SAC NAS

Title page under construction

Contents

List of figures

1. Introduction

Over the last years, agent systems and especially Multi-Agent Systems (MASs) [3], [4], [5], [6] have emerged as one of the important tools to facilitate management of complex energy systems. As swarm logic, they can handle numerous tasks, such as maintaining real power equilibria, voltage control, or automated energy trading [7]. The fact that MASs implement proactive and reactive distributed heuristics allows to analyze their behaviour and give certain guarantees, a property that has helped in their deployment. However, modern energy systems have also become valuable targets. Cyber-attacks have become more common [8], [9], and establishing local energy markets, although being an attractive concept of self-organization, can also be to manipulation, e. g., through artificially created congestion [10]. Attacks on power grids are no longer carefully planned and executed, but also learned by agents, such as market manipulation or voltage band violations [11]. Thus, carefully designing software systems that provide protection against a widening field of adversarial scenarios have become a challenge, especially considering that complex, inter-connected Cyber-Physical Systems (CPSs) are inherently exploitable due to their complexity itself [12].
Learning agents, particularly those based on Deep Reinforcement Learning (DRL), have gained traction as a potential solution: If a system faces unknown unknowns, a learning agent can devise strategies against it. In the past, researchers have published using DRL-based agents for numerous tasks related to power grid operations—e.g. voltage control [13]—, but the approach to use DRL for general resilient operation is relatively new [14], [15]. DRL—the notion of an agent with sensors and actuators that learns by "trial and error"—is at the core of many noteworthy successes, such as MuZero [16], with modern algorithms such as Twin-Delayed DDPG (TD3) [17], Proximal Policy Gradient (PPO) [18], and Soft Actor Critic (SAC) [19] having proved to be able to tackle complex tasks. All modern DRL use deep Artificial Neural Networks (ANNs) at least for the policy (or multiple, e.g. for the critic). Actual parameter optimisation is commonly done with gradient descent algorithms. However, these ANNs' architectures still need to be provided by the user, in addition to the hyperparameters 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 [20]. 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 [21].

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). [22] Quelle auch vorm Punkt, wenn es für den ganzen Absatz gilt?

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. [23]

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. [24]

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. [25]

Albeit being a fairly new trend [14], [15], 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. [13]

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. [26]

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). [27]

Machine learning encompasses several types of learning paradigms, each suited to different types of problems and data: [28]

- 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. [27]

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

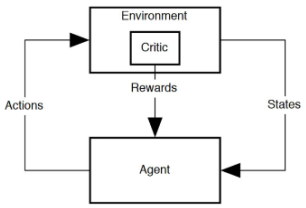

Figure 1: A basic representation of a neural network [1]

'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. [29]

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.

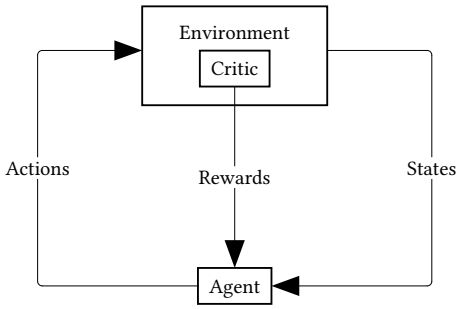

Figure 2: The feedback-loop of reinforcement learning algorithms [2]

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 [30].

2.3.2.3. History (Chat GPT, if useful improve else remove)

The history of Deep Reinforcement Learning combines advancements in both reinforcement learning and deep learning.

Around the end of the last century (1980s-2000s), early work in reinforcement learning focused on developing foundational algorithms such as Q-learning and temporal difference learning. These algorithms, however, struggled with high-dimensional state spaces due to their reliance on hand-crafted features and tabular methods.

Around the 2000s, parallel advancements in deep learning, driven by increased computational power and large datasets, led to the development of powerful neural networks capable of handling complex data. Convolutional Neural Networks (CNNs) and advancements in backpropagation were pivotal.

The integration of deep learning with reinforcement learning began to take shape around 2013, leading to the emergence of DRL. A seminal moment was the introduction of the Deep Q-Network (DQN) by researchers at DeepMind in 2013. DQN combined Q-learning with convolutional neural networks to play Atari games at superhuman levels, demonstrating the potential of DRL. This success spurred a wave of research and applications, leading to more sophisticated DRL algorithms like Deep Deterministic Policy Gradients (DDPG), Asynchronous Advantage Actor-Critic (A3C), and Proximal Policy Optimisation (PPO).

2.3.2.4. 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. [31]

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. [32] 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. [31]

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. [33]

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) [34], 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. [35]

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 [20].

