Report

Neuroevolution

Index

List of Abbreviations

1 Introduction

Over the last couple of years, Deep Neural Networks (DNN) have drawn great attention in several fields of research. Due to increased data availability combined with increasingly inexpensive computing capacities, DNN have been applied to many research topics such as image processing, natural language processing or decision making. However, DNN are driven by an enormous amount of parameters that must be defined prior to their training process and decide on success or failure. Since the parameter setting often relies on the designers expertise and thus, is error prone, neuroevolution approaches have gained momentum. Neuroevolution describes the process of automated neural network optimization and and construction by combining DNN's with Evolutionary Algorithms (EAs) so that EAs are used to adjust the network's design during its training, or to optimize the network's parameters. In contrast to classical network training, neuroevolution enables the network optimization independent of any gradient information. Deep Reinforcement Learning (DRL) is a rather new approach for optimal control problems and decision making under uncertainty. It combines Reinforcement Learning (RL) with DNN to approximate a decision policy for a decision maker. DRL approaches can be applied for e.g. automated robot control, but also for economic decision making such as stochastic Order Acceptance and Allocation Problem (OAAP). This work indicates how the neuroevolution approach can be applied in DRL and if it is competitive to other DRL methods such as Deep Q Learning (DQL). Therefore, the basics of EAs and DNN are introduced in Chapter 2, as well as neuroevolution approaches to train DNN and to automate the network's design. Chapter 3 introduces the concept of RL and illustrates DQL as one state of the art DRL method. Additionally, it is clarified, how EAs can be applied in DRL. Chapter 4 examines the performance of a DQL agent compared to a DRL agent that is trained with a neuroevolution approach. The agents are evaluated on the example of a stochastic OAAP. Lastly, a proposition about the suitability of neuroevolution in DRL is made, as well as a prospect about possible optimizations of the evaluated agents.

2 Evolutionary Algorithms in Neural Networks

Deep Learning algorithms are a subset of Machine Learning (ML) algorithms and thus, provide a subset of artificial intelligence. ML algorithms constitute methods, which are able to recognize correlations in data after completing a learning process. (Welsch et al. (2018)) Deep learning describes the training of neural networks with a large number of layers and parameters. Finding the right set of parameters is challenging, since DNN possess a large number of parameters that are independent of each other. However, every single parameter influences the training result of the networks. Critical parameters for DNN are e.g the network architecture, the optimization method, activation functions, number of neurons or learning rates. The determination of all parameters often relies on the designers expertise and thus, is error prone. Further, the setting of parameters influences the computational effort for the network's training. The dominant method for training a DNN is backpropagation, which optimizes the network's parameters based on the gradient of the error, produced by the network (Stanley et al. (2019)). Recently, EAs are gaining momentum as a computationally feasible method for training networks (Galván and Mooney (2021)). At this juncture, EAs can not only optimize the networks internal parameters, moreover they provide a possibility to automate the network construction. The remainder of this section is organized as followed. First, the basics of EAs and DNN are presented. Afterwards, the backpropagation approach is presented, together with the evolutionary approach for optimizing the network. Lastly, the evolutionary approach for the construction of DNN is presented.

2.1 Evolutionary Algorithms

EAs describe to a set of stochastic optimization algorithms that are based on evolutionary principles. The algorithms feature a population of $\mu-$encoded potential solutions to a particular problem, whereat each potential solution, referred as individual, constitutes exactly one instance of the solution space. (Galván and Mooney (2021)) EAs utilize simulated evolution to explore possible solutions to a particular problem. The algorithms are based on the principle

of the survival of the fittest which means, the best individuals are carried into an evolutionary step to create new a generation of individuals. The generations change dynamically over time in order to search the solution space. The algorithm's parameters under change are referred as genes and a set of genes forms a chromosome. The objective of EAs is the identification of the best possible combination of genes and chromosomes to solve a particular problem. (Vikhar (2016)) The population is evolved by a genetic operator in order to optimize the solution search and find an optimal solution. Therefor, each individual is evaluated with a so called fitness function, which indicates the suitability of a problem's solution. At this juncture, better solutions achieve higher scores in maximization problems and lower scores in minimization problems. The evolutionary process is driven by genetic operators that select individuals for reproduction and base a new generation upon them. In most EAs, the genetic operators include selection, crossover and mutation. The selection operator chooses individuals based on their fitness values. The crossover operator recombines normally two individuals to form a new one and thus, exploits the search space. The mutation operator introduces diversity by performing random changes to only one individual at a time. By doing so, the mutation operator explores the search space. (Galván and Mooney (2021)) The algorithms start with the creation of an initial population with random individuals. Afterwards, an iterative approach evolves this initial population (Figure 1). The individuals are evaluated by computing objective values. Based on these objective values, the fitness of each individual is determined by applying the fitness function. Next, the fittest individuals are selected for reproduction, at which mutation and/or crossover operations are performed. (Vikhar (2016)) These steps are repeated until a condition is met. The termination criterion is required since EAs do not guarantee to reach an optimal solution and would be executed endlessly without. The final population contains the best evolved potential solutions for a problem and is a result of the exploration and exploitation of the search space. (Galván and Mooney (2021)) EAs comprise four main paradigms: (Vikhar (2016)): Evolutionary Strategies (ES), Evolutionary Programming (EP), Genetic Algorithms (GA) and Genetic Programming (GP), whereat GA are often applied for ML problems.

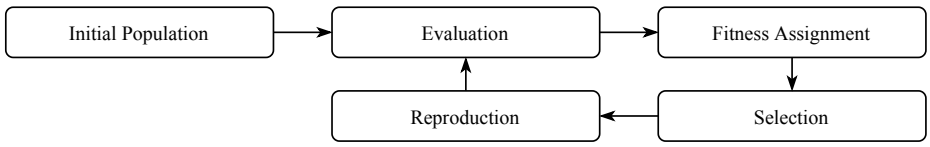

Figure 1 – Overview genetic algorithm, based on Vikhar (2016)

Within the ES approach, a number of single parents are mutated through random changes in order to produce an offspring. Afterwards, the offspring and the parents are evaluated against each other and the best performer is selected to form a new parent. (Coello (2005)) Additionally, an optional crossover operation can be applied. Depending on the number of offspring, the ES algorithms are divided into $(\mu, \lambda)-$ES and $(\mu + \lambda)-$ES where μ denotes the number of parents and λ defines the number of offspring. In the former, the offspring replace their parents and in the latter, both the parents and the offspring are considered to form the new generation.(Galván and Mooney (2021))

EP emphasizes the behavioral link between parents and offspring and is rather similar to the ES approach. Accordingly to ES, a population of individuals is mutated to generate offspring. However, the main difference between the two approaches is that several types of mutation operators are utilized and crossover operations are not possible. Furthermore, the number of offspring for each parent is limited to exactly one and a probability determines whether the parent is respected for the selection step, or not. (Coello (2005))

