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, 27.03.2025

Abstract

Agent systems have become more and more important in the management of complex energy systems. To ensure the security of these systems, it is necessary to protect them from cyber-attacks. Learning agents, particularly those based on deep reinforcement learning (DRL), have shown great potential in this area. All modern DRL algorithms use deep artificial neural networks (ANNs), whose architecture has a great influence on the performance and quality of the learning. However, the choice of the architecture is still mostly up to the user, for whom the task of finding the optimal architecture for the task at hand can be challenging. Thus, three Neural Architecture Search (NAS) algorithms were implemented to automate the choice of the architecture. The algorithms (NEAT, Bayesian optimisation, and a reinforcement learning approach) were tested as agents tasked to stabilise a simulated powergrid in the adversarial resilience learning (ARL) architecture of the palaestrAI framework. All three algorithms failed at learning in the task and performing better than the baseline, which suggests a faulty implementation or a too complex task for these NAS algorithms.

Contents

List of figures

1. Introduction

In recent years, agent systems, and in particular multi-agent systems (MAS) [1], [2], [3], [4], 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 [5]. 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 [6], [7], 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 [8]. Attacks on power networks are no longer carefully planned and executed, but also learned by agents, e.g. market manipulation or voltage violations [9]. 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 [10]. 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 [11] - but the approach of using of using DRL for general resilient operation is relatively new. [12], [13]. 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 [14], with modern algorithms such as such as Twin-Delayed DDPG (TD3) [15], Proximal Policy Gradient (PPO) Policy Gradient (PPO) [16], and Soft Actor Critic (SAC) [17] 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 choice of the architecture of these neural networks has a great influence on the performance and quality of the learning, as shown e.g. by [18]. But despite its importance, the choice is still mostly up to the user, 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; thus the user may choose by themself or stay with the standard parameters, which are not adapted to the current task. Both cases can result in a subpar choice of a architecture, thus leading to an unsatisfying result. Additionally, because the agents are part of a critical infrastructure, the choice of architecture has to be reasoned upon.

A solution to the problem may be to automate the choice of the architecture by the use of neural architecture search (NAS) algorithms, which have shown its worth in the field of machine learning, but mostly for image recognition tasks [19], [20]. This approach may be adaptable to agents tasked to stabilise the powergrid as part of the adversial resilience learning (ARL) approach.

This leads to this work's hypothesis that choosing a neural network's architecture can be automated with a NAS algorithm, which does this task in a way that improves upon the performance of a user picked architecture in a reasonable amount of time in the ARL architecture.

2. Basics

2.1. Agents

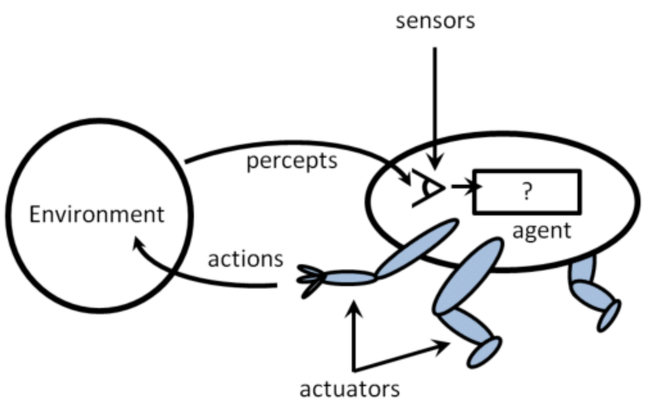


Figure 1: A depiction of an agent with its core functions
after Russell and Norvig [21][1]

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

[1] https://www.researchgate.net/figure/ntelligent-Agents-Russell-and-Norvig-2009_fig2_303545602

taking the initiative in order to satisfy their design objectives; and social ability, by interacting with other agents (and possibly humans). [23]

Russell and Norvig [21] group intelligent agents into five classes based on their capabilities and perceived intelligence:

Simple reflex agents
These agents act only upon what they perceive in the current state of the environment, and not on past states; a simple if condition then action approach. They only succeed when the environment is fully observable and infinite loops may only be avoided if the agent is capable of randomising its actions.

Model-based reflex agents
By storing its current state, which cannot be seen by the agent, inside, the agent gets a model of the world and knows “how the world works”. Like this, it is able to handle partially observable environments. The agent chooses its actions in the same way as the simple reflex agent.

Goal-based agents
Goal-based agents expand the model-based agents by using a “goal” information. The goal information describe situations that are desirable for the agent and the agent tries to reach these goals with its chosen actions. While sometimes less efficient, goal-based agents are more flexible as their decision-making knowledge is explicitly represented and modifiable.

Utility-based agents
Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is by using a utility function, which maps a state to a measure of the utility of the state. They choose actions that maximise their expected utility, allowing for more nuanced decision-making in complex environments.

Learning agents
Learning enables agents to operate in unknown environments and improve beyond their initial knowledge. The key components are the “learning element”, which makes improvements, and the “performance element”, which selects actions. The learning element uses feedback from the “critic” to adjust the performance element for better future actions. The “problem generator” suggests actions to gain new and informative experiences.

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. The distribution grid faces the challenge of integrating distributed generation (DG) in the form of renewable energy sources like wind and solar, which are variable by nature. The main impacts of DG in the distribution grid are power flow inversion, voltage increase, overcurrent, and the risk of overheating the cables. Furthermore, the increasing use of electric vehicles (EVs) and heat pumps can further strain the distribution grid by increasing peak demand. Thus, monitoring the distribution grid is essential, but currently not done enough. [24], [25], [26]

Power generation primarily relies on electromagnetic induction, where a changing magnetic field creates a current in a conductor. Large-scale generation uses mechanical motion to provide a consistent changing magnetic field. The rotating magnetic core (rotor) induces current in the fixed wire (stator), which interfaces with the power transmission system. The rotation of the magnetic core causes the current to alternate, with the grid frequency fixed (e.g. 50 Hz in continental Europe). Deviations from this frequency can damage the grid and connected components. In alternating current, there are three types of power.
Real Power P is the “visible force” that makes a motor run.

$$P = U \cdot I \cdot \cos\phi$$

Every conductor that carries power maintains an electric field. With alternating currents, these fields are created and inverted 50 times a second, which requires

its own power. This power does not do actual, visible work, such as driving a motor; it “works” only to maintain the EM field.
This is called reactive power (Q).

$$Q = U \cdot I \cdot \sin\phi$$

Real and reactive power are shifted by the phase angle, denoted by φ. Coils and conductors change this angle.
Real and reactive power combined give the apparent power S:

$$S = U \cdot I = \sqrt{P^2 + Q^2}$$

A power flow study calculates how real and reactive power flow. The nodes in a grid are called buses and can be classified into three types:

- Generators supply real power (P) and voltage (V); they are called *PV buses*.
- Load buses consume real and reactive power (Q); they are called *PQ buses*.
- At a special bus, called the slack bus, the voltage and phase angle are known; it is therefore the *VD bus*.

The study is based on Kirchhoff’s Law stating that the sum of all currents at a node must be 0. The flow of current is defined by the voltage difference between i and its k-th neighbour as well as the admittance of the grid elements between these two nodes. $\underline{I}_i = \sum_{k=1}^{n} \underline{I}_{ik} = \sum_{k=1}^{n} (\underline{V}_i - \underline{V}_k)\underline{Y}_{ik}$

The equations can be reformulated to a matrix equation $I = Y \cdot V$, which constitutes a system of non-linear functions. To solve this the Newton-Raphson method is often used – especially in the case of power flow study. The method iteratively refines an approximation:

$$x_{t+1} = x_t J_t^{-1}[y - f(x_i)]$$

[27]

2.3. Artificial Intelligence

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. [28]

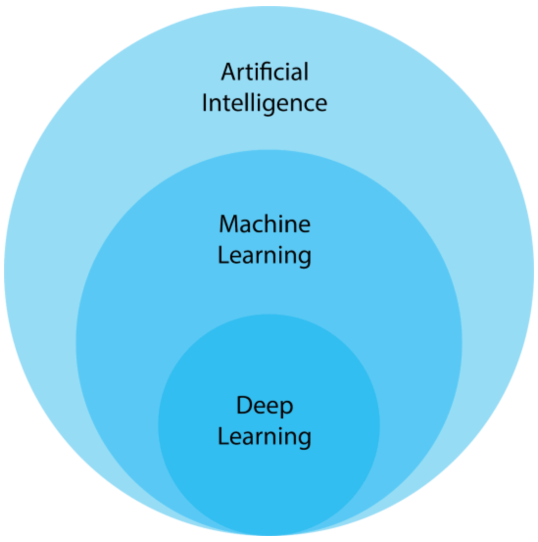


Figure 2: Venn diagram illustrating the relationship between artificial intelligence, machine learning, and deep learning.[2]

2.4. Machine Learning

As shown in Figure 2, Machine learning (ML) is a subset of artificial intelligence (AI) and it 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). [29]

Machine learning encompasses several types of learning paradigms, each suited to different types of problems and data: [30]

[2]https://www.researchgate.net/figure/Fig-1-Venn-diagram-showing-relationship-between-Artificial-Intelligence-Machine_fig1_362150447

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. [29]

2.4.1. Independent and identically distributed random variables

The independent and identically distributed (iid) assumption is a fundamental concept in statistics and machine learning. It means that given a set of data $\{x_i\}$, each of these x_i observations is an independent draw from a fixed probabilistic model. Independent means that the probability of observing two values x_1 and

x_2 is the product of the probabilities of observing each value separately. [31] This assumption simplifies the analysis of data and allows for the application of various statistical methods. In machine learning, the iid assumption is often used in the context of training and testing datasets. When the data points are drawn independently from the same distribution, the iid assumption ensures that the model's performance on the training set generalises well to unseen data.

2.4.2. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a popular optimisation algorithm used in machine learning and a modifications of gradient descent.

To explain gradient descent following example is used:
Taking a loss function $\ell(\hat{y}, y)$ that measures the difference between the predicted output $\hat{y}$ and the true output y for a given input x and a family $\mathcal{F}$ of functions $f_\omega(x)$ parameterised by a weight vector ω. A function $f \in \mathcal{F}$ that minimises the loss averaged on the examples is sought. The empirical risk function $E_n(f)$ measures the training set performance:

$$E_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

To minimise the empirical risk $E_n(f_\omega)$ using gradient descent, each iteration updates the weights ω on the basis of the gradient of $E_n(f_\omega)$:

$$\omega_{t+1} = \omega_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_\omega \ell\Big(f_{\omega_t}(x_i), y_i\Big),$$

where γ is an adequately chosen gain.

SGD drastically simplifies this process. Instead of computing the gradient of $E_n(f_\omega)$ exactly, each iteration estimates this gradient on the basis of a single randomly picked example (x_t, y_t):

$$\omega_{t+1} = \omega_t - \gamma \nabla_\omega \ell\Big(f_{\omega_t}(x_t), y_t\Big).$$

The stochastic process $\{w_t, t = 1, ...\}$ depends on the examples being randomly picked at each iteration. It is hoped that the SGD calculation behaves like the one in gradient descent. Because the stochastic algorithm does not need to remember which examples were visited in previous iterations, it can process examples on the fly in a deployed system. [32]

2.4.3. Adaptive Moment Estimation

Adaptive Moment Estimation (Adam) is an optimisation algorithm that combines the benefits of two other popular optimisation algorithms: AdaGrad (adaptive gradient algorithm) [33], and RMSProp (Root Mean Square Propagation) [34], both of which improve upon the SGD with a per-parameter learning rate.

