Automated machine learning for dynamic energy management using time-series data

Wang, C.

Chapter 3

Automated machine learning

This chapter provides an overview of the automated machine learning (AutoML) research field. We review and compare available AutoML systems and neural architecture search techniques, a particular case of AutoML for automatically generating deep learning models. While this thesis is mainly focused on AutoML for time-series data, here we present a more general overview of AutoML. The topics presented in this chapter lay the foundation of developing more advanced and specialised AutoML systems.

3.1 Automated machine learning problem definitions

Machine learning *models* are generated by machine learning *algorithms* by internally optimising a number of *parameters*. Machine learning algorithms typically have several *hyperparameters* that need to be externally set by users and control various aspects of the training process; their values usually do not change during training. For example, a neural network model can have millions of parameters, such as the weights and bias values. The learning algorithm that determines these parameters has several hyperparameters, such as the learning rate, the optimisation schedule and how to perform data augmentation. Next to the choice of the algorithm, the choice of hyperparameters has a strong influence on the final performance of the model and the efficacy of the learning process.

AutoML aims to produce the best model for a data set of interest, given a large set

of machine learning algorithms and their hyperparameter spaces. There are two major goals in finding these models: (1) to obtain a model with the highest performance for a given machine learning task (e.g., classification or regression accuracy) on the data set, and (2) to ensure that good "out of sample generalisation" of the model on unseen data is achieved. Hyperparameter optimisation (HPO) strategies facilitate reaching the first goal, while towards the second, techniques such as cross-validation are used to compare models obtained using different hyperparameter configurations.

There are three closely related AutoML problems, namely, (i) *algorithm (or model) selection*, which deals with multiple machine learning algorithms with fixed hyperparameter settings; (ii) *hyperparameter optimisation*, which deals with a single algorithm; and (iii) *combined algorithm selection and hyperparameter optimisation* (Thornton et al., 2013). To formally define these problems, we use the following notation:

- $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$ denotes the configuration space of algorithm A induced by n hyperparameters, where Λ_i denotes the domain of the ith hyperparameter.

- $\mathcal{A} = \{A^{(1)}, \cdots, A^{(R)}\}$ denotes a set of learning algorithms. The hyperparameters of each algorithm $A^{(j)}$ are defined over the configuration space $\mathbf{\Lambda}^{(j)}$.

- $\mathcal{D} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)\}$ denotes the training data set, where $\mathbf{x_i}$ and y_i represent the values of features and target variables, respectively. To perform k-fold cross-validation, $\mathcal{D}$ is further split into k equally-sized partitions composed of training $(\mathcal{D}^{(1)}_{train}, \cdots, \mathcal{D}^{(k)}_{train})$ and validation sets $(\mathcal{D}^{(1)}_{valid}, \cdots, \mathcal{D}^{(k)}_{valid})$ such that $\mathcal{D}^{(i)}_{train} = \mathcal{D} \setminus \mathcal{D}^{(i)}_{valid}$. Note that there is usually also a test set, but this is considered to be completely external to the selection or optimisation procedure.

- Generalisation performance is evaluated by running training algorithm A on $\mathcal{D}^{(i)}_{train}$ and evaluating the resulting model on $\mathcal{D}^{(i)}_{valid}$ by measuring the loss using a performance metric, such as classification accuracy, denoted by $\mathcal{L}(A, \mathcal{D}^{(i)}_{train}, \mathcal{D}^{(i)}_{valid})$. This evaluation approach corresponds to k-fold cross-validation, which has also been the method of choice in the AutoML literature thus far. In principle, however, other validation techniques can be used.

Definition 3.1 (Algorithm (or Model) Selection Problem)**.** Given a set $\mathcal{A}$ of learning algorithms (associated with specific classes of models), the algorithm selection problem can then be defined as finding $A^* \in \mathcal{A}$ that yields the model with the best performance on the validation set (Thornton et al., 2013):

$$A^* \in \underset{A \in \mathcal{A}}{\operatorname{argmin}} \frac{1}{k} \cdot \sum_{i=1}^{k} \mathcal{L}(A, \mathcal{D}_{train}^{(i)}, \mathcal{D}_{valid}^{(i)}) \tag{3.1}$$

We note that the $A \in \mathcal{A}$ can also correspond to the same learning algorithm with different hyperparameter settings or even just different sequences of random numbers; equivalently, the problem can also be formalised for sets of models (irrespectively how these were obtained) rather than sets of algorithms.