GA uses crossover operations as main genetic operator and adopt mutation as secondary operator. As in EP, the GA approach is based on a probabilistic selection. The individuals are represented as a set of chromosomes that capture all possible solutions to a particular problem There are several ways to perform the crossover, e.g. a single-point crossover at which exactly one, randomly chosen crossover point is used to merge the parents, or a uniform crossover, at which a probability distribution decides which parent provides which of its genes for the next generation (Figure 2). Additionally , mutation operations can be performed on individual genes that change e.g. in a binary case a gene with value 0 to value 1 and vice versa. (Coello (2005))

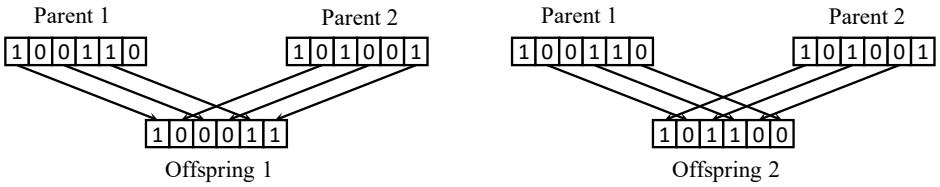

Figure 2 – Uniform crossover with 50 percent probability, based on Coello (2005)

GP is a subclass of GAs at which individuals are randomly created through functional and terminal sets. These algorithms are driven by tree encoding where terminals denote variables and functions represent e.g. boolean operations, arithmetic operations or conditionals. Each vertex of the encoded tree defines either a function or a terminal. Crossover operations are enabled by numbering the tree nodes of two parents and randomly selecting a point in each of the parent's tree. This point determines where the two parents are combined to create offspring, such that the offspring consists of the vertices above this point from one parent and the vertices below this point from the other parent. Each crossover results in two offspring. GP also allows mutation operations at which a vertex and the subtree below is replaced by a randomly generated tree.

2.2 Deep Neural Networks

DNN are information processing systems that are simulated in computers. Even though the structure and exercise of DNN are different, their functionality is always the same. The network receives a m dimensional, numerical input vector X_m and transforms it into a n dimensional, numerical output vector Y_n. (Scherer (2013)) The networks are compound by input-, hidden- and output layers, whereat all layers consist of a number of single neurons (Figure 3). The neurons receive input signals that are multiplied with a connection weight $w_{i,j}$ that indicates the strength of the connection between the neurons. The connection weights are adjusted during the learning process. Therefor, the weighted input signals of a single neuron are summed to a transfer function $u(t) = \sum_j x_j \cdot w_j$ that determines the activation condition of the neurons by means of an activation function. (Styczynski et al. (2017)) From a mathematical perspective, a DNN is a directed graph, where each node implements a neuron model. Each neuron model is characterized by a weighted sum of the input signals that are transferred by a transfer function. (Floreano et al. (2008))

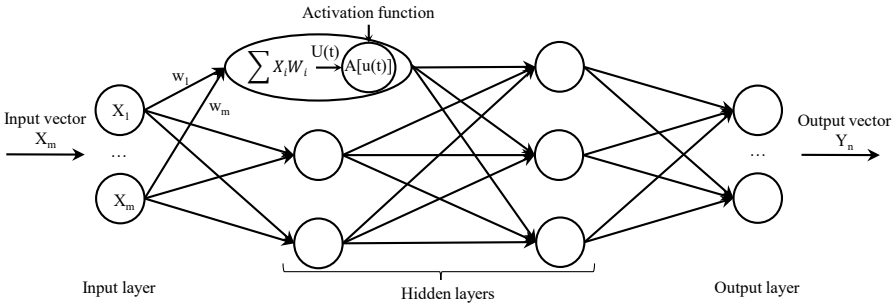

Figure 3 – Structure neural networks

The activation function computes the output value $y(t)$ of the neurons using the transfer function $u(t)$ as input. Depending on the desired range of output values, different activation functions are applied. At this juncture, activation functions feature non-linear or linear, as well as bounded or unbounded output ranges. In example, the *sigmoid* function $sigmoid(x) = \frac{1}{1+e^{-x}}$ is a non-linear activation function and constitutes an output range between $[0, 1]$. In contrast, the $ReLU$ activation function is non-linear with a lower bound that outputs x for $x \geq 0$ and 0 otherwise. (Sharma (2017))

2.3 Parameter Optimization in Deep Neural Networks

The objective of the learning process in neural networks is to find a set of internal network parameters that minimize the error between the computed output by the network and the desired output. (Rumelhart et al. (1986)) Backpropagation is the most commonly used optimization approach for neural networks, where the adjustment of each weight is based on a gradient vector. This vector indicates by what amount the error would increase or decrease, if the weights are increased by a small amount. The weights are then adjusted into the opposite direction of the gradient vector. This procedure is repeated iteratevly until the error is minimized. (LeCun et al. (2015)) In contrast, the parameter optimization with EAs is independent of any gradient information and depends only on the fitness assignment (Ding et al. (2013)). In the following, these two approaches will be explained in more detail.

2.3.1 Learning with Backpropagation

The learning process of a DNN can be accomplished in three steps (Figure 4) that are repeated iteratively until some condition is met. Normally, the

termination condition takes hold, if the error converges towards zero. In the first step, the network receives an input and propagates it forward, from the input to the output layer, and outputs y. In the second step, y is evaluated against the desired output t by means of a loss function. The loss function constitutes an error function and calculates a distance value E between the computed and desired output. (Chollet (2018)) Commonly used loss functions are e.g. the Mean Absolute Error (MAE) $E = MAE = \frac{1}{n} \cdot \sum_{j=1}^{n} |y_j - t_j|$ or the Mean Square Error (MSE) $E = MSE = \frac{1}{2} \cdot \frac{1}{n} \cdot \sum_{j=1}^{n} (y_j - t_j)^2$ among others, where n denotes the size of the employed data set. (Aunkofer (2019)) In the last step, an optimizer updates the network's parameters corresponding to the error E. Since the parameter updates are performed from the output layer to the input layer, the last step is also called backward pass. (Chollet (2018))

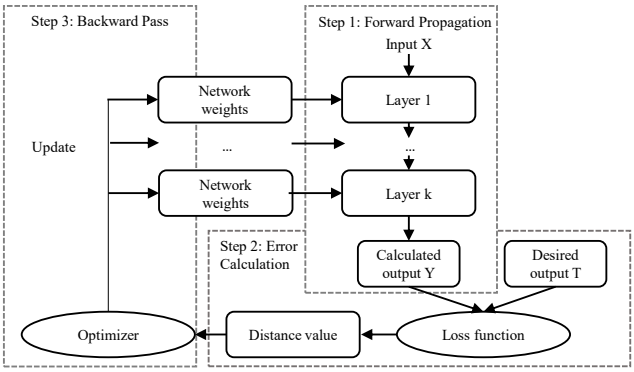

Figure 4 – Learning process of a neural network, based on Chollet (2018)