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. [36]

2.5.2. Network Optimisation/ 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 [20]. 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) [37], 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. [38]

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 [38]. The other way around – using neural architecture search to improve reinforcement learning – is, however, not common. One of the works with this premise is [39], 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 [40]. 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:

[41] 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* [42] and *HyperNEAT* [43] are mentioned as working solutions, whilst for the HPO variants of the *Genetic Algorithm* (GA) [44], *Whale Optimisation* [45], *online meta-learning by parallel algorithm competition* (OMPAC) [46], and *population based training* (PBT) [47] were successfully used.

[48] 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 [49], SEARL [50]); and combinations of both evolutionary methods, like an evolutionary stochastic gradient descent (ESGD) [51].

[52] 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 (Temp until better name)

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 [SOURCE] is the execution framework. It offers packages to implement or interface to agents, environments, and simulators. The main concern of palaestrAI is the orderly and reproducible execution of experiment runs, orchestrating the different parts of the experiment run, and storing results for later analysis.

palaestrAI's Executor class acts as overseer for a series of experiment runs. Each experiment run is a definition in YAML format. Experiment run definitions are, in most cases, produced by running arsenAI on an experiment definition. An experiment defines parameters and factors; arsenAI them samples from a space filling design and outputs experiment run definitions, which are concrente instanciations of the experiment's factors.

ExperimentRun objects represent such an experiment run definition as is executed. The class acts as a factory, instanciating agents along with their objectives, environments with corresponding rewards, and the simulator. For each experiment run, the Executor creates a RunGovernor, which is responsible for governing 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 favor 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 only has to react to the step(·) method returning done. Complex simulations for CPSs are often realized as co-simulations, meaning that they couple domain specific simulators. Through co-simulation software packages like mosaik (Ofenloch et al., 2022), these simulators can exchange data; the co-simulation software synchronizes these simulators and takes care of proper time keeping. This, however, means that palaestrAI's agents act just like another simulator from the perspective of the co-simulation software. The flow of execution is controlled by the co-simulator.

palaestrAI's loose coupling is realized using ZeroMQ (Hintjens, 2023), which is a messaging system that allows for a reliable request-reply patterns, such as the majordomo pattern (Górski, 2022; Hintjens, 2023). palaestrAI starts a message broker (MajorDomoBroker) before executing any other command; the modules then either employ a majordomo client (sends a request and waits for the reply), or the corresponding worker (receives requests, executes a task, returns a reply). Clients and workers subscribe to topics, which are automatically created on first usage. This loose coupling through a messaging bus enables the co-simulation with any control flow.

In palaestrAI, the agent is split into a learner (Brain) and a rollout worker (Muscle). The muscle acts within the environment. It uses a worker, subscribed to the muscle's identifier as topic name. During 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), supplying state and reward, requesting an update to its policy. Only then does the muscle infer actions, which constitute the reply to the request to act. In case of DRL brains, the algorithm trains when experiences are delivered 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. I. e, a trajectory is always given by a state, followed by an action, which then results in 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, τ = (s0, a0, …). Thus, the control flow as it is realized 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 synchronizes 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 results storage facilities. Currently, SQLite for smaller and PostgreSQL for larger simulation projects are supported, through SQLalchemy [https://www.sqlalchemy.org/, retrieved: 2023-01-04.]. 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 shared 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 allow for flexibility in offline learning or adversarial learning through autocurricula (Baker et al., 2020). Within a phase, agents can be employed in any combination and any sensor/actuator mapping. Moreover, 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 (If own chapter worth elaborate else remove)

https://gitlab.com/arl2/harl

harl 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. [19]

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. [37]

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 approximation 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. [53]

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.

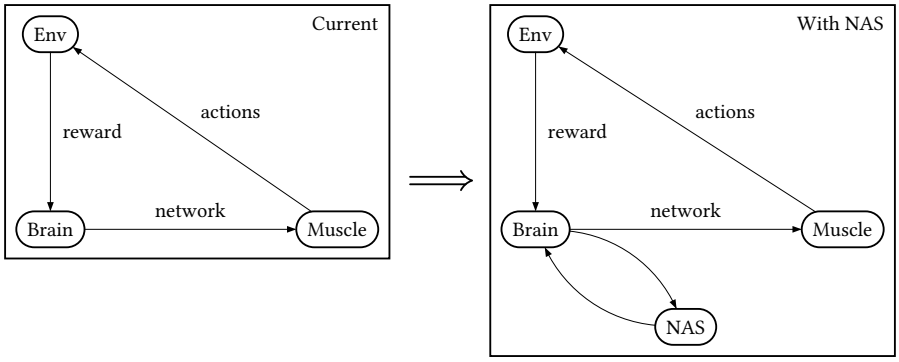

Figure 3: Adding NAS to the palaestrAI cycle

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 replaced by NAS for the duration of the search. 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 is finished, the original learning method replaces NAS to resume normal learning behaviour.