Adam computes adaptive learning rates for each parameter, as well, by storing exponentially decaying averages of past squared gradients and past gradients. This allows Adam to adapt the learning rate during training, making it well-suited for a wide range of machine learning tasks. The algorithm is known for its robustness, efficiency, and ease of use, and it has become a popular choice for training deep neural networks.
In every timestep t, Adam's parameter update is calculated as follows:

$$\begin{aligned} m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot (\nabla_\theta f_t(\theta_{t-1})) \\ v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla_\theta f_t(\theta_{t-1}))^2 \\ \alpha_t &= \alpha \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \\ \theta_t &\leftarrow \theta_{t-1} - \alpha_t \cdot \frac{m_t}{\sqrt{v_t} + \hat{\epsilon}} \end{aligned}$$

$f(\theta)$ is a noisy objective function that is differentiable and whose expected value $\mathbb{E}[f(\theta)]$ is to be minimised w.r.t. its parameters θ. The algorithm updates exponential moving averages of the gradient m_t and the squared gradient v_t, which are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively. The two hyperparameters $\beta_1, \beta_2 \in [0,1)$ control the exponential decay rates of these moving averages. ϵ is a small scalar (e.g. 10^{-8}) to prevent division by zero.

An important property of Adam's update rule is its careful choice of step sizes. It establishes a 'trust region' around the current parameter value, in order to prevent the algorithm from taking large steps in regions where the current gradient estimate does not provide sufficient information.

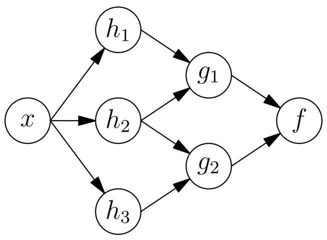
Adam's initialisation bias correction is another key feature. The algorithm corrects the bias of the first and second moment estimates by dividing them by $1-\beta_1^t$ and $1-\beta_2^t$, respectively. This ensures that the estimates are unbiased during the initial timesteps of training. [35]

Due to its robustness and efficiency, Adam has become a popular choice for training deep neural networks and is widely used in various machine learning applications. This popularity inspired the development of several variants and enhancements, like AdaMax [35], AdamW [36], and AdamX [37], but the original Adam algorithm is still used in popular sets of reinforcement learning algorithm implementations like Stable-Baselines3[3] or cleanRL[4].

2.5. Deep Reinforcement Learning

One branch of machine learning (Figure 2) 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.

2.5.1. Deep Learning



$$\begin{tikzcd} & h_1 \arrow[dr] & & \\ x \arrow[ur] \arrow[r] \arrow[dr] & h_2 \arrow[r] \arrow[dr] & g_1 \arrow[r] & f \\ & h_3 \arrow[r] & g_2 \arrow[ur] & \end{tikzcd}$$

Figure 3: A basic representation of a neural network[5]

'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 3. 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. [38]

[3] https://stable-baselines3.readthedocs.io/en/master/
[4] https://cleanrl.dev/
[5] https://commons.wikimedia.org/wiki/File:Ann_dependency_(graph).svg

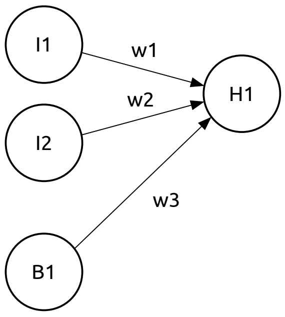


Figure 4: A representation of single hidden neuron with its inputs

The node H1 in Figure 4 has three inputs: I1, I2, and B1. I1 and I2 are simply the input values that the neural network was provided with to compute the output. B1 is a bias neuron and thus always 1. Bias neurons allow the output of the activation function to be shifted to the left or right on the x-axis.

The formula to calculate a neurons output generally is:

$$h_1 = A\left(\sum_{c=1}^{n}(i_c * w_c)\right)$$

with h_1 being the output of the neuron,
A being the activation function,
i_c being the output of the previous neuron I,
w_c being the weight of the connection,
and n the amount of connections.

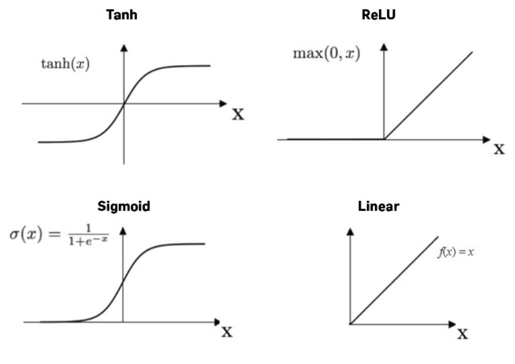


Figure 5: A selection of common activation functions with their formula[6]

Some common activation functions are displayed in Figure 5. Activation functions determine the output of a neuron and are used to introduce non-linearity to the neural network, thus making it capable of learning non-linear functions. [39]

[6]https://machine-learning.paperspace.com/wiki/activation-function

This way, every neuron calculates its output according to the formula and passes its output to the next neuron. With the output of the last neuron being the result of the network.

It has been shown that "multilayer feedforward networks are universal approximators". This means that any measurable function can be approximated to any desired degree of accuracy. A lack of success is due to inadequate learning, insufficient number of hidden units, or the lack of a deterministic relationship between input and target.[40]

2.5.2. Reinforcement Learning

As mentioned in Section 2.4, '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.

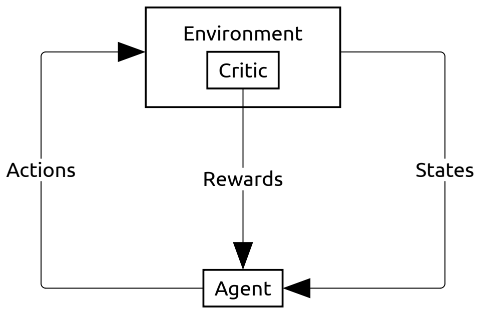


Figure 6: The feedback-loop of reinforcement learning algorithms)

In Figure 6, the general feedback loop for a reinforcement learning algorithm is illustrated. The environment is in a state, which is observed by the agent. The agent then decides upon an action, which is executed in the environment. The environment then transitions to a new state and gives feedback to the agent in the form of a reward. The agent uses this feedback to adjust its policy, which is the strategy it uses to decide upon actions. By adjusting the policy, the agent learns to take actions that maximise the rewards received. [41]

2.5.2.1. Applications

Reinforcement learning has many applications, such as in robotics [42], finance [43], and for this scope most relevant the energy sector, for example as a control system for powergrids [44]. As shown in [45], the application of reinforcement learning can be worthwhile, as many publications regarding the use of it in energy systems report a 10-20% performance improvement; however, many of these publications are not using state-of-the-art algorithms and instead half of them opt for Q-learning [45].

Another application of reinforcement learning is adversarial reinforcement learning, Adversarial reinforcement learning is an extension of traditional reinforcement learning techniques applied to multi-agent, competitive environments. In adversarial 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. [46] This approach can also be brought to the energy sector.

A survey [47] shows the possibilities of attacking machine learning agents deployed in smart grids with adversarial machine learning. [48] use the adversarial attacker to train and improve the robustness of their reinforcement learning agent controlling the power grid. A highly related work is [12], in which adversarial resilience learning (ARL) is introduced; agents take the roles of attackers or defenders that aim at worsening or improving – or keeping, respectively – defined performance indicators of the system, e.g. a simulated power system, and learn to adapt their strategies to the opponent's actions.

2.5.2.2. Bellman Equation

A central part of reinforcement learning is the Bellman equation, based upon Richard Bellman's principle of optimality [49]:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."
>
> — Richard Bellman

That means that the optimal policy consists of always picking the optimal action in a given state.

The Bellman equation can be defined as:

$$V^{\pi^*}(x) = \max_a \left[\mathfrak{R}_{x(a)} + \gamma \sum_y P_{\text{xy}}(a) V^{\pi^*}(y) \right]$$

Where:
x is the state, y a possible next state, a an action, π the policy,
V the value of a state, which is the long-term reward for being in that state,
$\mathfrak{R}_{x(a)}$ the immediate reward for taking action a in state x, and
$P_{\text{xy}}(a)$ the probability of transitioning to state y from state x by taking action a.

In order for a reinforcement learning agent to have an optimal policy, it has the task to find a policy π that maximises the total expected reward when in state x. The total expected reward is the immediate reward $\mathfrak{R}_{x(a)}$ received by taking action a plus the sum of all values V of the possible next states y multiplied by the probability P_{xy} of reaching that state y from current state x by taking the action a. The sum is discounted by a factor γ ($0 < \gamma < 1$), which determines the importance of future rewards compared to the immediate reward.

This way, by always taking the path with the highest expected reward, an agent is said to be able to find the optimal policy.

2.5.2.3. On-policy – Off-Policy

There are mainly two different approaches to reinforcement learning: on-policy and off-policy learning. On-Policy algorithms evaluate and improve the same policy that is also used to select actions. Off-Policy algorithms, however, have two different policies: the behaviour policy that is used to select actions and the target policy that is learned and improved. The former concept is a straightforward and simple way to learn a neural network policy. They also tend to be more stable and due to the simpleness easier to implement and use. A major drawback of on-policy algorithms is that they are inclined to be data inefficient, because they look at each data point only once. Not being able to reuse data points makes them unable to learn from past experiences, too. Off-Policy algorithms, on the other hand, use a memory replay buffer to store and reuse samples. This way, data effiency is improved, but often at a cost in stability and ease of use. [50]

2.5.2.4. Policy Gradient Method

Policy gradient methods are reinforcement learning approaches that directly optimise the policy. They are centered around a parametrised policy π_θ with parameters θ that allow the selection of actions a given the state s. The policy can either be deterministic $a = \pi_\theta(s)$ or stochastic $a \sim \pi_\theta(a|s)$.

The goal of the algorithm is to optimise the expected return

$$J(\theta) = Z_\gamma E\left\{\sum_{\mathrm{k}=0}^{H} \gamma^k r_k\right\},$$

where γ denotes a discount factor, Z_γ is a normalisation factor, H is the horizon, and r_k is the reward at time step k.

The policy gradient methods follow the gradient of the expected return

$$\theta_{\mathrm{k+1}} = \theta_k + \alpha_k \nabla_\theta J(\pi_\theta)|_{\theta=\theta_k}$$

Policy gradient approaches have the advantages that they often have fewer parameters for representing the optimal policy than value-based methods, and they are guaranteed to converge to at least a locally optimal policy. They also can handle continuous states and actions, and often even imperfect state information. However, they are slow to converge in discrete problems, global optima are not attained and they are difficult to use in off-policy settings. [51]

2.5.2.5. Q-Learning

An early off-policy reinforcement learning algorithm that uses the Bellman equation is Q-learning [52]. It defines Q values (or action-values) for a policy π:

$$Q^\pi(x,a) = \Re_{x(a)} + \gamma \sum_y P_{\mathrm{xy}}[\pi(x)]V^\pi(y),$$

where θ_k denotes the parameters after update k with initial policy θ_0 and α_k is the learning rate.

