Carl von Ossietzky Universität Oldenburg

Studiengang:
Masterstudiengang Informatik

Masterarbeit

Titel:
Optimisation of Reinforcement Learning using Evolutionary Algorithms

vorgelegt von:
Mario Fokken

Betreuender Gutachter:
Dr.-Ing Eric Veith
Zweiter Gutachter:
M.Sc. Torben Logemann

Oldenburg, 20.03.2025

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Contents

List of figures

1. Introduction

In recent years, agent systems, and in particular multi-agent systems (MAS) [4], [5], [6], [7], have emerges as one of the most important tools to facilitate the management of complex energy systems. As swarm logic, they can perform a wide range of tasks, such as maintaining real power balances, voltage control or automated energy automated energy trading [8]. The fact that MASs implement proactive and reactive distributed heuristics, their behaviour can be analysed and to analyse their behaviour and provide certain guarantees, a feature that has has helped their deployment. However, modern energy systems have also become valuable targets. Cyber-attacks have become more common [9], [10], and While the establishment of local energy markets is an attractive concept of of self-organisation, can also be open to manipulation, for example through through artificially created congestion [11]. Attacks on power networks are no longer carefully planned and executed, but also learned by agents, e.g. market manipulation or voltage violations [12]. The careful design of software systems against a widening field of adversarial scenarios has become a challenge. scenarios has become a challenge, especially given the complex, interconnected cyber-physical systems (CPSs) are inherently exploitable by their very complexity [13]. Learning agents, particularly those based on Deep Reinforcement Learning (DRL), have gained as a potential solution: When a system is faced with faced with unknown unknowns, a learning agent can develop strategies to against them. In the past, researchers have used DRL-based agents for many tasks related to power system operations - e.g. voltage control [14] - but the approach of using of using DRL for general resilient operation is relatively new. [15], [16]. DRL - the idea of an agent with sensors and and actuators that learns by trial and error - is at the heart of many many notable successes, such as MuZero [17], with modern algorithms such as such as Twin-Delayed DDPG (TD3) [18], Proximal Policy Gradient (PPO) Policy Gradient (PPO) [19], and Soft Actor Critic (SAC) [20] which have proven their ability to tackle complex tasks. All modern DRLs use deep artificial neural networks (ANNs) at least for the policy (or several, e.g. for the critic). The actual parameter optimisation is usually done with gradient descent algorithms. descent algorithms. However, the architectures of these ANNs still have to be by the user, in addition to the hyperparameters of the algorithm. of the algorithm, which in itself has some disadvantages: The user may not have the extensive knowledge in the field of machine learning needed to optimise the network's architecture and hyperparameters. This may lead to the user choosing

by themself or staying with the standard parameters, which are not adapted to the current task. Both cases can result in a subpar choice of hyperparameters and architecture, thus leading to an unsatisfying result since theses settings have a severe influence on the performance and quality of the learning, as shown e.g. by [21]. Additionally, because the agents are part of a critical infrastructure, the choice of hyperparameters and architecture has to be reasoned upon.
The goal of this work is thus to establish an algorithm to chose the architectures and hyperparameters for the machine learning algorithm automatically, in order to always have a good choice without the need for user input, which solves the problems stated above.

2. Background

The implementation and results of this thesis rely on an understanding of machine learning and the environment in which the thesis's study takes place. Thus, this chapter outlines the basics of Agents, Machine Learning (Deep Reinforcement Learning and Evolutionary Algorithms), and ways to optimise ML.

2.1. Agents

An agent is an autonomous computer system capable of perceiving its environment and deciding upon action in it to fulfil their given role and objective. Agents are designed to operate independently and without human intervention on its decisions and task performance. The functionality of an agent can be split into three parts:

1. Sensing: The agent perceives the environment through its sensors and thus gathers data from their surroundings.
2. Processing: The agent processes the gathered information, assesses the state of the environment and then makes a decision based on predefined rules or learned experiences.
3. Acting: By using their actuators, the agent is able to perform the action decided upon in the environment.

A modern example of an agent is the robotic vacuum cleaner, which has a designated area to be cleaned by it – the environment –, sensors like cameras or infrared sensors, and actuators in form of motors for driving and vacuuming. The robot's objective is to clean its territory, which it achieves independently [22].

Agents are intelligent, when they are capable of flexible autonomous actions. 'Flexible' means, that the agent is capable of reactivity, by responding to changes

in the environment; pro-activeness, by exhibiting goal-directed behaviour and taking the initiative in order to satisfy their design objectives; and social ability, by interacting with other agents (and possibly humans). [23]

2.1.1. Multi-Agent Systems (MAS)

The just mentioned ability of interaction between agents makes it also possible to form systems of multiple agents, called multi-agent system (MAS). In a MAS several agents with different objectives, information, and abilities work together towards a single collective objective. Thus, these MASs are able of solving complex problems that lie beyond the capabilities of a single agent. Besides the coordination and collaboration of the agents, MASs as a whole are – like a single agent – also able to adapt to changes in the environment, making them flexible. Furthermore, MASs are decentralised, meaning that control and decision-making are distributed amongst multiple agents rather than centralised in a single entity, making the system more robust by avoiding single points of failure for example. Due to consisting of multiple single entities that take over different tasks, MASs have a great scalability and are capable of parallelism. [24]

MASs are used in several different fields and have many applications. For example, in cloud computing, MAS are employed for resource monitoring, ensuring optimal load distribution on Virtual Machines based on predefined policies. They enhance security by monitoring and responding to potential threats, assist in discovering available resources within the cloud infrastructure, and manage and automate services to improve efficiency and reliability.
In healthcare, agents are used to monitor patient health data in real-time, providing timely alerts to healthcare providers, and assist in the allocation of medical resources like hospital beds and medical staff to improve patient care.
Or in smart grids, where agents manage the energy production, storage, and consumption to enhance efficiency and profitability. Energy producers use agents to analyse price signals and decide whether to store or sell energy. MASs also assist in demand response by adjusting energy usage based on supply conditions. [25]

2.2. Powergrid

A power grid is an interconnected network that delivers electricity from producers to consumers. It is the backbone of modern electrical infrastructure, ensuring that power generated in various plants, such as coal, natural gas, nuclear, hydroelectric, wind, and solar, reaches homes, businesses, and industries across vast distances. The power grid operates through a complex system of transmission

lines, substations, transformers, and distribution lines, all coordinated to maintain a stable and continuous supply of electricity.

The power grid can be broadly divided into three main components: generation, transmission, and distribution. Generation refers to the process where power is produced at power plants. Transmission involves moving this electricity over long distances through high-voltage lines to different regions. Distribution is the final step, where electricity is delivered to end-users through lower-voltage lines.

To function efficiently, power grids rely on real-time monitoring and control systems to balance supply and demand, ensuring the grid remains stable despite fluctuations. This involves sophisticated technology like Supervisory Control and Data Acquisition (SCADA) systems that provide operators with real-time data and the ability to manage electricity flows.

One of the key challenges for power grids is maintaining reliability, especially as they integrate renewable energy sources like wind and solar, which are variable by nature. The grid must adapt to fluctuations in generation from these sources while continuing to meet the steady demand. Smart grid technology is increasingly being implemented to address this challenge, incorporating digital communication, automation, and energy storage solutions to enhance grid flexibility and resilience. [26]

Albeit being a fairly new trend [15], [16], machine learning has shown promise in addressing various power grid challenges, such as optimizing energy distribution, predicting electricity demand, and enhancing grid security. By leveraging data analytics and AI algorithms, machine learning can help power grid operators make more informed decisions, improve system efficiency, and respond to dynamic grid conditions in real-time. [14]

2.3. Machine Learning

Artificial Intelligence (AI) is a branch of computer science focused on creating systems capable of performing tasks that typically require human intelligence. These tasks include learning from experience, understanding natural language, recognising patterns, solving problems, and making decisions. AI encompasses a variety of techniques and approaches, including machine learning, neural networks, and natural language processing. [27]

2.3.1. Machine Learning

Machine learning (ML) is a subset of artificial intelligence (AI) that focuses on developing algorithms and statistical models that enable computers to perform tasks without explicit instructions. Instead, these systems learn from data by identifying patterns, making decisions, and improving over time through experience. This capability to learn and adapt autonomously has made machine learning a pivotal technology in various fields, from healthcare and finance to transportation and entertainment.