Definition 3.2 (Hyperparameter Optimisation Problem)**.** Given a learning algorithm A with hyperparameters $\mathbf{\Lambda}$, and letting $A_{\boldsymbol{\lambda}}$ denote that A with the hyperparameter vector $\boldsymbol{\lambda} \in \mathbf{\Lambda}$, the HPO problem can be defined as finding $\boldsymbol{\lambda}^*$ that yields the model with the best performance on the validation set (Thornton et al., 2013):

$$\boldsymbol{\lambda}^* \in \underset{\boldsymbol{\lambda} \in \mathbf{\Lambda}}{\operatorname{argmin}} \frac{1}{k} \cdot \sum_{i=1}^{k} \mathcal{L}(A_{\boldsymbol{\lambda}}, \mathcal{D}_{train}^{(i)}, \mathcal{D}_{valid}^{(i)}) \tag{3.2}$$

Definition 3.3. (Combined Algorithm Selection and Hyperparameter Optimisation (CASH) problem) Given a set of learning algorithms $\mathcal{A}$, where the hyperparameters of each algorithm $A^{(j)} \in \mathcal{A}$ have the configuration space $\mathbf{\Lambda}^{(j)}$. The CASH problem can be defined as finding the algorithm A^* and its hyperparameter configuration $\boldsymbol{\lambda}^*$ that yields the model with the best performance on the validation set (Thornton et al., 2013):

$$A^*, \boldsymbol{\lambda}^* \in \underset{A^{(j)} \in \mathcal{A}, \boldsymbol{\lambda} \in \mathbf{\Lambda}^{(j)}}{\operatorname{argmin}} \frac{1}{k} \cdot \sum_{i=1}^{k} \mathcal{L}(A_{\boldsymbol{\lambda}}^{(j)}, D_{train}^{(i)}, D_{valid}^{(i)}). \tag{3.3}$$

Dealing with the algorithm selection problem requires finding an approach for setting the hyperparameters of the algorithms. For instance, this can be done by using the default hyperparameter values or optimising the hyperparameters of each algorithm separately before performing algorithm selection. This approach is less relevant for modern AutoML research and thus will not be covered in this chapter. The HPO problem is still frequently tackled in NAS systems, where only the hyperparameters of one type of learning algorithm are optimised. HPO procedures are also used within approaches for solving the CASH problem. Section 3.2 covers HPO strategies, and in Section 3.3, we will discuss how these techniques are used in NAS. The CASH problem is the most general AutoML problem and typically considers a much more diverse search space based on different machine learning algorithms and other elements of a machine learning pipeline, such as pre-processors. In Section 3.4, we will review AutoML systems that are defined based on the CASH problem.

3.2 Hyperparameter optimisation

In some cases, we are already committed to using a specific machine learning algorithm (e.g., a support vector machine) because of external constraints, and the goal is to optimise the hyperparameters of this specific algorithm. In this case, we deal with a hyperparameter optimisation (HPO) problem. In this section, we will describe solutions to the HPO problem.

HPO procedures can be described in terms of a search space, a search strategy and an evaluation mechanism, which we address in Sections 3.2.1, 3.2.2 and 3.2.3, respectively. In Section 3.2.4, we describe several popular libraries for HPO.

3.2.1 Search spaces

The search space of HPO typically consists of all hyperparameters that are relevant to be optimised (numerical as well as categorical, and in many cases conditional). To find a good hyperparameter configuration, the search space needs to contain at least some regions where such good configurations can be found. When the search space is too large, the success of the optimisation procedure highly depends on the quality of the search algorithm and might take more time.

HPO frameworks often take as input a search space, requiring the user to bring in expertise on relevant and important hyperparameters and how to search for good values. Various research lines went in the direction of generating this knowledge (Hutter et al., 2014; Snoek et al., 2014; van Rijn and Hutter, 2018), whereas other works aim to automatically construct a good search space based on historical data (Hoos, 2012; Perrone et al., 2019; Pfisterer et al., 2021; Wistuba et al., 2015).

Important design criteria are the set of hyperparameters to be optimised, the dependencies between these hyperparameters (i.e., conditional hyperparameters), the ranges of values to be considered for numerical hyperparameters, and potential transformations.

3.2.2 Search strategy

In this section, we describe various search strategies for HPO. We perform the HPO with search strategies on the aforementioned search space. Different types of optimisation algorithms are used in this domain. We start from the most simple ones, grid search, and random search. Then we introduce Bayesian optimisation and variants, which aim to reduce the number of function evaluations by incorporating knowledge

about the performance of configurations encountered earlier in the search process. We also cover reinforcement learning-based approaches, evolutionary algorithms, Monte Carlo tree search and gradient-based optimisation.

Grid search and random search

Grid search and random search are the earliest and simplest search techniques used for HPO. Grid search (also known as a parameter sweep) selects the best configuration by exhaustively evaluating all possible combinations of hyperparameter values. To use grid search on continuous hyperparameters, the respective domains have to be mapped to a set of discrete values. In general, grid search performs reasonably well in the low-dimensional search spaces induced by small numbers of hyperparameters. However, performing a grid search in high-dimensional search spaces or with high resolution for discretised hyperparameters involves the evaluation of very large numbers of configurations. To improve efficiency in such scenarios, Larochelle et al. (2007) proposed a multi-resolution approach in which, first, a configuration is selected from a coarse-grained configuration space, and next, a higher resolution search is performed in the vicinity of the selected configuration.

