Universiteit Leiden

Master Computer Science

A Genetic Algorithm Parameter Control Method using Deep Reinforcement Learning for Offshore Wind Farm Maintenance Planning.

Name:	Damian Domela Nieuwenhuis Nyegaard
Student ID:	s1853767
Date:	16/02/2022
Specialisation:	Artificial Intelligence
1st Supervisor:	Dr. J.N. van Rijn & M. Huisman
2nd Reader:	Prof.dr. A. Plaat
External Superv.:	S. Mancini & K. Hermans

Master's Thesis in Computer Science

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

Abstract

The increasing demand for renewable energy sources has caused a growing interest in offshore wind energy. Offshore wind farm managers face the NP-hard optimization challenge of scheduling daily service operations, which is subject to constraints such as weather conditions, shifts, vessel and technician capabilities, and availability. Stock-Williams and Swamy [1] proposed a genetic algorithm that uses a wind farm simulator as the basis for its objective function to automate and solve daily maintenance planning for offshore wind farms. The performance of this approach greatly depends on its parameter setting and the proposed default configuration potentially limits the algorithm. The work presented here aims to improve the genetic algorithm in this context by applying a reinforcement learning agent to dynamically control the key parameters at runtime based on the fitness of the population and features of the underlying problem. We show that the reinforcement learning agent together with an improved re-insertion operator increases the convergence speed of the genetic algorithm by 69,7% with only a 0,02% solution quality decrease. This was illustrated using an experiment on the Princess Amalia Wind Park in the Netherlands.

Contents

1 Introduction

In the present-day scenario, the rate of global warming is harmful to the environment and the human race. The non-renewable energy production sector contributes nearly 75% of the total CO_2 emission in the world [2]. As a result, the United Nations is urging every government in the world to adopt more renewable energy sources in order to comply with the Sustainable Development Goals [3] to slow down the effects of climate change.

The increased need for renewable energy sources has caused a considerable rise in attention towards offshore wind energy. In 2020, European countries owned 5.402 wind turbines that are connected to the grid, with the Netherlands being the leading country in terms of newly added capacity in 2020 [4]. With Dutch companies representing an estimated 25% of the offshore wind energy market share globally and the Dutch government planning to increase the energy produced by offshore wind farms from 3,3% to 8,5% of the total energy production in the Netherlands by 2030 [5], there is a valuable opportunity to develop the Dutch international economic position. Alongside the Netherlands, offshore wind farms in Germany are considered subsidy-free under any price scenario. With recent cost reductions indicating that offshore wind power will become cheaper than conventional power generation [6]. This research suggests that subsidy-free wind farms will be the norm in 2023 globally.

Given the practical constraints imposed by offshore operations, the selection of maintenance strategies influences the overall efficiency, profit margin, safety, and sustainability of offshore wind farms. *Operation & Maintance* (O&M) costs have typically accounted for 17% of an offshore wind farm's levelized cost of energy [7]. These O&M activities range from preventive to corrective turbine work, with individual O&M activities (such as the inspection of the wind turbine blades or the repair of a power supply) being referred to as *Service Orders*. On a daily basis, Service Orders need to be scheduled in order to manage and maintain the wind farm. This creates a problem in which different schedules can result in varying O&M costs. Given larger wind farms, creating these schedules by hand becomes infeasible, as a list of 10 Service Orders has more than 3,6 million possible orderings in a schedule. The addition of choices between vessels, allocation of technicians, and uncertainties about the weather creates an even more complex problem.

In Stock-Williams and Swamy [1], the authors proposed a method that automates the process of scheduling daily Service Orders for offshore wind farms, in order to reach better scheduling solutions and minimize the associated costs for offshore wind farms. Over a 5 month period, this method managed to increase the net income by 302 k€ relative to scheduling these Service Orders by hand. This was illustrated by a case study on the Princess Amalia Wind Park in the Netherlands (see Fig. 1a). This automatization is achieved by implementing a genetic algorithm that allocates technicians to Service Orders while also determining an efficient work order. Each individual in the genetic algorithm population represents a work schedule for a given workday. A *genetic algorithm* is an algorithm inspired by Charles Darwin's theory of natural evolution, emulating the process of natural selection, reproduction, and mutation in order to produce offspring for the succeeding generation. By regulating the selection and re-insertion operators

according to a certain fitness function, generational selective pressure causes the population in memory to converge towards a better selection of individuals.

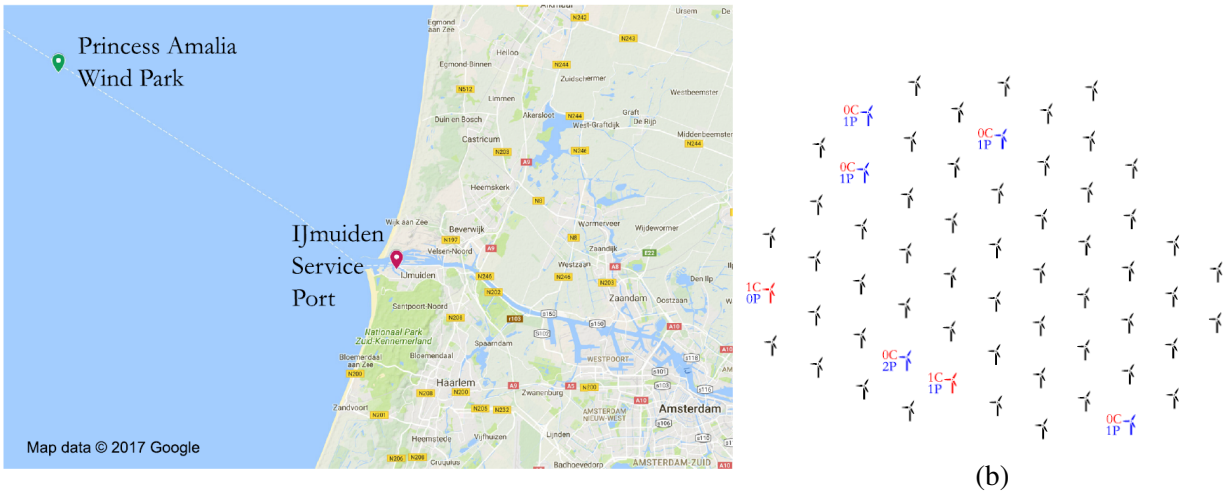


Figure 1: Princess Amalia Wind Park. (a) Location in the North Sea of wind farm and O&M port; (b) turbine layout and status (number of preventive P and corrective C Service Orders open per turbine) on an example day. (Taken from [1])

Although genetic algorithms have reached significant performance in very large, high-dimensional search spaces in terms of pattern recognition and clustering (Maulik and Bandyopadhyay [8]), the performance of genetic algorithms greatly depends on the values of their key parameters. In order to achieve good performance, parameter values should be carefully chosen. Manual parameter adjustment can be difficult and tedious. Therefore, automating this task has received a lot of attention [9]. Autonomous parameter adjustment approaches can be divided into two categories based on the challenges that they attempt to solve [10]:

- **Parameter tuning**: The challenge to select a static parameter setting a priori which is likely to result in good performance based on the performance of previous runs.
- **Parameter control**: The challenge to adjust the parameter setting during runtime, as the optimal parameter values may change over time. Usually, a parameter setting is chosen and applied for a given timeframe, based on the performance, the control method will know how good that choice was and will adjust the parameter setting accordingly.

A recent line of research aims to improve the performance of genetic algorithms by applying reinforcement learning. *Reinforcement Learning* [11] is an influential machine learning algorithm that has had a considerable impact across many different fields and communities [12]. reinforcement learning utilizes an agent which interacts with a complex environment, mapping actions to the said environment based on maximizing the associated rewards.

In Chen et al. [13], reinforcement learning is applied to control the key parameters during the runtime of a genetic algorithm. Two key parameters for a genetic algorithm are the probability of crossover (P_c) and the probability of mutation (P_m). These values indicate the probability

of applying the crossover or mutation operators for each generation of the genetic algorithm. Too large values for P_c and P_m could result in promising individuals being lost; while too small values for P_c and P_m could result in the test cases becoming similar after a certain amount of iterations [14]. In addition to this, a dynamic key parameter configuration grants the genetic algorithm the ability to search in big leaps in the beginning and fine-tune to the near-optimal solution in small steps in the later stages of the search. By using a parameter control method, the user has no need to select a parameter setting a priori, implicitly solving the parameter tuning problem [10].

In this paper, we aim to answer the following research question:
To what extent does the addition of a (deep) reinforcement learning agent as a parameter control method improve the genetic algorithm developed by Stock-Williams and Swamy [1] in terms of computation time and quality for offshore wind farm maintenance planning?

Our contributions are the following:

- An improved re-insertion operator which decreases the original genetic algorithm its computation time by 51,6% at the cost of a 0,02% decrease in performance.
- A Q-Table parameter control method based on the work of Chen et al. [13] which adjusts the key parameter setting for the genetic algorithm during runtime.
- A Deep-Q Network parameter control method which allows for an extended continuous input state definition, while the Q-Table requires a discretized input state definition.
- A Quantile Regression Deep-Q Network parameter control method generates a quantile function for each available action with the aim of better approximation of the stochastic genetic algorithm environment.

2 Background

In this Section, we provide necessary background information to better understand the structure of the genetic algorithm used in [1] as well, and outline related work concerning combining reinforcement learning with genetic algorithms.

2.1 Despatch GUI Tool

To make the genetic algorithm accessible and easily usable, a GUI tool which was written in the C# language and was developed by TNO [15] called *Despatch*. This API adds a visual frontend to the genetic algorithm backend which requires the user to define preventive and corrective Service Orders to be carried out on the wind turbines of a wind farm for that day by loading a file into the API. In addition to this, the user must load the weather forecast for the day into the API as well. During a workday simulation, the technician team is transported from the port to the wind turbines using one or more vessels. These vessels, the technician teams, and Service Orders together define a transfer list, in which the available work, transport, and workers are independently listed. By assigning the technician teams to the vessels and Service Orders while also including additional information such as the time of the simulation, the weather data and the array of turbines to be worked on, a Transfer Plan is defined. *Transfer plans* are the input for the simulator engine (*UWiSE*), which in turn simulates and evaluates the plan. The resulting Transfer Plan can be investigated in the GUI. Fig. 2 shows an overview of the Despatch tool.

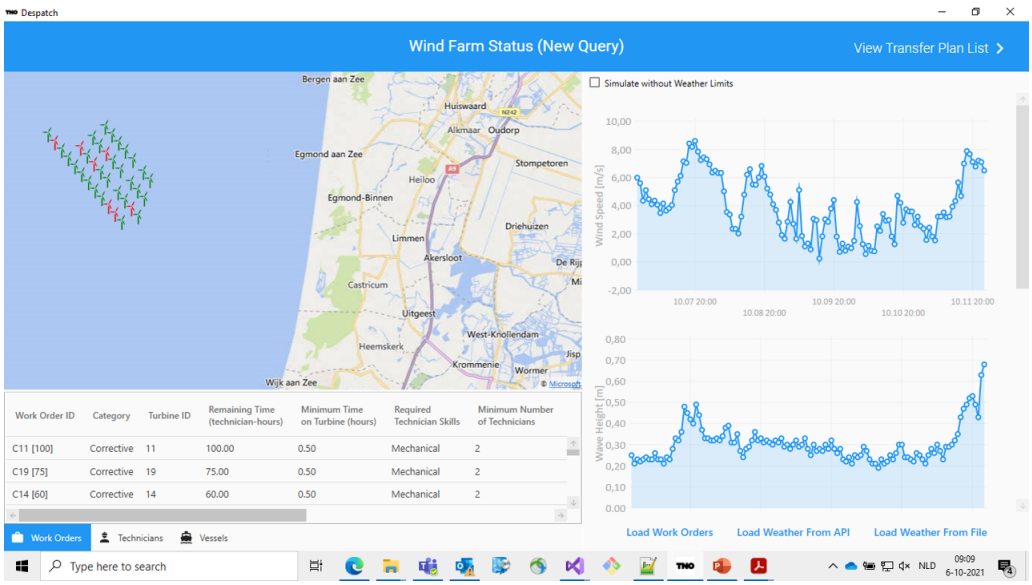


Figure 2: Screenshot of Despatch GUI frontend controlling the genetic algorithm in the backend of the tool.

On a daily basis, O&M activities for wind farms are scheduled according to the structure visible in Fig. 3, which is explained below:

1. At the start of each day, the wind farm manager is presented with a list of pending O&M activities, which will be referred to as Service Orders. These range from preventive main-

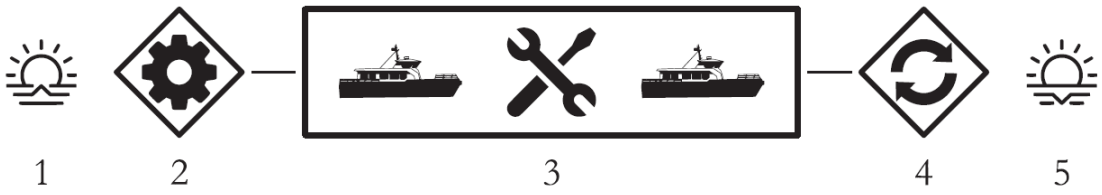


Figure 3: A schematic of the offshore wind farm daily maintenance process. (Taken from [1])

tenance tasks to corrective repairs.

2. Given this list and that morning's weather forecast for the day, a scheduler creates a Transfer Plan which assigns technicians to vessels that travel to wind turbines associated with pending Service Orders.
3. The Transfer Plan is executed, although inaccuracies in the weather forecast, mispredictions in time required to complete Service Orders, and unforeseen health issues for the technicians can cause deviations in the application of the Transfer Plan.
4. When a vessel returns to the port, the work done on the Service Orders is registered, which possibly affects the scheduling done for future Transfer Plans.
5. Before the end of the day, a temporary Transfer Plan can be made for the following day, which will be updated the next morning based on the new weather forecast.

The Despatch tool aims to automate step 2 of this cycle to create a quicker and more accessible way to create Transfer Plans for the Wind Farm in question.

2.2 Traveling Merchant Problem

The abstract form of the problem to be solved by the genetic algorithm, i.e., Service Order prioritization, is in the class of Vehicle Routing problems. More specifically, a variation of the *Traveling Salesman Problem* (TSP) [16] denoted by the authors of Stock-Williams and Swamy [1] as the *Traveling Merchant Problem* (TMP). TMP is a variation of TSP where the salesman is allowed to spend time selling at each city, instead of just considering the distance between cities.