Machine learning is broadly defined as the study of computer algorithms that improve automatically through experience. Tom M. Mitchell, a prominent figure in the field, provides a more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" (Mitchell, 1997). This definition highlights three key elements: the task (T), the experience (E), and the performance measure (P). [28]

Machine learning encompasses several types of learning paradigms, each suited to different types of problems and data: [29]

- Supervised Learning: In supervised learning, the algorithm is trained on a labeled dataset, which means that each training example is paired with an output label. The goal is to learn a mapping from inputs to outputs that can be used to predict the labels of new, unseen data. Common supervised learning algorithms include linear regression, decision trees, and neural networks.
- Unsupervised Learning: Unsupervised learning deals with unlabeled data. The algorithm's goal is to identify underlying patterns or structures in the data without any specific guidance. Techniques such as clustering (e.g. k-means) and dimensionality reduction (e.g. principal component analysis) are typical examples of unsupervised learning.
- Semi-supervised Learning: This approach combines both labeled and unlabeled data to improve learning accuracy. It is particularly useful when labeling data is expensive or time-consuming.
- Reinforcement Learning: In reinforcement learning, an agent interacts with an environment and learns to make decisions by receiving rewards or penalties. The agent's objective is to maximise cumulative rewards over time. This paradigm is often applied in robotics, game playing, and autonomous systems.

Machine learning's ability to analyse vast amounts of data and extract meaningful insights has revolutionised many industries. In healthcare, ML algorithms can predict disease outbreaks, diagnose medical conditions from imaging data, and personalise treatment plans. In finance, they are used for credit scoring, fraud detection, and algorithmic trading. Autonomous vehicles rely on machine learning for navigation, obstacle detection, and decision-making. Additionally, ML powers recommendation systems for online shopping and content streaming platforms, enhancing user experiences by suggesting products and media based on individual preferences. [28]

2.3.2. Deep Reinforcement Learning

One branch of machine learning is the previously mentioned Deep Reinforcement Learning (DRL), which combines deep learning with reinforcement learning in order to enhance the capabilities of the learning algorithm.

2.3.2.1. Deep

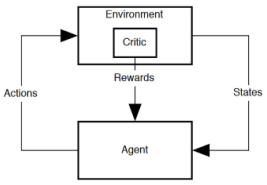

Figure 1: A basic representation of a neural network [1]

'Deep Learning' refers to the use of deep neural networks. Neural networks are models based on the human brain with multiple layers of interconnected neurons – a simple variant is displayed in Figure 1. Neurons are either part of the input, hidden, or output layer. The former layer gets the data given to the network, whilst the latter layer outputs the result of the network. The hidden layers are for the calculation of the result and are not seen by the user of the network, thus the name 'hidden'. The amount of layers in a network is referred to as 'depth' of the network, which is also the reason for the name 'deep learning'.
Each neuron has at least one input as well as one output connection. The output of the neuron is based on its input(s) and its inherent function. Each connection between two neurons holds a weight by which the output of the one neuron is multiplied with and the result then is given to the other neuron as input. Changing the weights of the network leads to a change in the result; fine-tuning weights to get a desirable result for every input is the goal of the deep learning algorithms. [30]

2.3.2.2. Reinforcement

As mentioned in Section 2.3.1, 'Reinforcement Learning' is one of the learning paradigms of machine learning. In Reinforcement Learning, the agent learns to make decisions by performing actions in an environment to maximise some notion of cumulative reward. Reinforcement learning is inspired by behavioural psychology and involves the agent learning from the consequences of its actions, rather than from being told explicitly what to do. The agent receives feedback in the form of rewards or penalties, which it uses to adjust its actions to achieve the best long-term outcomes.

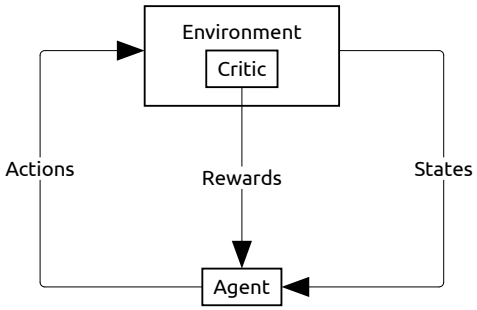

Figure 2: The feedback-loop of reinforcement learning algorithms [2]

In Figure 2, the standard feedback loop for a reinforcement learning algorithm is illustrated. An agent gets the current state of the environment, which it uses to decide upon actions. These actions and their effect on the environment are evaluated by the critic and handed to the agent along the environment's state. By using the reward to determine whether the proposed actions were good or bad, the agent is able to select the optimal actions corresponding to the highest reward [31].

2.3.2.3. Abilities of DRL

Deep Reinforcement Learning stands out for several reasons:
Firstly, it is able to handle high-dimensional data; DRL can process and learn from high-dimensional data inputs, such as images and videos, directly from raw pixels, without requiring manual feature engineering.
It is also capable of end-to-end learning; DRL frameworks can learn policies directly from input to output in an end-to-end manner. This holistic approach streamlines the process of training agents to perform complex tasks.
Scalability is another specialty of DRL; The use of deep neural networks enables DRL to scale to problems with vast state and action spaces, which were previously infeasible with traditional reinforcement learning methods [32].
Furthermore, DRL has been successfully applied to a wide range of real-world

problems, including robotics, autonomous driving, game playing, finance, and healthcare. Notable achievements include AlphaGo defeating human champions in Go and DRL agents excelling in complex video games and simulations [33]. And lastly, DRL agents can continuously learn and adapt to changing environments, making them suitable for dynamic and uncertain scenarios where the environment evolves over time [32].

2.3.3. Adversarial Reinforcement Learning

Adversarial reinforcement learning is an extension of traditional reinforcement learning techniques applied to multi-agent, competitive environments. In adversial reinforcement learning, at least two agents interact in an environment with opposing goals, often competing in a zero-sum game where one agent's gain is the other's loss. Both agents (or all agents respectively) learn and adapt their strategies simultaneously as they interact with each other and the environment. By learning the estimate of the value of state-action pairs, the agents may consider the opponent's potential moves. The agents must balance exploration of new strategies and exploitation of known effective actions to achieve their goals. [34]

2.4. Evolutionary Algorithms

Evolutionary algorithms (EAs) are a subset of artificial intelligence (AI) techniques inspired by the mechanisms of biological evolution. These algorithms are employed to solve optimization and search problems by simulating the process of natural selection, where the fittest individuals are selected for reproduction to produce the next generation of solutions.

(HISTORY | If useful improve else remove)
The conceptual roots of evolutionary algorithms date back to the 1950s and 1960s, influenced heavily by the works of John Holland, Ingo Rechenberg, and Lawrence Fogel. John Holland's seminal book, "Adaptation in Natural and Artificial Systems" (1975) [35], laid the groundwork for genetic algorithms (GAs), a prominent class of EAs. In parallel, Ingo Rechenberg and Hans-Paul Schwefel in Germany were pioneering evolutionary strategies (ES), another critical branch of EAs, through their work on optimisation problems in engineering. Lawrence Fogel's evolutionary programming (EP) further diversified the field by focusing on the evolution of finite state machines and problem-solving strategies.

Evolutionary algorithms typically go through a circle of following steps:

1. Initialisation: A population of potential solutions (individuals) is generated, often randomly.
2. Evaluation: Each individual's fitness is assessed based on a predefined fitness function.
3. Selection: Individuals are selected for reproduction based on their fitness, favouring the fittest individuals.
4. Crossover: Selected individuals are paired, and their genetic material is recombined to produce offspring with mixed traits.
5. Mutation: Offspring undergo random mutations to introduce new genetic variations.
6. Replacement: The new generation replaces the old population, and the cycle repeats until a stopping criterion is met (e.g. a solution is found or a maximum number of generations is reached).