As an alternative to grid search, random search samples configurations from the search space at random; this does not require the discretisation of continuous hyperparameters. Grid and random search approaches share the advantage of simplicity, ease of implementation and excellent parallelisability. Bergstra and Bengio (2012) demonstrated that random search tends to outperform grid search in high-dimensional spaces when using the same computational budget. This is explained by the fact that each hyperparameter contributes differently to the overall loss. Grid search unnecessarily allocates too many resources to the evaluation of unimportant hyperparameters. Figure 3.1 provides an illustrative example comparing grid search and random search. Assuming that nine configurations are evaluated for optimising a function f, which is highly influenced by a function h, it can be seen that with random search, all nine evaluations explore distinct values over this function h, whereas grid search only explores three distinct values. Bergstra and Bengio (2012) further compared various sampling strategies for random search and found that Sobol sequences (Antonov and Saleev, 1979) offer a particularly effective way to perform random sampling. Sobol sequences have later been adopted by systems such as SMAC3 (Lindauer et al., 2022).

Random and grid search are both commonly used as baselines, and many popular machine learning libraries include implementations of these simple search techniques (see, e.g., Scikit-learn (Buitinck et al., 2013), Tune (Liaw et al., 2018), Talos (software,

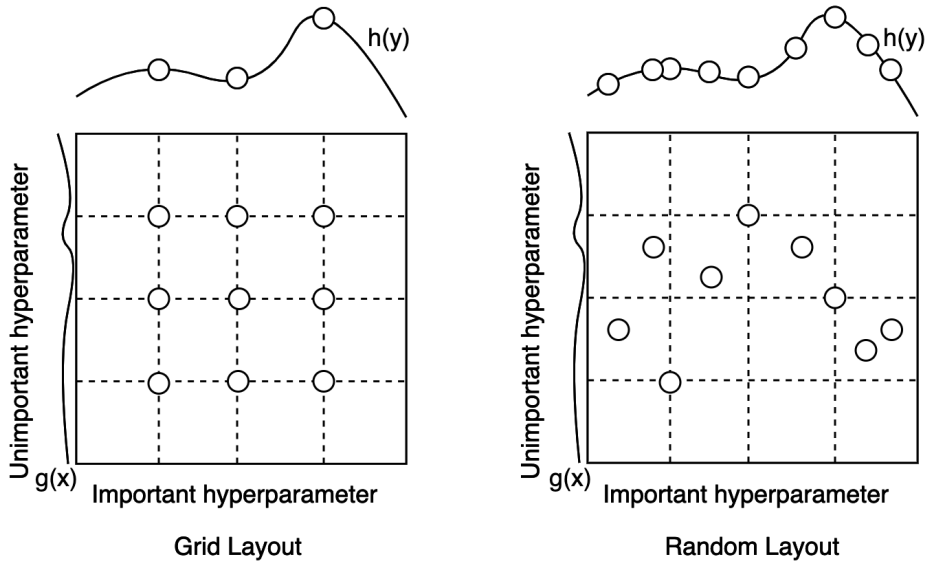

Figure 3.1: The difference between random search and grid search. Each approach performs 9 evaluations for optimising a function $f(x, y) = g(x) + h(y)$ that depends on two parameters, x and y. The functions $g(x)$ and $h(y)$ are shown on the left and above the representation of the search space, respectively. Assume that $f(x, y)$ is strongly influenced by $h(y)$ and only weakly by $g(x)$. Grid search only evaluates $h(y)$ for three distinct values of y. However, with random search, all the nine evaluations explore distinct y and hence $h(y)$, increasing the chance of also finding a good value of $f(x, y)$ (Bergstra and Bengio, 2012).

2020), and H2O (H2O.ai, 2017)).

Bayesian optimisation and variants

Evaluating a hyperparameter configuration can be a computationally expensive task since it requires training and validating a machine learning model. Both grid search and random search require a relatively large number of such function evaluations, especially when the number of hyperparameters is large. Bayesian optimisation, sometimes referred to as sequential model-based optimisation, aims to reduce the number of function evaluations by incorporating knowledge about the performance of configurations encountered earlier in the search process. It uses concepts from Bayesian statistics, in particular in the way a statistical model is used to estimate the probability distribution over the possible performance (loss) values of a previously unseen hyperparameter

configuration.

Bayesian optimisation utilises knowledge about the search space by fitting a model to the data collected from previously evaluated configurations (the objective function f); this model is often referred to as a surrogate model or empirical performance model. The two key ingredients of Bayesian optimisation are the (1) surrogate model and the (2) acquisition function. The surrogate model is a probabilistic model for approximating the objective function f that maps hyperparameter configurations to the respective performance values. Bayesian optimisation uses a prior belief of the shape of f and updates this prior with new evaluations of f (on selected hyperparameter settings) to achieve a posterior that better approximates f (Jones et al., 1998; Shahriari et al., 2016). The acquisition function is used to guide the process of sampling from the hyperparameter space in the next round by evaluating the utility of candidate hyperparameter settings based on the surrogate model. The acquisition function, in principle, uses the mean and the uncertainty in the posterior distribution derived from the surrogate model to determine which hyperparameter configuration should be evaluated next. Figure 3.2 shows a snapshot from the 1-D Bayesian optimisation procedure after seven iterations.