Meaning, the Q value is the expected discounted reward for executing action a at state x and following policy π therafter. Theoretically, by using the policy to always take the action with the highest Q value, the Q-learning agent is able to find the optimal policy. In Q-learning, the agent has a set of acts that it repeats during so called episodes: in the nth episod, the agent:

1. observes its current state x_n
2. selects and performs an action a_n
3. observes the following state y_n
4. receives an immediate reward r_n
5. adjusts its $Q_{\mathrm{n\text{-}1}}$ values using a learning factor α_n, according to

$$Q_n(x,a) = \begin{cases} (1-\alpha_n)Q_{\mathrm{n\text{-}1}}(x,a) + \alpha_n[r_n + \gamma V_{\mathrm{n\text{-}1}}(y_n)] & \text{if } x = x_n \text{ and } a = a_n, \\ Q_{\mathrm{n\text{-}1}}(x,a) & \text{otherwise,} \end{cases}$$

where $V_{\text{n-1}}(y) \equiv \max_b\{Q_{\text{n-1}}(y, b)\}$ is the best the agent thinks it can do from state y.

State	Action1	Action2	...	ActionN
A	0.5	−0.2	...	0.8
B	0.9	0.1	...	−0.3
...	...	...	...	...
Z	0.2	0.8	...	0.9

Table 1: An example of a look-up table for Q values

To represent the Q values $Q_n(x, a)$, the algorithm uses a look-up table like shown in Table 1 since other representations may not converge correctly. [52]

2.5.2.6. Deep Q-Network

An advancement of Q-learning is Deep Q-Network (DQN) [53], which combines Q-Learning with deep neural networks. Instead of the look-up table representation, which becomes impractical for large state spaces due to high memory usage, DQN uses a neural network to approximate the optimal Q values:

$$Q^*(s, a) = \max_a \mathbb{E}[r_t + \gamma r_{\text{t+1}} + \gamma^2 r_{\text{t+2}} + ... \ |s_t = s, a_t = a, \pi],$$

which is the maximum sum of rewards r_t discounted by γ at each time step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a).

It stores samples of experience (s_t, a_t, r_t, $s_{\text{t+1}}$) in a replay buffer at each time-step t in a data set $D_t = \{e_1, ..., e_t\}$. The Q-learning updates are applied on samples of experience drawn uniformly at random from the replay buffer.

The loss function, a function that measures the difference between the predicted and the actual target values, to quantify how well or poor the model has performed and to update the parameters of the neural network, is defined as:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q_{\text{target}}(s', a'; \theta_i^-) - Q(s, a; \theta)\right)^2\right]$$

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters – or rather weights – of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network's weights θ_i^- held fixed between individual updates and are only updated with the Q-network's parameters (θ_i) at fixed intervalls. [53]

2.5.2.7. Deep Deterministic Policy Gradients

While DQN can solve problems high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. To address these issues and to be able to solve continuous action spaces, Deep Deterministic Policy Gradients (DDPG) [54] was introduced, which combines the insights of DQN with the actor-critic approach. The actor-critic approach itself combines the policy gradient methods of Section 2.5.2.4 as the actor with the value-based methods of Section 2.5.2.5 as the critic, to take advantage of the strengths of both approaches.

Initially, DDPG randomly initialises the critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with their corresponding weights θ^Q and θ^μ.
It then repeatedly goes through episodes, each of which consists of first initialising a random process N for action exploration, then observing the current state s_t, followed by repeating the following steps a predetermined number of times:

1. Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ with N_t being the exploration noise
2. Execute action a_t and observe reward r_t and new state $s_{\mathrm{t+1}}$
3. Store experience (s_t, a_t, r_t, $s_{\mathrm{t+1}}$) in replay buffer R
4. Sample a random minibatch of N transitions (s_i, a_i, r_i, $s_{\mathrm{i+1}}$) from R
5. Set $y_i = r_i + \gamma Q'\big(s_{i+1}, \mu'\big(s_{i+1}|\theta^{\mu'}\big)|\theta^{Q'}\big)$
6. Update critic by minimising the loss $L = \frac{1}{N}\sum_i \big(y_i - Q(s_i, a_i|\theta^Q)\big)^2$
7. Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)}\ \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

8. Update the target networks:

$$\theta^{Q'} \leftarrow r\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow r\theta^\mu + (1-\tau)\theta^{\mu'}$$

This leads to DDPG finding solutions for Atari games in a factor of 20 fewer steps than DQN. [54]

2.5.2.8. TD3

While DDPG can sometimes achieve great performance, it is often brittle with respect to hyperparameters and other types of tuning. A common failure mode for DDPG is that the learned Q-function starts to dramatically overestimate Q-values, which then leads to the policy breaking because it exploits the errors in the Q-function. To address these issues, Twin Delayed Deep Deterministic Policy Gradient (TD3) [15] was introduced. TD3 applies three modifications to DDPG to increase the stability and performance with consideration of the function approximation error:

1. Clipped Double Q-Learning: TD3 uses two Q-functions to mitigate the overestimation of Q-values. The minimum of the two Q-values is used to compute the target value, resulting in the target update being:

   $$y_1 = r + \gamma \min_{i=1,2} Q_{\theta'_i}\big(s', \pi_{\phi_1}(s')\big),$$

   with $Q_{\theta'_i}\big(s', \pi_{\phi_1}(s')\big)$ being the Q-value of the target network $Q_{\theta'_i}$ for the next state s' and the action $\pi_{\phi_1}(s')$.

2. "Delayed" Policy Updates: The policy is updated less frequently than the Q-function, which prevents the policy from overfitting to the current Q-function. One policy update for every two Q-function updates is recommended.

3. Target Policy Smoothing: TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action:

   $$y = r + \gamma Q_{\theta'}\big(s', \pi_{\phi'}(s') + \epsilon\big), \epsilon \sim \operatorname{clip}(\mathcal{N}(0, \sigma), -c, c),$$

   with ε being the noise sampled from a clipped normal distribution.

These modifications lead to TD3 greatly improving both learning speed and performance of DDPG in a number of challenging tasks in the continuous control setting. [15]

2.5.2.9. SAC

Soft Actor Critic (SAC) is like DDPG and TD3 an off-policy actor-critic algorithm, but instead of only trying to maximise the expected return, it also maximises the entropy of the policy. That way, the algorithm strives to succeed at the task while acting as randomly as possible.

To factor in the entropy, it uses a maximum entropy objective which favours stochastic policies by adding the expected entropy of the policy over $\rho_{\pi(s_t)}$ to the objective:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t, a_t) \sim \rho_\pi}[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot\,|s_t))],$$

where α is the temperature parameter that determines the relative importance of the entropy term $\mathcal{H}$ against the reward and controls the stochasticity of the optimal policy. As the temperature approaches zero ($\alpha \to 0$), the maximum entropy objective reduces to the conventional maximum expected return objective.

The objective has several advantages like being incentivised to explore more widely, while abandoning unpromising routes. It can also capture multiple modes of near-optimal behaviour; in problem settings where multiple actions appear equally attractive, the policy will give equal probability mass to those actions. Lastly, it has been observed that it improves learning speed over other state-of-art algorithms, which use the conventional objective of maximising the expected sum of rewards, like DDPG and TD3.

```
SOFT ACTOR CRITIC
Input: $\theta_1, \theta_2, \phi$                              // Initial parameters
$\theta_1 \leftarrow \theta_1, \theta_2 \leftarrow \theta_2$                    // Initialise target network weights
$\mathcal{D} \leftarrow \emptyset$                                    // Initialise an empty replay pool
for each interation do
    for each environment step do
        $a_t \sim \pi_\phi(a_t|s_t)$.                       // Sample action from the policy
        $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$                  // Sample transition from the environment
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t), a_t, r(s_t, a_t), s_{t+1}\}$   // Store the transition in the replay pool
    end for
    for each gradient step do
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$   // Update Q-function parameters
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$            // Update policy parameters
        $\alpha \leftarrow \alpha - \delta \hat{\nabla}_\alpha J(\alpha)$             // Adjust temperature
        $\theta_1 \leftarrow \tau\theta_1 + (1 - \tau)\theta_1$           // Update target network weights
    end for
end for
Output: $\theta_1, \theta_2, \phi$                             // Optimised parameters
```

Figure 7: Pseudocode of the Soft Actor Critic algorithm

SAC uses a parameterised soft Q-function $Q_\theta(s_t, a_t)$ and a tractable policy $\pi_\phi(a_t|s_t)$, whose parameters are θ and ϕ, respectively. The Q-function can be modelled as a neural network and trained to minimise the Bellman residual, which is the difference between the left and right side of the Bellman equation, thus quantifying how well the Q-function satisfies the Bellman equation:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\frac{1}{2}\Big(Q_\theta(s_t, a_t) - \Big(r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\theta}}(s_{t+1})]\Big)\Big)^2\right],$$

where the value function is implicitly parameterised through the soft Q-function parameters via $V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)]$, and it can be optimised with stochastic gradients:

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(a_t, s_t)(Q_\theta(s_t, a_t) - \big(r(s_t, a_t) + \gamma\big(Q_{\bar{\theta}}(s_{t+1}, a_{t+1}) - \alpha \log\big(\pi_\phi(a_{t+1}|s_{t+1})\big)\big)\big).$$

The update uses a target soft Q-function with parameters $\bar{\theta}$ obtained as an exponential moving average of soft Q-function weights, which has been shown to stabilise training.

The policy parameters, however, are learned by directly minimising the expected Kullback-Leibler divergence

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}\Big[\mathbb{E}_{a_t \sim \pi_\phi}\left[\alpha \log\big(\pi_\phi(a_t|s_t)\big) - Q_\theta(s_t, s_t)\right]\Big].$$

By reparameterising the policy using a neural network transformation
$a_t = f_\phi(\epsilon_t : s_t)$, where ϵ is an input noise vector, sampled from some fixed distribution the actor's function is now:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D, \epsilon_t \sim \mathcal{N}}\left[\alpha \log \pi_\phi\big(f_\phi(\epsilon; s_t)|s_t\big) - Q_\theta(s_t, f_\theta(\epsilon_t; s_t))\right],$$

where π_ϕ is defined implicitly in terms of f_ϕ. The gradient of the equation can be approximated:

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \alpha \log\Big(\pi_{\phi(a_t|s_t)}\Big) + \Big(\nabla_{a_t} \alpha \log\Big(\pi_{\phi(a_t|s_t)}\Big) - \nabla_{a_t} Q(s_t, a_t)\Big)\nabla_\phi f_\phi(\epsilon_t; s_t),$$

where a_t is evaluated at $f_\phi(\epsilon_t; s_t)$.

Like TD3, SAC uses two Q-function with parameters θ_i, train them indepenently to optimise $J_Q(\theta_i)$ and use the minimum of both for the stochastic gradient and policy gradient.

The gradient for the temperature α are computed with the objective:

$$J(\alpha) = \mathbb{E}_{a_{t \sim \pi_t}}\left[-\alpha \log \pi_t(a_t|s_t) - \alpha \bar{\mathcal{H}}\right]$$

SAC has been shown to be robust and sample efficient enough to perform in real-world robotic tasks like underactuated walking of a quadrupedal robot and even outperforms other state-of-art algorithms such as TD3 on several continuous control benchmarks. [17], [55]

2.6. Neuroevolution

Neuroevolution is subfield within artificial intelligence and machine learning that consists of trying to trigger an evolutionary process to evolve neural networks. It uses evolutionary algorithms (EAs) to generate artificial neural networks, parameters, and rules.

The first neuroevolution algorithms appeared in the 1980s. Researchers had to decide on the neural architecture themselves, and the evolution of the weights was done by evolution instead of stochastic gradient descent (Section 2.4.2).