To minimize E, the backpropagation algorithm uses the gradient descent that computes the partial derivation of the error E in respect to each weight $w_{i,j}$. The backward pass starts with the computation of the partial derivation for each output neuron:

$$\frac{\partial E}{\partial y_j} = y_j - d_j \tag{2.1}$$

According to the chain rule the partial derivation (Equation 2.2) for each transfer function u_j is computed and describes how a change in the input will affect the error. With this relation, the correlation between E and each of the network's weights $w_{i,j}$ can be determined (Equation 2.3).

$$\frac{\partial E}{\partial u_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial u_j} \tag{2.2}$$

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w_{i,j}} = \frac{\partial E}{\partial u_j} \cdot y_j \tag{2.3}$$

Taking into account all connections from neuron i, $\frac{\partial E}{\partial y_j}$ is determined for all neurons in the previous layer by:

$$\frac{\partial E}{\partial y_j} = \sum_j \frac{\partial E}{\partial u_j} \cdot w_{i,j} \tag{2.4}$$

This procedure is repeated for each layer of the DNN. Afterwards, the gradient descent changes each weight $w_{i,j}$ by a proportional amount of the accumulated $\frac{\partial E}{\partial w}$ (Equation 2.5), where α is the learning rate that indicates the amount of change to the weights according to the error E.

$$w_{t+1} = w_t - \Delta w_t = w_t - \alpha \cdot \frac{\partial E}{\partial w_t} \tag{2.5}$$

A major challenge in the backpropagation approach are vanishing and exploding gradients that describe a phenomenon, which occurs from applying backpropagation over a large number of layers. At this juncture, the network's weights are subject to too much adjustment in a exploding gradient scenario, or a subject to too little adjustment in a vanishing gradient scenario. In both cases, the weights are adjusted in an improper way, which results in an inefficient training of the network and prevents that the error function converts towards zero.

2.3.2 Learning with Evolutionary Algorithms

Advantages of evolutionary algorithms in optimization problems are their ability to approximate functions that are noncontinuous, non-differentiable or multimodal. Thereby, EAs perform a directed stochastic global search and reach near optimum solutions. Since EAs evaluate multiple points simultaneously, their optimization processes inherent parallelism. (Ding et al. (2013)) Furthermore, they overcome some shortcomings of backpropagation learning such as a learning stagnation in a local optimum, an appropriate parameter setting for e.g. the learning rate, the optimizer or the loss function, and do not suffer from vanishing or exploding gradients (Repetto (2017)). Besides the weight optimization, EAs offer additional approaches to a learning problem such as designing the network architecture or adapting learning rules. (Ding et al. (2013)) The performance of the network is determined by each individual's fitness, which is, in the simplest case, based on the produced error E of the network in a validation data set. The individuals that produce the lowest error are the fittest and are considered for the reproduction step. (Elsken et al. (2019)) The form in which the genetic operators are applied depends on the representation of the network and the information, each individual codifies. Both choices offer different implementation possibilities. Encoding is performed either direct or indirect. For

the former, the maximum detail of each parameter is specified by its representation, such that a network with N neurons creates a $N \times N$ matrix, where each element $c_{i,j}$ indicates the connection between neuron i and j. The indirect encoding approach reduces the representation by not explicitly codifying each individual connection weight. Instead, the characteristics of an individual are represented in a predefined way that is either based on the designer's knowledge or deterministic rules. The more compact representation of the characteristics reduces the high computational time that results from a big weight matrix for large neural networks. (Castillo et al. (2003)) The representations also offer different implementation possibilities, such as a direct, a binary or a developmental representation. While all approaches can be used to evolve both, the network architecture and its parameters, the direct representation is most suitable for evolving network parameters in fixed networks and the developmental approach offers more flexibility for evolving the network's architecture. In a direct representation scenario, the parameter values of a neural network are one-to-one mapped to the genes. The binary representation concatenates the weights of a single neuron, such that the genetic operator interchanges the complete hidden unit, instead of its single weights. Lastly, the developmental representation deploys a developmental process based on genetically encoding in order to construct the neural network. (Floreano et al. (2008))

2.4 Evolutionary Algorithms for Network Construction

Besides optimizing the network's parameters, finding an appropriate network structure is another challenge. Even though there are several rules to determine the number of layers and the number of neurons for each layer (e.g. Zhang et al. (2016), Renda et al. (2020)), the networks construction relies often on the designers expertise. At this juncture, the network construction influences the training results, as well as the computational effort. Lu and Lu, have shown, that DNN with only two hidden layers and a sufficient large number of neurons are capable of approximating any given function. However, it still has to be determined, which number of hidden neurons is sufficiently large. The hidden neurons highlight the characteristic of the data set and map the nonlinear relation between input and output. Frequently, the determination of the number of neurons is based on experiments or rules that define the number of neurons in subject to the number of input neurons (Equation 2.6 Hunter et al. (2012)), where N_i is the number of input and N_h the number of hidden neurons.

$$N_h = N_i \cdot 2 + 1 \tag{2.6}$$

To overcome the challenges of a manual network design, Neural Architecture Search (NAS) approaches were introduced in neuroevolution. NAS describes the process of automated network engineering and already outperforms manual designed architectures in image classification task. NAS features three main categories. The *search space* limits the considered network architectures, the *search strategy* details how the solution space is explored and the *performance estimation strategy* specifies the architecture evaluation. The NAS approach is compatible with different parameter optimization methods such as evolutionary methods or gradient based methods. Recently, NAS with evolutionary network search and backpropagation optimization outperform NAS where both, the network design and the parameter optimization is based on evolutionary principles. (Elsken et al. (2019))

3 Reinforcement Learning

RL is a machine learning approach, where an agents makes decisions in a uncertain environment and information is provided stochastically (Wang (2019)). The agent is an entity that acts depending on observations of an environment. The environment describes the scope of a particular optimization problem and is altered by the agent's actions. At each time step t, the agent receives an observation ω_t that defines the current state s_t of the environment (Figure 5). Based on some decision making process, the agent chooses and performs an action a_t. Afterwards, the environment changes to a new state s_{t+1} and the agent receives a reward r_t. (François-Lavet et al. (2018))

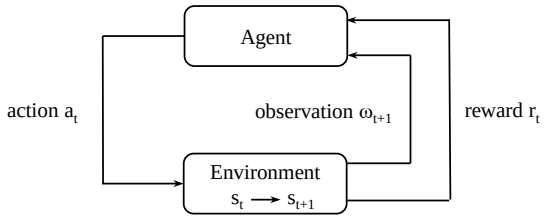

Figure 5 – Agent-environment interaction, based on François-Lavet et al. (2018)

3.1 Deep Reinforcement Learning and Markov Decision Processes

The optimization model designed for stochastic decision making under uncertainty is characterized by the Markov Decision Process (MDP). RL provides near optimal solutions to such optimization problems and enables a numerical function approximation by learning from sampled data and rewards of the environment. (Buşoniu et al. (2018)) Since the RL methods approximate a functional solution for the optimization problem, it is not guaranteed that an optimal solution is found.