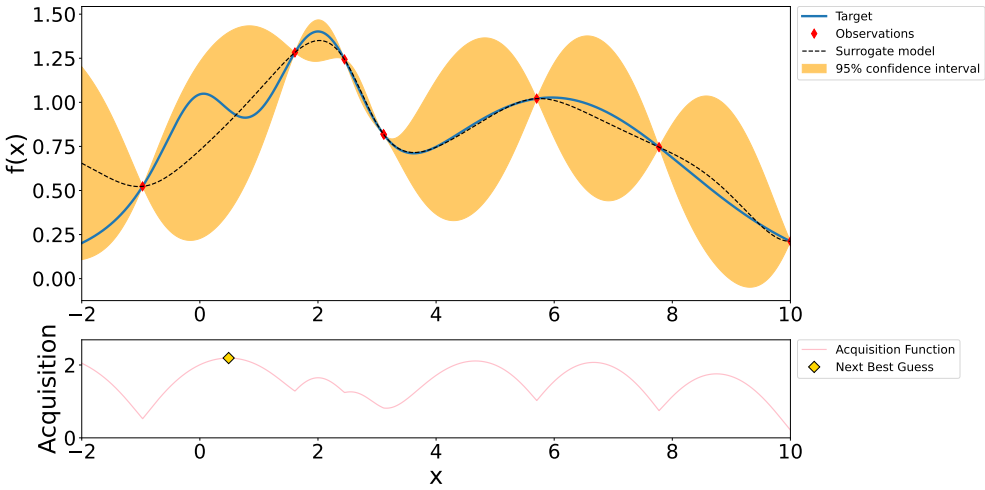

Figure 3.2: Bayesian optimisation using a Gaussian process-based surrogate model and the upper confidence bound as acquisition function after seven function evaluations. The solid blue line (target) is the function that is to be optimised, and the dashed line and yellow surface are the mean performance and associated uncertainty derived from the surrogate model, respectively. There is no uncertainty at points where the target function has been evaluated. The acquisition function determines the utility of the configurations to be evaluated (Nogueira, 2014).

Many different functions can be used as surrogate models and acquisition functions in Bayesian optimisation; for further details, we refer the interested reader to the survey on Bayesian optimisation by Shahriari et al. (2016). Many researchers have developed Bayesian optimisation methods for HPO (Bergstra et al., 2011; Hutter et al., 2011; Snoek et al., 2012). The key difference between these approaches lies in the choice of the surrogate model and the acquisition function.

We briefly introduce widely used surrogate models and acquisition functions below and refer interested readers to Jones et al. (1998) for a more detailed discussion.

Surrogate models: The two most widely used surrogate models are Gaussian processes and random forests; the use of the former has been explored by various authors (Desautels et al., 2014; Ginsbourger et al., 2010; Snoek et al., 2012). A Gaussian process is a non-parametric statistical model defined based on its prior mean and covariance functions. The posterior mean and covariance at any evaluated point represent the model's prediction and its uncertainty. Such a model is suitable as a surrogate model in Bayesian optimisation, as it provides the uncertainty estimates that are needed for the acquisition function to choose the next evaluation point. The computational complexity of training a Gaussian process model is $O(n^3)$, the cost of inversion of its covariance matrix, where n is the number of observations. In the context of Bayesian optimisation, each observation is a function evaluation carried out earlier in the optimisation process. One of the main drawbacks of Gaussian processes arises from the fact that the computational cost of training increases cubically. Gaussian processes with most standard kernels do not scale to high-dimensional data sets. However, methods such as sparsifying Gaussian processes (Seeger et al., 2003) have improved applicability to large data sets by reducing the rank of the covariance matrix.

Another limitation of Gaussian-process-based Bayesian optimisation methods is that they are only applicable to continuous hyperparameter spaces and cannot natively handle integer, categorical or conditional hyperparameters. Additional approximations need to be introduced to handle these commonly encountered types of hyperparameters. For instance, for integer-valued hyperparameters, continuous values are rounded to the closest integer after optimising the acquisition function. For categorical hyperparameters, an approximation based on a one-hot encoding can be used. Garrido-Merchán and Hernández-Lobato (2020) show that these approximations may lead to the failure of the Bayesian optimisation process, as they ignore that some configurations are invalid and may put probability mass on points where f cannot be evaluated. They further provide a systematic transformation of the categorical and

integer variables that permits the assumption that the value of f remains constant in certain areas of the underlying search space.

An alternative class of surrogate models are based on random forest regression (Breiman, 2001); these have been demonstrated to be more scalable than Gaussian processes (Hutter et al., 2011). A random forest model creates an ensemble of decision trees that can collectively approximate the response surface of the given objective function f. Since training individual trees is parallelisable, and each tree is trained only on a sample of data, this technique scales much better to large data sets. Furthermore, it can quite easily handle different hyperparameter types, including conditional hyperparameters. The drawbacks of random forest regression models are that the uncertainty estimation is known to be less accurate than those of a Gaussian process and that they do not extrapolate well outside the observed data points; the latter also applies to most Gaussian process models.