By going through this cycle iteratively, evolutionary algorithms mimic the process of natural selection, where the fittest individuals are more likely to survive and pass on their genetic material to the next generation. Over time, the population evolves towards better solutions. Ideally, the population converges towards the global optimum, representing the best solution to the optimisation problem, but early stopping criteria like a predetermined threshold are often used to prevent overfitting or stagnation.

Evolutionary algorithms may be classified into three subcategories, each inspired by different evolutionary theories. Firstly, there are Darwinian Evolution algorithms, which are rooted in Charles Darwin's theory of natural selection. These algorithms emphasise survival of the fittest, where individuals are selected based on their fitness, and crossover and mutation create variation. Genetic algorithms (GAs) are a primary example of this approach. Secondly, Lamarckian Evolution algorithms are based on Jean-Baptiste Lamarck's theory, which posits that traits acquired during an individual's lifetime can be passed to offspring. In EAs, this translates to individuals that can adapt and improve within their lifetime before passing on their enhanced traits, merging the ideas of learning and evolution. Lastly, there are Swarm Intelligence algorithms, inspired by the collective behaviour of social animals, such as birds flocking or fish schooling. Algorithms like Particle Swarm Optimisation (PSO) and Ant Colony Optimisation (ACO) fall under this category. These algorithms utilise simple agents that interact locally with one another and with their environment, leading to the emergence of complex, intelligent behaviour at the global level.

The distinctive feature of evolutionary algorithms lies in their robustness and flexibility. Unlike traditional optimisation methods, EAs do not require gradient information or continuity of the search space. They are highly adaptable, capable of finding global optima in complex, multimodal landscapes where other algorithms might get trapped in local optima. Moreover, EAs can be parallelised effectively, making them suitable for solving large-scale problems across diverse domains, from engineering and economics to biology and artificial intelligence. [36]

2.5. Optimisation of Machine Learning Algorithms

In order to get the most of machine learning algorithms, optimisation is needed. Generally, there are two big areas that are more or less algorithm independent and whose improvement has a big influence on the performance of the machine learning algorithm: the algorithm's hyperparameters and the used network.

2.5.1. Hyperparameter Optimisation

Hyperparameters in machine learning are configuration settings used to control the learning process of a model. Unlike parameters that the model learns during training, such as weights in a neural network, hyperparameters are set before the training process begins and remain constant. Examples of hyperparameters include learning rate, number of epochs, batch size, and algorithm specific parameters like a gamma for a discounting function. The selection of hyperparameters significantly affects the performance and effectiveness of a machine learning model, shown e.g. in [21].

Hyperparameter optimisation, also known as hyperparameter tuning, refers to the process of finding the optimal set of hyperparameters for a machine learning model. This process aims to improve the model's performance on a given dataset by searching through different combinations of hyperparameters to identify the best-performing set. Effective hyperparameter optimisation can lead to better model accuracy, generalisation, and overall performance.

Several methods are used for hyperparameter optimisation. One common approach is Grid Search, which involves exhaustively searching through a manually specified subset of the hyperparameter space. This method is straightforward but can be computationally expensive, especially for large datasets or complex models.

Another popular method is Random Search, where hyperparameter values are selected randomly from a defined range. Random Search can be more efficient than Grid Search as it is not limited to a fixed grid and has a higher chance of finding a good combination in fewer iterations.

More advanced techniques include Bayesian Optimisation, which builds a probabilistic model of the function mapping hyperparameters to the objective to be optimised, and uses this model to select the most promising hyperparameters to evaluate next. This method is generally more efficient than Grid or Random Search, particularly for expensive function evaluations.

Lastly, methods like Gradient-based optimisation and evolutionary algorithms can also be applied to hyperparameter tuning, though they are less common. These methods adapt the hyperparameters iteratively based on performance feedback, either by calculating gradients (as in gradient descent) or using evolutionary strategies to evolve a population of hyperparameter sets.

For instance, in Bayesian Optimisation, an acquisition function is used to balance exploration (searching through new, unexplored hyperparameter values) and exploitation (refining the best-known hyperparameter values). This iterative process continues until a specified budget (time or computational resources) is exhausted or the performance improvement plateaus. [37]

2.5.2. Network Optimisation and Neural Architecture Search

As already mentioned in Section 2.3.2.1, networks are a vital part of deep reinforcement learning algorithms. Their topology, which describes the arrangement of the neurons and how they are connected amongst each other, has a big influence on the learning performance of the algorithm [21]. Automating the design of these neural networks is a challenging task that has been addressed by the field of neural architecture search (NAS). NAS aims to find the optimal network architecture for a given task without human intervention, by searching through a vast space of possible network architectures to identify the most effective design for a specific problem.

There are several approaches to NAS, ranging from reinforcement learning-based methods to evolutionary algorithms and gradient-based optimisation. Reinforcement learning-based NAS methods treat the search for network architectures as a sequential decision-making process, where the agent learns to select the best architecture based on rewards obtained from evaluating different architectures. These methods can be computationally expensive due to the large

search space and the need for extensive training and evaluation of architectures. Gradient-based optimisation such as Bayesian optimisation or gradient descent can also be used for NAS, where the network architecture is treated as a continuous space that can be optimised using gradient-based methods. However, the high-dimensional and discrete nature of network architecture search makes gradient-based methods challenging to apply directly.
Amongst the evolutionary-based approaches to NAS are algorithms like NEAT (NeuroEvolution of Augmenting Topologies) [38], which evolve neural network topologies and weights simultaneously. These are known under the TWEANN (Topology and Weight Evolving Artificial Neural Networks) moniker. Roughly, these TWEANN-algorithms will create a population of networks with different topologies and weights, whose individual networks will, based on their fitness to the task, either be removed or used as an basis for the next generation. This way, a network with optimal fit to the task will be found. [39]

3. Related Work

3.1. Using NAS for DRL

Using reinforcement learning for neural architecture search has been done for a long time and thus has been heavily researched [39]. The other way around – using neural architecture search to improve reinforcement learning – is, however, not common. One of the works with this premise is [40], introduces a framework that optimises their DRL agent through NAS. The authors argue that their framework outperforms manually designed DRL in both test scores and effiecency, potentially opening up new possibilities for automated and fast development of DRL-powered solutions for real-world applications. Another paper that uses NAS to improve DRL is [41]. In which the authors – similar to the other paper – report that modern NAS methods successfully find architectures for RL agents that outperform manually selected ones and that this suggests that automated architecture search can be effectively applied to RL problems, potentially leading to improved performance and efficiency in various RL tasks.

3.2. Combining EA with DRL

Several surveys attend to the combination of evolutionary algorithms and deep reinforcement learning:

[42] talks about reinforcement learning in the context of automated machine learning and which methods currently exist. As already mentioned, the success of machine learning depends on the design choices like the topology of the network or the hyperparameters of the algorithm. The field of automated machine learning tries to automate these design choices to maximise the success. Automated reinforcement learning (AutoRL) is a branch of automated machine learning which focuses on the improvement and automation of reinforcement learning algorithms. The survey shows that the evolutionary approach is possible and was done for NeuroEvolution and HPO, both of which are part of the undertaking in this paper. For the former, *NEAT* [43] and *HyperNEAT* [44] are mentioned as working solutions, whilst for the HPO variants of the *Genetic Algorithm* (GA) [45], *Whale Optimisation* [46], *online meta-learning by parallel algorithm competition* (OMPAC) [47], and *population based training* (PBT) [48] were successfully used.

[49] covers several uses cases and research fields of *Evolutionary reinforcement learning*, like policy search, exploration, and HPO. According to the survey, HPO in RL faces several challenges. However, the challenges, namely extremely expensive performance, too complex search spaces, and several objectives, can be addressed by using evolutionary algorithms instead. The algorithms for the latter are put into three categories: Darwinian, like GAs,; Lamarckian, like PBT and its variations (FIRE PBT [50], SEARL [51]); and combinations of both evolutionary methods, like an evolutionary stochastic gradient descent (ESGD) [52].

[53] focuses on using evolutionary algorithms for policy-search, but also has a section for HPO, in which several different algorithms are mentioned, such as PBT and SEARL.