3.1.1 Markov Decision Process

Formally, the MDP can be illustrated as a tuple $(S, A, p(s_{t+1}, r_t|s_t, a_t), r, \gamma)$, where S defines the state space that describes all possible states of the envi-

ronment. The action space A contains all possible actions a that the agent can perform. $p(s_{t+1}, r_t|s_t, a_t)$ specifies the transition dynamic function and provides the probability of transitioning to state s_{t+1} and receiving reward r_t after performing action a_t in state s_t. r characterizes the expected reward from the environment after the agent performs action a_t in state s_t. Lastly, γ is a discount factor with $0 < \gamma \leq 1$ and indicates how strong future rewards are taken into account. (Nian et al. (2020)) The agent follows a policy π that maps a probability distribution over actions from states $\pi : S \rightarrow p(A = a|S)$. An optimal action selection is represented by an optimal policy π^* that achieves an optimal reward for each action $a \in A$ in each state $s \in S$. The sequence of states, actions and rewards in a MDP is a trajectory of the policy and the expected return is denoted by R (Equation 3.1). Depending on whether the MDP is episodic or not, the discount factor is taken into account. In an episodic MDP, where the state is reset to the initial state s_0 after each episode of length T, γ is set to one and in a non-episodic case, γ is chosen to a value $\gamma < 1$ so that the expected return has a finite value for an infinite planning horizon. (Van Otterlo and Wiering (2012))

$$R = \sum_{t=0}^{\infty} \gamma^t \cdot r_{t+1} \tag{3.1}$$

The agent searches for an optimal policy π^* that maximizes the expected return R. Additionally, the agent features two objective functions, the state-value function $V^{\pi,a}(s)$ (Equation 3.2) and the state-action function $Q^{\pi}(s, a)$ (Equation 3.3). The former describes the expected return of state s, following policy π. The latter estimates the expected reward for a given state s and action a. (Ding et al. (2020))

$$V^{\pi,a}(s) = E\{\sum_{t=0}^{\infty} \gamma^t \cdot r^{\pi}_{s,t}\} \tag{3.2}$$

$$Q^{\pi}(s, a) = E\{\sum_{t=0}^{\infty} \gamma^t \cdot r^{\pi}_{s,a,t}\} \tag{3.3}$$

3.1.2 Deep Reinforcement Learning

RL provides different solution approaches for stochastic optimization. Depending on the knowledge about the environment's transition dynamics, *model-based* or *model-free* RL is implemented, whereat *model-based* RL requires complete knowledge about the transition dynamics. The *model-free* algorithms either estimate the optimal policy π^* directly in *policy-search methods*, or approximate

one of the objective functions in *value-based methods*. Further differentiation is possible between *on-policy* and *off-policy* learning (Figure 6). For the former, one policy is optimized by means of collecting a batch of behavior from the agent's interaction with the environment and deploying this batch of behavior to update the agent's policy. The agent uses every batch of behavior only once and collects new data with the updated policy for the next learning step. (Gu et al. (2016)) The latter reuses samples from the agent's interaction by storing them in a buffer D. Each learning step produces a new policy π_t and D consists of samples from $\pi_0 \dots \pi_t$ at time t. (Levine et al. (2020))

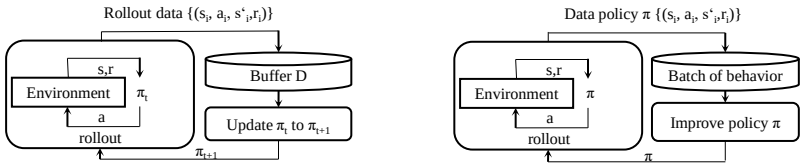

Figure 6 – On-policy and Off-policy Reinforcement learning, based on Levine et al. (2020)

DRL is one method of RL in which various components of the agent, such as the policy π, the state-value function $V(s)$, or the state-action function $Q(s, a)$ is estimated with a DNN. At this juncture, the network's parameters are trained to optimize the policy or one of the objective functions respectively (Figure 7). As mentioned before, in an *off-policy* RL approach, the agent's behavior is stored in a buffer. Thereby, the network is trained with experience replay by taking advantage of previously experienced transitions that are uniformly sampled from the buffer and after each action, the transition (s_t, a_t, r_t, s_{t+1}) is stored in the buffer. (Hessel et al. (2018)) The buffer provides uncorrelated training data for the network and thus, prevents it from over-fitting recent experience. As a result, the network's training is stabilized. The Temporal Difference (TD) algorithm is one possibility for the network optimization and is relied on by many state-of-the-art DRL solutions. The algorithm is used to update the network's parameters based on the so called TD-error δ_t (Equation 3.4, 3.5) that is measured between the value of the objective function in state s_t and state s_{t+1}, discounted by γ. In respect to the error, the objective function is adjusted until it converges and the optimal policy π^* is estimated. (Sutton and Barto (2018))

$$\delta_t = R_{t+1} + \gamma \cdot (V(s_{t+1}) - V(s_t)) \tag{3.4}$$

$$\delta_t = R_{t+1} + \gamma \cdot (Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3.5}$$

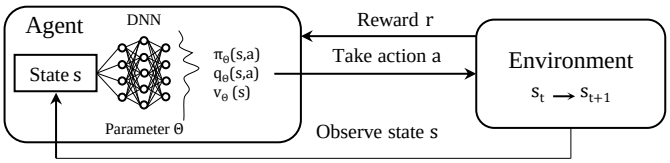

Figure 7 – Deep reinforcement learning, based on Mao et al. (2016)

A great challenge in RL is the trade-off between exploration and exploitation, whereat the agent has to decide whether to investigate its environment or leverage past experiences to choose an action. To optimize its action selection the agent must do both. The exploration is supposed to improve the agent's knowledge about the reward generation process and the exploitation utilizes the agent's experience to optimize its objective. (Chen et al. (2009))

3.2 Deep Reinforcement Learning Models

One state of the art DRL method is DQL, where a DNN is trained to approximate the state-action function $Q(s, a)$. In the following, DQL is explained in more detail. Afterwards, it is explained, how GA can be utilized to train deep reinforcement agents. At this juncture, the agent also estimates the state-action function $Q(s, a)$ for comparable results in the experiments.