In this problem, shown in Fig. 4, there are k cities $c_1, c_2, ..., c_k$, each located at $l_i = (x_i, y_i)$ for $i \in \{1, 2, ..., k\}$, with the merchant his home city being denoted by c_b. Each city has a population size of p_k, signifying the number of potential customers in that city. The merchant should choose a route in which he spends time selling his wares in the visited cities. Each valid route is required to start and end at c_b. The distance between these cities is defined as $d_{i,j} = ||l_j - l_i||$, with i and j as city indices. Unlike the salesman in TSP, the merchant is not required to find the shortest Hamiltonian cycle in which he visits every city, instead, he can choose what amount of cities r to visit on his route:

$$R = \{c_b, c_{r_1}, ..., c_{r_n}, c_b\}, \quad i \in \{1, 2, ..., k\}, \quad r \subset \{1, 2, ..., k\}$$

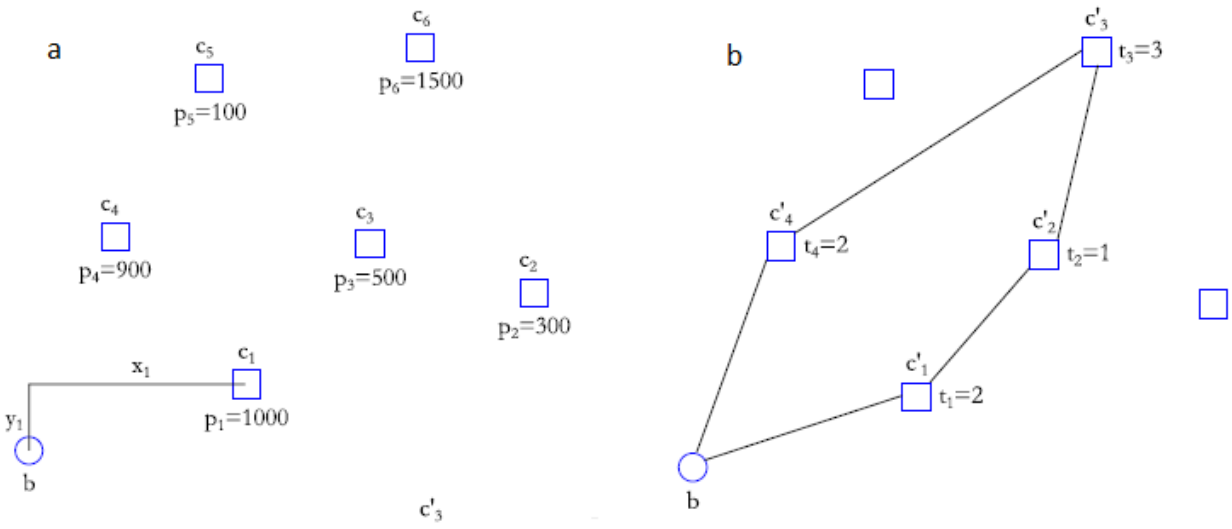


Figure 4: The Travelling Merchant Problem, an abstract problem equivalent to Service Order resource allocation. (a) problem set-up; (b) an example solution. (Taken from [1])

where r is an ordered list of the n visited city indices excluding the home city. At each city, the merchant can choose to spend a certain amount of time t_i to sell his goods. Given that the merchant sells at a steady rate of s people per hour, receiving an income of q per sale, the merchant his income after t_i hours in city i is:

$$I_i = min(s \cdot t_i, p_i) \cdot q$$

The merchant is given the constraint to return home after a time period τ has elapsed. The merchant's transport travels with a certain speed v, meaning that the time spent selling and traveling must never exceed the allowed time, resulting in the following constraint:

$$\frac{1}{v} \cdot \left(\sum_{j=2}^{n} (d_{r_{j-1},r_j}) + d_{b,r_1} + d_{r_n,b} \right) + \sum_{j=1}^{r} t_{r_j} \leq \tau$$

The merchant his objective in this problem is to choose a route which maximizes his income given τ:

$$I_{opt} = \max \left\{ \sum_{i=1}^{n} \min(s \cdot t_{r_i}, p_{r_i}) \cdot q \right\}$$

Since TSP is NP-hard, TMP is also at least NP-hard. A possible solution is to solve a TSP for each possible subset of cities and allocate selling time by solving the Continuous Knapsack Problem [17].

In this problem, the cities represent the Service Orders, while the time spent by the merchant at each city represents the allocation of technician resources to that Service Order.

2.3 Genetic algorithm

The Despatch tool utilizes a genetic algorithm backend as its fundamental optimization algorithm. genetic algorithms belong to the family of metaheuristic global optimization algorithms, which consist of four main steps [18] as illustrated in Fig. 5:

1. **Generation:** For a genetic algorithm to propose new solutions to the problem, a population is stored in memory consisting of a set of solutions. Each possible solution, or individual, is encoded into a *decision vector*: a vector of binary, integer, or real numbers. During the initialization of the population, individuals are generated randomly. Once the desired population size has been reached, new individuals are generated by a process mimicking natural evolution: through applying *crossover* and *mutation* operators on existing individuals selected from the population. By carefully choosing the parameter settings for the selection, crossover, and mutation operators, a good balance can be found for a given problem between exploration across the search space and convergence towards the near-optimal solution.

2. **Conversion:** The newly generated individuals their corresponding decision vectors need to be converted into a representation that is applicable to the current problem context and allows for evaluation in the next step. Often additional information needs to be added to form this representation.

3. **Evaluation:** Each converted decision vector is evaluated according to some optimization objective(s), generating a fitness value that can be associated with the respective individual.

4. **Re-insertion:** The fitness value(s) for the newly generated individual(s) are compared to the fitness values of the individuals in the current population (based on rules, for example that the new individual replaces the current worst individual if it has a better fitness value). After this step, the genetic algorithm checks if convergence is reached, if not, it repeats the loop, if so, it ends the computation.

These steps are also visualized in Fig. 5. Note that the selection of parents chronologically precedes the crossover operator, after which the mutation operator is applied to the offspring generated by the crossover operator. In addition to this, each genetic algorithm generation leads to s a singular new individual, significantly decreasing the volume of executed simulations, as the evaluation step is quite computationally heavy. This figure further details the *Conversion* and *Evaluation* steps in the cycle, in which it is highlighted that a decision vector is transformed from its vector presentation (see Section 2.4) into a Transfer Plan that allocates technicians to available Service Orders. Given this Transfer Plan, a simulation is made of the current day considering the weather forecast in terms of wind speed and wave height. This simulation outputs a fitness value according to a chosen optimization objective, with further intricacies detailed in Section 2.7.

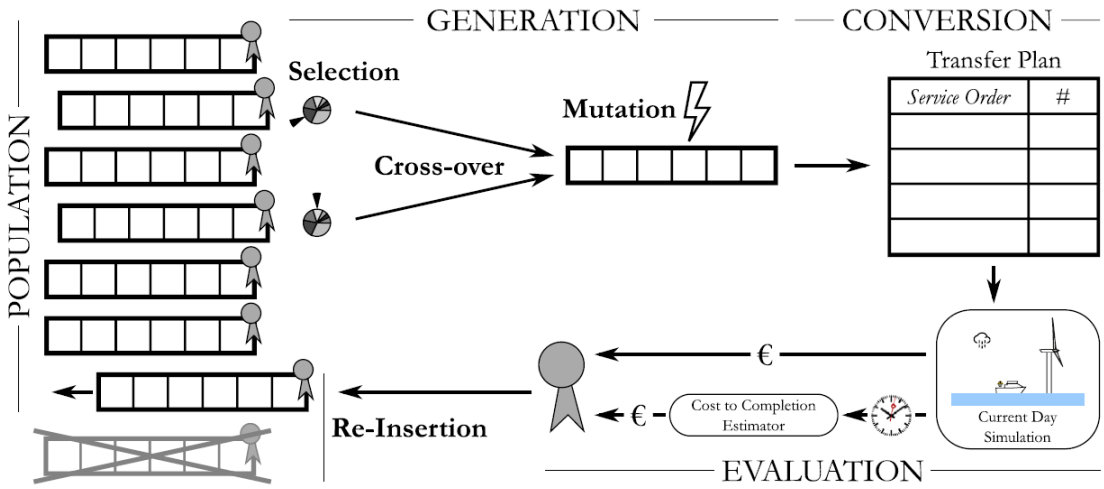


Figure 5: The process involved in completing one iteration of a genetic algorithm search, with details specific to the offshore wind farm daily maintenance scheduling problem. (Taken from [1])

2.4 Individual Representation

Each individual in the population is represented by a decision vector. The challenge is to represent not only the order in which the Service Orders will be worked on but also the technician resource allocation. This is achieved by using a real-coded decision vector of a fixed length. Each element consists of a real number which indicates both the priority and the position on a cumulative distribution of resources (see Fig. 6, 7).

Converting a decision vector into a Transfer Plan is done in four main steps:

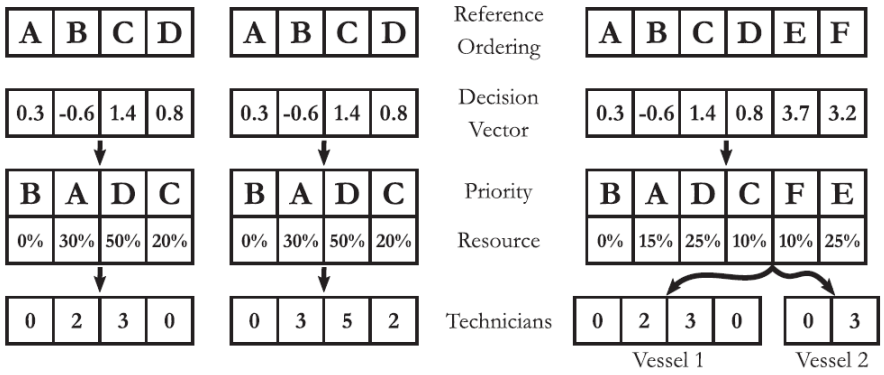


Figure 6: Example decision vector conversion for Service Order resource allocation (far left: with one vessel and five technicians; middle: with one vessel and ten technicians; right: with two vessels and ten technicians). The minimum number of technicians allowed to work on a Service Order is 2. (Taken from [1])

1. As displayed in Fig. 6 on the far left, Service Orders are prioritized in ascending order according to their value. Service Order B has the highest priority with a value of −0,6, while Service Order C has the lowest priority with a value of 1,4.
2. The value for each Service Order is matched with a novel cumulative curve proposed by Stock-Williams and Swamy [1] (see Fig. 7) to determine what portion of the available technician resources is allocated to that specific Service Order. Service Order B receives

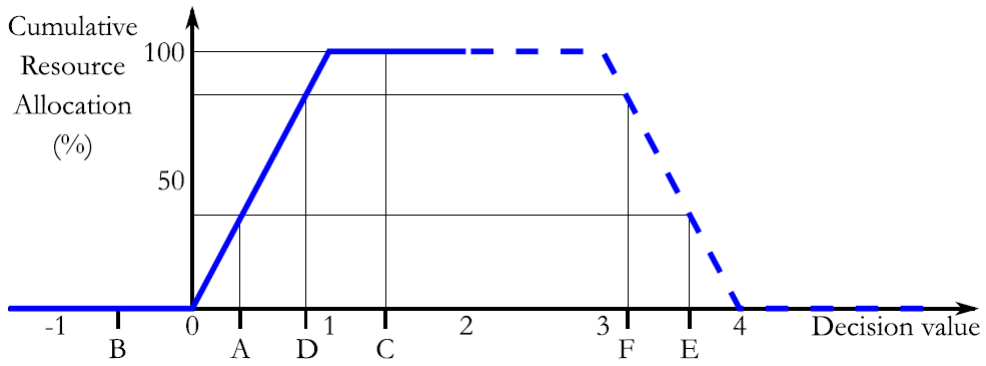


Figure 7: Decision vector conversion for Service Order resource allocation. See Fig. 6 for the use of the values associated with the Service Orders denoted A-F. (Taken from [1])

0% of the available resources, as it has a value below 0. While Service Orders A, D, and C receive 30%, 80% and 100% of the cumulative resources in that order based on their decision value as is visible in Fig. 7 (e.g.: Service Order A has a decision value of 0,3, which corresponds to 30% on the Cumulative Resource Allocation curve). This results in a 30%, 50% and 20% split respectively.

3. The resource allocation percentages must be converted into integer values of technicians. Given the fixed amount of technicians available, they are allocated in order of most resources required (while respecting the skill demands, minimum and maximum technicians requirements).

4. In order to enable multiple vessel compatibility, the resource conversion function is extended by mirroring the function, as shown by the dashed line in Fig. 7. Values between -1 and 2 are associated with the first vessel, while values between 2 and 5, such as the Decision values of Service Orders F and E, are associated with the second vessel. Note that the Service Orders C & F are assigned 10% of the resources each, but due to all Service Orders requiring a minimum of 2 technicians, only 8 out of the 10 total technicians are allocated.

2.5 Evolutionary operators

The fundamental elements of the genetic algorithm are chromosome representation (see Section 2.4), fitness selection (see Section 2.5.1) and the crossover and mutation operators (see Sections 2.5.2 & 2.5.3). Chromosomes are considered as points in the solution space, with a fitness function assigning a fitness value to the chromosomes. The selection operator chooses what chromosomes should be used for further processing based on their fitness values. The crossover operator generates offspring by combining the genetic information of two or more parents. The mutation operator receives the generated offspring from the crossover operator and alters the chromosome to diversify the existing population.

2.5.1 Selection operator

In order to apply the crossover operator, a selection of parents needs to be made from the available population. In contrast to the widely used fitness-proportional selection strategy, Despatch uses the tournament selection strategy as it improves the fitness of each succeeding generation more efficiently while being less computationally costly [19]. As shown in Fig. 8, a random selection is made from the population according to an arbitrary tournament size. Given this selection, the best individual is chosen. Despatch uses a tournament size of 20 while selecting the best 2 individuals in terms of fitness as parents for the crossover operator.

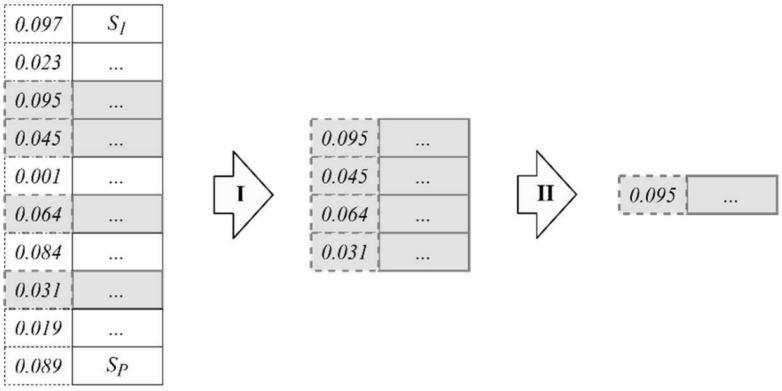


Figure 8: Tournament selection with a tournament size of 4, in a P-size population $\{S_1, \ldots, S_P\}$: I—random selection of 4 individuals, II—selection of the best (most fit) individual. (Taken from [20])

2.5.2 Crossover operator

With a clear definition for an individual its decision vector, it is important to explore the applied crossover operator. The crossover operator is responsible for generating new permutations which combine genetic information from all selected parents, without repetition of values. The crossover operator used in the original Despatch algorithm is the uniform crossover operator [21]. As shown in Fig. 9, the uniform crossover produces new offspring by collectively iterating over the parent vectors, with each parent element having an equal chance to be redistributed to the offspring. This unbiased operator strikes a balance between initially performing global search of the search space when the parents are distant in the genotype space while eventually performing local search of the search space when the parents are close in the genotype space [22]. While this operator was designed for binary vectors, it is applicable to real-encoded vectors as well. Despatch handles a static crossover probability of 1,0, meaning that the crossover operator will always be applied for every generation.