4. Software Stack

To build up upon the background knowledge, this section covers the environment in which the optimisation is set, as well as a deeper explanation of the algorithms used.

4.1. palaestrAI

palaestrAI[1] is used as the execution framework. It provides packages to implement or interface with agents, environments and simulators. The main concern of palaestrAI is the orderly and reproducible execution of experiment runs, the

[1] https://gitlab.com/arl2/palaestrai

orchestration of the different parts of the experiment run, and the storage of results for later analysis.

palaestrAI's Executor class acts as an overseer for a series of experiment runs. Each experiment run is a definition in YAML format. Run definitions are in most cases created by running arsenAI on an experiment definition. An experiment defines parameters and factors; arsenAI samples them from a space-filling design and outputs experiment run definitions that are concrete instantiations of the experiment's factors.

ExperimentRun objects represent such an experiment run definition as it is executed. The class acts as a factory, instantiating agents along with their objectives, environments with their corresponding rewards, and the simulator. For each experiment run, the Executor creates a RunGovernor, which is responsible for managing the run. It takes care of the different stages: For each phase, setup, execution, and shutdown or reset, and error handling.

The core design decision that was made for palaestrAI is to favour loose coupling of the parts in order to allow for any control flow. Most libraries enforce an OpenAI-Gym style API, meaning that the agent controls the execution: The agent can reset() the environment, call step(actions) to advance execution, and need only respond to the step(·) method that returns done. Complex simulations for CPSs are often realised as co-simulations, i.e. they couple domain-specific simulators. Co-simulation software packages such as mosaik [54] allow these simulators to exchange data; the co-simulation software synchronises these simulators and takes care of proper time keeping. However, this means that from the perspective of the co-simulation software, palaestrAI's agents behave just like any other simulator. The execution flow is controlled by the co-simulator.

palaestrAI's loose coupling is realised using ZeroMQ [55], which is a messaging system that allows for a reliable request-reply pattern, such as the majordomo pattern [55], [56]. palaestrAI starts a message broker (MajorDomoBroker) before executing any other command; the modules then either use a majordomo client (sends a request and waits for the response), or the corresponding worker (receives requests, executes a task, returns a reply). Clients and workers subscribe to topics, which are automatically created the first time they are used. This loose coupling through a messaging bus allows the co-simulation with any control flow.

In palaestrAI, the agent is divided into a learner (brain) and a rollout worker (muscle). The muscle acts within the environment. It uses a worker that subscribes to

the muscle's identifier as a topic name. During the simulation, the muscle receives requests to act with the current state and reward information. Each muscle then first contacts the corresponding brain (acting as a client), provides state and reward, and requests an update of its policy. Only then does the muscle infer actions that are the response to the action request. In the case of DRL brains, the algorithm trains as experience is provided by the muscle. As many algorithms simply train based on the size of a replay buffer or a batch of experiences, there is no need for the algorithm to control the simulation.

But even for more complex agent designs, this inverse control flow works perfectly fine. The reason stems directly from the MDP: Agents act in a state, st. Their action at triggers a transition to the state st+1. That is, a trajectory is always given by a state, followed by an action, which then leads to the follow-up state. Thus, it is the state that triggers the agent's action; the state transition is the result of applying an agent's action to the environment. A trajectory always starts with an initial state, not an initial action, i.e. $\tau = (s0, a0, \ldots)$. Thus, the control flow as implemented by palaestrAI is actually closer to the scientific formulation of DRL than the Gym-based control flow.

In palaestrAI, the SimulationController represents the control flow. It synchronises data from the environment with setpoints from the agents, and different derived classes of the simulation controller implement data distribution/execution strategies (e.g. scatter-gather with all agents acting at once, or turn-taking, etc.)

Finally, palaestrAI provides facilities for storing results. Currently, SQLite is supported for smaller and PostgreSQL for larger simulation projects, via SQLalchemy[2]. There is no need to provide a special interface, and agents, etc. do not need to take care of results storage. This is thanks to the messaging bus: Since all relevant data is exchanged via message passing (e.g. sensor readings, actions, rewards, objective values, etc.), the majordomo broker simply forwards a copy of each message to the results storage. This way, the database contains all relevant data, from the experiment run file through the traces of all phases to the "brain dumps," i.e. the saved agent policies.

arsenAI's and palaestrAI's concept of experiment run phases allows for flexibility in offline learning or adversarial learning through autocurricula [57]. Within a phase, agents can be employed in any combination and with any sensor/actuator

[2] https://www.sqlalchemy.org/, retrieved: 2025-01-04

mapping. In addition, agents – specifically, brains – can load “brain dumps” from other, compatible agents. This enables both offline learning and autocurricula within an experiment run in distinct phases.

4.2. HARL

harl[3] is repository containing palaestrAI-agents that are capable of machine learning. Currently, two deep reinforcement learning algorithms are implemented: Proximal Policy Optimisation (PPO) and Soft Actor-Critic (SAC).

4.3. SAC

Soft Actor-Critic (SAC) is an off-policy deep reinforcement learning algorithm developed for continuous control tasks. SAC uses a setup with an actor and a critic. The actor outputs a probability distribution over actions, emphasizing randomness for exploration, and is used to act in the environment. The critic evaluates the expected cumulative reward (Q-value) for a given state-action pair, augmented by the entropy term and is used to learn from the actions of the actor. This setup of using different networks to act and learn is known as off-policy and has the advantage of reusing experiences from a replay buffer, enhancing sample efficiency for example as data can be repurposed multiple times without requiring new interactions with the environment for each learning update.

Another key feature of SAC is its maximum entropy framework. Traditional reinforcement learning optimizes the expected cumulative reward. In contrast, SAC augments this objective by maximizing the entropy of the policy. The entropy term encourages exploration by favoring stochastic policies. This makes the algorithm robust and allows it to discover multiple modes of optimal behavior. The entropy bonus is weighted by a temperature parameter, balancing exploration and exploitation. Additionally, the temperature parameter is automatically adjusted to maintain a target level of entropy, eliminating the need for manual tuning and adapting exploration based on the task’s complexity.

SAC has been successfully applied to various benchmark environments, outperforming both on-policy algorithms like PPO and off-policy algorithms like DDPG and TD3. It combines high sample efficiency, stability, and robustness, making it suitable for real-world tasks such as robotic locomotion and manipulation. [20]

[3] https://gitlab.com/arl2/harl

4.4. NEAT

NeuroEvolution of Augmenting Topologies (NEAT) is a genetic algorithm for the generation of artificial neural networks. It was developed by Kenneth O. Stanley and Risto Miikkulainen in 2001. NEAT not only evolves the topology of an artificial neural network, but also its weights and differs from other Darwinian evolutionary algorithms in three implementation aspects:

Firstly, starting with a minimal structure, NEAT gradually increases the the complexity of the networks over generations. This is based on the idea that it is easier to evolve a simple network to solve a problem and then add complexity to it, rather than starting with a complex network.
Secondly, NEAT has a direct encoding of the network structure, which allows for the evolution of complex networks without the need for a pre-defined structure and crossovers between different network topologies.
Lastly, NEAT protects structural innovations through speciation, which allows new topologies to optimise before competing with the general population.

These key aspects of NEAT lead to it outperforming fixed-topology methods and other topology-evolving systems on challenging reinforcement learning tasks. [38]

4.5. Bayesian Optimisation

Bayesian Optimization (BO) is a powerful framework designed to efficiently locate the global maximizer of an unknown function $f(x)$ – also known as black-box function – within a defined design space X. This methodology is particularly advantageous in scenarios where function evaluations are expensive or time-consuming, such as hyperparameter tuning in machine learning or experimental design in scientific research.

The optimization process unfolds sequentially. At each step, BO selects a query point from the design space, observes the (potentially noisy) output of the target function at that point, and updates a probabilistic model representing the underlying function. This iterative approach refines the model and guides subsequent search decisions, progressively improving the efficiency of finding the global optimum.