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

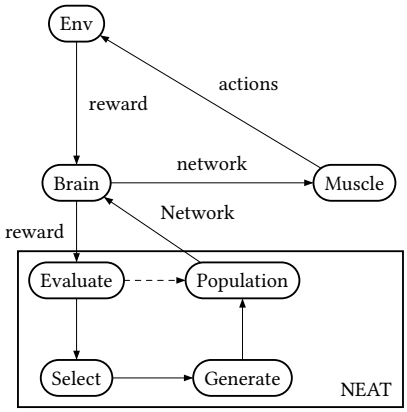

Figure 4: Using NEAT as NAS

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, but further improvement with the normal learning process is still possible.

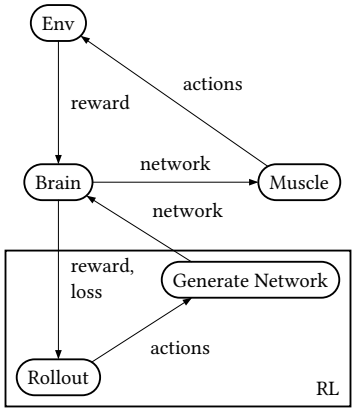

Figure 5: Using a RL approach as NAS

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 a network, which is then run through the cycle. 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. By repeating the cycle, the algorithm should be able to generate networks yielding better results in the environment. The search finishes after a set amount of steps; afterwards the latest – and ideally best – network is run through the normal learning process, where its weights are improved.

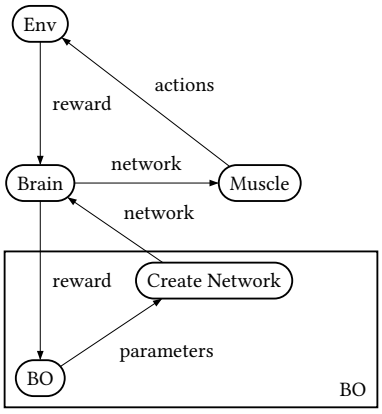

Figure 6: Using BO as NAS

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, the network is run through the normal learning process after the search is finished to improve the weights.

6. Implementation

For the implementation, each of the NAS methods is implemented in a separate Python class. The Python 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. Due to wanting all important parameters to be adaptable by the user, this dictionary is taken from the .yaml-file, which is used in palaestrAI to configure an experiment. If the user does not provide a parameter, the standard values are used.

Secondly, the method to return the initial network is used to get a network for the setup of the SAC actor in the brain. This is put into a method to provide a similar usage of each NAS method.

Finally, the 'run'-method is used to run the NAS algorithm. It gets called in every turn of the brain. The method receives the last reward and saves it in a way that allows to get the reward of the network using its network-describing properties. To ensure the network's reward corresponds the network's properties, each network is run for several steps. This way, a network that would perform poorly in the long run will not get a high reward due to a lucky natural fluctuation of the powergrid or vice versa. The 'run'-method returns a network, if a new one shall be run, and whether the NAS algorithm has finished, in which case the brain will stop calling the NAS and proceed with the SAC algorithm's learning.

Each of the NAS-methods also has a file describing the network. These contain a class for the NAS actor as well as a class for this actor's network. Both classes are heavily based on the network implementation of SAC in harl[1]. 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.

6.1. NEAT

The implementation of NEAT is PyTorch-NEAT, a GitHub repository by ddehueck[2]. 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.

6.2. RL

The reinforcement learning based NAS method is based on the 'minimal-nas' implementation by nicklashansen[3]. A controller class contains the reinforcement learning algorithm; each time a new network is to be run, the controller is called to optimise its algorithm's network and generate a rollout. The 'run'-method of the NAS receives a loss value in addition to the reward. This loss value is used in optimising the controller's network, whilst the reward is used to determine the network's performance, in order to be able to select the best performing network. The rollout consists of a list of actions, which can be either a number of features in a layer, an activation function, or a stop. A net is generated with these actions by first adding an input layer, then iterating over the actions and adding the corresponding layer or stopping respectively and finally adding an output layer.

6.3. BO

The Bayesian Optimisation NAS method is based on the python 'bayesian-optimization' implementation[4]. 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 it can be used as suck a black box function. In this use case, the network is encoded as six parameters each reaching from 0 to 256,

[1] https://gitlab.com/arl2/harl/-/blob/development/src/harl/sac/network.py?ref_type=heads
[2] https://github.com/ddehueck/pytorch-neat/
[3] https://github.com/nicklashansen/minimal-nas
[4] https://github.com/bayesian-optimization/BayesianOptimization

depicting the number of features in the corresponding layer. A 0 means that the layer is skipped. The reward of each network is 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 net set of parameters for the black box function and network respectively.

7. Experiment setup

8. Evaluation

9. Conclusion

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

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

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

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

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

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

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

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

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

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

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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.

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

[48] 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.

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

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

[51] 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.

[52] 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.

[53] 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.