2.5.3 Mutation operators

Given the crossover operator, the second fundamental operation of the genetic algorithm is the mutation operator. The original Despatch algorithm applies two mutation operators sequentially. The first is the *Replace with random number* mutation operator. As shown in Fig. 10, one or more random elements are selected in the decision vector, after which a legal random number

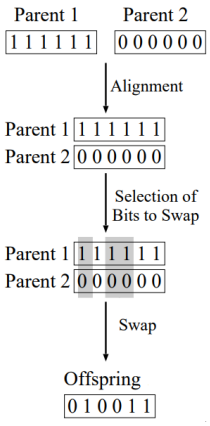


Figure 9: *Uniform* crossover operator in binary genetic algorithms. (Taken from [22])

replaces the associated values. Legal in this context indicates a random value within a lower- and upper bound which corresponds to the number of available vessels. Despatch handles a static mutation probability of 0,15 for this operator, with the number of replacements in the decision vector σ being calculated as follows:

$$\sigma = max(1, \lceil (amount\ of\ Service\ Orders)/15 \rceil)$$

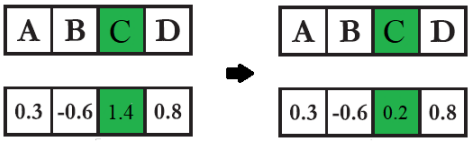


Figure 10: *Replace with random number* mutation operator for real-encoded vectors.

The second sequential mutation operator is the *Random swap* operator. As shown in Fig. 11, two different random elements of a decision vector are selected, after which their corresponding values are switched. This simple operation interchanges the priority and resource allocation of two Service Orders. Despatch handles a static mutation probability of 0,65 for this operator.

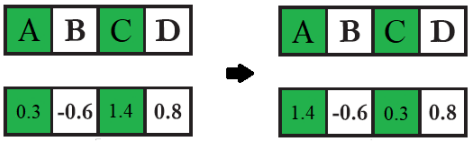


Figure 11: *Random swap* mutation operator for real-encoded vectors.

2.6 Re-insertion operator

After the selection, crossover, and mutation operators are applied in that order, the re-insertion operator receives the newly evaluated individual and concludes the generational cycle of the genetic algorithm by conditionally inserting it into the population. Despatch uses the *Replace random* re-insertion operator, which simply selects a random individual from the current population and compares it to the newly evaluated individual. If the new individual has a superior

fitness value, it replaces the selected individual. This operator ensures that diversity is not hastily decreased, as better individuals have a higher chance to be re-inserted, but do not necessarily eliminate the worst individuals, which might hold useful information as well.

2.7 Evaluation of Transfer Plan

Given a decision vector which was converted into a Transfer Plan following the steps mentioned in Section 2.4. The UWiSE simulator is run which considers the weather forecast in terms of wave height and wind speed and must abide by the following logic:

- Comply with technician shift time constraints, accessibility given the weather, and the number of technicians allowed per wind turbine.
- Realistic transit, work and break activities while respecting technician work times such that they are returned to the port before the end of their shift.
- Calculate the fitness value in terms of the chosen objective function (for example Net income) by computing the relevant physics formulas with respect to the objective function. The assigned fitness value is the computed simulation result with respect to the objective function without its associated unit (e.g.: A Transfer Plan with a net income of €3.000 corresponds to a fitness value of 3.000).

In order to achieve closure in this optimization, the future cost, or the consequence of choosing not to do something today, needs to be determined. If the simulator were to assume that all maintenance activities can be performed in the future without added cost, further degradation of equipment could occur, causing simple maintenance activities to develop into major maintenance activities. With all the work hours today being denoted by t_1, the average cost of an hour of work today being c_1, the penalty cost of doing work after today's forecast being c_x and the number of hours of work done today being denoted by d_1, the following must hold:

$$t_1 \cdot c_1 < d_1 \cdot c_1 + (t_1 - d_1) \cdot c_x$$

The left-hand side of the equation represents the cost of doing work today, while the right-hand side shows the cost of the actual work done today summed with the cost of doing the leftover work after today. Consequently, the cost of doing work today is lower than the cost of doing work at a later date, as re-arranging the equation shows that $c_x > c_1$. Despatch implements this by adding a small artificial amount to the objective in case of a minimization objective function while subtracting a small amount in case of a maximization objective function. To ensure that critical Service Orders are handled as quickly as possible, the issue date can be manually moved forward or a small artificial penalty can be added to postponing it.

2.8 Related Work

In recent years, the integration of machine learning techniques into *metaheuristics* (algorithms designed to solve approximately a wide range of hard optimization problems without having

to deeply adapt to each problem [23]) has been a growing research interest. This integration aims to improve metaheuristics with regards to efficient and robust search and improve their performance in terms of quality, convergence rate, and robustness [24].

The parameter control for genetic algorithms is one of the subfields of this machine learning integration. This method can be performed in three manners [10]:

- In a *deterministic* manner in which the parameters are adjusted according to pre-defined schedules without any feedback of the search process.
- In an *adaptive* manner in which the parameters are adjusted based on the feedback of performance provided by the search process.
- In a *self-adaptive* manner in which the parameter values are encoded into the solution genomes and evolve along with the problem solutions during the search process. This is a subcategory of the adaptive approach.

This research belongs to the adaptive genetic algorithm parameter control category.

Aleti et al. [25] use a Linear regression model to predict the quality of parameter values used for the genetic algorithm in the next generation. Adaptable parameter values for the linear regression model are the mutation rate, crossover rate, the mutation and crossover operators, the population and mating pool sizes. This work managed to outperform similar methods at the time of its publication.

Leung et al. [26] use a parameter control system using the entire search history to conditionally update the crossover and mutation rates. By recording a search history of evaluated individuals and their associated fitness, an approximation is made of the fitness landscape. The decrease in the slope of average fitness acts as a trigger to change the parameter configuration. This is done by generating trial solutions and estimating their fitness, the parameter setting is updated to the configuration which resulted in the best-estimated fitness.

The majority of research within the field of meta-heuristic machine learning integration uses conventional machine learning techniques such as k-means clustering, k-nearest neighbors, linear regression, Support Vector Machines, etc. With the recent developments of machine learning techniques, more advanced and modern machine learning techniques such as (deep) reinforcement learning have been employed as a research direction towards genetic algorithm parameter control.

Eiben et al. [27] implemented a Q-Table reinforcement learning agent which adjusted the crossover rate, mutation rate, tournament size, and population size based on the improvement of the best fitness value as a reward function. The states were defined as a vector of the previous action vector concatenated to statistics such as the standard deviation of the fitness and the mean fitness. In terms of fitness, this method was able to outperform the benchmark genetic algorithm, while in terms of computation time and amount of generations, this method was outperformed by the benchmark genetic algorithm due to the additional reinforcement learning overhead.

Karafotias et al. [28] developed a Q-Table reinforcement learning agent which adjusted a dynamic set of numerical and categorical parameters based on the improvement in best fitness

normalized by the number of evaluations needed to achieve this improvement. The authors argue that the absolute fitness value should be left out of the state definition as it is unlikely to occur more than once during the execution of the genetic algorithm. Instead, the a priori discretized state definition is based on diversity, fitness improvement, standard deviation, and stagnation. Across 15 problems, the reinforcement learning controller showed an existing margin of improvement relative to the static parameter genetic algorithm.

Buzdalova et al. [29] and Sakurai et al. [30] construct a Q-Table reinforcement learning agent which selects the crossover and mutation operators to be used during the runtime of a genetic algorithm. This adaptive approach allows the agent to utilize the advantages that each specific operator offers dynamically throughout the execution of the genetic algorithm. The reward in this context was derived from the generational improvement of the best fitness.

All the related work regarding reinforcement learning mentioned so far, along with this work, concerns value-based reinforcement learning. Value-based methods aim to find the optimal policy for an agent by finding the best action-value of a state, after which the accompanying actions are found. This functions well for discrete action spaces, but fails for continuous action spaces. Given that this work utilizes probabilities in the actions space, policy-based reinforcement learning could work better in this context. Schulman et al. [31] proposed Proximal Policy Optimization algorithms. This method aims to find the optimal policy for the agent by starting with a policy function and adjusting it with a policy gradient method which alternates between data sampling through interactions with the environment, and optimizing a surrogate objective function using stochastic gradient ascent.

This work aims to implement a Q-Table reinforcement learning agent based on [13], as well as two more complex deep reinforcement learning agents which abolish the need for discretized input state definitions, as parameter control methods. This agent adjusts the probabilities of crossover and mutation during the execution of the genetic algorithm. This was done in the context of a genetic algorithm solving the efficient allocation of technicians to wind farm Service Orders as discussed by Stock-Williams and Swamy [1].

3 Methodology

As the TMP is an NP-hard problem, it cannot be deterministically solved in polynomial time. The genetic algorithm is used as an approximation algorithm to find approximate solutions for the problem instances. In order to improve the efficiency of this approximation algorithm, several methods are introduced. An improved re-insertion operator is introduced in Section 3.1, the usage of a reinforcement learning agent as a parameter control method is introduced in Section 3.2 and its subsections.

3.1 Despatch Bottleneck & Re-insertion Operator Switch

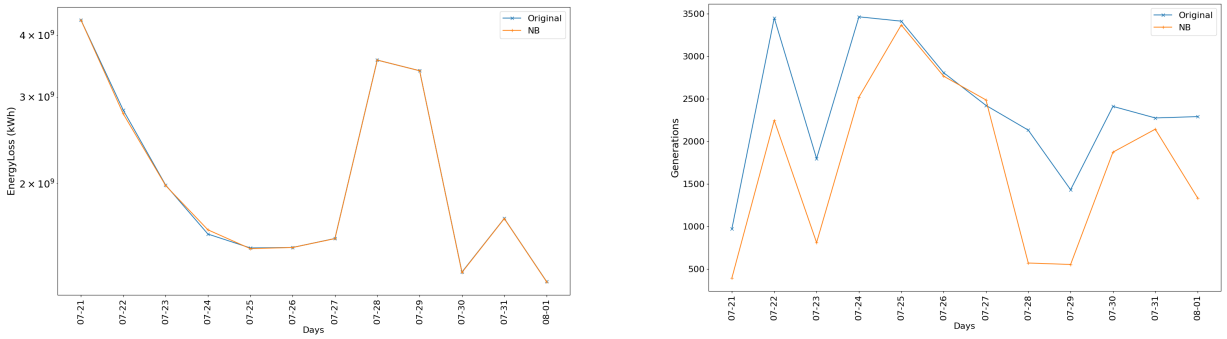
The genetic algorithm in the Despatch backend was programmed to have a certain criterion that must be met if the algorithm were to converge to a solution. This is the *Absolute Fitness Convergence* (*AFC*) condition, which is defined as follows:

$$AFC = \begin{cases} True, & \text{if } |maxf(x_i) - minf(x_i)| < c_T \\ False, & \text{otherwise} \end{cases} \tag{1}$$

With $maxf(x_i)$ and $minf(x_i)$ denoting the maximum and minimum individual fitness in the population respectively and c_T indicating the convergence tolerance. This constant is manually defined before runtime, with the Despatch algorithm defining $c_T = 1$. Empirically, we observe that practically all runs of the genetic algorithm contains individuals with fitness values that are far greater than 1, this convergence condition approximately requires the whole population to consist of the best-found individual.

In Section 2.6, the operator used for re-insertion is outlined. Although this operator is useful when it comes to quickly avoiding a reduction in diversity, the combination with the AFC condition creates a profound problem. Towards the end of convergence, the population will almost entirely consist of the best-found individual. The last few individuals that are inferior in terms of fitness need to be replaced if the algorithm were to converge. Using the *Replace random* re-insertion operator creates a bottleneck situation in which the last few inferior individuals are very rarely selected to be replaced by a newly evaluated individual, making the final steps of convergence very time-consuming due to the unnecessary repetition of the costly simulation function.

We propose a modified re-insertion operator which eliminates this problem by switching to a different operator if more than 70% of the population consists of the current best-found individual. If this condition is met, the *Replace worst* re-insertion operator is applied. This operator always compares the newly evaluated individual to the worst individual in the population in terms of fitness. If the new individual is superior, it replaces this worst individual. This operator is greedy in nature, as it significantly reduces the diversity in the population as the genetic algorithm progresses. This is why this operator is only used after said threshold condition, meaning it is only applied to finish converging towards a solution. The addition of this dynamic operator will be reffered to as the *No Bottleneck* method (NB). A performance comparison between the



(a) End performance comparison.

(b) Generational duration comparison.

Figure 12: Visual comparison between the NB method and the Original Despatch algorithm across the validation set.

original Despatch algorithm and the NB method is shown in Fig. 12a and Fig. 12b.

In Fig. 12a, for a single run of both methods, the NB method approximately performs equally well relative to the original Despatch algorithm across the validation dataset. While in Fig. 12b, it is shown that the NB method considerably decreases the number of generations used to reach the same solution.

3.2 Reinforcement learning for parameter control

A reinforcement learning agent aims to map actions to environmental states with the aim of achieving the maximum cumulative reward. The agent continuously interacts with an environment to optimize its value-based action selection according to the feedback signals of the environment (Sutton and Barto [11]). The reinforcement learning model framework is shown in Fig. 13, at time step t, the agent receives a corresponding state s_t and responds to it using action a_t. The environment recognizes this action and proceeds to time step $t+1$ and to the associated state s_{t+1}. After executing action a_t, the agent receives the associated reward r_t from the environment, which will give the agent feedback on how useful a_t was and allow it to update its action selection for the next time step. After the agent has experienced extended exposure to the environment, it will find an optimal policy π^* which will aim to achieve maximum long-term rewards [32]. In this work, three value-based reinforcement learning agents are implemented to control the discrete action space of crossover and mutation probabilities, inspired by the action space definition in Chen et al. [13]. The value-based reinforcement learning agent attempts to find the optimal policy by utilizing the value function $V(s)$ and selecting the most valuable action for all observed states. This policy is expressed as Eq. 2 (Plaat [33]).

$$\pi^*(a|s) = \arg\max_{\pi} V^{\pi}(s_0) \tag{2}$$

where $\pi^*(a|s)$ is the optimal policy which selects a certain action a when in state s; where $V^{\pi}(s_0)$ is the expected return when the agent follows policy π starting in the initial state s_0. The $\arg\max$ function is used to select the policy with the highest expected value.