Acquisition functions: The acquisition function determines, based on a given surrogate model, which point in the search space to evaluate next. Expected improvement is a commonly used acquisition function; it is composed of two terms relating to the (1) expected value at a given point of the function to be optimised and (2) the associated variance (or uncertainty). This combination naturally provides a trade-off between exploitation (around a promising point) and exploration (in unknown areas). By considering the expected value, the search is focused on regions of the search space containing good candidate solutions with high probability. In contrast, the maximum uncertainty term encourages the exploration of regions that are weakly explored or in which candidate solutions of highly variable quality have been observed (Jones et al., 1998). There are other acquisition functions, such as the upper confidence bound (Hoffman and Shahriari, 2014) of uncertainty for every query point, or information-theoretic approaches, such as entropy search (Hennig and Schuler, 2012).

Parallelism: Bayesian optimisation takes advantage of all information collected during the optimisation process by evaluating samples sequentially, one by one. This makes it much more data-efficient than the grid and random search approaches. However, as the surrogate model typically suggests only a single best configuration to evaluate next, it is not trivial to parallelise Bayesian optimisation. To address this issue, Ginsbourger et al. (2011) extend the original Bayesian optimisation framework by proposing a multi-point expected improvement criterion for the simultaneous selection of multiple points that are evaluated in parallel. In this approach, some evaluations can be performed while the results of previous evaluations are not yet fully available. This is enabled by injecting partial knowledge of ongoing evaluations into the expected

improvement formulation.

Sequential model-based algorithm configuration (SMAC) (Hutter et al., 2011) is a Bayesian optimisation procedure for general algorithm configuration. SMAC uses random forests as surrogate models and expected improvement as an acquisition function. The random forest model allows SMAC to optimise conditional and categorical hyperparameters. Further improvements to SMAC have been proposed (e.g., pruning the search space to increase efficiency (Li et al., 2022)).

Tree-structured Parzen estimator (TPE) (Bergstra et al., 2011) is another approach based on Bayesian optimisation that uses expected improvement as an acquisition function and TPE to model the probabilities and distributions of the target function. Gaussian process approaches model the probability $p(y \mid \boldsymbol{\lambda})$ directly, where $\boldsymbol{\lambda}$ denotes the configurations, and y indicates the performance observed for a given configuration $\boldsymbol{\lambda}$. TPE, however, calculates $p(\boldsymbol{\lambda} \mid y)$ and $p(y)$. $p(\boldsymbol{\lambda} \mid y)$ is modelled by replacing the distributions of the configuration prior with two non-parametric densities that make a distinction between good and bad configurations: (1) the density of configurations that have a loss below a given threshold (set to a quantile of observed y values) and (2) the density of those with higher loss. Through maintaining a sorted list of configurations in memory, the run-time of TPE scales linearly with respect to the number of hyperparameters. TPE can also be used for conditional hyperparameters and tree-structured configuration spaces. TPE is implemented in the HyperOpt library (Bergstra et al., 2013).

Reinforcement learning-based approaches

Reinforcement learning concerns sequential decision processes in state space (Sutton and Barto, 2018). The agent or a controller learns to find optimal paths as a Markov decision process with a number of states $\mathcal{S}$ and an action space $\mathcal{U}$. At each state $s_i \in \mathcal{S}$, there are a number of actions $\mathcal{U}(s_i) \subseteq \mathcal{U}$ that can be selected by the agent. Taking an action $u \in \mathcal{U}(s_i)$ will create a state transition from state s_i to state s_j with probability $p_{s'|s,u}(s_j \mid s_i, u)$. The agent interacts with the environment at points in time. At each timestamp t, the agent receives an immediate reward r_t, based on its action u and the transition between s_i and s_j. The goal in reinforcement learning is to determine a policy, i.e., a function that determines for each state a probability distribution over actions, such that a cumulative reward function computed from the immediate rewards r_t (in many cases, a weighted sum over t) is maximised.

Value-based and policy-based approaches are two well-known classes of reinforcement learning methods. In value-based approaches, the agent estimates the optimal

value function Q that defines which action to take in a particular state to achieve the maximum reward. The policy-based methods directly learn the optimal policy π.

Q-learning (Watkins, 1989) is an example of a value-based approach where the agent learns a look-up table of actions and states by iteratively updating the equation (Baker et al., 2017):

$$Q_{t+1}(s_i, u) = (1 - \alpha) \cdot Q_t(s_i, u) + \alpha \cdot (r_t + \gamma \cdot \max_{u' \in \mathcal{U}(s_j)} Q_t(s_j, u')) \tag{3.4}$$

Here, α is the Q-learning rate determining the weight of new information to old information, and γ is a discount factor which defines the weight given to rewards depending on how far in the future they will be collected. Originally, Q-learning is defined for discrete spaces and which makes it useful for optimising discrete (or discretised) hyperparameters. There are, however, extension of Q-learning to continuous action spaces as well (Millán et al., 2002), which can potentially be used to optimise continuous hyperparameters.