Evolutionary algorithms generally start with creating a population of iid individuals, like neural networks with random weights. Each of the individuals is evaluated by using it on the task at hand. Based on the quality of its performance, the individual is given a "fitness" score. The fittest individuals are selected to reproduce; the offspring is constructed by slightly altering the parents. The best performing individuals amongst the offsprings are selected as parents as well, and so forth. By repeating this process, each generation should have individuals better adapted to the task than the previous one. [56]

Later, the fixture of the topology was not needed anymore and TWEANN (Topology and Weight Evolving Artificial Neural Networks) algorithms were developed. These algorithms could change the topology of the neural network as well, i.e. by adding a connection during the creation of an offspring. The addition of the topolgy made the evolutionary algorithms more flexible and powerful.

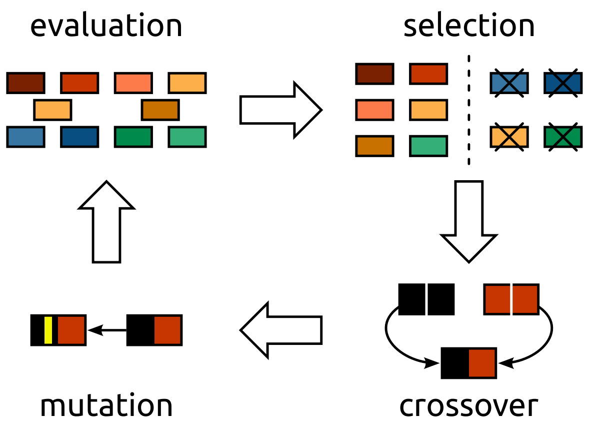


Figure 8: A diagram showing the general steps of a evolutionary algorithms[7]

Evolutionary algorithms may be classified into three subcategories, each inspired by different evolutionary theories.

[7]https://www.strong.io/blog/evolutionary-optimization

Darwinian Evolution

Darwinian Evolution algorithms 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.

Lamarckian Evolution

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.

Swarm Intelligence

Swarm Intelligence algorithms are 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. Basically, these algorithms have a swarm of agents that move around in the search space, guided by their own best-known position and the whole swarm's best-known position. When improved positions are found, the whole swarm is moved towards them. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered [57].

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. [58]

2.6.1. 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 [59]. The peculiarity of NEAT is that it addresses three major challenges of TWEANNs:

1. Is there a genetic representation that allows disparate topologies to crossover in a meaningful way?

2. How can topological innovation that needs a few generations to optimize be protected so that it does not disappear from the population prematurely?
3. How can topologies be minimised throughout evolution without the need for a specially contrived fitness function that measures complexity?

Compared to the traditional genetic algorithms, NEAT has three key differences:

Genetic Encoding

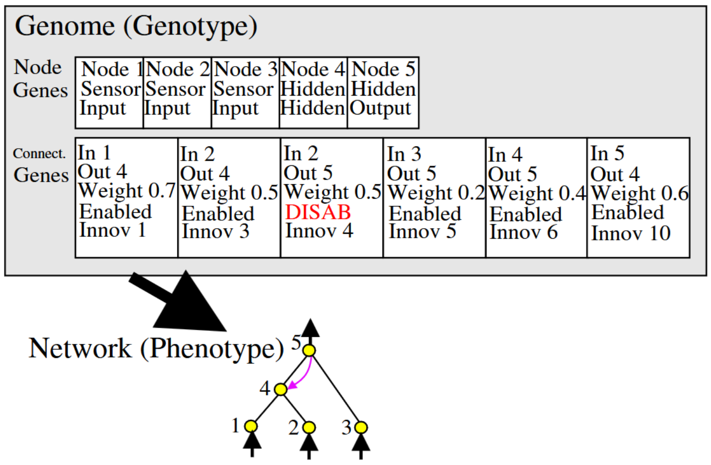

Genome (Genotype)
Node Genes
Node 1 Sensor Input
Node 2 Sensor Input
Node 3 Sensor Input
Node 4 Hidden Hidden
Node 5 Hidden Output
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.5 DISAB Innov 4
In 3 Out 5 Weight 0.2 Enabled Innov 5
In 4 Out 5 Weight 0.4 Enabled Innov 6
In 5 Out 4 Weight 0.6 Enabled Innov 10
Network (Phenotype)
5
4
1
2
3

Figure 9: The scheme NEAT uses for its genomes [59]

The individuals in NEAT are called genomes, based upon the biological term, which is the complete genetic information of an organism. As shown in Figure 9, each genome contains a list of node genes and one of connection genes, that specify the in-node and out-node as well as contain information about the connection's weight, the innovation number, and whether the connection is enabled or not. The innovation number identifies the original historical ancestor of each gene, which allows finding corresponding genes during crossover.

The weight mutation in NEAT functions as in any neuroevolution system, with each connection either perturbed or not. The structural mutation can occur in two ways depicted in Figure 10:

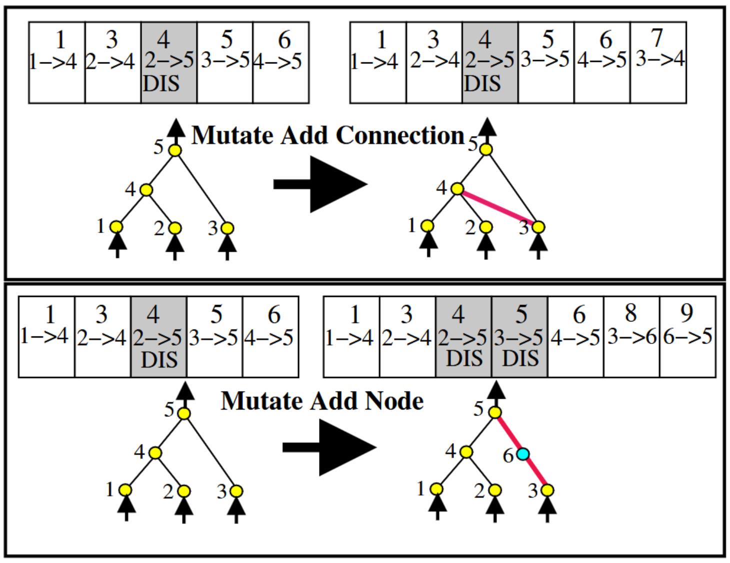


Figure 10: The two ways of structural mutation in NEAT [59]

Adding a connection is done by expanding the list of connection genes with one connecting two previously unconnected nodes. A node is added by splitting an existing connection into two, with the new node in between. The old connection gene is disabled and two new connection genes are added.

Tracking Genes through Historical Markings

The tracking of the origins of a gene is done by the aforementioned innovation number. Whenever a new gene appears through structural mutation, the global innovation number is incremented and assigned to the new gene. Figure 11 shows the crossover of two genomes:

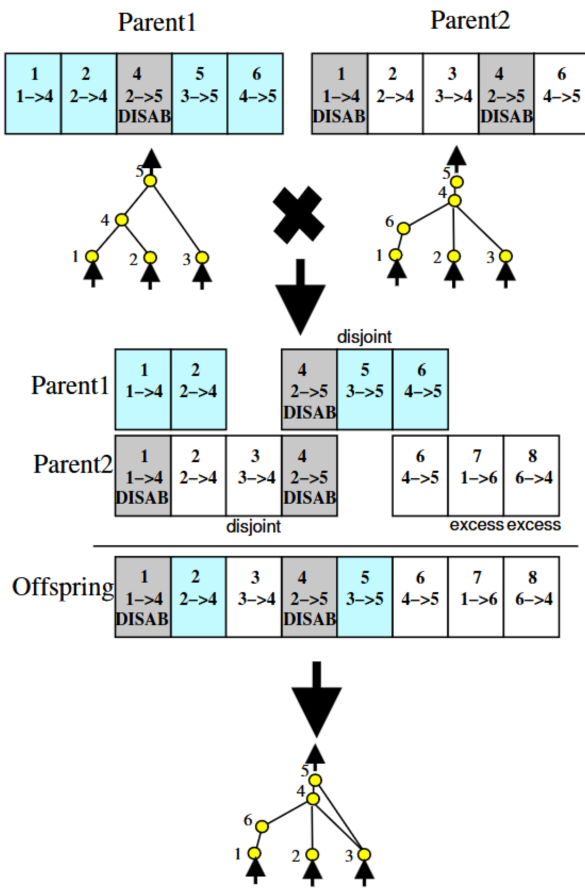


Figure 11: The matching of genes during crossover in NEAT;
The innovation number is shown on top of each parent [59]

Thanks to the innovation number, the algorithm knows which genes match up and represent the same structure during crossovers. Genes that do not match are either disjoint or excess, depending on if they occur within the innovation numbers of the other genome. If genes are not matching up, they are inherited from the fittest parent or randomly chosen if both parents have the same fitness. This way, NEAT can crossover genomes with different topologies without the need for any topological analysis.

Protecting Innovation through Speciation

Adding a new structure to a genome can be detrimental to its fitness, as the new structure tends to be unoptimised. To prevent the genome with the new structure from being eliminated too quickly, NEAT uses speciation. The population is divided into species based on their topological similarity, which can be measured by the number of disjoint D and excess E genes and the average weight difference for matching genes $\overline{W}$:

$$\delta = \frac{c_1 \cdot E}{N} + \frac{c_2 \cdot D}{N} + c_3 \cdot \overline{W}.$$

The coefficients c_1, c_2, and c_3 are constants that determine the importance of the three factors and the factor N – the number of genes in the larger genome – normalises for genome size.

Genomes are tested for compatibility with a species calculating their δ to a random member of the species and if the δ is below a threshold, the genome is added to the species. Genomes are placed into the first species that fits.

NEAT uses explicit fitness sharing, where organisms in the same species share their niche's fitness, preventing any one species from dominating. Fitness is adjusted by dividing by the number of individuals in the species. Species grow or shrink based on whether their average adjusted fitness is above or below the population average:

$$N'_j = \frac{\sum_{i=1}^{N_j} f_{ij}}{\overline{f}},$$

where N_j and N'_j are the old and new size of species j, f_{ij} is the adjusted fitness of individual i in species j, and $\overline{f}$ is the mean adjusted fitness of the entire population. The best performing $r\%$ of each species is randomly mated to generate N'_j offsprings, replacing the entire species.

Minimizing Dimensionality

TWEANN algorithms usually start with a population of random topologies to introduce diversity, as new structures often do not survive without protection. However, this diversity may be unnecessary and costly, as random topologies contain untested structures. Optimizing these complex structures increases the search dimensions, potentially wasting effort. Thus, NEAT starts with a uniform population of simple networks with no hidden nodes. By using speciation to protect innovation, NEAT can grow new structures incrementally as needed. Only useful structures survive through fitness evaluations, reducing the number of weight dimensions and generations needed to find a solution.

Each of these features introduced by NEAT has been shown to be necessary as NEAT performs significantly worse when one of them is removed [59]. Through these innovations, NEAT quickly became the most popular and widely used algorithm in neuroevolution, its feats amongst many others being used to find the most accurate mass estimate of the top quark or controlling Mario in Super Mario Bros [56].

2.7. Evolutionary algorithms in Reinforcement Learning

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

[60] 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* [61] and *HyperNEAT* [62] are mentioned as working solutions, whilst for the HPO variants of the *Genetic Algorithm* (GA) [63], *Whale Optimisation* [64], *online meta-learning by parallel algorithm competition* (OMPAC) [65], and *population based training* (PBT) [66] were successfully used.