The key components of Bayesian Optimization include are the probabilistic surrogate model, the acquisition function, and sequential updating. The probabilistic surrogate model is central to BO and provides a computationally cheap approxi-

mation of the target function. The surrogate starts with a prior distribution that encapsulates initial beliefs about the function's behavior. This prior is updated based on observed data using Bayesian inference to yield a posterior distribution, which becomes increasingly informative as more evaluations are performed.
The Acquisition function directs the exploration of the design space by quantifying the utility of candidate points for the next evaluation. It balances exploration (sampling regions with high uncertainty) and exploitation (refining areas likely to yield high values). Popular acquisition functions include Thompson sampling, probability of improvement, expected improvement, and upper confidence bounds, each offering a unique strategy to navigate the trade-off between discovering new regions and capitalizing on known promising areas. Lastly, sequential ensures that after observing the function value at a new query point, the surrogate model is updated to incorporate this information. The prior distribution is adjusted to produce a posterior that better reflects the function's behavior, enhancing the precision of subsequent predictions.

These aspects of Bayesian optimisation lead to several advantages: Firstly, BO is data-efficient, requiring fewer evaluations to identify the global optimum compared to traditional optimization methods. This makes it particularly suitable for scenarios where function evaluations are costly or time-consuming. Secondly, Bayesian Optimization is well-suited for optimizing black-box functions, where the underlying function is unknown or lacks a closed-form expression. This flexibility allows BO to handle a wide range of optimization problems without requiring derivative information. Moreover, Bayesian Optimization is effective for optimizing non-convex and multimodal functions, where the objective landscape is complex and contains multiple local optima. The probabilistic nature of the surrogate model enables BO to explore diverse regions of the design space, increasing the likelihood of finding the global maximizer. Lastly, Bayesian Optimization leverages the full optimization history to make informed search decisions. By iteratively updating the surrogate model and acquisition function, BO incorporates past evaluations to guide the search towards promising regions, enhancing the efficiency of the optimization process. [58]

5. Concept

In order to establish a neural architecture search (NAS) into the reinforcement learning agents of palaestrAI, breaking up the learning cycle is needed. Due to the nature of reinforcement learning, namely using the result of the agent's actions

in an environment to calculate a reward, which is then used for improvement, it is not possible to execute a NAS before starting the whole process of palaestrAI, making it necessary to run the NAS in parallel.

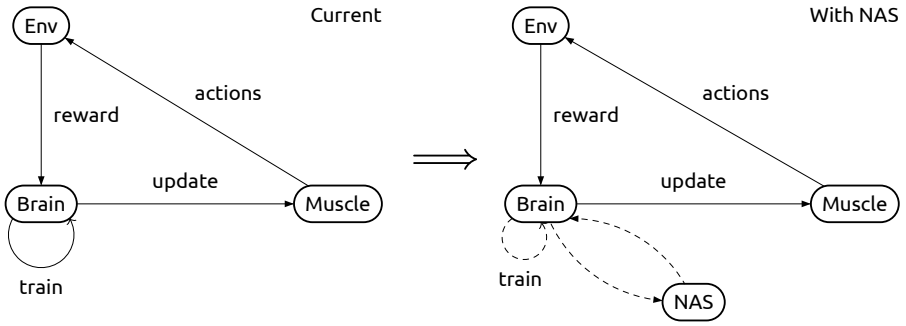

Figure 3: Adding NAS to the palaestrAI cycle

On the left side of Figure 3, the basic cycle of palaestrAI is shown. The agent's muscle suggests actions to take to the environment, which in result returns a reward to the agent's brain. In the brain the learning process takes places which changes the agent's network. Based on this network, the actions to suggest are calculated.

To add NAS to this cycle, the learning method of the brain is either completely or at times replaced by the NAS. After getting set up and creating an initial network, NAS generates a new network in every of its turns, which is then given to the brain to replace its current network. After the search finishes, the NAS is turned off and the normal learning behaviour is resumed.

Three different approaches of NAS were integrated as shown above:

1. An evolutionary algorithm in the form of NEAT
2. Reinforcement learning based NAS
3. Bayesian Optimisation

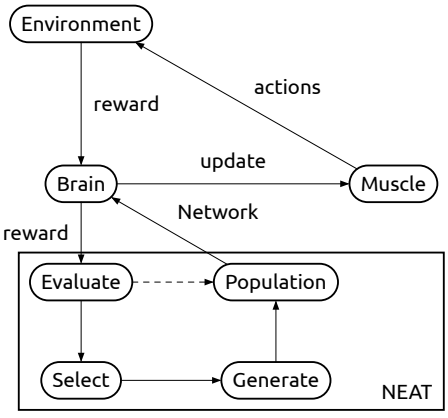

Figure 4: Using NEAT as NAS

Figure 4 shows the cycle with NEAT as the NAS method. During the setup, the first population of NEAT is generated. The network to be used is then taken from this population and run through the cycle. The resulting reward is forwarded by the brain to NEAT and saved to be used for the network's evaluation. If every network of the population has been evaluated, NEAT uses their resulting rewards to select the best ones and generate a new population. This process is repeated until the search is finished, ideally by finding a network that performs well in the environment. Due to the nature of NEAT, the network's weights are also improved throughout generations, making the original learning process obsolete. However, after NEAT has finished, further improvement with the normal learning process is still possible.

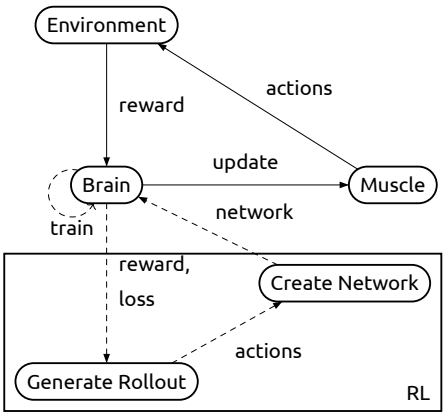

Figure 5: Using a RL approach as NAS

In Figure 5, the cycle is shown with reinforcement learning based NAS. Here, the NAS uses its reinforcement learning algorithm to generate a list of actions. These actions are used to create the initial network, which is then run through the cycle. During the runs, the network is trained normally in the brain where its weights are improved until a specified amount of runs is reached. Afterwards, the brain forwards the result as well as a loss to the NAS method, which are used to improve the NAS's reinforcement learning algorithm and generate a new network. By repeating the cycle, the algorithm should be able to improve the yielded results of its generated in the environment. After repeating the generation and running cycle a certain amount of times, the search is finished and the latest and/or best performing network is used for the normal learning process.

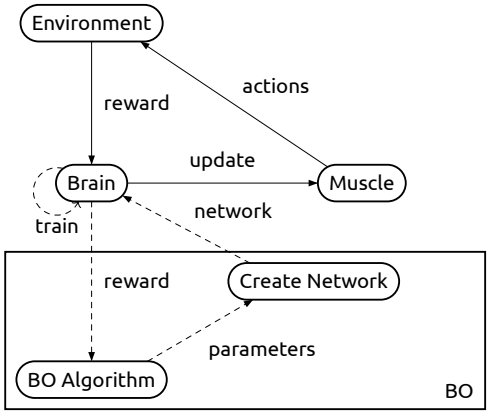

Figure 6: Using BO as NAS

Lastly, Figure 6 shows the cycle with Bayesian Optimisation as the NAS method. The black box function used for the Bayesian optimisation is made of an encoding of the network's topology. The result of the network's run is given to the algorithm to improve and select the next batch of parameters. By iterating through this process, the algorithm should be able to find a network with a high reward. Same as with the reinforcement learning approach above, each network is run through the normal learning process to improve the weights during its turn.

6. Implementation

For the implementation, each of the NAS methods is implemented in a separate Python module. Each module contains a class for the NAS method, which is used to link the brain to the NAS algorithm. This class is getting instantiated in the brain and is used to run the NAS algorithm. By setting a flag in the palaestrAI experiment file, the method of NAS can be selected.

Each of the NAS-methods also has their own network implementation. These contain a class for the NAS actor as well as a class for this actor's network. Both classes are almost identical to the network implementation of SAC in harl[4]. The only change made to the original implementation is the net itself; it was made changeable to allow the NAS methods to change the network's architecture.