3.2.1 Deep Q-Learning

In DQL a DNN, referred as Q-network, is optimized to find an optimal decision policy. The network is trained with backpropagation and uses the temporal difference to determine the error of the network. Additionally, a second network, referred as target network $\hat{Q}$, which possesses the parameters of the Q-network from a previous state, calculates the estimated return of the next state s_{t+1}. This target value $Y_t^{DQN} = r_t + \gamma \underset{a}{argmax}\ \hat{Q}(s_{t+1}, a_{t+1}; \theta_i^-)$ is then used to calculate the loss of the Q-network (Mnih et al. (2015)):

$$L_i^{DQN}(\theta_i) = \mathbb{E}[(Y_t^{DQN} - Q(s_t, a_t; \theta_i))^2] \tag{3.6}$$

where θ denotes the parameters of the Q-network and θ^- represents the target network's parameters. After every C learning steps, the parameters of the target network are replaced by the current parameters of the Q-network. The complete Q-learning algorithm is shown in the appendix (Page 28). As stated before, the loss function is based on the TD error:

$$\delta_t^{DQN} = r_t + \gamma \underset{a}{max}\ \hat{Q}(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t; \theta_i). \tag{3.7}$$

The TD error is measured between random chosen samples of the agent's experience. To handle the trade-off between exploration and exploitation, the DQL agent incorporates the ϵ-greedy exploration, where the agent chooses a random action with probability ϵ and selects the best known action with probability $1 - \epsilon$. Since the exploration behavior is most important at the beginning of the training and the exploitation behavior is most important at the end of training near convergence, the probability ϵ is reduced over time (Masadeh et al. (2018)).

3.2.2 Genetic Algorithm for Deep Reinforcement Learning

In contrast to DQL, the evolutionary algorithm to estimate the state-action function $Q(s, a)$ is completely independent of any gradient. The algorithm evolves a population of N individuals, where each individual represents a vector with the parameters of the DNN. At every generation, the different networks are evaluated by producing a fitness score. (Such et al. (2017)) The fitness score is based on the agent's cumulative reward over one or more episodes within the environment. The best performers of the generation are selected based on the selection operator. There are many kinds of selection methods such as *Truncation Selection*, where all individuals are ranked based on their fitness values (AbuZekry et al. (2019)), or *Tournament Selection*, where the individuals are grouped into a pool and the best individual of each pool is selected (Galván and Mooney (2021)). Afterwards, the crossover and mutation operators are applied to produce the next generation. Again, there are several methods for choosing these operators which include single-point or uniform crossover as well as mutation by applying Gaussian noise (Such et al. (2017)) or a random mutation, where a probability determines whether a gene is mutated or not (Mallawaarachchi (2017)). However, some state of the art DRL solutions do not consider the crossover operation at all and base the network's evolution only on mutation and selection operators (Such et al. (2017), AbuZekry et al. (2019)) A pseudo code for the GA is shown in the appendix (Page 29).

One advantage of using the GA is that the algorithm handles the agent's trade-off between exploration and exploitation implicitly. The exploration is accomplished by the creation of new individuals whereby new instances of the solution space are created and therefore, new action selection policies are introduces. The exploitation is utilized by the selection operator that favors better performing individuals to produce offspring and thus, uses the best action selection policies to create the next generation.

4 Experiments

To gather further insights into the performance of DRL agents that are trained with GA, this chapter presents a technical evaluation. In this context, each agent is trained to optimize a stochastic OAAP problem, where the agent needs to decide whether to accept an order or not. Each order can be processed on one of three identical production lines and the agent allocates each accepted order to one of them. Thereby two aspects are focused: Firstly a set of test is performed in order to find an appropriate parameter setting for the training process with GA. Therefor, the influence of different probabilities for mutation and crossover operations are examined. Since Such et al. (2017) and AbuZekry et al. (2019) have shown that GA with no crossover operation and a low mutation probability outperformed DQL agents, it is expected that agents with a low crossover and mutation probability outperform agents with a high probability. Afterwards, the influence of the number of episodes, an agent interacts with the environment during the individual's fitness assignment, affects the agent's training. At this juncture, it is expected, that more episodes in the validation data accelerate the training process, as the policy is evaluated over more episodes. Lastly, the size of the population and the number of parents are subject of examination. In this context, a larger population size and more parents to produce offspring should lead to a better policy after a lower amount of training steps, since more instances of the solution space are evaluated. Secondly, th best parameter set up for an agent trained with GA is evaluated against a DRL agent that is trained with DQL. Since evolutionary approaches for optimizing DNN parameters have already outperformed DQL approaches in DRL, it is excepted that the GA approach achieves at least similar results as the DQL approach. The following section describes the environment of the agents, as well as the parameter setting and network architecture for each agent. Afterwards, the results of the experiments are presented.

4.1 Experimental Setup and Model Details

The environment describes a stochastic OAAP where a random order arrives at each time step $t \in \{1, \dots, T\}$. Each order requires capacity on multiple re-

sources of each production line and offers a capacity related reward. Each production line consists of six resources. The agent's objective is the maximization of the accumulated return over an episode and its action space contains four actions: decline the order, accept the order on production line one, accept the order on production line two, or accept the order on production line three. If an accepted order exceeds the capacity on one of the resources of a production line, the agent receives a penalty reward. The mathematical formulation of the stochastic OAAP formulated as MDP is presented in the appendix (Page 29).

All agents share the same network architecture. The network is compound by one input, two hidden and one output layers. The number of input neurons corresponds to the agents observation, which characterizes the available capacity on each resource of each production line, the required capacity of the arriving order, the remaining time steps of the episode and the order's reward. Hence, the number of input neurons is 26. The number of hidden neurons is determined in respect to the number of input neurons (Equation 2.6) and the number of output neurons equals the size of the action space (Table 1).

Table 1 – Network Architecture

Learning Approach	Input Neurons	Hidden Neurons	Activation Function	Output Neurons
GA	26	53	ReLU	4
DQL	26	53	ReLU	4

To evaluate each individual of a generation, the GA approach interacts with a fixed validation data set that is changed after each generation. The best performers of each generation are considered for the crossover and mutation step and are carried into the next generation. The genetic operators use a direct encoding and representation so that all network weight are mapped one-to-one to a vector. The networks are randomly mutated and the crossover operator performs a uniform crossover to create offspring from two parents. In order to find an appropriate setting for the crossover and mutation probability, the size of each generation and the number of parents, as well as the size of the validation data, the number of generations during the tests is set to 1000. The DQL agent interacts directly with the environment and after each decision, the agent performs a learning step. The learning rate is set to $\alpha = 0.0005$, the batchsize is 512 and the *RMSprop* serves as optimizer. To save the best policy of the DQL agent, each policy that achieves an average score of 640 or higher over 50 training episodes is evaluated with a validation data set that contains 1000 episodes. The validation data is not changed during the agent's training so

that the policy which achieves the highest return in the validation data is saved and considered for the comparison with the GA approach. For the exploration exploitation trade-off, the agent utilizes the ϵ-greedy exploration strategy. The start value is set to $\epsilon_{start} = 1$ and the end value is $\epsilon_{end} = 0.01$. After each learning step, ϵ is reduced by $\epsilon_{dec} = 0.000033$ and the target network is updated after every $C = 300$ transitions. While validating or testing an agent's policy, ϵ is set to $\epsilon = 0$ and the agent only exploits.

