y. Sakurai, K. Takada, T. Kawabe, and S. Tsuruta. 2010. A method to control parameters of evolutionary algorithms by using reinforcement learning. In *2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*. IEEE, 74–79.

[23] M. Sharma, M. López-Ibáñez, and D. Kazakov. 2018. Performance Assessment of Recursive Probability Matching for Adaptive Operator Selection in Differential Evolution. In *Parallel Problem Solving from Nature - PPSN XV*, A. Auger et al. (Eds.). Lecture Notes in Computer Science, Vol. 11102. Springer, Cham, 321–333.

[24] M. Sharma, M. López-Ibáñez, and D. Kazakov. 2019. Deep Reinforcement Learning Based Parameter Control in Differential Evolution: Supplementary material. https://github.com/mudita11/DE-DDQN. (2019).

[25] R. Storn and K. Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization* 11, 4 (1997), 341–359.

[26] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y. P. Chen, A. Auger, and S. Tiwari. 2005. *Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization*. Technical Report. Nanyang Technological University, Singapore.

[27] R. S. Sutton and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[28] H. van Hasselt, A. Guez, and D. Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*, D. Schuurmans and M. P. Wellman (Eds.). AAAI Press.