Policy-based approaches have better convergence properties than Q-learning approaches and are suited for higher-dimensional and continuous actions (i.e., optimising continuous hyperparameters). The policy-gradient approach is an example of a policy-based approach that has been used for HPO. The approach taken is to directly learn the optimal policy $\pi(u_t \mid s_t)$ given the history of state-action pairs (s_t, u_t). Policies are defined by a number of parameters θ, and the general goal is to optimise the parameters of the policy such that the total expected reward is optimised. The REINFORCE algorithm (Williams, 1992) is a well-known algorithm to calculate the policy parameters θ by maximising the expected reward

$$J(\theta) = \mathbb{E}_{P(u_{1:T};\theta)}[R] \tag{3.5}$$

Here, $P(u_{1:T})$ denotes the probability of a sequence of T actions that have resulted in reward R after T time steps.

The search for good hyperparameter settings is usually seen as a *sequential* decision process, in which the values of one or more hyperparameters are modified in each step. Different approaches have been taken so far to formulate the HPO problem within an RL framework. Each state is commonly defined by hyperparameters and their values. The way states and actions are defined could allow step-wise changes to only a single hyperparameter (Wu et al., 2020a; Zoph and Le, 2017), to a selected group of hyperparameters (Baker et al., 2017) or to all hyperparameters at once (Jomaa

et al., 2019). Additionally, the states could hold other information, such as data set meta-features or the history of evaluated hyperparameter configurations (Jomaa et al., 2019). Formulating states that allow changes to only a single hyperparameter at each step, rather than to a group, allows treating the configuration of individual hyperparameters as a sequential decision process as well. This allows to implicitly consider the conditionality among hyperparameters in the search space by remembering previous decisions. This also impacts the size of the action space, which implicitly determines the search space. Let us assume that n hyperparameters are to be optimised, each with a domain denoted by Λ_i.

Treating the hyperparameters individually will reduce the action space size from $\mathbf{\Lambda} = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$ to $\mathbf{\Lambda} = \Lambda_1 + \Lambda_2 + \cdots + \Lambda_n$ (Wu et al., 2020a).

Hyp-RL (Jomaa et al., 2019) is a general HPO method based on Q-learning. In Hyp-RL each action corresponds to a hyperparameter configuration (to set all hyperparameters) being rewarded based on the validation loss of the configured model r_t. The total reward is calculated by accumulating the validation loss of the models configured in a sequence of actions. The policy selects the actions that maximise the discounted cumulative reward through iterative Q-learning updates from an initial state (i.e., hyperparameters initialised randomly or to their default value).

In the policy-based approach proposed by Zoph and Le (2017), each action corresponds to setting one hyperparameter, and a sequence of T actions in a trajectory leading to the configuration of all hyperparameters is rewarded by the validation accuracy of the configured model R. The optimal policy is selecting the trajectory that maximises the expected reward J based on computing the policy gradient to update the controller parameters θ based on the reward.

In general, reinforcement learning methods have been used for HPO (Jomaa et al., 2019; Wu et al., 2020a) and more commonly in neural architecture search (examples for the latter will be given in Section 3.3.2).

Evolutionary Algorithms

Evolutionary algorithms (Bäck et al., 1997; Simon, 2013) are population-based global optimisation algorithms inspired by biological evolution. They work on a set (*population*) of P solution candidates (*individuals*). Starting from an initial population, evolutionary algorithms iteratively vary the population (giving rise to a sequence of *generations*), as illustrated in Algorithm 1 (Bäck et al., 2018). In each generation, a population $P(t)$ (*parent individuals*) of size μ are selected, from which new individuals $P'(t), P''(t)$ (*offspring individuals*) are generated using variation operators; then, the

Algorithm 1 Evolutionary Algorithm

Input: population size μ, mutation group size λ, termination, recombination, mutation and selection parameters $\mathbf{\Theta_t}, \mathbf{\Theta_r}, \mathbf{\Theta_m}, \mathbf{\Theta_s}$

Output: the best individual a^* or the best population P^* found during the run

$t \leftarrow 0$;
$P(t) \leftarrow \textsc{Initialise}(\mu)$;
$F(t) \leftarrow \textsc{Evaluate}(P(t), \mu)$;
while $(\textsc{Termination}(P(t), \mathbf{\Theta_t}) \neq \text{True})$ **do**
    $P'(t) \leftarrow \textsc{Recombine}(P(t), \mathbf{\Theta_r})$;
    $P''(t) \leftarrow \textsc{Mutate}(P'(t), \mathbf{\Theta_m}))$;
    $\mathbf{F}(t) \leftarrow \textsc{Evaluate}(P''(t), \lambda)$;
    $P(t+1) \leftarrow \textsc{SelectReplace}(P''(t), \mathbf{F}(t), \mu, \mathbf{\Theta_s})$
    $t \leftarrow t + 1$;
end while

next set of individuals (*survivors*) are selected from the previous population and the offspring. These selected individuals form the new population for the next generation. The selection of parent and survivor individuals is typically based on the *fitness* values $\mathbf{F}(t)$ (i.e., objective values) of the individuals, where individuals with higher fitness are preferred over individuals with lower fitness. Typical variation operators are *crossover* and *mutation*. Crossover combines two or more parent individuals to transfer the beneficial features of the parents to the offspring. The mutation operator applies small random changes to individuals to increase the diversity within the population.