4.2 Results

The results of the different agents are evaluated based on the training time, the average return over a test data set of 10000 random sampled episodes and the number of penalty rewards each agent received. Receiving a penalty reward lowers the return of an agent hence, actions that cause one should be avoided at all cost and thus, policies that generated a lower number of penalty rewards outperform policies with a higher number. The agents are implemented in python with the *PyTorch-Library* for the neural networks and the *PyGad-Library* (Gad (2021)) for the GA.

4.2.1 Experiment 1

For the first test, the crossover probability that determines the probability with which a parent is selected for applying the crossover operation is set to a high and a low value. The same applies to the mutation probability that indicates the percentage of genes that are mutated. All values are combined with each other so that four agents are trained (Table 2). The number of episodes for the agent's validation is set to 3, the size of the population is 125 and the number of parents is 15. All parents are carried into the next generation.

Table 2 – Parameter Setting Test 1

Agent	Crossover Probability	Mutation Probability
1	0.8	0.025
2	0.1	0.75
3	0.8	0.75
4	0.1	0.025

The average return of the individual with the best policy of each agent in each generation is shown in Figure 8. The return of all agents is rather similar and converge to a value of about 575. The oscillating of the return is caused by the

different rewards in the different validation data, as well as different decision making of different policies.

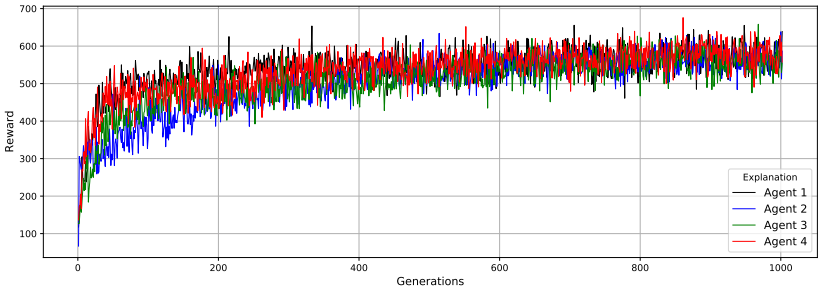

Figure 8 – Training Results Test 1

The test data better clarifies the performance of the agents (Table 3). The agents that use a low mutation rate seem to perform better than these with a high mutation rate. Moreover, a higher mutation probability increases the training time of the agents, while a high crossover probability does not strongly influence the training time. All agents achieve similar returns on each of their resources and agent 4, that has a low mutation and a low crossover probability, achieves the highest overall return. Even though agent 4 causes more penalty rewards compared to agent 1, its overall reward is about 30 units higher and thus it found a better policy and is considered for the second test.

Table 3 – Test Results Test 1

Agent	Overall Reward	Reward Resource 1	Reward Resource 2	Reward Resource 3	Training Time	Penalty Rewards
1	489.8427	162.5285	165.0213	162.2929	02:38:14	7094
2	462.9949	155.204	155.6415	152.1494	03:19:41	23995
3	496.5119	160.4848	169.6061	166.421	03:17:09	17282
4	521.003	181.1188	177.4589	162.4253	02:35:25	10132

For the second test, the mutation and crossover probability of agent 4 are adapted for agent 5 and only the size of the validation data is increased from 3 to 25 episodes. As in the first test, the average return during the training is rather similar (Figure 9). However, the average return of agent 5 does not oscillate that strong which can be explained by the fact that the additional episodes better indicate the expected return over an episode.

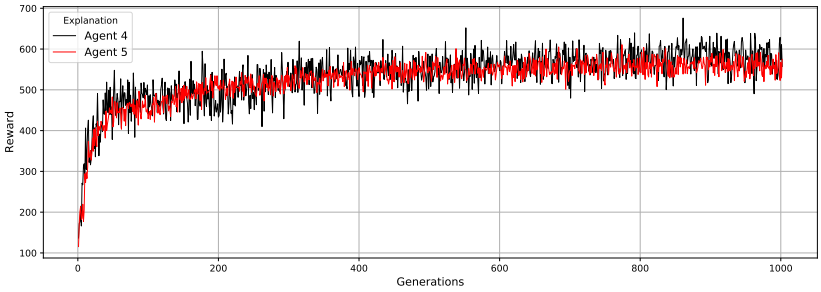

Figure 9 – Training Results Test 2

The overall reward of agent 5 in the test data is about 10 units lower than the reward of agent 4 (Table 4). Agent 5 achieves a better return on resource 3 compared to agent 4 and its rewards over all episodes are in a closer range. However, its number of penalty rewards is significantly higher and the additional episodes in the validation data dramatically increased the training time. Since agent 4 achieves a better overall reward with less computational time, its parameter setting is also used for the last test.

Table 4 – Test Results Test 2

Agent	Overall Reward	Reward Resource 1	Reward Resource 2	Reward Resource 3	Training Time	Penalty Rewards
4	521.003	181.1188	177.4589	162.4253	02:35:25	10132
5	511.8328	167.5284	166.3227	177.9817	08:29:44	15768

The subject of the last test is the influence of the population size and the number of parents on the agent's performance. Therefore, two additional agents with the parameter setting of agent 4 are introduced with a deviant population size and number of parents. For agent 6 both values are decreased and for agent 7, both are increased (Table 5).

Table 5 – Parameter Setting Test 3

Agent	Population Size	Number of Parents
4	125	25
6	50	10
7	250	50

The achieved average return of the agents during training is rather similar. However, the return of the agent with the highest population converges to a slightly higher value and the agent with the lower population size to a slightly lower average value (Figure 10).

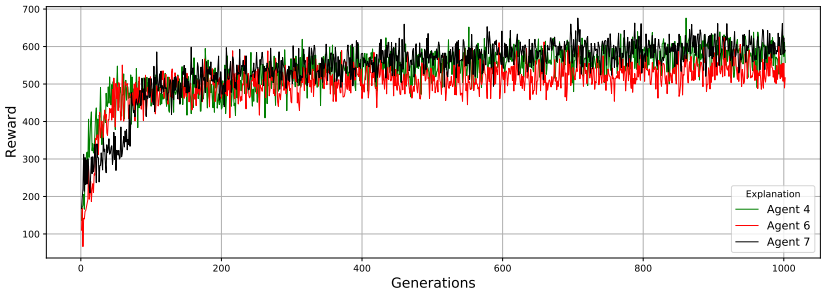

Figure 10 – Training Results Test 3

Even though agent 7 seems to perform better during training, its test results are slightly worse compared to agent 4 (Table 6). The achieved return and the received penalty rewards of agent 7 are worse compared agent 4 and its training time is considerably higher. Agent 6 required the lowest training time, however it performed significantly worse in each other category. Including the training time of the agents, agent 4 outperformed the other two and thus, is considered for the second experiment.