We propose a reinforcement learning agent which controls the probabilities of crossover and

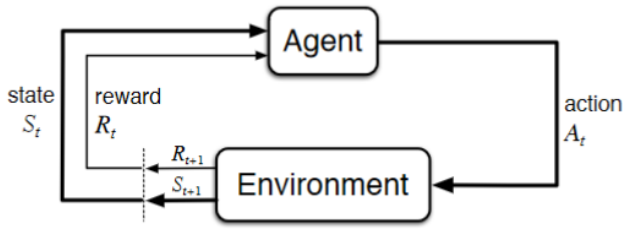


Figure 13: The reinforcement learning model framework. (Taken from [11])

mutation for each generation based on an assessment of the genetic algorithm population and optionally also on features of the genetic algorithm. A reinforcement learning agent acting as a parameter control method for the genetic algorithm is referred to as a *Self-learning Genetic Algorithm* (SLGA). The SLGA execution flow is shown in Algorithm 1.

Algorithm 1 SLGA for Wind Farm Maintenance Planning Q-Table DQN QRDQN

Input: Service Order list, Weather forecast
Output: Population of Transfer Plans
Initialize Genetic Algorithm
- Set iteration $t = 0$, calculate the fitness of initial population
Initialize RL, calculate initial population diversity d^*, best fitness m^* and fitness sum f^*
- Set initial $P_c = 1{,}0$ and $P_m = \{0{,}15, 0{,}65\}$
while $AFC == False$ **do**
  Select parents using tournament selection
  Calculate state s_t of Genetic Algorithm according to Eq. 6 Table 1 Table 1, $s \leftarrow s_t$
  Choose action a_t with $\varepsilon - greedy$, $a \leftarrow a_t$
  Execute action a ▷ action a will set new values for P_c and P_m
  Apply crossover operator using P_c
  Apply mutation operator using P_m
  Compute fitness of newly generated individual by UWiSE simulator
  Calculate the reward r_t according to Eq. 7 and Eq. 8, $r \leftarrow r_t$
  Update Q value according to Eq. 9 Eq. 14 Eq. 17 using r
  $t = t + 1$
return Best-found Transfer Plan

The exact learning procedure for this policy depends on the structure of the reinforcement learning algorithm. The respective learning procedures will be explained in the following sections for all applied methods.

3.2.1 Q-Table parameter control Method

Based on the method proposed by Chen et al. [13], this work implements a Q-Table reinforcement learning agent as a parameter control method for genetic algorithms, previously also defined as an SLGA. The SLGA is divided into the environment, the learning module, and the reinforcement process. The continuous state acquisition, agent reaction, feedback retrieval, and policy adjustment collectively form the reinforcement process. The reinforcement learning agent combined with the Q value table functions as the learning module.

3.2.1.1 State set

The genetic algorithm is regarded as the environment, with the derived state being based on the following aspects of the population fitness:

1. Average fitness of the population
2. Population diversity
3. The best individual fitness

f^* gives the current average fitness of the population, normalized by the average fitness of the initial population (see Eq. 3).

$$f^* = \frac{\sum_{i=1}^{N} f(x_i^t)}{\sum_{i=1}^{N} f(x_i^1)} \tag{3}$$

d^* gives the current population diversity normalized by the diversity of the initial population (see Eq. 4).

$$d^* = \frac{\sum_{i=1}^{N} \left| f(x_i^t) - \frac{\sum_{i=1}^{N} f(x_i^t)}{N} \right|}{\sum_{j=1}^{N} \left| f(x_j^1) - \frac{\sum_{j=1}^{N} f(x_j^1)}{N} \right|} \tag{4}$$

m^* gives the best individual fitness of the current population normalized by the initial population's individual best fitness (see Eq. 5).

$$m^* = \frac{max f(x_i^t)}{max f(x_i^1)} \tag{5}$$

These assessments are normalized by the initial population to illustrate the changes in best fitness, population diversity, and fitness average relative to the starting point of the algorithm. Diversity and average fitness represent the whole population state, while the best fitness value only reflects the best individual. Only an outstanding individual could change the state and consequently the behavior of the agent.

The state value is calculated by weighting and summing these three assessments (Chen et al. [13]), as is shown in Eq. 6.

$$S^* = w_1 * f^* + w_2 * d^* + w_3 * m^* \tag{6}$$

Given a minimization problem, the state value is approximately between 0 and 1. As the reinsertion operator only allows for improvements in the population in terms of fitness, f^* and m^* start at 1,0 and progressively decrease towards 0,0. The same holds for a maximization problem if Eq. 3 and Eq. 5 are inversed. Although Eq. 4 could theoretically be above 1,0 if a series of large improvements are made and consequently the diversity is increased, the convergence condition in Eq. 1 ensures that, as the genetic algorithm progresses, $d^* \approx 0{,}0$ as the whole population consists of the best-found individual (with $c_T = 1$).

The weights should sum to 1 such that $w_1 + w_2 + w_3 = 1$, meaning that each individual weight

represents the relative importance of the associated variable. In SLGA, w_1, w_2 and w_3 are set to 0,35, 0,35 and 0,3 respectively (Chen et al. [13]).

With S^* being a continuous variable, the amount of possible states is enormous. A larger state set leads to a more expressive policy for the reinforcement learning agent, but it requires more exploration to learn the policy, which will affect the convergence of the genetic algorithm. A smaller state set leads to a less expressive policy for the agent, but it requires less exploration to learn the policy. In order to control the state set size, it is discretized based on the implementation by Shahrabi et al. [34] to define 9 state ranges.

$$S = \{[0{,}0, 0{,}15), [0{,}15, 0{,}25), [0{,}25, 0{,}35), [0{,}35, 0{,}45), [0{,}45, 0{,}55), \\ [0{,}55, 0{,}65), [0{,}65, 0{,}75), [0{,}75, 0{,}85), [0{,}85, 1{,}0]\}$$

If S^* its value falls in between one of the above ranges, the current state is associated with its index in $S = \{s_0, s_1, ..., s_8\}$, such that $s_0 = [0{,}0, 0{,}15)$. If $S^* > 1{,}0$ then $s = s_8$.

3.2.1.2 Action set

Given the current state s_t, the reinforcement learning agent will react by executing action a_t. In the SLGA, an action by the reinforcement learning agent represents both a probability of crossover P_c and a probability of mutation P_m. Each action corresponds to a range with a lower- and upper bound. Executing this action will select a random value within this range and set it as the probability for that operator. The action set for crossover A_c features 10 actions for probability ranges which were manually defined to cover a large range of values around the static parameter P_c used in the Despatch algorithm (see Section 2.5.2), as is shown below.

$$A_c = \{[0{,}5, 0{,}55), ..., [0{,}95, 1{,}0]\}$$

Similarly, the action set for mutation A_m covers a wide range of probabilities around the static parameters P_m used in the Despatch algorithm (see Section 2.5.3). However, seeing as the Despatch algorithm applies two mutation operators sequentially, the action set for mutation A_m concatenates two sets of 10 probability ranges, one for each mutation operator, such that:

$$A_{m_1} = \{[0.01, 0.06), ..., [0.46, 0.51]\} \\ A_{m_2} = \{[0.1, 0.17), ..., [0.73, 0.8]\}$$

If the reinforcement learning agent would have both of these action sets at its disposal, the action set for mutation would consist of the cartesian product between these two sets. This action set would contain 100 actions. To limit the size of the Q-Table, a different approach was chosen based on the contributions in Buzdalova et al. [29], in which reinforcement learning was used to select the evolutionary operator to apply. This results in one of the probabilities for either mutation operator being set to 0,0, while the other mutation probability is set to a specific setting. The subsequent action set for mutation contains 20 actions in total.

3.2.1.3 Reward functions

In reinforcement learning, each action set should be associated with a reward function to define the short-term value of the available actions in the environment. The reward function definitions in this Section were taken from Chen et al. [13]. The reward for the crossover operator is defined as the best individual fitness in the population resulting from the applied actions, relative to the previous generation (see Eq. 7). The reward for the mutation operator(s) is defined as the total fitness of the current population relative to the previous generation (see Eq. 8). Given that the re-insertion operator only allows for improvements in terms of fitness, the rewards are strictly subject to:

$r_{c_{max}} \geq 0, r_{c_{min}} \geq 0, r_{m_{max}} \geq 0, r_{c_{min}} \geq 0.$

$$r_{c_{max}} = \frac{maxf(x_i^t) - maxf(x_i^{t-1})}{maxf(x_i^{t-1})}, \quad r_{c_{min}} = \frac{minf(x_i^t) - minf(x_i^{t-1})}{minf(x_i^{t-1})} \tag{7}$$

$$r_{m_{max}} = \frac{\sum_{i=1}^{N} f(x_i^t) - \sum_{i=1}^{N} f(x_i^{t-1})}{\sum_{i=1}^{N} f(x_i^{t-1})}, \quad r_{m_{min}} = \frac{\sum_{i=1}^{N} f(x_i^{t-1}) - \sum_{i=1}^{N} f(x_i^t)}{\sum_{i=1}^{N} f(x_i^{t-1})} \tag{8}$$

where $f(x_i^t)$ represents the fitness of the i-th individual in the t-th generation. Note that the reward functions are corrected for the nature of the objective function (minimization or maximization).

3.2.1.4 Learning procedure

The $Q(s_t, a_t)$ value represents the estimated value of action a_t in-state s_t. Assuming there are n states in the state set and m actions in the action set, a $m \times n$ Q value table is used to record the learning experience of the agent. The Q value table is initialized to a zero matrix with the number of columns being equivalent to the action set size and the number of rows being equivalent to the state set size, as is shown in Fig. 14.

Initial Q value table.

$$Q(s_t, a_t) = \begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_n \end{array} \begin{array}{c} \begin{array}{ccccc} a_1 & a_2 & a_3 & \cdots & a_m \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \end{array}$$

Figure 14: Initial Q value table. This table is continuously updated with the aim of representing the expected sum of discounted future rewards for each action in each state. (Taken from [13])

The calculation of the Q value is a combined consideration of the observed state, the selected action, and the associated reward. The action selection strategy for the reinforcement learning agent should offer a trade-off between exploration and exploitation. The environment

should be explored to acquire knowledge on the values of the available actions for all the states in the state set. Once the environment has been sufficiently explored, the agent should exploit the most valuable action. ε-greedy is an action selection strategy in which the reinforcement learning agent chooses a random action with probability ε, which induces exploration of the environment to improve the policy. With the probability of $(1-\varepsilon)$, the agent chooses an action that has maximal estimated action value in the current state, which induces exploitation of the policy in order to get the maximum reward from the environment.

To update the reinforcement learning agent's policy, a learning algorithm is necessary. The two well-known reinforcement learning learning methods are the *Q-learning* (Watkins and Dayan [35]) and *Sarsa* (Sutton and Barto [11]) algorithms. Generally speaking, the Sarsa algorithm has faster convergence characteristics, while the Q-learning algorithm has better final performance [36]. Seeing as the SLGA spends a number of generations exploring the environment, a faster-converging learning algorithm would spend fewer generations executing suboptimal actions and getting stuck in local optima. Due to this reasoning, the Sarsa algorithm (see Eq. 9) was used to update the reinforcement learning policy.

$$Q(s_t, a_t) \leftarrow (1-\alpha) * Q(s_t, a_t) + \alpha * (r_{t+1} + \gamma * Q(s_{t+1}, a_{t+1})) \tag{9}$$

where $Q(s_t, a_t)$ represents the Q value of taking action a_t in the current state s_t; α represents the learning rate; γ is the discount rate, discounting future Q value. $Q(s_{t+1}, a_{t+1})$ represents the expected Q value in state s_{t+1} for action a_{t+1} which is chosen by the ε-greedy strategy. To guarantee exploration in new unknown states and exploitation of the policy once the state has been explored, an exponential ε decay schedule is used (see Eq. 10 and Fig. 15).

$$\varepsilon = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min}) * e^{-\lambda * i} \tag{10}$$

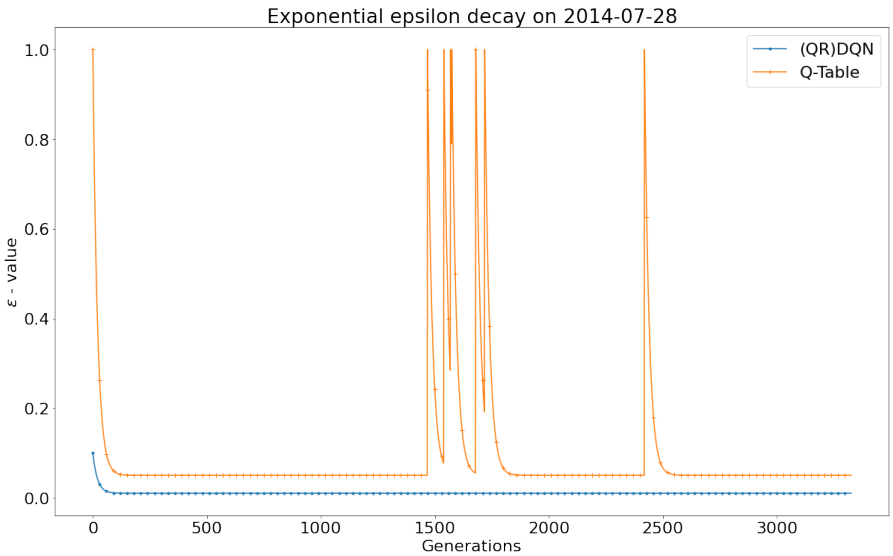


Figure 15: ε decay schedule for the parameter control methods on an example day of the validation set. With ε representing the probability that the reinforcement learning agent acts randomly, this schedule illustrates the balance between exploration and exploitation for the agent.

with ε_{max} indicating the maximal epsilon value and ε_{min} indicating the minimal epsilon value. For the Q-Table implementation, ε_{max} is set to 1,0 and ε_{min} is set to 0,05. The λ decay constant is set to 0,05 and i represents the number of generations spent in the current state, meaning that ε exponentially decreases to the asymptote of 0,05 until a new state is reached. The DQN and QRDQN methods have undergone a preceding training procedure, consequently, the decision has been made to only do a small portion of initial exploration for each specific day by setting ε_{max} to 0,1 and ε_{min} to 0,01.