The classes contain an initialisation or setup method, a method to return the initial network, as well as a 'run'-method. The former is used to setup the NAS algorithm and receives a dictionary of parameters. By taking a dictionary from the experiment file, the parameters of the NAS Method can be freely chosen by the user; standard values are used if the user does not provide a parameter.

[4]https://gitlab.com/arl2/harl/-/blob/development/src/harl/sac/network.py?ref_type=heads

The method to return the initial network is used during the setup process of the brain. When SAC's actor network is set up, this method is called to get the first network of the NAS algorithm.

Finally, the 'run'-method is used where the neural architecture search takes place. In every step inside the brain – as long as the nas is not finished yet –, a `nas_update` method is called. This method calls the corresponding NAS method's 'run'-method at the right times (depending on the NAS method) with the specific preparations needed. The 'run'-methods return a new network (or parameters to create a network) and whether the NAS algorithm has finished. Every new network is put into the SAC actor and an update containing the changed actor is sent to the muscle. If the NAS algorithm has finished, the brain will stop calling the NAS and proceed with the SAC algorithm's learning.

6.1. NEAT

For the implementation of NEAT a GitHub repository by ddehueck called PyTorch-NEAT[5] was used. Minor additions were conducted to make the implementation compatible with this use case: A list of rewards and a method to add a reward to it was given to the genomes. Further, the fitness function of NEAT was changed to base the genome's fitness on its reward list; the standard fitness is the mean of the rewards, but other functions like the sum or the maximum are also selectable by the user. These changes are necessary, since the use case makes it impossible to calculate the fitness of the genome directly and without prior runs.

The for this implementation of NEAT necessary config file was created based on the repository's example config file. This config file contains the parameters of the genomes, populations, and the NEAT algorithm itself. All these parameters are adjusted to the user specified values. The aforementioned adapted fitness function is also set in this config file.

In every step of the brain, the NEAT run method is called with the last step's reward. This reward is added to the current genome's reward list. If the genome and thus its corresponding network was run enough times through the process, the next genome of the population is popped and their network is returned to the brain. When the population has no more unrun genomes, a new population is created. This is repeated until a genome is found with a fitness over the user

[5] https://github.com/ddehueck/pytorch-neat/

specified threshold or the maximum number of generations is exceeded. Both cases lead to NEAT finishing and the brain proceeding with the SAC algorithm.

6.2. RL

The NAS method using reinforcement learning is based on the 'minimal-nas' implementation by nicklashansen[6]. A controller class contains the reinforcement learning algorithm, which itself uses a simple neural network with a single hidden layer. In order to get the network to be optimised, the controller generates a rollout. During the rollout, several steps are taken; in each step, the network returns an integer representing the next action: The action is used to determine the next layer of the network and is either a number of features in a layer, an activation function, or a stop. Steps are taken by the algorithm until either a maximum depth is reached, or a stop is returned. To penalise the creation of certain networks, a reward is adapted. This way, an early stop (leading to a network without hidden layers) is heavily penalised (-1) and having two layers of the same kind (two feature layer or two activation layers) directly after each other is mildly penalised (-0.1). After terminating the rollout, the list of actions is used to generate the new network. This is done by first adding an input layer with use case specific in features, then iterating over the actions and adding the corresponding layer or stopping respectively, and finally adding an output layer.

Each network is run several times: The `nas_update`-method mentioned before calls the RL NAS's run method to create a new network every time the step counter is a multiple of the amount of `runs_per_network` times the SAC implementation's `update_every` parameter. Every time the step counter is only a multiple of the `update_every` parameter, the network is trained with the normal SAC algorithm's training method. This way, the network gets trained `runs_per_network`-times before the run-method is called with a loss value calculated from the network's runs. The loss value is adapted into a reward value that is higher the nearer the loss is to 0, with 2 being the highest value; it is then added to the controller's internal reward of the network and used to optimise the reinforcement learning algorithm. The external reward coming from the environment is getting saved in a dictionary which maps the network to the reward and is used to select the best performing network to be used in the SAC algorithm after the NAS finishes.

[6] https://github.com/nicklashansen/minimal-nas

The RL NAS finishes when a set number of networks were created and run through the whole process.

6.3. BO

The Bayesian Optimisation NAS method is based on the python 'bayesian-optimization' implementation[7]. For BO, a function to optimise – the black box function – is needed. In order to let BO generate a usable network, it has to be encoded in a way that can be used as such a black box function. In this use case, the network is encoded as six parameters each reaching from 0 to 256, depicting the number of features in the corresponding layer; a 0 means that the layer is skipped. The rewards of each network accumulated during their runs are used to tell BO how well the network performed, which in turn uses the info to step the search in the right direction and propose a new set of parameters for the black box function and network respectively.

Like the RL NAS, the BO NAS's network is trained every `update_every` steps with the SAC algorithm and exchanged when the network was trained `runs_per_network` times. Besides the black box function, the BO NAS has two more settable parameters: `init_points` and `n_iter`. The former is the number of random points to probe by BO before starting the optimisation. The latter is the number of total iterations to run the optimisation, leading to BO NAS finishing as the number is reached.

7. Experiment setup

Eight experiments were conducted to test the performance of the NAS methods. Two different scenarios were used, each with the three different NAS approaches as well as a baseline using the default SAC implementation without NAS.

7.1. Scenarios

In every scenario, an AI agent is tasked with keeping the power grid stable. A reactive power controller, realised as a muscle in palaestrAI and thus named reactive power muscle (RPM), is also present in every scenario.

7.1.1. 1. Scenario: CIGRE

This scenario uses the CIGRE Medium Voltage grid model of [3], also depicted

[7] https://github.com/bayesian-optimization/BayesianOptimization

in Figure 7. The ‘defender’ agent is given the ARL defender objective[8], which considers the overall mean voltage of the grid, the agent’s voltage sensor values, as well as the number of busses in service to calculate the reward.

7.1.2. 2. Scenario: CIGRE + COHDARL

In this scenario, the grid model used is the same as in the first scenario. The difference to the first scenario is the usage of COHDARL. COHDARL is a distributed heuristic that applies self-organization mechanisms to optimize a global, shared objective. It is used to optimize the scheduling of energy resources in virtual power plants and operates by representing each distributed energy resource as a self-interested agent, allowing both global scheduling objectives and individual local objectives to be efficiently integrated into a distributed coordination paradigm [6]. Consistent with the usage of COHDARL, the ‘defender’ agent is given the COHDARL defender objective[9], whose reward is calculated based on the global objective of COHDARL.

7.2. Parameters

The experiment’s parameters are shown in Table 1 and the whole experiment file can be found in the ‘nas’-branch of harl. Every scenario is run for ten episodes, each representing one year, which is equivalent to 31,536,000 seconds. A step size of 900 seconds is used, resulting in 35,040 steps per episode.

Every scenario is run for ten episodes, each representing a hundred days, which is equivalent to 8,640,000 seconds. A step size of 900 seconds is used, resulting in 9,600 steps per episode or 28,800 steps for the whole simulation. As reward function, the ExtendedGridHealthReward[10], a reward based on the grid’s “healthiness” – meaning the deviation from the best possible status –, is used.

The parameters for SAC shown in Table 2 are mostly kept to the default values of the implementation. In the baseline experiment, the muscle’s `start_steps`-parameter, which determines the number of random actions taken before SAC chooses the actions, plus the `update_after`-parameter, which determines the amount of steps taken before SAC starts to learn, are both set to `1000`. During the experiments with NAS, both parameters are set to `0`. This is because the NAS

[8] https://gitlab.com/midas-mosaik/midas-palaestrai/-/blob/main/src/midas_palaestrai/arl_defender_objective.py

[9] https://gitlab.com/midas-mosaik/midas-palaestrai/-/blob/main/src/midas_palaestrai/cohdarl_defender_objective.py

[10] https://gitlab.com/midas-mosaik/midas-palaestrai/-/blob/main/src/midas_palaestrai/rewards.py#L88

methods have their own way of exploring the search spaces and thus do not need the random actions taken by SAC. The `batch_size`-parameter, which determines the size of the minibatches for SAC's stochastic gradient descent, is set to 1000 in the baseline experiment and kept this way for the NAS experiments, to have similar learning as the baseline after NAS finishes. `fc_dims`, determining the size of the fully connected layers, is set to `[48, 48]` in the baseline experiment; this parameter is adopted for the NAS experiments, but has no influence on the NAS methods since they each use their own networks.