Table 6 – Test Results Test 3

Agent	Overall Reward	Reward Resource 1	Reward Resource 2	Reward Resource 3	Training Time	Penalty Rewards
4	521.003	181.1188	177.4589	162.4253	02:35:25	10132
6	440.0248	152.3698	147.2877	140.3673	00:59:34	36581
7	520.0313	170.4048	168.3614	181.2651	05:07:32	11441

4.2.2 Experiment 2

The second experiment examines the performance of an GA agent that has been trained over 2000 generations against a DQL agent, so that 250000 policies from the GA learning approach are tested, compared to 360000 policies of the DQL agent. During training, the policies of the GA agent start with a higher return (Figure 11). The low average return of the DQL agent can be explained by the fact that the exploration probability ϵ of the DQL agent is high, which leads it to perform many random actions and thus, the agent receives penalty rewards at a high rate. With the decreases of the exploration probability, the average return increases and its return converge to a average above 600, which is slightly better than the return of the GA agent. Since ϵ never reaches zero during the training process, the exploration probability prevents the agent from receiving even higher average returns.

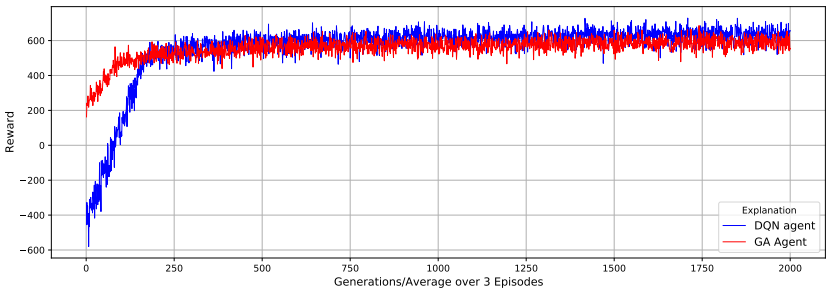

Figure 11 – Training Results Experiment 2

During testing, the DQL agent outperforms the GA agent in every measurement (Table 7). The achieved overall reward is over 100 units higher and the number of penalty rewards significantly lower. In addition, the DQL agent required only about one fifth of the GA agent's training time. It has to be noted that the training time of the DQL agent includes the time for the policy validation, which lasted about 13 minutes for evaluating 63 policies. Thus, the pure training time of the agent was only 46 : 39 minutes.

Table 7 – Test Result Experiment 2

Agent	Overall Reward	Reward Resource 1	Reward Resource 2	Reward Resource 3	Training Time	Penalty Rewards
GA	535.0799	176.0068	184.7298	174.3433	04:54:26	9294
DQN	655.3575	234.1577	209.7353	211.4645	00:59:53	497

5 Discussion and Prospect

The experiments in the previous section have shown that GA algorithms are capable of training neural networks and besides, can be applied to DQL. However, with the tested parameter setting for the GA, this approach is not competitive to DQL tasks. At this juncture, the second experiment illustrates that DQL required less computational time and achieve a higher average return in the considered OAAP. The first experiment indicates that varying the GA parameters rather influences the computational effort than its results. Thereby, additional generations, a larger population size or a higher crossover probability lead to considerably higher training times and achieve no or only slightly better results. GA lead the DRL agent to find an valuable action selection after a short training period so that the return starts to convert. However improving the action selection has itself proven to be difficult. Neither additional generations, nor a larger population size strongly influence the action selection so that small improvements in the action selection comes at the cost of high computational times.

However, it must be noted that a different parameter setting might lead to different results and have not been considered for the presented experiments. While the crossover operator has been proven to be not that influential to the network's optimization, a deviant result might be achieved with e.g a one-, or two-point crossover operation. Moreover, a direct representation and encoding result in high computational times since every network weight is considered for crossover and mutation operations. Indirect encoding approaches reduce the total number of parameters by compressing network parameter's so that single crossover and mutation operations are performed on a set of parameters at a time. As a result, it is expected that the computational time decreases since new values for fewer parameters have to be computed. Further improvements can be achieved with a different performance evaluation. While the fitness evaluation in the experiment are completely based on the agent's return, Such et al. (2017) and AbuZekry et al. (2019) have shown that DRL agents that are trained with GA in combination with a novelty search approach that assign the fitness value based on trying new action selection policies, achieve competitive result to DQL.

Last but not least, it is worth mentioning that the fact that all GA agents, as well as the DQL agent have receive penalty rewards, proves that none of the agents have found an optimal decision policy. To further improve the agent's policies, a NAS approach to optimize the network architecture may be a good solution. At this juncture, DQL agents can be extended with a GA to optimize the network's architecture, or algorithms such as NeuroEvolution of Augmenting Topologies (NEAT) that use the GA to optimize the network's architecture and its parameters, might find even better action selection policies.

List of Figures

List of Tables

List of Algorithms

Appendix

Deep Q Learning Algorithm

Initialize replay memory D
Initialize action-value function Q with random weights Θ
Initialize target action-value function $\hat{Q}$ with weights Θ^{-}
for planning horizon $m = 1, \ldots, M$ **do**
  for time step $t = 1, \ldots, T$ **do**
    if $exploration \ = \ True$ **then**
      Select random action a_t
    else
      Select $a_t = argmax\ Q(s_t, a_t; \Theta)$
    end if
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Compute $y_j = \begin{cases} r_j, & \text{if j=T} \\ r_j + \gamma \cdot \max \hat{Q}(s_{j+1}, a_{j+1}; \Theta^{-}), & \text{otherwise} \end{cases}$
    Compute $Loss = (y_j - Q(s_j, a_j; \Theta))^2$
    Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ
    Update the current state $s_t \leftarrow s_{t+1}$
  end for
  Reset state to the initial state $s_t \leftarrow s_0$
  Every C steps reset $\hat{Q} = Q$
end for

Algorithm 1: Deep Q-Learning

Deep Reinforcement Learning with Genetic Algorithm

Initialize population P_0 with random weights Θ
Initialize validation data set V with size K
for generation $g = 1, \ldots, G$ **do**
  for Individual $i = 1, \ldots, I$ **do**
    Compute fitness in V_t
  end for
  Select best performers B based on the fitness
  for indivual $b = 1, \ldots, B$ **do**
    Perform crossover on B's weights Θ
    Perform mutation on B's weights Θ
  end for
  Update validation data set $V_t \leftarrow V_{t+1}$
end for