3.2.2 Deep-Q Network parameter control method

The application of the Q-Table reinforcement learning agent as a parameter control method to the genetic algorithm in this context is unfortunately subject to several shortcomings, particularly in the following aspects:

1. Having a discretized state set is a requirement for the Q-Table implementation, given that there must be a finite amount of entries in the table. This discretization (see Eq. 3.2.1.1) requires a definition of a lower- and upper bound for each respective state. Seeing as each respective day has a different fitness landscape in the genetic algorithm, a static state discretization will work well for some days, while working poorly for other days.
2. The three population assessments (see Eq. 3, Eq. 4 & Eq. 5) are weighted and summed to calculate the state value S^* (see Eq. 6). The values of w_1, w_2 and w_3 are subject to interpretation, as their value sets the relative importance regarding the state value.
3. The Q-Table reinforcement learning agent visits a state until enough progress has been made, after which a new state is visited. A repeated increase for the ε value was necessary for the agent to explore yet unknown states. Unfortunately, this also resulted in taking suboptimal actions for numerous generations due to the agent exploring the environment (often multiple times). These shortcomings can be overcome by using a Deep-Q Network.

In order to tackle these shortcomings, a *Deep Q-Network* (DQN) is implemented which uses a neural network its weights to store the relation between the input state and the Q value for each action.

3.2.2.1 State set

By using the unweighted continuous values of f^*, m^* and d^* as the input state for the reinforcement learning agent, the first two of these shortcomings can be mitigated. A DQN (Mnih et al. [37]) can be used to handle continuous state spaces, in contrast to the Q-Table implementation, but does require an extensive training phase to sufficiently learn the genetic algorithm environment. The discretized state set definition for the Q-Table method only contains population assessments for a single day, which causes a lack of general knowledge on the environment and nullifies the possibility for a training phase. A successful training phase for the DQN mitigates the third listed shortcoming, as the agent should do a small portion of initial exploration (see

Input Features	Category
Amount of vessels available	Wind farm & work day features
Amount of turbines in wind farm	
Amount of technicians available	
Corrective Service Order hours today	
Preventive Service Order hours today	
Average wind speed at turbine hub height	
Standard deviation of wind speed at turbine hub height	
Hours of excessive wave height in the weather forecast	
Workable hours w.r.t. excessive wind speed in the weather forecast	
Diversity of population relative to initial population (d^*)	Population features
Fitness sum of population relative to initial population (f^*)	
Best fitness in current population relative to initial population (m^*)	
Parent 1's fitness relative to fitness sum of population	Selected parent features
Parent 2's fitness relative to fitness sum of population	
Parent 1's fitness relative to the best fitness in the population	
Parent 2's fitness relative to the best fitness in the population	

Table 1: All 16 (numerical) input features for the DQN and QRDQN models, divided into three categories. This extended input feature set aids the Deep-Q Network agents in learning the various fitness landscapes represented by the individual days in the dataset.

Eq. 15). The f^*, m^*, and d^* assessments are informative metrics with respect to the fitness landscape for that specific day, but it fails to approximate all genetic algorithm fitness landscapes. For the DQN to approximate the environment for all days, an extension of the state is made (see Table 1):

The first category of the input features are decimal metrics used to inform the (QR)DQN of the inputs of the genetic algorithm. These describe settings for the wind farm, the available list of Service Orders, and the weather forecast for that day. The second category contains the three continuous unweighted population assessments which will vary during runtime. The third category contains fitness-based features for both parents which were selected by the tournament selection operator. The fitness value of a parent relative to the fitness sum of the population provides a diversity metric, while fitness relative to the best fitness in the population provides a normalized elitist metric.

3.2.2.2 Action set

To facilitate the learning procedure of the (QR)DQN, the action set for the Q-Table implementation is used in a nonrandom manner, as the Q value of an action is easier to learn if there is no variation in the action execution.

$$A_c = \{0{,}55, 0{,}6, ..., 1{,}0\}$$

The DQN crossover action set A_c was constructed by copying the upper bound of each action in the Q-Table's crossover action set, consequently, the new crossover action set consists of 10 static crossover probabilities.

$$A_{m_1} = \{0{,}01, 0{,}06, ..., 0{,}46\}$$
$$A_{m_2} = \{0{,}1, 0{,}17, ..., 0{,}73\}$$

The DQN mutation action sets A_{m_1} and A_{m_2} were constructed by copying the lower bound of each action in the Q-Table's mutation action set, consequently, the new mutation action set consists of 20 static mutation probabilities, 10 for each operator.

3.2.2.3 Learning procedure

Given the neural network of the DQN, each action is represented by a node in the output layer which outputs a decimal value representing the expected Q value for that action in the current input state. A representation of the DQN model is visible in Fig. 16.

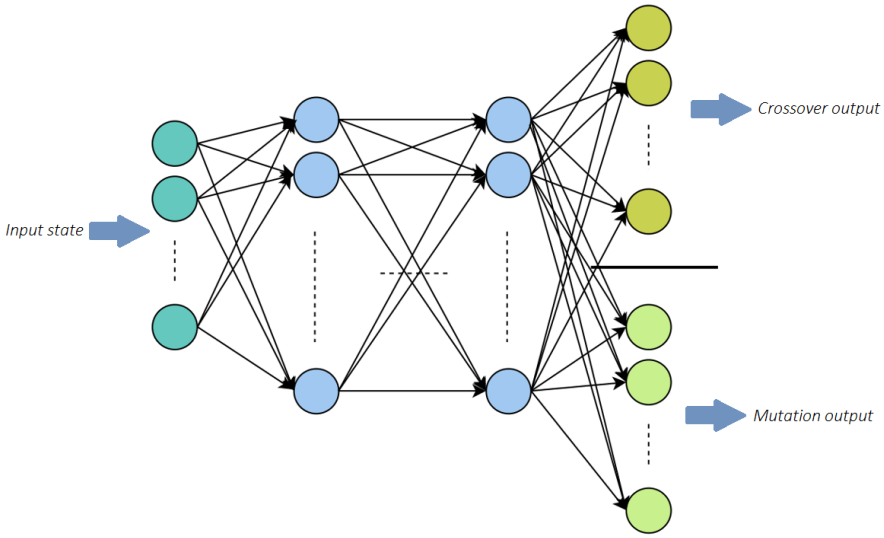


Figure 16: Schematic of the DQN model architecture. The output layer features two output heads, one for crossover and one for mutation. Weight sharing is used prior to the output layer.

Given a specific input state, the hidden layers of the neural network form an encoded representation of its relation to each action its Q value. For the output layer, weight sharing of the final hidden layer was used, after which the crossover and mutation output heads transform the encoded representation into the expected Q values. By designating one output node per action in the output layer, the DQN output head for crossover output contains 10 nodes, while the output head for mutation contains 20 nodes.

The DQN approximates the optimal policy by training on a representative dataset. This dataset consists of workdays and their associated Service Order list and weather forecast. The (untrained) DQN was applied to all days in the training data portion of the dataset with a non-repetitive exponential ε decay schedule with $\varepsilon_{max} = 1{,}0$ and $\varepsilon_{min} = 0{,}05$ (see Eq. 10). In practice, the DQN agent chooses many suboptimal actions which lead to exploration due to it lacking a prior training phase. The knowledge acquired during this application was stored in an *experience replay buffer*. Each tuple element in this buffer was structured as shown below (see Eq. 11).

$$(s_t, a_{c_t}, r_{c_t}, a_{m_t}, r_{m_t}, s_{t+1}) \tag{11}$$

with t denoting the current timestep; a_{c_t} being the executed crossover action in timestep

t; r_{c_t} being the observed reward after this crossover action; a_{m_t} being the executed mutation action in timestep t and r_{m_t} being the observed reward after this mutation action. The respective experience replay buffers for all days were concatenated and stored in a single data file which will act as the training dataset for the DQN agent.

In this learning procedure, a *Double Deep Q-Network* (DDQN) is used in the interest of a stable learning procedure. To illustrate this, take into consideration the target Q value equation for ordinary Q-learning:

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * Q(s_{t+1}, a_{t+1})) \tag{12}$$

Every iteration in which the DQN is updated according to Eq. 12 so that its predictions get closer to $Q(s_t, a_t)$, the $Q(s_{t+1}, a_{t+1})$ output is changed. As a result, the next time the Q function is updated, $Q(s_t, a_t)$ will be different even for the same state. Training the DQN in this context will cause its predictions to chase a moving target, leading to an unstable training procedure. To mitigate this, a duplicate of the Q function is used which is denoted by $Q'(s_t, a_t)$. Substituting this duplicate function as a target network relative to the main network into the update equation results in the equation shown below (see Eq. 13).

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * Q'(s_{t+1}, a_{t+1})) \tag{13}$$

By seperating the main network and the target network, the training procedure is more stable as it is no longer shasing a constantly moving target. Every 100 iterations the main network weights are copied to the target network in order to periodically transfer the learning progress. For each training iteration of the DQN, a random batch of experience replay buffer elements was sampled from the training dataset (with a batch size of $N_b = 32$). The Deep-Q Networks were trained for 200.000 iterations in total. In this context, the loss is calculated by (see Eq. 14):

$$\begin{aligned} L = &\left(\frac{1}{N_b} * \sum_{i=1}^{N_b} (Q(s_{t_i}, a_{c_{t_i}}) - (r_{c_{t_i}} + \gamma * max(Q'(s_{t+1_i}))))^2 \right) + \\ &\left(\frac{1}{N_b} * \sum_{i=1}^{N_b} (Q(s_{t_i}, a_{m_{t_i}}) - (r_{m_{t_i}} + \gamma * max(Q'(s_{t+1_i}))))^2 \right) \end{aligned} \tag{14}$$

where s_{t_i} is the i-th input state for timestep t in the sampled batch; $a_{c_{t_i}}$ is the executed crossover action in timestep t at index i in the sampled batch and $a_{m_{t_i}}$ is the executed mutation action in timestep t at index i in the sampled batch. The first line of the equation corresponds to the mean squared error between the main network prediction of the Q value for the executed crossover action in the input state, and the observed crossover reward added to the discounted max Q value generated by the target network for the consecutive input state. The second line is similar to the left-hand side, but applied to the mutation output head of the DQN.

3.2.3 Distributional Deep-Q Network parameter control method

Each action in the action set corresponds to a probability for crossover or mutation. These probabilistic actions have stochastic behavior in the genetic algorithm environment, as a certain input state might yield different rewards for multiple applications of the same action. Consider the following probability distribution which represents the possible immediate rewards for a certain input state in the genetic algorithm environment for a single action (see Fig. 17).

The DQN implementation would predict a Q value that takes into account an immediate re-

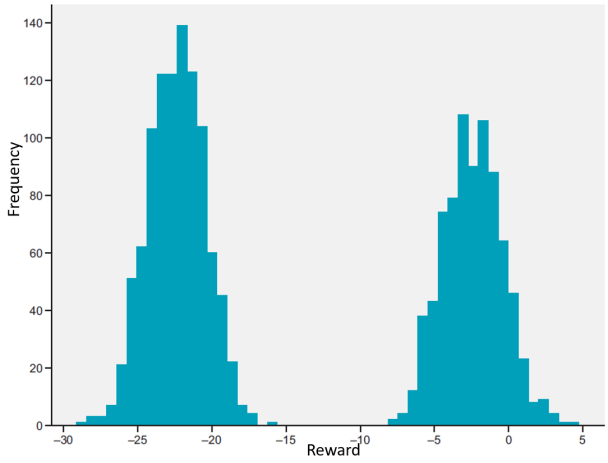


Figure 17: Example reward probability distribution for a hypothetical reinforcement learning agent's action on a genetic algorithm environment. (Adapted from Zai and Brown [38])

ward of approximately −14, which is incorrect for all the occurring rewards in this distribution. A singular expected Q value in decimal form on the output layer of the DQN lacks the resolution to properly represent a stochastic genetic algorithm environment. A probability distribution on the output layer of the reinforcement learning network is required to fully approximate this context.

3.2.3.1 C51; The Distributional DQN

A reinforcement learning agent which alleviates this problem, the *C51 Distributional DQN*, was proposed by Bellemare et al. [39]. This name was based on the changes made to the output layer of the DQN, in which a probability distribution is formed by designating a separate output head with 51 nodes to every action. These 51 outputs correspond to a support vector of the same size which forms a range with a constant interval starting at the minimal observable reward in the environment and ending at the maximal observable reward. The discrete probability distribution is updated by binning the observed reward at the index of the element of the support vector which is closest in value.

The C51 method was successful in picking up on stochasticity in the environments provided by the experiment in its publication and was able to outperform a fully trained DQN in 45 out of the 57 Atari games. Unfortunately, the C51 method requires a predefined support vector and corresponding minimal and maximal observable rewards for a given environment. This poses an obstacle in its application to Despatch, as every workday has a different fitness landscape

and consequently a different maximal observable reward (see Fig. 18).

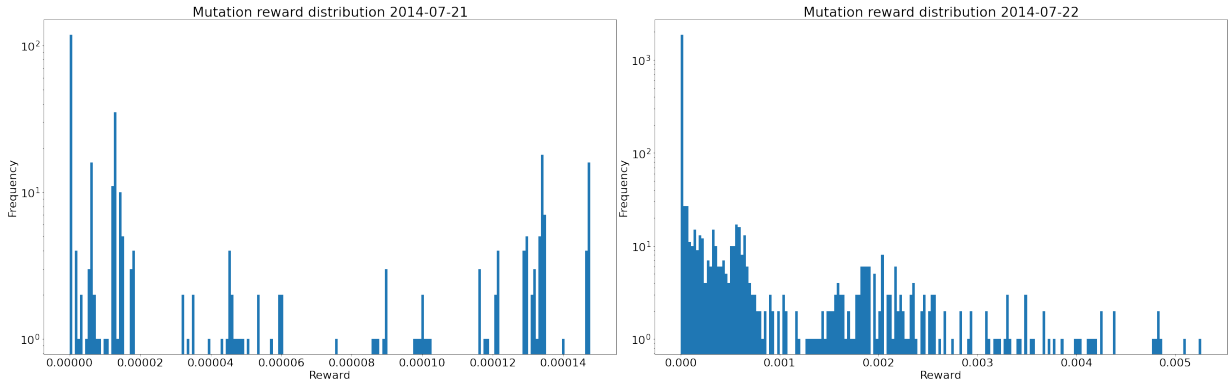


Figure 18: Mutation reward probability distributions for a single run on two example days of the validation dataset; the 21st and 22nd of July 2014. These probability distributions illustrate that each day in the dataset has a different range of observable rewards and consequently should not be represented by a single static support vector by the C51 method.

Not only do the mutation reward probability distributions differ for the two days, they also have a significantly different maximal reward each: 0,00014 and 0,005. By using a single support vector to form a probability distribution for each of these days, a lot of information is lost in representing a corresponding reward probability distribution. The support vector is statically defined ahead of the training procedure, therefore the C51 method is incompatible with the genetic algorithm in this context.