In the context of HPO, mutation and crossover are analogous to exploitation and exploration, respectively. When applying an evolutionary algorithm to an optimisation problem, one has to decide how solutions are encoded as individuals. For example, an integer variable can be encoded as an integer but also as a list of binary variables.

In evolutionary algorithms, the *genome* encoding of an individual includes the representation of a possible solution to the problem, the actual solution after evolution is termed *phenotype* and its encoding is termed *genotype* (Bäck et al., 2018). Tani et al. (2021) evaluated an evolutionary algorithm and particle swarm optimisation (Kennedy and Eberhart, 1995) on two HPO tasks, concluding that both can outperform random search and gradient descent. There are two special types of evolutionary algorithms which are frequently used in the context of HPO and AutoML: genetic programming and CMA-ES. In the following, these approaches are briefly outlined.

Genetic programming (Koza, 1994) is a form of an evolutionary algorithm that evolves programs composed of functions, which work on primary inputs and/or outputs of other functions. An example of such a program could be a mathematical expression,

where the functions are mathematical operators (e.g., addition, sine, logarithm), and the actual optimisation task could be to find an expression, which best fits some experimental data. Often a tree representation is used to represent programs in genetic programming. TPOT (Olson et al., 2016a) is an example of an AutoML system that uses genetic programming for the optimisation of machine learning pipelines and their hyperparameters (see Section 3.4.4 for more details).

Covariance matrix adaptation evolution strategy (CMA-ES) (Hansen et al., 2003) is an evolutionary algorithm, which has been demonstrated to be very efficient for a number of black-box optimisation tasks, including HPO (Jedrzejewski-Szmek et al., 2018; Loshchilov and Hutter, 2016; Loshchilov et al., 2012; Watanabe and Roux, 2014). It samples candidate solutions from a multivariate normal distribution whose mean, and covariance matrix is updated based on the performance of the individuals in the population. CMA-ES works well on non-linear and non-convex optimisation tasks; it is typically used for problems with search spaces with three up to a hundred dimensions. CMA-ES has shown good performance compared to other black-box optimisers, such as Bayesian optimisation, on continuous black-box optimisation benchmarks (Loshchilov et al., 2013). While Bayesian optimisation is recommended for conditional search spaces, CMA-ES is recommended if the search space only contains continuous hyperparameters and the objective function is cheap, or the evaluation budget is large (Mendoza et al., 2016).

Monte Carlo tree search

Monte Carlo tree search (MCTS) is an approach for addressing state-space Markovian sequential decision problems (Kocsis and Szepesvári, 2006) working based on a randomised evaluation of a search tree. This approach has been successfully used in game AI to predict the best moves to reach a winning position in a game (Chaslot et al., 2008a), such as Go (Silver et al., 2017) or Chess (Silver et al., 2018). For a given search problem, the MCTS algorithm builds a tree where each node (representing states) includes information about the current value of the position (usually the average of the results of simulated games visiting this node) and the visit count of this position. The MCTS algorithm repeats the following steps (Chaslot et al., 2008b): (1) traverse the tree through the selection of the best next node to move to (through balancing between exploitation and exploration based on the statistics stored), (2) expansion of the selected node by adding new child nodes to the tree to increase the options to win the game, (3) simulation, or playout, to finish the game by traversing the search tree multiple ways in a random way and further assigning a reward based

on calculating how close the output of random decisions was from the final winning output, and (iv) back-propagation to update each node that was traversed in the tree based on the result of the simulation.

The main power of MCTS-based approaches lies in addressing sequential problems and are thus better suited for optimising hyperparameters representing ordered decisions such as hyperparameters involved in creating a pipeline of a fixed number of components (e.g., first selecting a data pre-processors, afterwards feature selectors, and finally a machine learning algorithm). This works especially well in combination with pipelines with a fixed structure, as considered, for example, in MOSAIC (Rakotoarison et al., 2019) (see Section 3.4). When the pipeline structure is fixed, it can be represented as a search tree that can be traversed by MCTS. Internal nodes in the search tree represent *partial configurations* in which only the first pre-processing operators are fixed, whereas a leaf node represents a full configuration. A surrogate model that generalises over the full configuration space, which can also assess the quality of partial configurations (as represented by internal nodes), can be employed to determine the performance of a given node (Rakotoarison et al., 2019). This can be done, for example, by means of sampling techniques, where a pre-defined number of leaf nodes (representing full configurations) are being evaluated, and the average of those represents the quality of the partial configuration. Once a playout-operation has determined a suitable leaf-node, the configuration that belongs to the leaf node is instantiated and evaluated on the real data, and the measured performance is back-propagated into the internal tree representation. Also, the surrogate is being updated with this additional information. MCTS algorithms for hyperparameter optimisation are only researched to a limited degree.