[67] 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 [68], SEARL [69]); and combinations of both evolutionary methods, like an evolutionary stochastic gradient descent (ESGD) [70].

[71] 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.

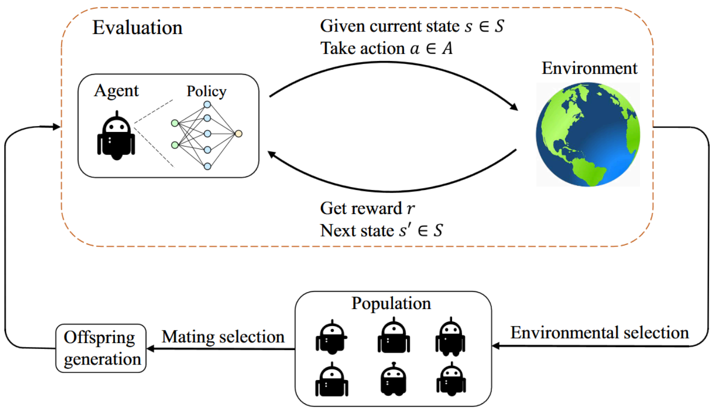


Figure 12: How a evolutionary algorithm is used to optimise RL [67]

Figure 12 shows the general way an evolutionary algorithm is used to optimise a reinforcement learning algorithm. The outer loop is the evolutionary algorithm, whose population consists of different RL agents. These RL agents are then run in the environment and their performance is evaluated.

Most of the papers mentioned above planning to use evolutionary algorithms to optimise RL use this approach. This, however, is not feasible for this thesis, as explained at a later point.

2.8. Bayesian Optimisation

Bayesian Optimization (BO) is a 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. [72]

2.9. Neural Architecture Search

Neural Architecture Search (NAS) is a field of research that aims to automate the design of neural network architectures. The goal is to find the optimal network topology 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. NAS has gained significant attention in recent years due to its potential to improve the performance and efficiency of deep learning models. NAS has been a popular research topic in recent years, as [73] could accumulate over 1000 papers on that topic in just two years.

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 [18]. 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 (BO) 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. [73]

Evolutionary algorithms can also be used to search for optimal network architectures as shown by; however, most of the existing papers on EAs in NAS focus on image classification tasks [20]. This focus is also true for the papers on NAS as a whole [19].

2.9.1. NAS in DRL

Using reinforcement learning for neural architecture search has been done for a long time and thus has been heavily researched [73]. The other way around – using neural architecture search to improve reinforcement learning – is, however, not common. [19] argues that even though many authors optimised some architectural choices of deep reinforcement learning algorithms, a full study of NAS for RL is still missing. One of the works trying to use NAS in DRL is [74], which 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 [75]. 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.

The author of this thesis reckons that the approach of using an evolutionary algorithm to search the neural architecture an inner loop of a DRL algorithm is unique and only done by a few. This uniqueness is only elevated by using the learning algorithm in the adversarial resilience learning [12] context, as an agent in a simulated powergrid.

3. Software Stack

This section covers the environment in which the optimisation is set, as well as further explanation of the algorithms used.

3.1. palaestrAI

palaestrAI[8] 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 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 [76] 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.

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

palaestrAI's loose coupling is realised using ZeroMQ [77], which is a messaging system that allows for a reliable request-reply pattern, such as the majordomo pattern [77], [78]. 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.

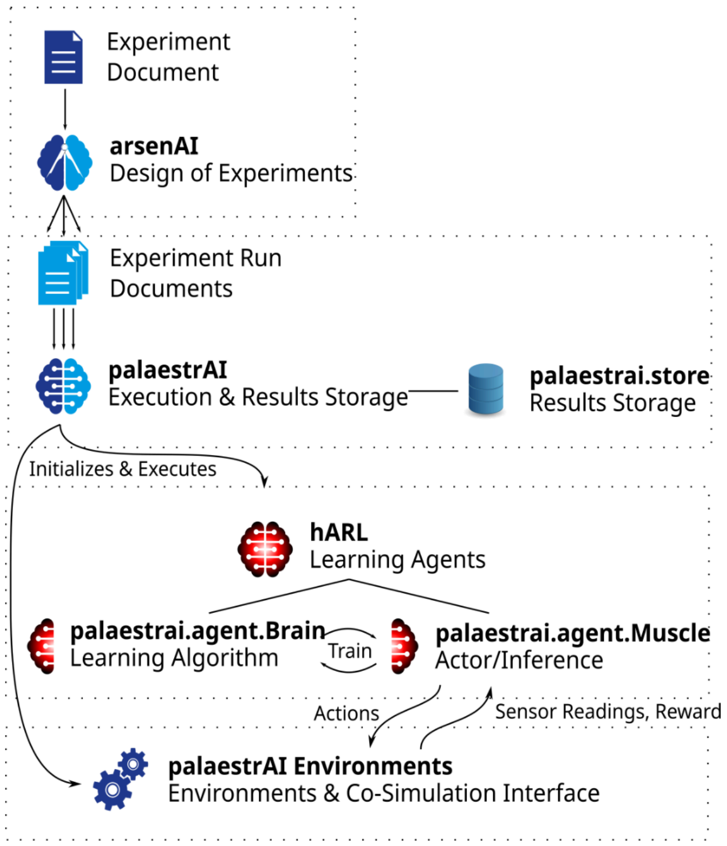


Figure 13: Execution Schema of the software stack's execution [79]

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. τ = (s0, a0, …). 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[9]. 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.

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

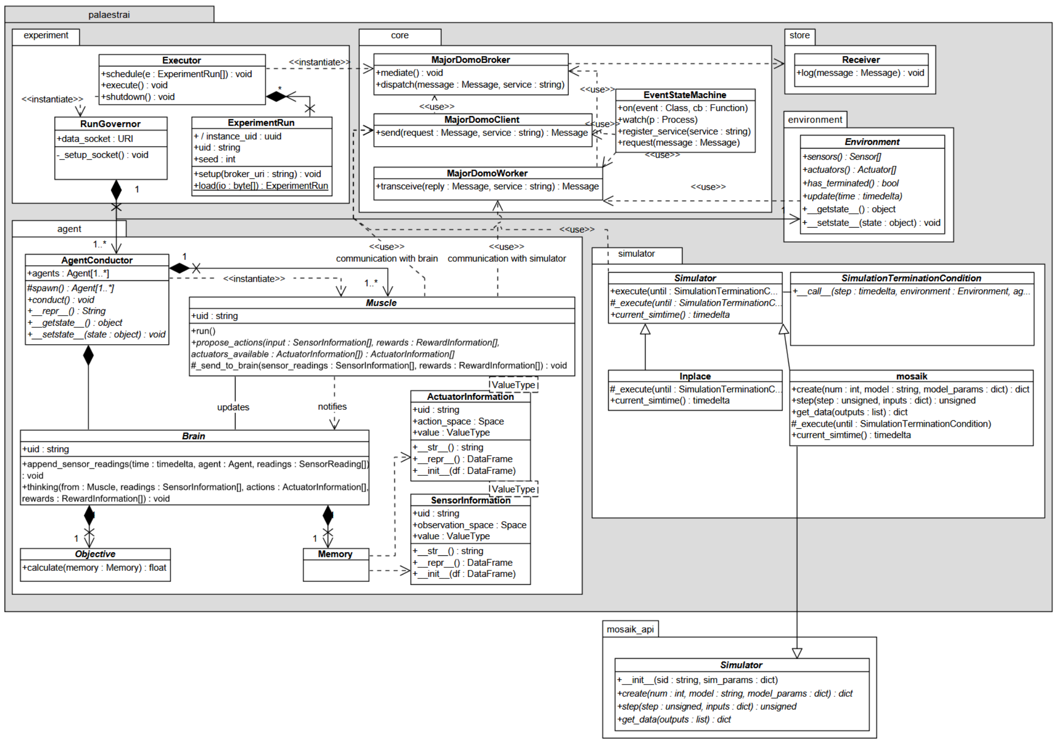


Figure 14: Most important classes in the palaestrAI software stack [79]

arsenAI's and palaestrAI's concept of experiment run phases allows for flexibility in offline learning or adversarial learning through autocurricula [80]. Within a phase, agents can be employed in any combination and with any sensor/actuator 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.

3.2. HARL

harl[10] 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).

The SAC implementation used in harl[11] is based on the 'OpenAI Spinning Up in DRL'-implementation[12]. It extends the implementation and uses α-annealing instead of Spinning Up's use of a fixed entropy coefficient α. PyTorch[13] is used as the library for machine learning and ADAM (Section 2.4.3) is used to optimise the temperature α, the policy π, and the Q-function Q.

[10]https://gitlab.com/arl2/harl
[11]https://gitlab.com/arl2/harl/-/blob/development/src/harl/sac/brain.py
[12]https://spinningup.openai.com/en/latest/algorithms/sac.html
[13]https://pytorch.org/

3.3. Choice of the algorithms

Algorithm	**Paper**	**Usable Impl.?**	**Usable for**
GA-DRL	[81], [82]	✓	Hyperparameter
NEAT	[59]	✓	Network + Weights
HyperNEAT	[62]	✓	Network + Weights
PBT	[66]	✓	Hyperparameter
FIRE-PBT	[68]	✗	Hyperparameter
SEARL	[69]	✓	Hyperparameter
AAC	[83]	✗	Hyperparameter
OMPAC	[65]	✗	Hyperparameter
BO-GA/BO-DE/BO-ES	[84]	○ Pseudocode	Hyperparameter
CMA-ES	[85]	✓	Hyperparameter
ESGD	[70]	○ Pseudocode	Network
WOA	[64]	✓	Hyperparameter
ALF	[86]	✗	Network + Weights
DEHB	[87]	✓	Hyperparameter
EARL, CERL, ERL, GEATL	[88], [89] [90], [91]	✗	Hyperparameter
ProxylessNAS	[92]	✓	Network
Minimal RL	Based on [93]	✓	Network
Bayesian Opt.	i.e. [94]	✓	Parameter
Smash	[95]	✓	Network
AutoDL	GitHub[14]	✓	Network
NASLib	GitHub[15]	✓	Network

Table 2: Comparison of possible algorithms on the availability of an useable implementation and their use.

Many evolutionary algorithms that were found i.i. in surveys like [60], [67], [71] and other NAS algorithms were assessed for the implementation. These algorithms are listed in Table 2. Of these algorithms, many had either no available implementation, only pseudocode, or an implementation in a different programming language. Since palaestrAI is written in python, the implementation had to be in python as well for compatibility and implementation reasons.

The goal of this thesis is to optimise the architecture of the reinforcement learning algorithm. Hence, NEAT that has several popular python implementations like neat-python[16] or pytorch-neat[17] available was readily chosen – latter was used as the implementation, because palaestrAI's SAC uses pytorch as well. It was picked over its successor HyperNEAT, which improves upon NEAT by using the geometric regularities of the task domain: Even though HyperNEAT performs better for

[14]https://github.com/D-X-Y/AutoDL-Projects
[15]https://github.com/automl/NASLib
[16]https://github.com/CodeReclaimers/neat-python
[17]https://github.com/ddehueck/pytorch-neat

simple fractured problems, it has no improvement for more complicated problems and was slow even on multi-core processors [96].