Algorithm 2: Genetic Algorithm

Order Acceptance and Allocation Problem as MDP

The OAAP describes an episodic MDP, where the environment is reset to the initial state s_0 after each episode of length T (Table 8). All incoming orders belong to a certain order class $n \in \{1, \ldots, N\}$ (Table 9) and consumes capacity $c_{n,j,p}$ on each resource $j \in \{1, \ldots, J\}$ of one production line $p \in \{1, \ldots, P\}$. The probability of each order's arriving is $p(n) = \frac{1}{N}$. The available capacity of each resource in each production line is $c_{j,p}^{max}$. The revenue $r_{c_{n,j,p}} \in \{1, \ldots, R_{c_{n,j,p}}\}$ of each order class is capacity related so that the reward of a specific order is determined by:

$$r_{n,r_{c_{n,j,p}}} = \sum_{j=1}^{J} r_{c_{n,j},p} \cdot c_{n,j,p} \qquad \forall\ r_{c_{n,j,p}} \in R_{c_{n,j,p}}, n \in N, p \in P$$

Table 8 – Required capacity $c_{n,j}$ of order class n on resource j of production line p

n \ j	1	2	3	4	5	6
1	1	2	3	2	1	0
2	1	1	1	1	0	0
3	0	2	2	0	0	0
4	2	2	1	1	0	0
5	1	3	2	2	1	0
6	2	1	1	1	3	3

The reward is randomly assigned with probability $p(R_{c_{n,j,p}}) = \frac{1}{R_{c_{n,j,p}}}$. The state of the environment is characterized by $(c_{n,j,p}, c_{occ,j,p}, t_{rem}, R_{n,t})$, where $c_{n,j,p}$ is the required capacity of the order, $c_{occ,j,p}$ denotes the occupied capacity on resource j of production line p and the agent's action space A contains the following actions:

$$A = \begin{cases} a_1, & \text{if the order is rejected} \\ a_2, & \text{if the order is accepted on production line 1} \\ a_3, & \text{if the order is accepted on production line 2} \\ a_4, & \text{if the order is accepted on production line 3} \end{cases}$$

If an order is rejected the agent's reward is $R^{a_1} = 0$. If an order is accepted, the agent receives either a penalty reward if it exceeds the capacity of one resource, or the order's reward.

$$R = \begin{cases} r_{n,r_{c_{n,j,p}}}, & \text{if } c_{occ,j,p} + c_{n,j,p} \leq c_j^{max} \qquad \forall j \in J,\ p \in P,\ s \in S \\ r_{pen}, & \text{else} \end{cases}$$

The agnt's objective is the maximization of the accumulated return R_T over all episodes.

$$Max!\ R_T = \sum_{t=0}^{T-1} R_t^A \tag{5.1}$$

Table 9 – Parameters of the environment

Parameter	Value	Meaning
T	60	number of episodes
N	6	number of order classes
J	6	number of resources
P	3	number of production lines
$c_{j,p}^{max}$	6	maximum capacity of resource j of production line p
$R_{c_{n,j,p}}$	9	number of possible rewards
r_{pen}	-20	penalty reward

References

Ahmed AbuZekry, Ibrahim Sobh, Mayada Hadhoud, and Magda Fayek. Comparative study of neuroevolution algorithms in reinforcement learning for self-driving cars. *European Journal of Engineering Science and Technology*, 2(4):60–71, 2019.

Benjamin Aunkofer. Training eines neurons mit dem gradientenverfahren, 2019. URL `https://data-science-blog.com/blog/2019/01/13/training-eines-neurons-mit-dem-gradientenverfahren/`.

Lucian Buşoniu, Tim de Bruin, Domagoj Tolić, Jens Kober, and Ivana Palunko. Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, 46:8–28, 2018.

PA Castillo, MG Arenas, JJ Castillo-Valdivieso, JJ Merelo, A Prieto, and G Romero. Artificial neural networks design using evolutionary algorithms. In *Advances in soft computing*, pages 43–52. Springer, 2003.

Jie Chen, Bin Xin, Zhihong Peng, Lihua Dou, and Juan Zhang. Optimal contraction theorem for exploration–exploitation tradeoff in search and optimization. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 39(3):680–691, 2009.

Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

Carlos A Coello Coello. An introduction to evolutionary algorithms and their applications. In *International Symposium and School on Advancex Distributed Systems*, pages 425–442. Springer, 2005.

Shifei Ding, Hui Li, Chunyang Su, Junzhao Yu, and Fengxiang Jin. Evolutionary artificial neural networks: a review. *Artificial Intelligence Review*, 39(3): 251–260, 2013.

Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong. Introduction to reinforcement learning. In *Deep reinforcement learning*, pages 47–123. Springer, 2020.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1):47–62, 2008.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

Ahmed Fawzy Gad. Pygad: An intuitive genetic algorithm python library. *arXiv preprint arXiv:2106.06158*, 2021.

Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2021.

Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, pages 2829–2838. PMLR, 2016.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

David Hunter, Hao Yu, Michael S Pukish III, Janusz Kolbusz, and Bogdan M Wilamowski. Selection of proper neural network sizes and architecturesâa comparative study. *IEEE Transactions on Industrial Informatics*, 8(2):228–240, 2012.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436–444, 2015.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

Yulong Lu and Jianfeng Lu. A universal approximation theorem of deep neural networks for expressing distributions. *arXiv preprint arXiv:2004.08867*.

Vijini Mallawaarachchi. Introduction to genetic algorithms â including example code, 2017. URL https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

Alaeddin Masadeh, Zhengdao Wang, and Ahmed E Kamal. Reinforcement learning exploration algorithms for energy harvesting communications systems. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, 139:106886, 2020.

Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning. *arXiv preprint arXiv:2003.02389*, 2020.

Anthony Repetto. The problem with back-propagation, 2017. URL https://towardsdatascience.com/the-problem-with-back-propagation-13aa84aabd71.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Andreas Scherer. *Neuronale Netze: Grundlagen und Anwendungen*. Springer-Verlag, 2013.

Sagar Sharma. Activation functions in neural networks, 2017. URL https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1 (1):24–35, 2019.

Zbigniew A Styczynski, Krzysztof Rudion, and André Naumann. *Einführung in Expertensysteme: Grundlagen, Anwendungen und Beispiele aus der elektrischen Energieversorgung*. Springer-Verlag, 2017.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

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

Martijn Van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement learning*, pages 3–42. Springer, 2012.

Pradnya A Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*, pages 261–265. IEEE, 2016.

V X Wang. *handbook of Research on Transdisciplinary Knowledge Generation*. IGI Global, 2019.

Andreas Welsch, Verena Eitle, and Peter Buxmann. Maschinelles lernen. *HMD Praxis der Wirtschaftsinformatik*, 55(2):366–382, 2018.

Yuchen Zhang, Jason D Lee, and Michael I Jordan. l1-regularized neural networks are improperly learnable in polynomial time. In *International Conference on Machine Learning*, pages 993–1001. PMLR, 2016.