The parameters for the NAS methods are shown in Table 3, Table 4, and Table 5. They are mostly kept to the default values of their implementations. NEAT's threshold is increased to a value of 1000, whilst the number of generations is set to 15. This increase to an unreachable threshold is done to ensure that NEAT finishes after the maximum number of generations is reached and not finishing early due to a genome exceeding an arbitrary set threshold, because a good value for the threshold was not known prior to starting the experiments. The RL approach's `INDEX_TO_ACTION`-dictionary was expanded with additional values for the hidden sizes to allow for bigger networks.

Parameters that were established for this thesis's implementation like NEAT's `runs_per_genome` or RL's `runs_per_network` as well as parameters like BO's `N_ITER` that define the algorithm's duration were adapted to the experiment's runtime and scaled accordingly. This leads to every NAS method running for a similar amount of time – 25.000 steps in total – with the remaining steps being run with the network that performed best during the NAS method's runtime.

8. Results

9. Further work

The results of the experiments show (good/bad/meh) … (Despite that/Thus) there are several opportunities to improve upon the NAS methods …

One of those potential improvements is NEAT's network; the implementation of the NEAT algorithm uses a unique feed forward network created after NEAT's genomes. Because it is based on torch, it is possible to use it in the harl framework. However, due to the difference in structure compared to the network used in harl's normal SAC implementation, it is not possible to continue with a normal training process after NEAT has finished. This network difference is also

presumed to be the reason for a (at the moment of writing) unidentifiable and untraceable problem with the “CIGRE + COHDARL”-experiment: an experiment run started finishes right after the first step, whilst the same experiment with other NAS methods run the whole distance as expected. This problem is still under investigation, but was circumvented by reducing the amount of actuators the agent has control over. If further work on the NEAT implementation is deemed to be worth the while, a possible starting point could be the change to a network structure similar to the one used in harl’s SAC implementation. Besides the possible erasure of this and similar problems and the possibility of a subsequent training of the network, this also increases the reusability of the network. Despite the advantages, this adaption is a non-trivial task and possibly requires a lot of work, because one has to make sure the conversion of NEAT’s genomes functions as intended and like the current unique network structure.

The bayesian optimisation approach to the NAS has a big room for improvement, as well. Currently, the devised black box function is rather simple by having six parameters, which represent the amount of features in the layer, and can assume values between 0, in which case no layer is employed, and 256. This leads to a network with six layers each having 128 features on average, which is a rather big network. By changing the black box function to a more complex one and/or one that is more tuned to the problem at hand, the performance of this NAS method is believed to be greatly improved upon.

Other opportunities for enhancement of the NAS methods’ performances is the optimisation of the parameters. At the moment, the parameters are mostly set to the default values of the implementations or set to values which are assumed to be good. Consultation of literature or usage of common practices like computational methods could lead to a better performance of the NAS methods.

Lastly, the implementation of other algorithms could also bring improvements. The implementations in this work have lead to an existing foundation to build upon, by having examples of three different NAS methods already connected to the harl framework. Thus, a good chunk of the implementation work is already done and a new NAS method with possibly greater performance could be implemented with less effort.

10. Conclusion

Appendix

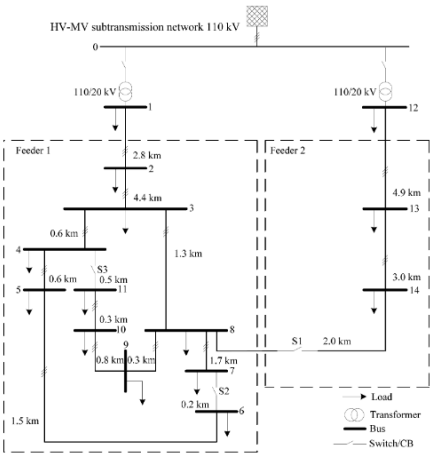

Figure 7: CIGRE benchmark grid used [3]

Parameter	Value
Net	CIGRE
Amount Steps	100 Days (8.640.000 seconds)
Episodes	3
Reward Function	ExtendedGridHealthReward
Step size	900

Table 1: Experiment parameters

Objective	ArlDefenderObjective/ COHDARLObjective
batch_size	1000
fc_dims	[48, 48]
gamma	0.99
learning rate	0.003
replay_size	1e6
update_after	1000 | 0
update_every	25
muscle start_steps	1000 | 0

Table 2: SAC parameters

	Parameter	Value
Genome	Runs per genome	20
	NUM_INPUTS	Amount of Sensors
	NUM_OUTPUTS	Amount of Actuators
	USE_BIAS	True
	ACTIVATION	ReLu
	SCALE_ACTIVATION	4.9
Population	FITNESS_THRESHOLD	1000
	POPULATION_SIZE	100
	NUMBER_OF_GENERATIONS	15
	SPECIATION_THRESHOLD	3.0
Algorithm	CONNECTION_MUTATION_RATE	0.80
	CONNECTION_PERTURBATION_RATE	0.90
	ADD_NODE_MUTATION_RATE	0.03
	ADD_CONNECTION_MUTATION_RATE	0.5
	CROSSOVER_REENABLE_CONNECTION_GENE_RATE	0.25
	PERCENTAGE_TO_SAVE	0.30
	FITNESS_FUNC	AVG

Table 3: NEAT parameters

Parameter	Value
NAS runs	50
Runs per network	20
NUM_ACTIONS	16
INDEX_TO_ACTION	0: 1, 1: 2, 2: 4, 3: 8, 4: 16, 5: 32, 6: 64, 7: 128, 8: 256, 9: 512, 10: 1024, 11: “Sigmoid”, 12: “Tanh”, 13: “ReLU”, 14: “LeakyReLU”, 15: “EOS”
HIDDEN_SIZE	64
EPSILON	0.8
GAMMA	1.0
BETA	0.01
MAX_DEPTH	6
CLIP_NORM	0

Table 4: RL based NAS parameters

Parameter	Value
Runs per network	20
INIT_POINTS	20
N_ITER	30

Table 5: BO parameters

Bibliography

[1] Glossar.ca, 'Artificial neural network with layer coloring'. 2013.

[2] A. G. Barto, S. Singh, N. Chentanez, and others, 'Intrinsically motivated learning of hierarchical collections of skills', in *Proceedings of the 3rd International Conference on Development and Learning*, 2004, p. 19.

[3] K. Rudion, A. Orths, Z. Styczynski, and K. Strunz, 'Design of benchmark of medium voltage distribution network for investigation of DG integration', in *2006 IEEE Power Engineering Society General Meeting*, 2006, p. 6. doi: 10.1109/PES.2006.1709447.

[4] E. Veith, *Universal smart grid agent for distributed power generation management*. Logos Verlag Berlin, 2017.

[5] E. Frost, E. M. Veith, and L. Fischer, 'Robust and deterministic scheduling of power grid actors', in *2020 7th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2020, pp. 100–105.

[6] C. Hinrichs and M. Sonnenschein, 'A distributed combinatorial optimisation heuristic for the scheduling of energy resources represented by self-interested agents', *International Journal of Bio-Inspired Computation*, vol. 10, no. 2, pp. 69–78, 2017.

[7] O. P. Mahela *et al.*, 'Comprehensive overview of multi-agent systems for controlling smart grids', *CSEE Journal of Power and Energy Systems*, vol. 8, no. 1, pp. 115–131, 2020.

[8] S. Holly *et al.*, 'Flexibility management and provision of balancing services with battery-electric automated guided vehicles in the Hamburg container terminal Altenwerder', *Energy Informatics*, vol. 3, pp. 1–20, 2020.

[9] B. A. Hamilton, 'When the lights went out: Ukraine cybersecurity threat briefing', *http://www. boozallen. com/content/dam/boozallen/documents/2016/09/ukraine-report-when-the-lights-wentout. pdf, checked on*, vol. 12, p. 20, 2016.