The implementation of the AutoDL-Project and NASLib algorithms as well as ProxylessNAS and Smash were tried. Problems arose as these implementations are generally intended for the use in image classification – NASLib additionally offers implementations for vision, speech recognition, and language processing tasks – and the used neural nets for this kind of task are not compatible with the use case faced in this thesis. An adaption to the kind of net useable by the rest of the stack revealed itself to be highly complicated due to the complexity of the implementations; changing the functionality of the net had implications for many parts of the implementation and could not be done without influencing the functionality of the algorithm.

Despite originally being for image classification as well, the minimal RL implementation could be adapted due to its simplicity.

Finally, Bayesian optimisation was chosen over the other hyperparameter optimisation algorithms, due to "Bayesian Optimisation" being a household name in the field for many years now and having a very popular maintained implementation[18] that is simple to integrate available.

Thus, three algorithms with different approaches to optimisation were chosen: an evolutionary algorithm, a reinforcement learning algorithm, and a sequential design strategy.

The evolutionary algorithm NEAT tries to find the best topology and weights for the neural network. The other two algorithm try to find the best architecture for the network without regards to the weights; the reinforcement learning algorithm by trying to find the best set of parameters that represent the architecture, and the Bayesian optimisation algorithm by optimising the black-box function describing the net's topology.

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

4. 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.

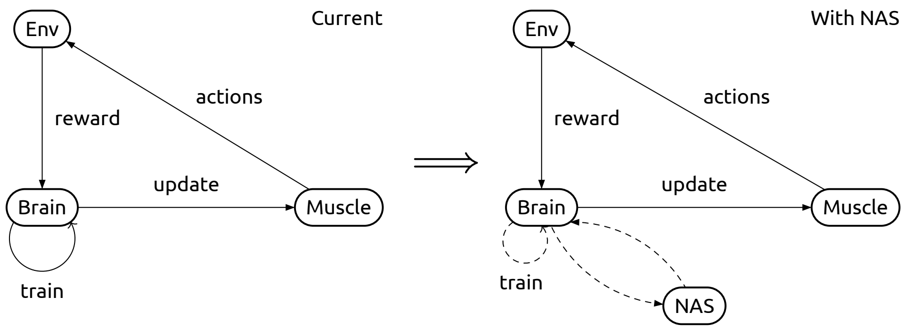


Figure 15: Adding NAS to the palaestrAI cycle

On the left side of Figure 15, a simplified version of Figure 13 with focus on the agent 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

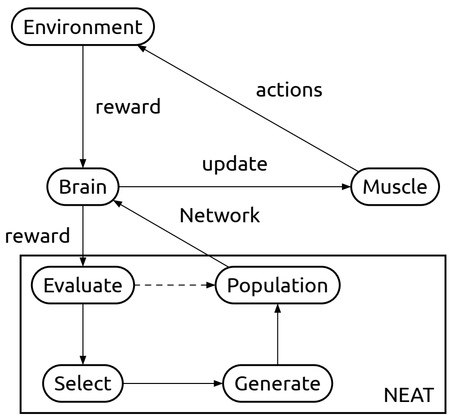


Figure 16: Using NEAT as NAS

Figure 16 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.

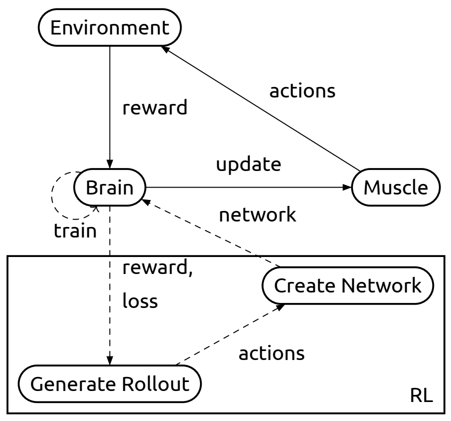


Figure 17: Using a RL approach as NAS

In Figure 17, 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.

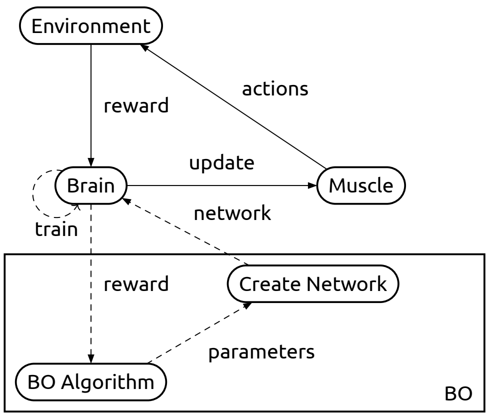


Figure 18: Using BO as NAS

Lastly, Figure 18 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.

5. 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[19]. 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.

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.

5.1. NEAT

For the implementation of NEAT a GitHub repository by ddehueck called PyTorch-NEAT[20] 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

[19]https://gitlab.com/arl2/harl/-/blob/development/src/harl/sac/network.py?ref_type=heads
[20]https://github.com/ddehueck/pytorch-neat/

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

5.2. RL

The NAS method using reinforcement learning is based on the 'minimal-nas' implementation by nicklashansen[21]. 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

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

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.

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

5.3. BO

The Bayesian Optimisation NAS method is based on the python 'bayesian-optimization' implementation[22]. 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

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

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.

6. 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.

6.1. 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.

6.1.1. 1st Scenario: CIGRE

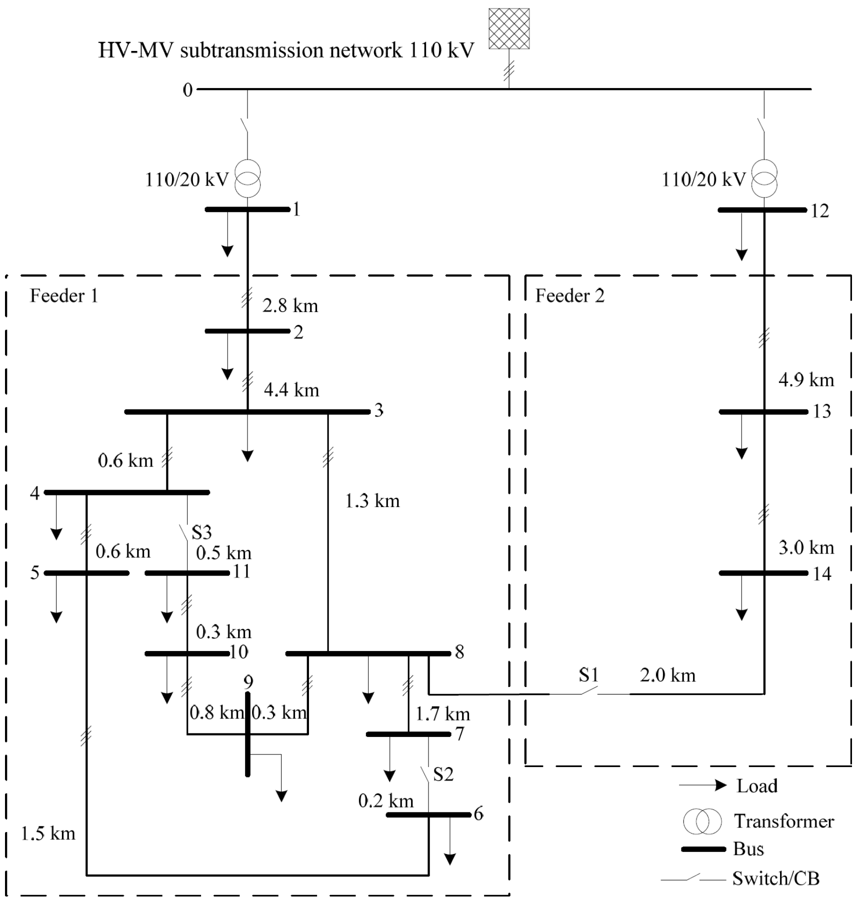


Figure 19: CIGRE benchmark grid used [97]

This scenario uses the CIGRE Medium Voltage grid model of [97], also depicted in Figure 19. The ‘defender’ agent is given the ARL defender objective[23], which adds the overall mean voltage of the grid, the agent’s voltage sensor values, as

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

well as the number of buses in service together, each with a coefficient of $\frac{1}{3}$ by default, to calculate the reward.

6.1.2. 2ND 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 [3]. Consistent with the usage of COHDARL, the 'defender' agent is given the COHDARL defender objective[24], whose reward is based on the global objective of COHDARL. The objective calculates its reward just like the ArlDefenderObjective, just with the addition of the cohdarl's power objective and a coefficient of $\frac{1}{4}$ for each of the summands.

6.2. Parameters

The palaestrAI framework allows to create schedules of experiments – so called phases – to start directly after one another with one command. Thus, an experiment for each of the two scenarios was conducted with the four different methods (Baseline, BO, NEAT, RL) being phases of it.

The experiment[25] for the first scenario is run for three episodes, each representing a hundred days, which is equivalent to 8,640,000 seconds. A step size of 900 seconds is used, which results in 5,760 steps per episode – since the palaestrAI simulation starts at around 40% of the total steps for experiments with timeseries – or 17,280 steps for the whole simulation. As reward function of the environment, the ExtendedGridHealthReward[26], a reward based on the grid's "healthiness" – meaning the deviation from the best possible status –, is used. It is a vectorised reward giving complete information about the net by encompassing the minimum, maximum, mean, median, and standard deviation of the values of the voltage bands and line loads as well as the number of buses in service and out of service. Each value is provided as a separate reward information with a corresponding id.

[24]https://gitlab.com/arl-experiments/cohdarl/-/blob/main/src/cohdarl/cohdarl_objective.py
[25]https://gitlab.com/arl2/harl/-/blob/nas/src/harl/sac/nas/experiments/NAS-Exp-TS.yml
[26]https://gitlab.com/midas-mosaik/midas-palaestrai/-/blob/main/src/midas_palaestrai/rewards.py#L88

The second scenario's experiment[27] uses almost the same parameters (also shown in Table 7) as the first scenario, but has an increased runtime of 15,000,000 seconds.

The parameters for SAC shown in Table 8 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 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 9, Table 10, and Table 11. They are mostly kept to the default values of their implementations. NEAT's threshold is increased to a value of 100,000, 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 both the RL-method and the BO-method running for a similar amount of time – 20.000 steps in total – with the remaining steps being run with the network that performed best during the NAS method's runtime. The NEAT method runs for 25.000 steps in total, because it also trains the weights and thus does not need the amount of time to train the best net.

[27] https://gitlab.com/arl2/harl/-/blob/nas/src/harl/sac/nas/experiments/NAS-Exp-TS-COHDARL.yml

7. Results

The outputs of the two experiments were saved to a database and visualised with Grafana[28], an open-source platform for monitoring, visualising, and analysing data from various sources by allowing users to create interactive dashboards and graphs to track metrics, logs, and other data in real-time.

7.1. 1st Experiment: Cigre

The experiment of the first scenario took 11 hours and 18 minutes to finish.