3.2.3.2 Quantile Regression Deep-Q Network

A modified version of the C51 method alleviates the static support issue by using a fixed set of probabilities and letting the reinforcement learning model learn the associated support values. This modified method is called the *Quantile Regression Deep-Q Network* (QRDQN) (Dabney et al. [40]). This name was based on the fact that the fixed probabilities end up representing quantiles of the probability distribution. By representing the quantiles on the output layer, any possible action value can theoretically be generated, abolishing the need for a statically defined minimal and maximal reward. The only hyperparameter which needs to be set beforehand is the number of quantiles the QRDQN will use to approximate the probability distribution ($N_q = 32$). Quantile regression is the unbiased stochastic approximation of the quantile function (Koenker [41]).

The quantile function is equivalent to the inverse cumulative distribution function (CDF). For a random variable X, the CDF $F_X(x)$ gives the probability that $X \leq x$. The quantile function is given by $F_X^{-1}(x)$, which, given a probability p, computes for what value of x the following comparison holds: $X \leq x$. For $N_q = 32$, the first of the output nodes on the designated output head for this action will predict what reward is at least immediately observable after executing this action with probability $p = \frac{1}{32}$.

3.2.3.3 Learning procedure

Similar to the DDQN, the QRDQN implementation uses a main network and a target network to stabilize the learning procedure. After every 100 learning iterations, the weights of the main network are copied to the target network to transfer the learning progress. The loss of the QRDQN method is calculated by using a modified version of the *Huber Loss* (Huber [42]):

$$\mathcal{L}_\kappa(u) = \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq \kappa \\ \kappa(|u| - \frac{1}{2}\kappa), & \text{otherwise} \end{cases} \tag{15}$$

$$u = (r_t + \gamma * QR'(s_{t+1})) - QR(s_t, a_t) \tag{16}$$

where u is the difference between the predicted N_q quantiles according to the main network QR and the observed reward r_t summed with the target network QR' its N_q quantiles for its greedy action selection in s_{t+1} (discounted by γ); where κ is a constant value used to clip gradients. Huber loss is less sensitive to outliers (when $u \leq \kappa$) and more sensitive to small errors. For outliers, the absolute value error is computed, while for small errors, the mean squared error is computed. The learning process computes a batch of u values, the loss is computed element-wise, taking either the MSE or the MAE for each individual u and returning the mean loss for the entire batch. To apply the Huber Loss to the seperate crossover and mutation output heads, the total loss is computed by Eq. 17.

$$L_\kappa(u_c, u_m) = |\tau - \delta_{\{u_c<0\}}| * \mathcal{L}_\kappa(u_c) + |\tau - \delta_{\{u_m<0\}}| * \mathcal{L}_\kappa(u_m), \quad \tau = \left\{ \frac{1}{N_q}, \frac{2}{N_q}, ..., \frac{N_q}{N_q} \right\} \tag{17}$$

where u_c is the difference between the main network QR its crossover prediction and the observed crossover reward summed by the target network QR' its crossover prediction (discounted by γ); where u_m is similar but applied to the mutation reward and predictions; where τ is a list representing the quantile weights for the output layer and where $\delta_{\{u_x<0\}}$ is a vector of equal length to u_x, which contains 1's at the indices where u_x is negative and 0's where it is not.

4 Experiment

The methods described in Section 3 were applied to the dataset specified in Section 4.1 and compared based on the experiment setup outlined in Section 4.2. Finally, the results are compared in Section 4.3.

4.1 Dataset

To train the Deep-Q Networks and assess the performance of all three reinforcement learning parameter control methods, a dataset is required that accurately resembles the workdays which the genetic algorithm backend in Despatch will receive as input. To this end, a dataset was provided by Eneco which covers the workdays for the period of January 30th to the September 2nd for the Princess Amalia Wind Park [43] (PAWP) in 2014. This dataset was also previously used to assess the performance of the Despatch algorithm [1]. The PAWP is located 23 kilometers off the coast of IJmuiden in The Netherlands. The Wind Park was developed by Ecoconcern with the help of investment from utility company Eneco and has been in operation since 2008. This dataset is used to form realistic simulations which act as an environment for the reinforcement learning agents to train on.

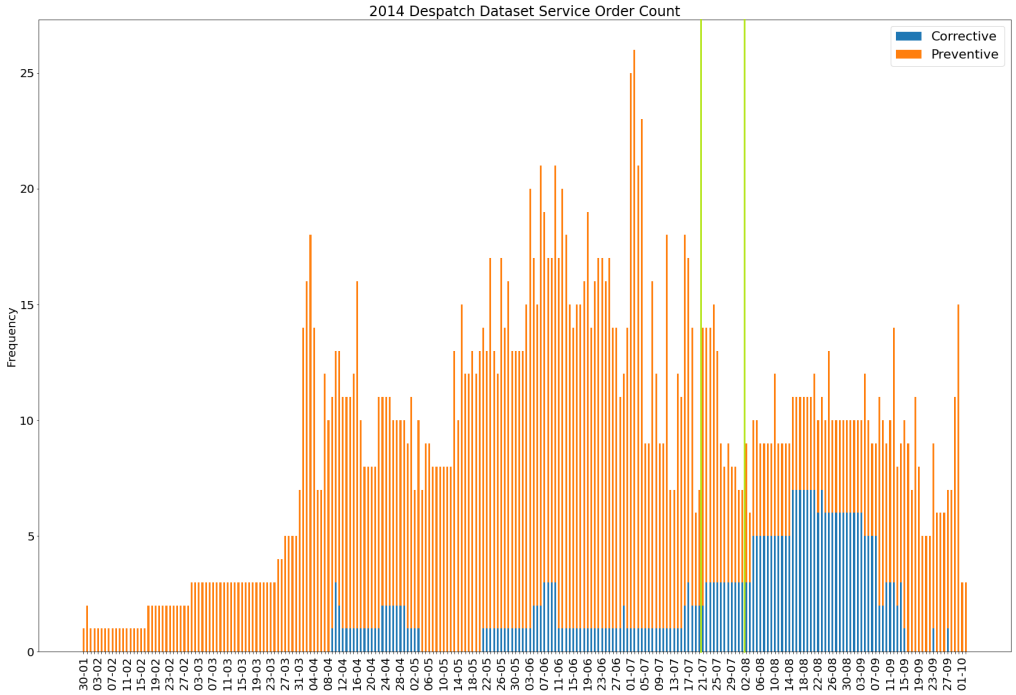
The PAWP uses only one crew transfer vessel to conduct day-to-day maintenance, creating a scenario in which parallel transport is impossible for the technicians. For each of the 246 days in the dataset, a weather forecast is provided for the coming 90 hours based on historical data, to simulate the current workday but also to estimate the cost of delaying Service Orders to the next days. In addition to the weather forecast, Service Orders are assigned to workdays based on if the current date falls between the associated start date and end date. Service Orders are divided into two categories: preventive or corrective. The resulting dataset's daily Service Order list can be expressed in terms of the number of Service Orders (see Fig. 19a) and amount of hours in that category (see Fig. 19b). Note that the amount of corrective Service Order hours per unit is significantly greater than the amount of preventive Service Order hours per unit. In this dataset, corrective Service Orders are often larger in proportion due to them being reactionary to immediate issues, while preventive Service Orders are precautionary maintenance activities and thus often less severe.

Category	**Training set**	**Validation set**	**Test set**
Ratio	70%	5%	25%
Duration (days)	172	12	62
Date range	01-30 to 07-20	07-21 to 08-01	08-02 to 10-02

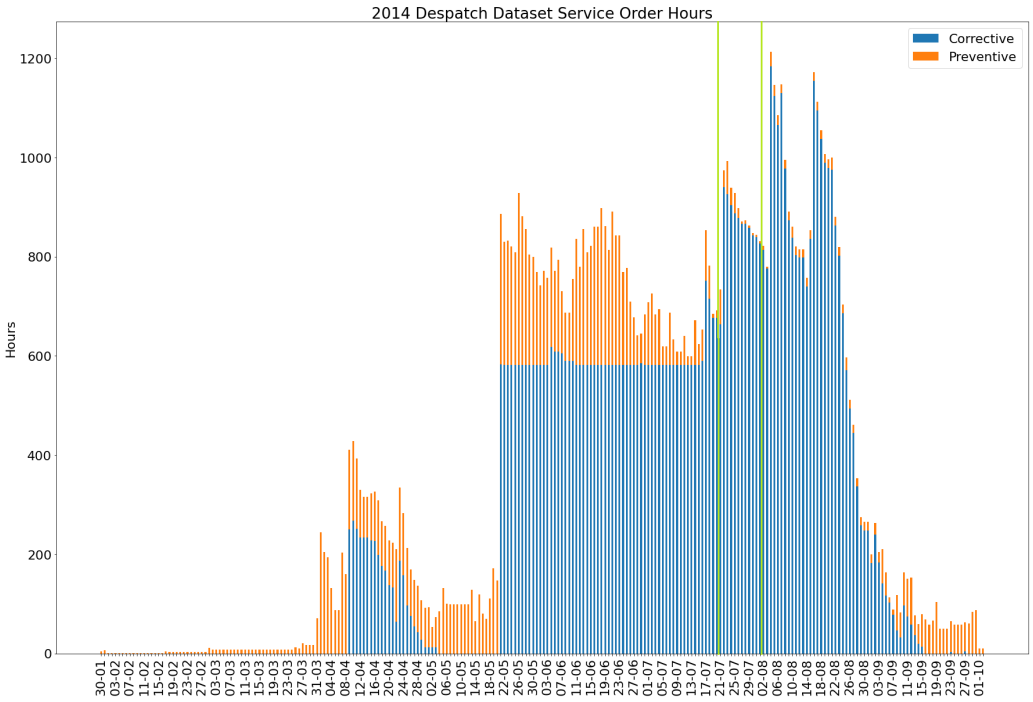
Table 2: Statistics regarding the dataset split of Fig. 19

The used dataset split is visible in Table 2. The first 172 days (70% of the dataset) functions as the training dataset and act as the foundation for the Deep-Q Networks their learning procedures, as explained in Section 3.2.2.3. The following 12 days (5% of the dataset) function as the validation dataset. These days are used to run a model architecture selection tournament for the fully trained Deep-Q Networks, to assess the relative performance of the NB method (see

Section 3.1) beforehand, to generate insightful mutation and crossover reward probability distributions and to generate the (repeated) exponential ε decay schedule (see Fig. 15). The final 62 days (25% of the dataset) function as the test dataset and are used the compare the performance of the original Despatch algorithm, Despatch using the NB operator, and all three reinforcement learning methods combined with the NB operator. Note that the training, validation and test sets are structured differently in terms of amount and duration of corrective/preventive Service Orders. The class imbalance between the training and test set can cause inaccurate initial predictions for the trained Deep-Q Networks. This is countered by unfreezing the neural network weights and allowing for some intitial exploration of the environment to optimize the policy.



(a) The amount of Corrective and Preventive Service Orders across the dataset.



(b) The duration (hours) of Corrective and Preventive Service Orders across the dataset.

Figure 19: Histogram statistics for the 2014 dataset describing the work days for the Princess Amalia Wind Park ranging from the 30th of January to the 2nd of September. Comparing figures (a) and (b) illustrates the relation between the amount of Service Orders and how many work hours are associated with those Service Orders. Generally speaking, corrective Service Orders are more time-consuming relative to preventive Service Orders in this dataset. To the left of the first vertical green line is the the training dataset, between the two green lines is the validation dataset and to the right of the second green line is the test dataset. There is a slight class imbalance between the training and test set, which can negatively affect the performance on the test set.

4.2 Experiment setup

All 5 methods were applied to the test set sequentially, each starting on the 2nd of August and ending after the 2nd of October. Once the method has converged and settled on the best-found Transfer Plan for that day, the Transfer Plan and relevant statistics are saved. Afterwards, the genetic algorithm environment is re-initialized to represent the next consecutive day, after which it once again attempts to find a suitable Transfer Plan. The performance assessments on the test set were all run on a computer cluster owned by TNO, with the random seed set to the same value for each individual day for all 5 methods.

4.2.1 Deep reinforcement learning model selection tournament

Before all 5 methods can be applied to the test set, the network architectures should be determined for both Deep-Q Networks. The input layer size is 16 for both deep reinforcement learning methods, based on the input features listed in Table 1. The output layer size is determined by the respective method and the crossover and mutation action set sizes. The DQN has a singular output for each available action, resulting in an output layer size of 30. The QRDQN has N_q outputs per available action, resulting in an output layer size of 960. The size of the hidden layers is not as easily determined and should be treated like a tunable hyperparameter as it significantly impacts the model's accuracy and time complexity [44]. The setting for this hyperparameter is determined by a model selection tournament. The preference is given to running a model selection tournament for each day twice over averaging the performance over all days due to the lengthy associated computation time.

After training all featured models for 200.000 iterations, they were applied to the first four days of the validation dataset twice. For the DQN implementation, one to four uniform hidden layers were tested with 16, 32, 64, 128, and 256 hidden nodes per layer. For the QRDQN implementation, one to four uniform hidden layers were tested with 64, 128, 256, and 512 hidden nodes per layer. The performance of the models was measured in terms of energy loss and in computation time (ms). Of the two runs, the average was taken to give a more reliable insight. After each day of the validation dataset, half of the best-performing models proceed to the next day. The first objective is to achieve the relatively lowest energy loss, in case of a tie, the shortest computation time is used to determine the winner.

The results for the tournament are visible in Fig. 20, note that all models who proceed to the next day are displayed in bold. For the 21st of July, all models were able to converge to the global optimum and as a result, the winners for that day were determined by the shortest computation time. The tournament winner for the DQN implementation was the model containing two hidden layers of 64 hidden nodes. In contrast, the tournament winner for the QRDQN implementation was the model containing four hidden layers of 256 hidden nodes. The greater amount of hidden layers and hidden nodes for the QRDQN model represents the need to compute the relatively more complex output layer.