[10] A. Aflaki, M. Gitizadeh, R. Razavi-Far, V. Palade, and A. A. Ghasemi, 'A hybrid framework for detecting and eliminating cyber-attacks in power grids', *Energies*, vol. 14, no. 18, p. 5823, 2021.

[11] T. Wolgast, E. M. Veith, and A. Nieße, 'Towards reinforcement learning for vulnerability analysis in power-economic systems', *Energy Informatics*, vol. 4, pp. 1–20, 2021.

[12] E. Veith, N. Wenninghoff, S. Balduin, T. Wolgast, and S. Lehnhoff, 'Learning to Attack Powergrids with DERs', *arXiv preprint arXiv:2204.11352*, 2022.

[13] O. Hanseth and C. Ciborra, *Risk, complexity and ICT*. Edward Elgar Publishing, 2007.

[14] R. Diao, Z. Wang, D. Shi, Q. Chang, J. Duan, and X. Zhang, 'Autonomous voltage control for grid operation using deep reinforcement learning', in *2019 IEEE Power & Energy Society General Meeting (PESGM)*, 2019, pp. 1–5.

[15] L. Fischer, J.-M. Memmen, E. Veith, and M. Tröschel, 'Adversarial resilience learning-towards systemic vulnerability analysis for large and complex systems', *arXiv preprint arXiv:1811.06447*, 2018.

[16] E. M. Veith, L. Fischer, M. Tröschel, and A. Nieße, 'Analyzing cyber-physical systems from the perspective of artificial intelligence', in *Proceedings of the 2019 International Conference on Artificial Intelligence, Robotics and Control*, 2019, pp. 85–95.

[17] J. Schrittwieser *et al.*, 'Mastering atari, go, chess and shogi by planning with a learned model', *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[18] S. Fujimoto, H. Hoof, and D. Meger, 'Addressing function approximation error in actor-critic methods', in *International conference on machine learning*, 2018, pp. 1587–1596.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, 'Proximal policy optimization algorithms', *arXiv preprint arXiv:1707.06347*, 2017.

[20] T. Haarnoja *et al.*, 'Soft actor-critic algorithms and applications', *arXiv preprint arXiv:1812.05905*, 2018.

[21] P. Probst, A.-L. Boulesteix, and B. Bischl, 'Tunability: Importance of hyperparameters of machine learning algorithms', *Journal of Machine Learning Research*, vol. 20, no. 53, pp. 1–32, 2019.

[22] M. Wooldridge, *An introduction to multiagent systems*. John wiley & sons, 2009.

[23] M. Wooldridge, 'Intelligent agents', *Multiagent systems: A modern approach to distributed artificial intelligence*, vol. 1, pp. 27–73, 1999.

[24] P. G. Balaji and D. Srinivasan, 'An introduction to multi-agent systems', *Innovations in multi-agent systems and applications-1*, pp. 1–27, 2010.

[25] A. Dorri, S. S. Kanhere, and R. Jurdak, 'Multi-agent systems: A survey', *Ieee Access*, vol. 6, pp. 28573–28593, 2018.

[26] M. Amin and J. Stringer, 'The electric power grid: Today and tomorrow', *MRS bulletin*, vol. 33, no. 4, pp. 399–407, 2008.

[27] J. McCarthy and others, 'What is artificial intelligence', 2007.

[28] T. M. Mitchell and T. M. Mitchell, *Machine learning*, vol. 1, no. 9. McGraw-hill New York, 1997.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[30] J. A. Anderson, *An introduction to neural networks*. MIT press, 1995.

[31] E. Puiutta and E. M. Veith, 'Explainable reinforcement learning: A survey', in *International cross-domain conference for machine learning and knowledge extraction*, 2020, pp. 77–95.

[32] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, 'Deep reinforcement learning: A brief survey', *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[33] D. Silver *et al.*, 'Mastering the game of Go with deep neural networks and tree search', *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[34] W. Uther and M. Veloso, 'Adversarial reinforcement learning', 1997.

[35] J. R. Sampson, *Adaptation in natural and artificial systems (John H. Holland)*. Society for Industrial, Applied Mathematics, 1976.

[36] D. Whitley, S. Rana, J. Dzubera, and K. E. Mathias, 'Evaluating evolutionary algorithms', *Artificial intelligence*, vol. 85, no. 1–2, pp. 245–276, 1996.

[37] M. Feurer and F. Hutter, 'Hyperparameter optimization', *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.

[38] K. O. Stanley and R. Miikkulainen, 'Evolving Neural Networks through Augmenting Topologies', *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002, doi: 10.1162/106365602320169811.

[39] C. White *et al.*, 'Neural architecture search: Insights from 1000 papers', *arXiv preprint arXiv:2301.08727*, 2023.

[40] Y. Fu, Z. Yu, Y. Zhang, and Y. Lin, 'Auto-agent-distiller: Towards efficient deep reinforcement learning agents via neural architecture search', *arXiv preprint arXiv:2012.13091*, 2020.

[41] N. Mazyavkina, S. Moustafa, I. Trofimov, and E. Burnaev, 'Optimizing the neural architecture of reinforcement learning agents', in *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 2*, 2021, pp. 591–606.

[42] J. Parker-Holder *et al.*, 'Automated reinforcement learning (autorl): A survey and open problems', *Journal of Artificial Intelligence Research*, vol. 74, pp. 517–568, 2022.

[43] K. O. Stanley and R. Miikkulainen, 'Efficient evolution of neural network topologies', in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, 2002, pp. 1757–1762.

[44] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, 'A hypercube-based encoding for evolving large-scale neural networks', *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[45] S. Mirjalili and S. Mirjalili, 'Genetic algorithm', *Evolutionary algorithms and neural networks: theory and applications*, pp. 43–55, 2019.

[46] S. Mirjalili and A. Lewis, 'The whale optimization algorithm', *Advances in engineering software*, vol. 95, pp. 51–67, 2016.

[47] S. Elfwing, E. Uchibe, and K. Doya, 'Online meta-learning by parallel algorithm competition', in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 426–433.

[48] M. Jaderberg *et al.*, 'Population based training of neural networks', *arXiv preprint arXiv:1711.09846*, 2017.

[49] H. Bai, R. Cheng, and Y. Jin, 'Evolutionary Reinforcement Learning: A Survey', *Intelligent Computing*, vol. 2, no. , p. 25, 2023, doi: 10.34133/icomputing.0025.

[50] V. Dalibard and M. Jaderberg, 'Faster improvement rate population based training', *arXiv preprint arXiv:2109.13800*, 2021.

[51] J. K. Franke, G. Köhler, A. Biedenkapp, and F. Hutter, 'Sample-efficient automated deep reinforcement learning', *arXiv preprint arXiv:2009.01555*, 2020.

[52] X. Cui, W. Zhang, Z. Tüske, and M. Picheny, 'Evolutionary stochastic gradient descent for optimization of deep neural networks', *Advances in neural information processing systems*, vol. 31, 2018.

[53] O. Sigaud, 'Combining Evolution and Deep Reinforcement Learning for Policy Search: A Survey', *ACM Trans. Evol. Learn. Optim.*, vol. 3, no. 3, Sep. 2023, doi: 10.1145/3569096.

[54] A. Ofenloch *et al.*, 'Mosaik 3.0: Combining time-stepped and discrete event simulation', in *2022 Open Source Modelling and Simulation of Energy Systems (OSMSES)*, 2022, pp. 1–5.

[55] P. Hintjens and others, 'Ømq-the guide', *Online: http://zguide. zeromq. org/page: all, Accessed on*, vol. 23, 2011.

[56] T. Górski, 'UML profile for messaging patterns in service-oriented architecture, microservices, and internet of things', *Applied Sciences*, vol. 12, no. 24, p. 12790, 2022.

[57] B. Baker *et al.*, 'Emergent tool use from multi-agent autocurricula', in *International conference on learning representations*, 2019.

[58] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, 'Taking the human out of the loop: A review of Bayesian optimization', *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

Eidesstaatliche Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Unterschrift

Mario Fokken | Oldenburg, 20.03.2025