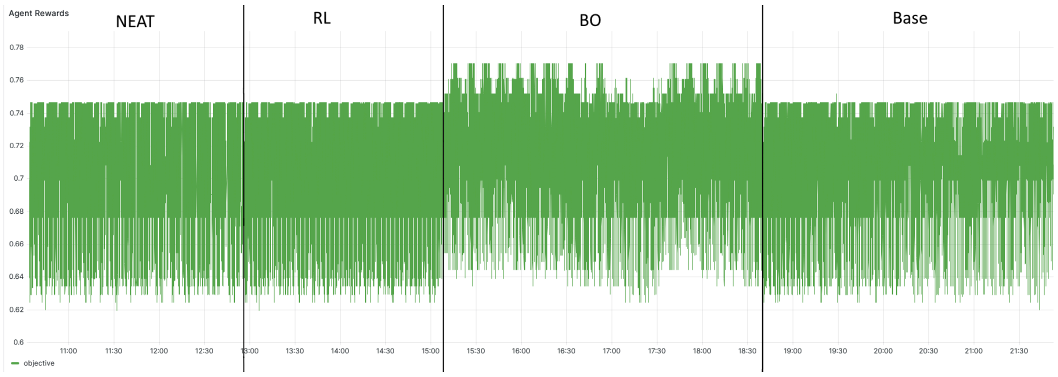


Figure 20: Rewards of the agents over time in the 1st experiment.

Figure 20 shows the rewards of the agents over time; the higher the rewards are, the better the agent is performing. The four phases of the experiment with their corresponding active NAS method are marked: the first phase was the NEAT phase, followed by RL, BO, and then the baseline. Apparent is the phase of the BO method, whose rewards are peaking several times a cut above the other three's highest rewards. The lows also tend to be higher than the other methods' lows. The graphs of the other three methods are highly similar, with their highs and lows being around the same values. Except for some time between 17:00 and 17:30, which resembles the other three methods' performance, the BO method's performance looks to be better than the other three methods' performance. All methods do not seem to learn as the rewards are not increasing over time.

Method	**First Step**	**Last Step**	**Total Duration**
NEAT	10:33:52	12:55:42	2:21:50
RL	12:56:28	15:07:48	2:11:20
BO	15:08:41	18:39:42	3:31:01
Base	18:40:30	21:52:44	3:12:14

Table 3: The duration of each phase in the 1st experiment.

[28]https://grafana.com/

Another information extractable from the graph is each phases duration, which is shown in Table 3. It shows that NEAT was the fastest of the methods, followed by RL, the baseline, and BO. The difference between RL's duration and BO's duration is 1:19:41, in other words, takes the BO agent around 60% longer than the RL agent. Both RL and NEAT are faster than the baseline by 1 hour 54 seconds (around 32% faster) and 50:24 minutes (ca. 26%), respectively. BO is slower than the baseline by 18:47 minutes (around 10% slower).

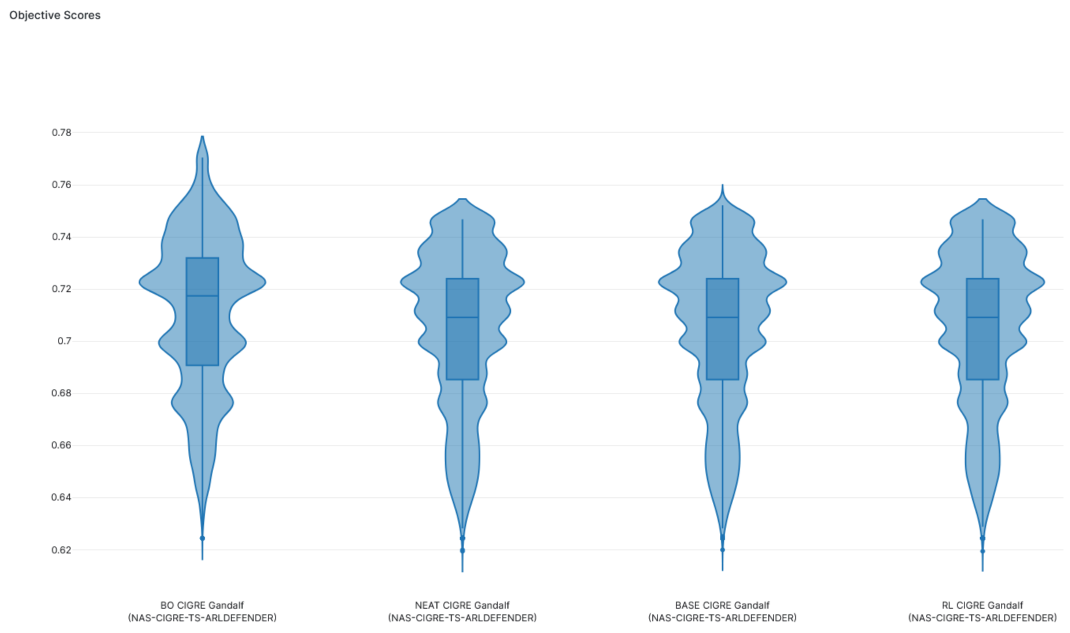


Figure 21: Violin plots of each method's objective scores in the 1st experiment.

The violin plots in Figure 21 show the distribution of the objective scores – or the agent rewards – of each method with the minimum, maximum, and median values of each method listed explicitly in the Table 4 below.

Method	Min	Max	Median
BO	0.6244281	0.770436	0.7174087
NEAT	0.6193798	0.7467596	0.709123
Base	0.6199054	0.7521687	0.709123
RL	0.6193798	0.7467596	0.709123

Table 4: The minimum, maximum, and median objective scores of each method in the 1st experiment.

Both Figure 21 and Table 4 reinforce what is seen in Figure 20: the BO method is a cut above the rest and has the highest median, maximum, and minimum objective scores, whilst the other three methods all have similar values for these three metrics.

The hypothesis made at the beginning of this thesis is that a NAS algorithm can improve upon the performance of a user-picked architecture in a reasonable amount of time in the ARL architecture. In regards to this experiment, the hypothesis is true. All three NAS methods have either similar performance as the baseline in less time (NEAT, RL) or a better performance with a slightly higher, but still reasonable, time investment (BO).

7.2. 2nd Experiment: Cigre + Cohdarl

Due to the increased amount of seconds to simulate for the second scenario, the experiment took 24 hours and 21 minutes to finish.

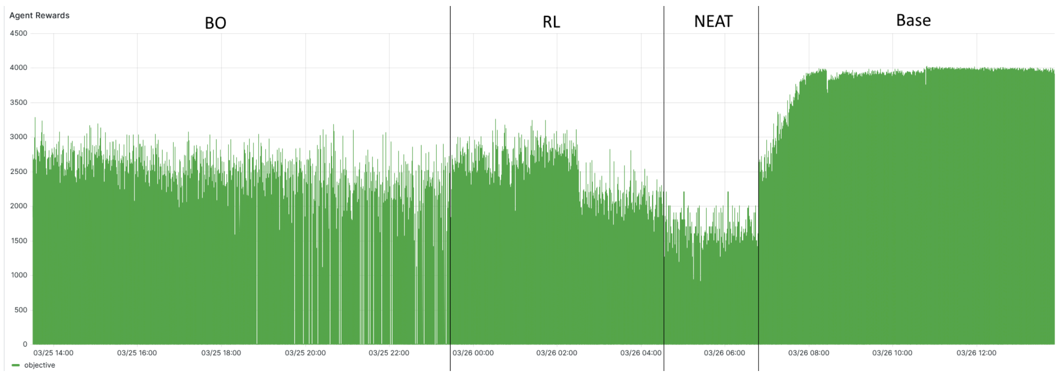


Figure 22: Rewards of the agents over time in the 2nd experiment.

In the graph of Figure 22, the four phases of the experiment are marked with the active NAS method: BO, RL, NEAT and lastly the baseline. Contrary to the first experiment, BO as well as RL and NEAT perform way worse than the baseline. BO's, RL's, and the baseline's rewards start at a similar level, but only the baseline's rewards increase over time – meaning the agent learns. Interestingly, the rewards of both NAS methods are getting worse with the BO agent's rewards slightly deteriorating over time and the RL agent's rewards visibly plummeting slightly after the half-way mark of its phase. The NEAT agent's rewards are the lowest of all four methods; they start at a lower level and do not increase visibly over time.

Method	**First Step**	**Last Step**	**Total Duration**
BO	13:30:11	23:27:42	9:57:31
RL	23:29:07	04:32:07	5:03:00
NEAT	04:33:13	06:47:43	2:14:30
Base	06:48:34	13:51:17	7:02:43

Table 5: The duration of each phase in the 2nd experiment.

The durations of the phases in the 2nd experiment are shown in Table 5. NEAT was the fastest of the methods by a large margin, followed by RL and the baseline. BO was the slowest of the methods, taking almost 3 hours longer than the baseline (40% longer). Both RL and NEAT were faster than the baseline by 1 hours 59 minutes 43 seconds (around 30% faster) and 4 hours 48 minutes (around 68%), respectively.

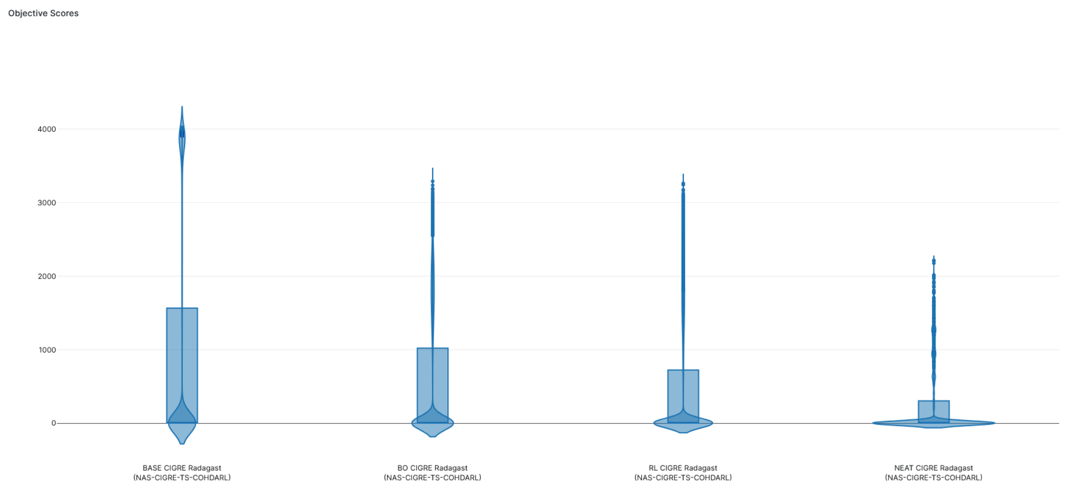


Figure 23: Violin plots of each method's objective scores in the 2nd experiment.

Method	**Min**	**Max**	**Median**
Base	0.7482757	4,025.185	3.777617
BO	0.7066692	3,292.298	2.249176
RL	0.6156467	3,264.563	2.238741
NEAT	0.4796731	2,211.526	1.598038

Table 6: The minimum, maximum, and median objective scores of each method in the 2nd experiment.

The violin plots in Figure 23 as well as the Table 6 clarify the results of the experiment: the baseline wins by a landslide, with the highest median, maximum, and minimum objective scores. NEAT loses by a large margin, with the poorest results in all three metrics.

The hypothesis of this work cannot be confirmed by the results of the second experiment. The BO method neither improves upon the baseline's performance nor does it perform this task in a reasonable amount of time. The RL and NEAT methods are faster than the baseline, but their performance is significantly worse. All three NAS methods do not seem to be learning, as their rewards are not increasing over time.

8. Evaluation and Further work

The two experiments show contrary results: Whilst the first experiment shows a clear advantage of the Bayesian optimisation method with a performance of the reinforcement learning approach and NEAT being similar to that of the baseline, the second experiment shows a clear loss of all the NAS methods compared to the baseline.