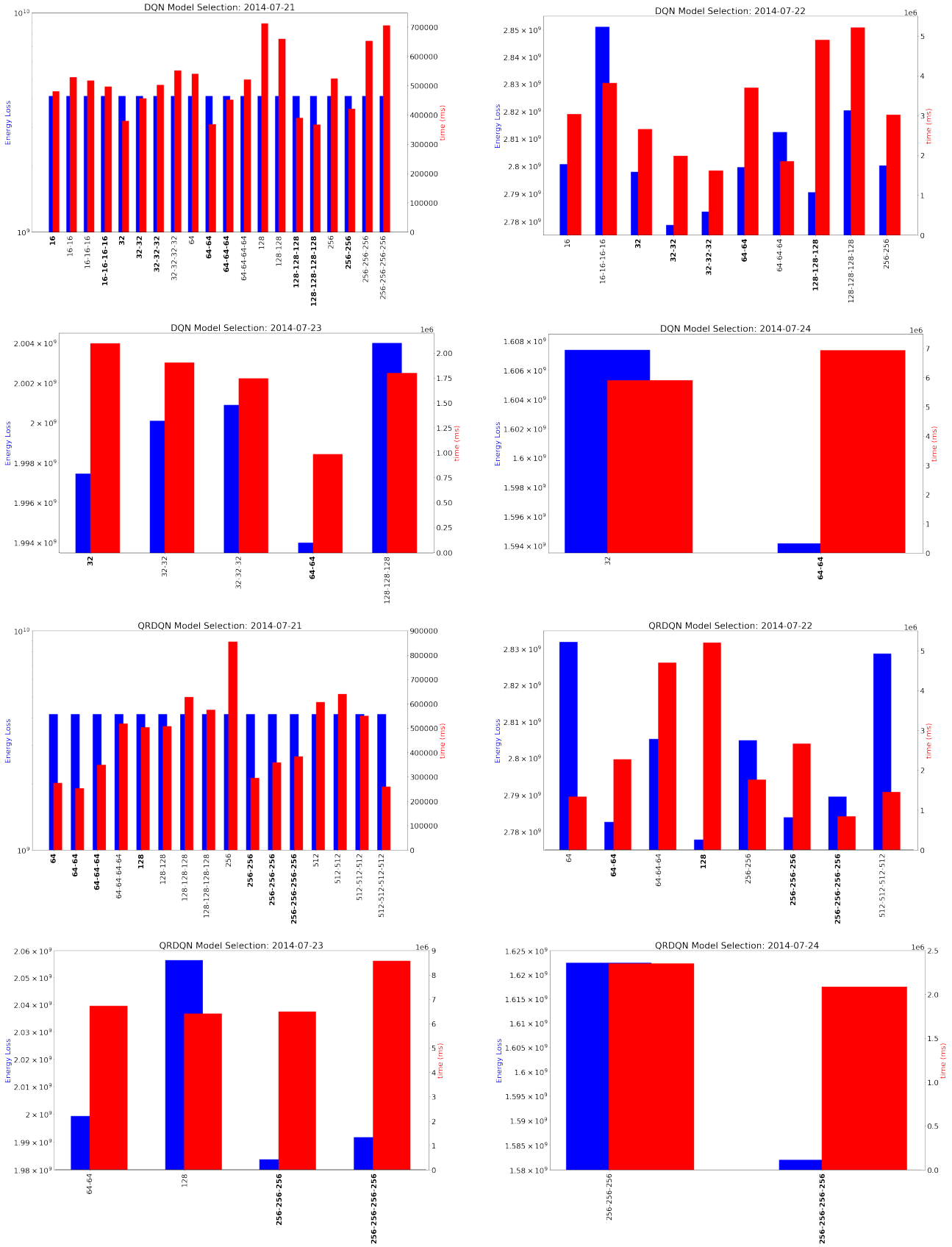


Figure 20: Model selection tournament for the DQN and QRDQN models, all featured models were applied twice to that day of the validation dataset. The red bar plots correspond to the computation time for that model, while the blue bar plots correspond to the energy loss performance for that model. The first four bar plots illustrate that two hidden layers of 64 nodes were selected as the DQN architecture. The last four bar plots illustrate that four hidden layers of 256 nodes were selected as the QRDQN architecture.

4.2.2 Training phase

As each day is subject to different work day features such as wind speed and amount of Service Order hours, each day in the dataset is associated with a different fitness landscape from the perspective of the genetic algorithm. These different fitness landscapes represent distinct problem instances for the reinforcement learning agents to learn. The training phase for the Deep-Q Networks aims to learn the different fitness landscapes based on the input features for each day.

The loss of the Deep-Q Networks is pipelined to the Adam optimizer (Kingma and Ba [45]), which iteratively adjusts the weights of the main network to minimize the loss function. For the two selected models, the progression of the respective loss functions during the training phase gives insight into the complexity of the problem the model is attempting to learn (see Fig. 21).

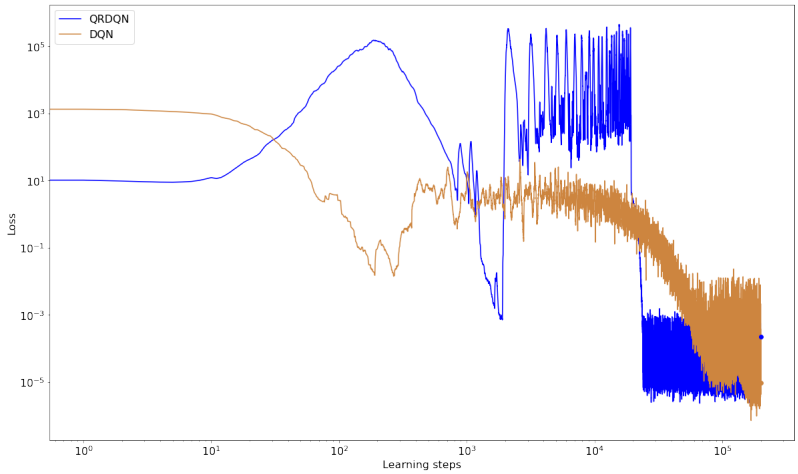


Figure 21: Loss function output during the 200.000 iterations long training phase for the DQN and QRDQN models (with moving average smoothing using a window size of 20). This graph illustrates that both the DQN and the QRDQN methods converge to a loss of nearly 0, even though the QRDQN experiences some forgetting around 10^4 iterations.

Note that both the x- and y-axis are plotted using a logarithmic scale to highlight the significant changes in loss during the early phases of training. While experiencing some fluctuation after around 10^3 training iterations, the DQN successfully and consistently converges with respect to the loss function. The QRDQN seemingly successfully learns the problem after around 10^3 training iterations but performs worse after around 10^4 training iterations. The erasure of a successful representation is often referred to as 'catastrophic forgetting' [46] and is caused by learning new input patterns which overwrite existing patterns. After 10^5 training iterations, the QRDQN consistently forms a successful representation and converges with respect to the loss function.

4.2.3 Hyperparameter settings

All featured reinforcement learning methods utilize a γ value, which determines the present value of future rewards. If $\gamma = 0$, the reinforcement learning agent becomes nearsighted, as it is only concerned about maximizing the immediate rewards. If $\gamma = 1$, the reinforcement learning agent becomes farsighted, as it is only concerned with future rewards. In the context of this work, γ was set to 0,98, highly valuing future rewards. This was done to motivate the parameter

control methods to assist the genetic algorithm in converging towards a global optimum over a local optimum, requiring heavy consideration of future rewards.
The α value represents the learning rate, which determines the ratio to which new information is accepted relative to old information. As the Adam optimizer's default learning rate is set to 0,001, this is applied here as well.

4.3 Results

We compare our results to the original Despatch algorithm with respect to three performance measurements:

- **Total computation time**: The total cumulative time taken to converge to the respective solutions for the test dataset.
- **Energy loss sum**: The sum of the total *energy loss* across the entire test dataset. This metric gives insight into the quality of the solutions found by each respective method since the energy loss was chosen as the objective function to minimize.
- **Generations sum**: The cumulative number of generations that the genetic algorithm took to converge to the respective solutions for the test dataset.

Although there is a correlation between the total computation time and the generations sum, the time needed to compute matrix operations for various neural network architectures prohibits a linear relationship between the two metrics. The results are visible in Table 3.

Model	**Time sum (hours)**		**EnergyLoss sum (kWh)**		**Generations sum**
Original	43,44	[0,0, 4,36]	$12{,}326 * 10^{10}$	[1,0]	133639
NB	21,03	[0,0, 1,46]	$12{,}329 * 10^{10}$	[1,0002]	97217
Q-Table + NB	2,44	[0,0, 0,13]	$12{,}985 * 10^{10}$	[1,05]	23599
DQN + NB	13,18	[0,0, 1,13]	$12{,}329 * 10^{10}$	[1,0002]	59732
QRDQN + NB	3,11	[0,0, 0,27]	$13{,}110 * 10^{10}$	[1,06]	23128

Table 3: Results of the featured models on the test set in terms of computation time, energy loss and amount of generations. The minimal and maximal runtime per day are also provided in the time sum column. The 'Original', 'NB' and 'DQN + NB' models were run twice to make the resuls more informative.

The original Despatch algorithm slightly outperforms the other methods with respect to the objective function but takes a significantly longer time to do so. The Despatch algorithm was able 43,44 hours, the addition of the NB operator speeds up the computation time to 21,03 hours while decreasing the cumulative solution quality by only 0,02%. The Q-Table method is very fast in comparison but reaches significantly lesser quality solutions. The same holds for the QRDQN method. The DQN method shortens the computation time to 13,18 hours relative to the NB method while reaching the same quality solutions.

Fig. 22 shows a visual comparison, illustrating that the original Despatch algorithm, the NB method, and the DQN method consistently find nearly equivalent solutions in terms of energy loss. While Fig. 23 shows a visual comparison between all methods in terms of cumulative computation time. Although the Q-Table and QRDQN methods overall converge the fastest, the convergence to lesser quality solutions makes these methods inferior to the other three.

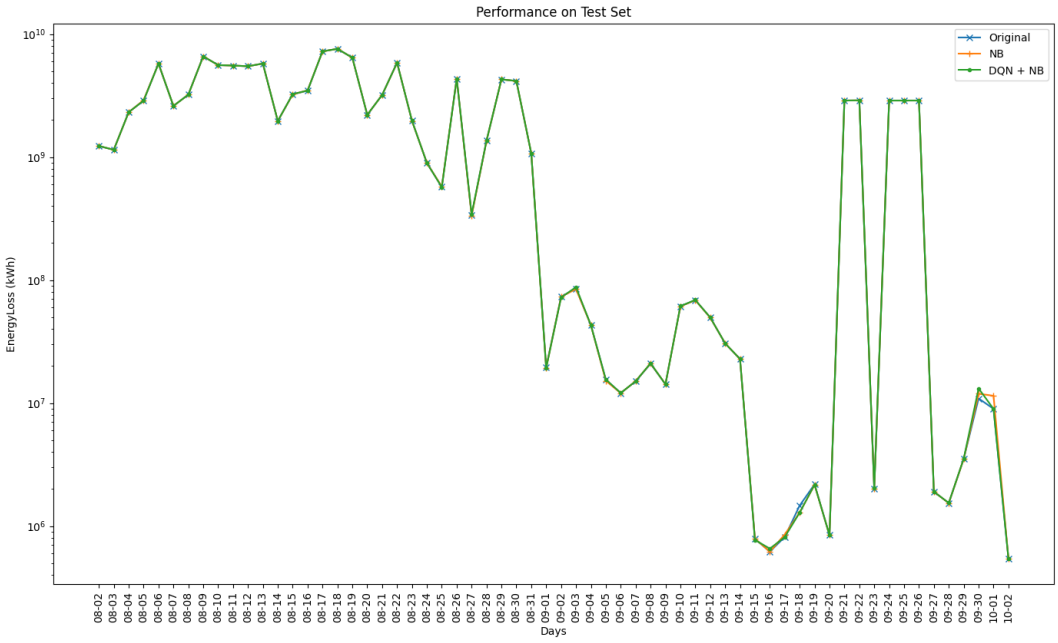


Figure 22: Visual cumulative energy loss comparison between the original Despatch algorithm, the NB method and the DQN + NB method, all applied sequentially to the 62 days in the test dataset. Each data point represents the end performance which was found by that method upon convergence. The y-axis was logarithmically scaled to encapsulate all end performance results despite their major differences. This graph illustrates that these three methods converge to nearly equivalent quality solutions across the entire test set.

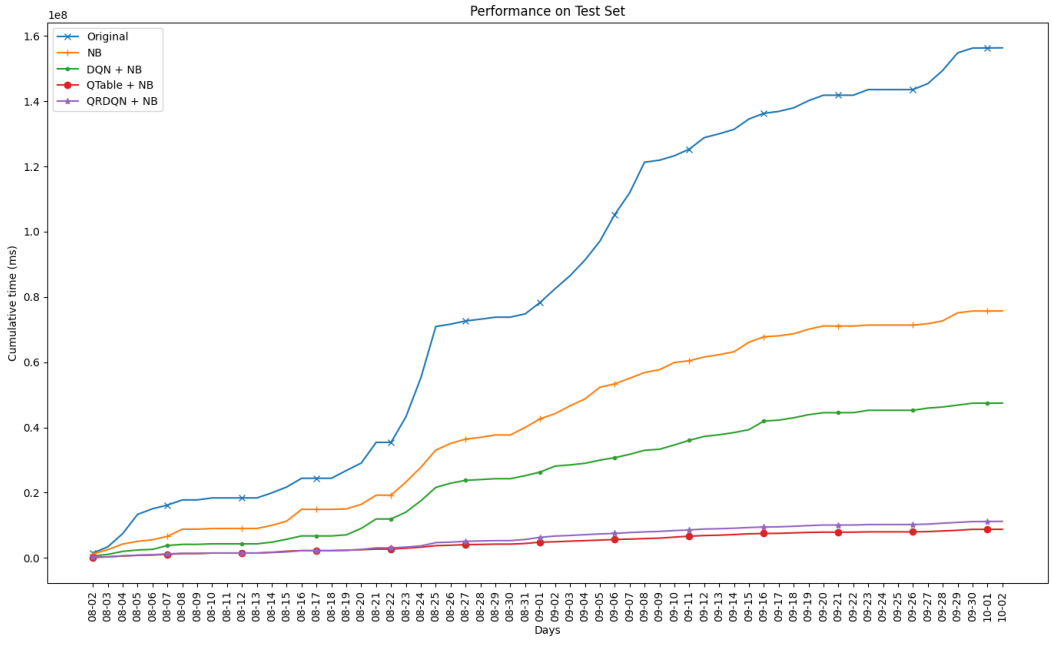


Figure 23: Visual cumulative computation time comparison between all 5 methods applied sequentially to the 62 days of the test dataset. The computation time monitoring for a single day starts when the target population size has been reached and ends when the method converges to a solution. The average computation time was taken for 2 runs of the Original, NB and DQN + NB methods, while the other 2 methods were run only once. This graph illustrates that the DQN + NB method consistently outspeeds the NB method, which in turn consistently outspeeds the original Despatch algorithm.