In neither the first nor the second experiment the NAS methods seem to be learning, as the rewards are not increasing over time. During the first experiment, the baseline is not learning either, which may be the reason for the NAS methods' performance being similar to the baseline's. Since the baseline is learning in the second experiment, it can be taken into consideration that the first experiment was not working as intended.

Thus the results of the second experiment may be more reliable than the first experiment's results. As both the RL method and the BO method should be using SAC's normal learning process – which is working as intended as seen in the baseline's case – after the NAS methods have finished at around two thirds of the phase time, the results of the second experiment are a clear indicator that parts of the NAS methods might not be working as intended.

Even though the software stack around palaestrAI went through some changes during the time of the experiments, it stands to reason that the implementation of the NAS methods into the harl framework may have some underlying problems and unintended behaviour. The possibility of the NAS methods simply not being able to handle the task at hand should also be taken into consideration. Further investigation and experimentation is needed to clarify the reasons for the NAS methods' performance.

The reason for the lower overall reward for NEAT in the second experiment is known; it is due to the reduced number of actuators over which the agent has control. This was done to circumvent a problem occurring when using the NEAT agent with all actuators: starting the experiment leads to palaestrAI finishing the simulation right after the first step, whilst the same experiment with other or no NAS methods run the whole distance as expected. This problem and its origins are still under investigation, but is presumed to be due to the network structure of the NEAT implementation used. Pytorch-NEAT uses a unique feed forward network structure, which, since the implementation uses torch as library, could be implemented into the harl framework. However, this network structure is not the same as the network structure used in the SAC implementation of harl and thus it is not possible to continue with a normal training process after NEAT has finished. 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 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; Consultation of literature or usage of common practices like computational methods to optimise the parameters for the specific task could lead to a better performance of the NAS methods; which is the like the premise of this thesis, just one level further.

At the moment, the parameters to determine the duration of the NAS methods are either set by the user or left to default. For the experiments these values were set manually so that the methods end before the experiment finishes. This adaption to the experiment's length could also be automated, reducing the amount of setting leaving the user to only activate the NAS method.

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.

9. Conclusion

The goal of this thesis was to confirm the hypothesis that a NAS algorithm can improve upon the performance of a user-picked architecture in a reasonable amount of time in the ARL architecture by implementing three different NAS methods – NEAT, a Bayesian optimisation, and a reinforcement learning based approach – into the SAC implementation of the palaestrAI framework and comparing their performance to a baseline agent in a simulated power system environment with ARL.

The two experiments conducted either confirm this hypothesis in case of the first experiment or do not confirm the hypothesis in case of the second experiment. Due to the first experiment not being able to show a learning baseline, the results of the second experiment are more reliable than the first experiment's results. Thus, the hypothesis can not be confirmed. More experimentation and investigation is needed in order to identify whether the reason for the NAS methods' performance is due to a faulty implementation of the NAS methods into the harl framework or if the NAS methods are simply not able to handle the task at hand.

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] T. Elsken, J. H. Metzen, and F. Hutter, ‘Neural architecture search: A survey’, *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[20] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, ‘A survey on evolutionary neural architecture search’, *IEEE transactions on neural networks and learning systems*, vol. 34, no. 2, pp. 550–570, 2021.

[21] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. pearson, 2016.

[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] I. Alymov and M. Averbukh, ‘Monitoring energy flows for efficient electricity control in low-voltage smart grids’, *Energies*, vol. 17, no. 9, p. 2123, 2024.

[25] W. A. Vilela Junior *et al.*, ‘Analysis and Adequacy Methodology for Voltage Violations in Distribution Power Grid’, *Energies*, vol. 14, no. 14, p. 4373, 2021.

[26] M. M. Albu, M. Sănduleac, and C. Stănescu, ‘Syncretic use of smart meters for power quality monitoring in emerging networks’, *IEEE Transactions on Smart Grid*, vol. 8, no. 1, pp. 485–492, 2016.

[27] E. M. Veith, ‘The Power Grid in a Nutshell’.

[28] J. McCarthy and others, ‘What is artificial intelligence’, 2007.

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

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

[31] A. Clauset, ‘A brief primer on probability distributions’, in *Santa Fe Institute*, 2011.

[32] L. Bottou, ‘Large-scale machine learning with stochastic gradient descent’, in *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, 2010, pp. 177–186.

[33] J. Duchi, E. Hazan, and Y. Singer, ‘Adaptive subgradient methods for online learning and stochastic optimization.’, *Journal of machine learning research*, vol. 12, no. 7, 2011.

[34] T. Tieleman, ‘Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude’, *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, p. 26, 2012.

[35] D. P. Kingma and J. Ba, ‘Adam: A method for stochastic optimization’, *arXiv preprint arXiv:1412.6980*, 2014.

[36] I. Loshchilov and F. Hutter, ‘Decoupled weight decay regularization’, *arXiv preprint arXiv:1711.05101*, 2017.

[37] P. T. Tran and others, ‘On the convergence proof of amsgrad and a new version’, *IEEE Access*, vol. 7, pp. 61706–61716, 2019.

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

[39] J. Heaton, ‘Introduction to the Math of Neural Networks (Beta-1)’, *Heaton Research Inc*, 2011.

[40] K. Hornik, M. Stinchcombe, and H. White, 'Multilayer feedforward networks are universal approximators', *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[41] 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.

[42] J. Kober, J. A. Bagnell, and J. Peters, 'Reinforcement learning in robotics: A survey', *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[43] B. Hambly, R. Xu, and H. Yang, 'Recent advances in reinforcement learning in finance', *Mathematical Finance*, vol. 33, no. 3, pp. 437–503, 2023.

[44] D. Chen *et al.*, 'Powernet: Multi-agent deep reinforcement learning for scalable powergrid control', *IEEE Transactions on Power Systems*, vol. 37, no. 2, pp. 1007–1017, 2021.

[45] A. Perera and P. Kamalaruban, 'Applications of reinforcement learning in energy systems', *Renewable and Sustainable Energy Reviews*, vol. 137, p. 110618, 2021.

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

[47] J. Hao and Y. Tao, 'Adversarial attacks on deep learning models in smart grids', *Energy Reports*, vol. 8, pp. 123–129, 2022.

[48] A. Pan, Y. Lee, H. Zhang, Y. Chen, and Y. Shi, 'Improving robustness of reinforcement learning for power system control with adversarial training (2021)'.

[49] R. Bellman, 'The theory of dynamic programming', *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[50] S. S. Gu, T. Lillicrap, R. E. Turner, Z. Ghahramani, B. Schölkopf, and S. Levine, 'Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning', *Advances in neural information processing systems*, vol. 30, 2017.

[51] J. Peters, 'Policy gradient methods', *Scholarpedia*, vol. 5, no. 11, p. 3698, 2010.

[52] C. J. Watkins and P. Dayan, 'Q-learning', *Machine learning*, vol. 8, pp. 279–292, 1992.

[53] V. Mnih *et al.*, ‘Human-level control through deep reinforcement learning’, *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[54] T. P. Lillicrap *et al.*, ‘Continuous control with deep reinforcement learning’, *arXiv preprint arXiv:1509.02971*, 2015.

[55] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, ‘Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor’, in *International conference on machine learning*, 2018, pp. 1861–1870.

[56] K. O. Stanley, ‘Neuroevolution: A different kind of deep learning – oreilly.com’. 2017.

[57] Y. Zhang, S. Wang, and G. Ji, ‘A comprehensive survey on particle swarm optimization algorithm and its applications’, *Mathematical problems in engineering*, vol. 2015, no. 1, p. 931256, 2015.

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

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

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

[65] 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.

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

[67] 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.

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

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

[70] X. Cui, W. Zhang, Z. T5üske, and M. Picheny, ‘Evolutionary stochastic gradient descent for optimization of deep neural networks’, *Advances in neural information processing systems*, vol. 31, 2018.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] E. Veith *et al.*, ‘palaestrAI: A training ground for autonomous agents’, in *Proceedings of the 37th annual European Simulation and Modelling Conference, EUROSIS*, 2023.

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

[81] A. Sehgal, H. La, S. Louis, and H. Nguyen, ‘Deep reinforcement learning using genetic algorithm for parameter optimization’, in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 2019, pp. 596–601.

[82] A. Sehgal, N. Ward, H. M. La, and S. Louis, ‘Deep reinforcement learning for robotic manipulation tasks using a genetic algorithm-based function optimizer’, *Encyclopedia with semantic computing and robotic intelligence*, 2023.

[83] J. Grigsby, J. Y. Yoo, and Y. Qi, ‘Towards automatic actor-critic solutions to continuous control’, *arXiv preprint arXiv:2106.08918*, 2021.

[84] A. M. Vincent and P. Jidesh, ‘An improved hyperparameter optimization framework for AutoML systems using evolutionary algorithms’, *Scientific Reports*, vol. 13, no. 1, p. 4737, 2023.

[85] I. Loshchilov and F. Hutter, ‘CMA-ES for hyperparameter optimization of deep neural networks’, *arXiv preprint arXiv:1604.07269*, 2016.

[86] R. Krauss, M. Merten, M. Bockholt, and R. Drechsler, ‘ALF: a fitness-based artificial life form for evolving large-scale neural networks’, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, in GECCO '21. Lille, France: Association for Computing Machinery, 2021, pp. 225–226. doi: 10.1145/3449726.3459545.

[87] N. Awad, N. Mallik, and F. Hutter, ‘Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization’, *arXiv preprint arXiv:2105.09821*, 2021.

[88] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette, ‘Evolutionary algorithms for reinforcement learning’, *Journal of Artificial Intelligence Research*, vol. 11, pp. 241–276, 1999.

[89] S. Khadka and K. Tumer, 'Evolutionary reinforcement learning', *arXiv preprint arXiv:1805.07917*, vol. 223, 2018.

[90] S. Khadka *et al.*, 'Collaborative evolutionary reinforcement learning', in *International conference on machine learning*, 2019, pp. 3341–3350.

[91] Q. Zhu *et al.*, 'A survey on Evolutionary Reinforcement Learning algorithms', *Neurocomputing*, vol. 556, p. 126628, 2023, doi: https://doi.org/10.1016/j.neucom.2023.126628.

[92] H. Cai, L. Zhu, and S. Han, 'Proxylessnas: Direct neural architecture search on target task and hardware', *arXiv preprint arXiv:1812.00332*, 2018.

[93] B. Zoph and Q. V. Le, 'Neural architecture search with reinforcement learning', *arXiv preprint arXiv:1611.01578*, 2016.

[94] J. Močkus, 'On Bayesian methods for seeking the extremum', in *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974 6*, 1975, pp. 400–404.

[95] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, 'Smash: one-shot model architecture search through hypernetworks', *arXiv preprint arXiv:1708.05344*, 2017.

[96] J. Lowell, K. Birger, and S. Grabkovsky, 'Comparison of NEAT and HyperNEAT on a Strategic Decision-Making Problem', *URL: http://web. mit. edu/jessiehl/Public/aaai11/fullpaper. pdf [As of: 2017.01. 15]*, 2011.

[97] 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.

Appendix

Parameter	Value
Net	CIGRE
Amount Steps	100 Days (8,640,000 seconds) / 15,000,000 seconds
Episodes	3
Reward Function	ExtendedGridHealthReward
Step size	900

Table 7: 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 8: 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	100000
	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 9: NEAT parameters

Parameter	**Value**
NAS runs	40
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 10: RL based NAS parameters

Parameter	**Value**
Runs per network	20
INIT_POINTS	16
N_ITER	24

Table 11: BO parameters

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