5 Discussion

The Despatch algorithm's computation time adds up to 43,44 hours across the entire test set. The average runtime per day was 0,70 hours, with the maximal runtime per day being 4,36 hours. The minimal runtime per day for all methods was 0,0 hours, as it is possible for a day to be solved during the population initialization as there is only one suitable Transfer Plan for that workday. While the NB operator was able to improve the cumulative computation time to 21,03 hours, the average runtime per day to 0,34 hours and the maximal runtime per day to 1,46 hours, illustrating the redundant time spent in the final convergence stage by the Despatch algorithm. The DQN agent was able to further improve the cumulative computation time to 13,18 hours, the average runtime per day to 0,21 hours and the maximal runtime per day to 1,13 hours, demonstrating the benefit of an adaptive parameter control method for the genetic algorithm.

First, we showed that the addition of the NB operator mitigated the redundant time spent fully converging by utilizing a threshold in which the population consists of the best-found individual for at least 70%. If this threshold was passed, the newly generated individual would be compared to the current worst individual by the re-insertion operator, instead of randomly selecting individuals from the population to compare to. This adjustment achieved its speedup at the cost of a very minor decrease in cumulative solution quality, which is the result of a rare scenario in which the genetic algorithm passes this threshold prior to eventually finding better solutions, causing early convergence.

The faster convergence of the genetic algorithm is valuable despite the mild decrease in cumulative solution quality as the wind farm managers use Despatch to generate a Transfer Plan for that day in the morning. This is done in order to use the most recent weather forecast and to address possible events occurring on the wind farm overnight. This creates a scenario in which the genetic algorithm is given only a short time budget, during which a faster computation time results in a superior solution given that the original Despatch algoritm did not fully converge. Although the NB operator solves the bottleneck of the original Despatch algorithm, it does not influence the gradient of the decrease in overall fitness of the population and simply causes a better Despatch user experience. Intermediate solutions are expected to have nearly equivalent quality between the NB method and the original Despatch algorithm. The DQN parameter control method also utilizes the NB operator but achieves an even faster convergence speed relative to the NB method. This indicates that the reinforcement learning agent causes superior intermediate solutions due to the steeper gradient of the decrease in overall fitness of the population.

Second, we see that Q-Table reinforcement learning agent performs relatively poorly as it is limited by its manual discretized state set definition. Consequently, the agent will greedily select one promising action until enough progress is made to progress to the next state. Often, this will result in one action being selected repeatedly for long periods of time even though it might be suboptimal for the genetic algorithm.

Furthermore, the QRDQN relatively performs the worst out of all the applied methods. Although it is difficult to analyze the behavior of neural networks, the convergence of the loss function during its training phase indicates that it was moderately successful in learning the

characteristics of the environment. A possible explanation for its poor performance would be that it exploited the imperfect reward functions due to its complex output layer and greedily decreased the population's diversity, causing early convergence.

Fortunately, the DQN agent achieves an equivalent cumulative solution quality relative to the NB addition while also causing a significant speedup. This equivalent cumulative solution quality indicates that the DQN does not introduce an additional decrease in performance. In contrast to the QRDQN, the DQN improves upon the convergence speed of the genetic algorithm. A singular expected return value for each action allowed for an expressive policy without greedily decreasing diversity.

The improvement that the DQN agent has brought to the Despatch algorithm illustrates that a deep reinforcement learning agent can successfully be applied as a dynamic parameter control method for a genetic algorithm by speeding up its converge process.

Although the final performance assessment was based on a relatively small time period (62 days) for a single wind farm, this research offers a valuable insight into the potential benefits in improving the convergence of genetic algorithms by applying a reinforcement learning agent as a parameter control method. Ideally, this DQN parameter control method should be trained and tested on a wider range of wind farms as the Princess Amalia Wind Park is dated and relatively small, suggesting that more recent wind farms could experience an even greater payback.

6 Conclusion and future work

In the interest of solving the NP-hard problem of scheduling daily service operations for offshore wind farms, a genetic algorithm was proposed by Stock-Williams and Swamy [1]. Given that the performance of genetic algorithms greatly depends on its parameter setting, the statically defined parameter configuration utilized in this approach allows for potential improvements. In this work, we investigate to what extent the addition of a (deep) reinforcement learning agent as a parameter control method improves this approach in terms of computation time and quality.

A review of the literature on this subject has shown that previous reinforcement learning agents acting as parameter control methods were tabular structured, posing constraints on the input state set. Additionally, to the knowledge of the authors, no (deep) reinforcement learning parameter control methods for genetic algorithms were previously implemented in the context of solving daily maintenance planning for offshore wind farms.

Three reinforcement learning agent structures were implemented and compared in terms of performance and computation time. Firstly, a Q-Table agent based on Chen et al. [13]. This method introduced several constraints in order to apply it to the genetic algorithm in this context, including the discretization of the input state and the introduction of a repeated learning phase during the reinforcement learning agent's application to the environment. Secondly, a Deep-Q Network agent which allowed for a continuous input state, abolished the need for a repeated learning phase during runtime and, due to a prior learning phase, mitigated the need for an extensive amount of explorative actions. Finally, a Quantile Regression Deep-Q Network agent which allowed for the more complex output layer relative to the Deep-Q Network to represent a probability distribution that can more accurately approximate the stochastic environment of the genetic algorithm. Given that all the applied reinforcement learning methods were value-based, a limitation of the scope of this work was that no policy-based method was implemented, which could theoretically utilize the complete continuous range of the probabilities used in the actions of the agent in this context. Value-based methods lack this characteristic as they require discretization of the action space.

In combination with an improved re-insertion operator, the addition of a deep reinforcement learning agent, specifically using a Deep-Q Network, as a parameter control method for the genetic algorithm was able to achieve a speedup from 43,44 hours to 13,18 hours. This speedup came at the cost of a mild decrease (0,02%) in cumulative energy loss across the test set.

Future work

The reward function definitions in Section 3.2.1.3 assume that the crossover operator is exclusively responsible for the improvement in the best fitness of the population and that the mutation operator is exclusively responsible for the improvement in the total fitness of the population. In reality, both operators have the ability to affect both metrics and as a result, these reward functions are flawed. The genetic algorithm's cycle computes the fitness function after the crossover

and mutation operators have been applied sequentially, making it difficult to distinguish the independent effect of both operators. Still, better reward function definitions should be explored in further work.

All three reinforcement learning methods that were implemented in this work are value-based agents, which generally perform well on discrete action spaces. Another line of work implements policy-based reinforcement learning agents, which allow for the use of continuous action spaces. This methodology would be interesting to apply in this context as it removes the need for the discrete action spaces as defined in Section 3.2.1.2 and might improve upon the achieved results as it leads to a more expressive reinforcement learning policy. Examples of such methods which could prove a good fit in this context include Proximal Policy Optimization algorithms (Schulman et al. [31]) and Soft Actor-Critic algorithms (Haarnoja et al. [47]).

The extended feature set (see Table 1) was manually constructed in accordance with an Aerodynamics employee working for TNO, but it is far from optimal. For instance, the future weather forecast, the number of turbines to be visited on the workday and the amount of corrective/preventive Service Orders are underrepresented in the feature set. Further extensions of this feature set could lead to better results.

Although the DQN agent showed positive results, the Q-Table and QRDQN agents showed negative results. The Q-Table implementation showed a less expressive policy due to the discrete input state set but the behavior of the Deep-Q Networks is more difficult to generalize. Further research is required to analyze the policies applied by the DQN and the QRDQN in this context.

References

[1] Clym Stock-Williams and Siddharth Krishna Swamy. Automated daily maintenance planning for offshore wind farms. *Renewable Energy*, pages 1393–1403, 2019. doi: https://doi.org/10.1016/j.renene.2018.08.112.

[2] Manish Shakdwipee, Indu Pillai, and Rangan Banerjee. Sustainability analysis of renewables for climate change mitigation. *Energy for Sustainable Development*, pages 25–36, 12 2006. doi: 10.1016/S0973-0826(08)60553-0.

[3] United Nations. Sustainable development goals (sdgs). `https://www.un.org/sustainabledevelopment/sustainable-development-goals/`, 2022.

[4] Wind Europe. Offshore wind in europe – key trends and statistics 2020. `https://windeurope.org/intelligence-platform/product/offshore-wind-in-europe-key-trends-and-statistics-2020/#findings`, 2020.

[5] Rijksoverheid. Windenergie op zee. `https://www.rijksoverheid.nl/onderwerpen/duurzame-energie/windenergie-op-zee`, 2021.

[6] Malte Jansen, Iain Staffell, Lena Kitzing, Sylvain Quoilin, Edwin Wiggelinkhuizen, B. Bulder, Iegor Riepin, and Felix Müsgens. Offshore wind competitiveness in mature markets without subsidy. *Nature Energy*, pages 1–9, 2020. doi: 10.1038/s41560-020-0661-2.

[7] Gavin Smart, Aaron Smith, Ethan Warner, Iver Bakken Sperstad, Bob Prinsen, and Roberto Lacal-Arantegui. Iea wind task 26: offshore wind farm baseline documentation. Technical report, National Renewable Energy Lab.(NREL), Golden, CO (United States), 2016.

[8] Ujjwal Maulik and Sanghamitra Bandyopadhyay. Genetic algorithm-based clustering technique. *Pattern recognition*, pages 1455–1465, 2000. doi: 10.1016/S0031-3203(99)00137-5.

[9] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, pages 124–141, 1999. doi: 10.1109/4235.771166.

[10] Giorgos Karafotias, Mark Hoogendoorn, and A. E. Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, pages 167–187, 2015. doi: 10.1109/TEVC.2014.2308294.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[12] Jess Whittlestone, Kai Arulkumaran, and Matthew Crosby. The societal implications of deep reinforcement learning. *J. Artif. Int. Res.*, page 1003–1030, 2021. doi: 10.1613/jair.1.12360.

[13] Ronghua Chen, Bo Yang, Shi Li, and Shilong Wang. A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem. *Computers Industrial Engineering*, page 106778, 2020. doi: 10.1016/j.cie.2020.106778.

[14] M.B. Bashir and A. Nadeem. Improved genetic algorithm to reduce mutation testing cost. *IEEE Access*, pages 3657–3674, 2017. doi: 10.1109/ACCESS.2017.2678200.

[15] TNO. Tno uwise: Cutting-edge simulation software tools for offshore energy operations. `https://www.tno.nl/en/focus-areas/energy-transition/roadmaps/renewable-electricity/wind-energy/innovative-logistics/simulation-software-tools/`, 2018.

[16] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, pages 393–410, 1954.

[17] Michael T Goodrich and Roberto Tamassia. Algorithm design and applications. chapter 10.1, pages 286–288. Wiley Hoboken, 2015.

[18] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., first edition, 1989. ISBN 0201157675.

[19] Jiaping Yang and Chee Kiong Soh. Structural optimization by genetic algorithms with tournament selection. *Journal of Computing in Civil Engineering*, pages 195–200, 1997. doi: 10.1061/(ASCE)0887-3801(1997)11:3(195).

[20] Przemysław Ignaciuk and Łukasz Wieczorek. Continuous genetic algorithms in the optimization of logistic networks: Applicability assessment and tuning. *Applied Sciences*, 2020. doi: 10.3390/app10217851.

[21] Gilbert Syswerda. Uniform crossover in genetic algorithms. 1989.

[22] Riccardo Poli and William B Langdon. On the search properties of different crossover operators in genetic programming. *Genetic Programming*, pages 293–301, 1998.

[23] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, pages 82–117, 2013. doi: 10.1016/j.ins.2013.02.041.

[24] Maryam Karimi-Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, pages 393–422, 2022. doi: https://doi.org/10.1016/j.ejor.2021.04.032.

[25] A. Aleti, I. Moser, I. Meedeniya, and L. Grunske. Choosing the appropriate forecasting model for predictive parameter control. *Evolutionary computation*, pages 319–349, 2014. doi: 10.1162/EVCO_a_00113.

[26] Shing Wa Leung, Shiu Yin Yuen, and Chi Kin Chow. Parameter control system of evolutionary algorithm that is aided by the entire search history. *Applied Soft Computing*, pages 3063–3078, 2012. doi: https://doi.org/10.1016/j.asoc.2012.05.008.

[27] A. E. Eiben, Mark Horvath, Wojtek Kowalczyk, and Martijn C. Schut. Reinforcement learning for online control of evolutionary algorithms. In Sven A. Brueckner, Salima Hassas, Márk Jelasity, and Daniel Yamins, editors, *Engineering Self-Organising Systems*, pages 151–160. Springer Berlin Heidelberg, 2007.

[28] Giorgos Karafotias, Agoston Endre Eiben, and Mark Hoogendoorn. Generic parameter control with reinforcement learning. page 1319–1326. Association for Computing Machinery, 2014. ISBN 9781450326629. doi: 10.1145/2576768.2598360.

[29] Arina Buzdalova, Vladislav Kononov, and Maxim Buzdalov. Selecting evolutionary operators using reinforcement learning: Initial explorations. page 1033–1036. Association for Computing Machinery, 2014. ISBN 9781450328814. doi: 10.1145/2598394.2605681.

[30] Yoshitaka Sakurai, Kouhei Takada, Takashi Kawabe, and Setsuo Tsuruta. 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*, pages 74–79, 2010. doi: 10.1109/SITIS.2010.22.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[32] Jiahu Qin, Man Li, Yang Shi, Qichao Ma, and Wei Xing Zheng. Optimal synchronization control of multiagent systems with input saturation via off-policy reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, pages 85–96, 2019. doi: 10.1109/TNNLS.2018.2832025.

[33] Aske Plaat. Deep reinforcement learning, 2022.

[34] Jamal Shahrabi, Mohammad Amin Adibi, and Masoud Mahootchi. A reinforcement learning approach to parameter estimation in dynamic job shop scheduling. *Computers Industrial Engineering*, pages 75–82, 2017. doi: 10.1016/j.cie.2017.05.026.

[35] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, pages 279–292, 1992. doi: 10.1007/BF00992698.

[36] Yin-Hao Wang, Tzuu-Hseng S Li, and Chih-Jui Lin. Backward q-learning: The combination of sarsa algorithm and q-learning. *Engineering Applications of Artificial Intelligence*, pages 2184–2193, 2013. doi: 10.1016/j.engappai.2013.06.016.

[37] 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.

[38] Alexander Zai and Brandon Brown. *Deep reinforcement learning in action*. Manning Publications, 2020.

[39] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458, 2017.

[40] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017.

[41] Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005. doi: 10.1017/CBO9780511754098.

[42] Peter J. Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, pages 73–101, 1964.

[43] Power Technology. The princess amalia offshore wind farm project, netherlands. `https://www.power-technology.com/projects/princess-amalia/`, 2021.

[44] Muhammad Uzair and Noreen Jamil. Effects of hidden layers on the efficiency of neural networks. In *2020 IEEE 23rd International Multitopic Conference (INMIC)*, pages 1–6, 2020. doi: 10.1109/INMIC50486.2020.9318195.

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

[46] Robert M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, pages 128–135, 1999. doi: 10.1016/S1364-6613(99)01294-2.

[47] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications, 2019.