Gradient-based optimisation

The gradient descent algorithm, classically used for setting the parameters of models, can be extended to jointly optimise the hyperparameters of the algorithms as well. As mentioned before, random search has shown promising results in the context of optimising small numbers of hyperparameters. For moderately higher dimensions, more complex methods (e.g., Bayesian optimisation) are preferred. However, when dealing with neural networks, besides a moderate amount of hyperparameters, there are also millions (if not billions) of parameters to optimise (i.e., the weights and bias values). Typically, the parameters of a neural network are optimised using a gradient descent method, whereas the hyperparameters are optimised by an HPO method. However, the optimised validation loss with respect to the hyperparameter can be estimated,

allowing gradient descent methods to traverse the loss landscape with respect to hyperparameter values. Recently, gradient-based optimisation has shown promising results for optimising very large numbers of (hyper)parameters (see, e.g., Lorraine et al., 2020) and also in meta-learning (see, e.g., Rajeswaran et al., 2019).

One of the earlier works in gradient-based optimisation for differentiable and continuous HPO was proposed by Bengio (2000). This approach uses reverse mode differentiation or backpropagation and focuses on small-scale continuous HPO based on differentiable objective functions, such as squared loss and logistic loss, that can be optimised with gradient descent. One of the main limitations of this approach is that it requires intermediate variables to be maintained in memory for the reverse pass of the backpropagation procedure. This gives rise to prohibitively large memory requirements and limits the practical applicability of this method.

There has been more recent research on memory-efficient methods for approximating the hypergradients (i.e., the gradient of the validation loss with respect to hyperparameters). These methods generally fall into two categories: (1) iterative differentiation (see, e.g., Franceschi et al., 2017; Maclaurin et al., 2015), which approximates the hypergradients by defining a sequence of functions that recursively approximate one another; and (2) approximate implicit differentiation (see, e.g., Lorraine et al., 2020), which defines an implicit function for the hypergradients through applying the implicit function theorem. Empirical results from several studies indicate that implicit differentiation methods tend to be more memory-efficient (see, e.g., Grazzi et al., 2020; Rajeswaran et al., 2019).

Gradient-based methods are only applicable to continuous hyperparameters and twice-differentiable loss functions. It is also possible to extend the use of these techniques to discrete hyperparameters using continuous relaxation methods (see, e.g., Jang et al., 2017). For instance, Jang et al. (2017) propose the Gumble-Softmax estimator to represent samples of one-hot encoded categorical variables with a differentiable distribution. As explained in Section 3.3.2, DARTS (Liu et al., 2019b) represents categorical hyperparameters by means of continuous variables, using the softmax operation in a similar manner.

3.2.3 Performance evaluation techniques

HPO techniques evaluate various configurations from the search space and, based on these evaluations, in each step, select the one deemed to be most promising. This is often done using nested cross-validation (Varma and Simon, 2006): the data set

is split into a training, a validation and a testing partition. All configurations are trained on the training set and evaluated on the validation set. The testing partition is set aside and only used for the final evaluation of the best configuration. We thus find evaluation procedures at an inner and outer level. The HPO method employs the evaluation procedure at the inner level to assess how well a specific configuration will perform on the given data set. Eventually, the evaluation procedure at the outer level assesses how well the HPO method as a whole works. The inner evaluation procedure is under the control of the (user of the) HPO method, whereas this is not the case for the outer evaluation procedure. The main focus of this section is on the inner evaluation procedure. Evaluating various candidate configurations can be computationally expensive, and therefore several procedures have been developed to speed up this process. We will review three general types of procedures: racing methods, methods that evaluate at lower budgets, and learning curve extrapolation methods.

Racing methods

One way of speeding up the internal evaluation procedure is by racing. When the internal evaluation procedure is subject to a cross-validation scheme, it will run each candidate various times, each time trained and tested on a different part of the data. In some cases, it becomes already clear after a few of those iterations that the candidate configuration will not be competitive with the best configuration found previously. Hoeffding races (Maron and Moore, 1993) discards bad models, and allocates more computational effort at differentiating between the better ones to find good models. F-race (Birattari et al., 2002) is a statistical approach-based racing algorithm used for configuring metaheuristics. Hoeffding races and F-race inspired iRace. iRace implements this design criterion by employing a statistical test, specifically, the Friedman test or the t-test (López-Ibáñez et al., 2016). When this test determines that the evaluations of a given candidate configuration are not showing statistically significantly better performance than the best configuration seen so far, no further evaluations are being conducted, and the evaluation procedure is stopped early.

Statistical tests can be unnecessarily conservative, therefore wasting compute time on candidate configurations that appear to be not competitive but have not shown to be statistically significantly dominated. An alternative to this is random aggressive online racing (ROAR), an extension to random search that applies the racing strategy in a more aggressive way (Hutter et al., 2011). It stops the evaluation of a candidate configuration after the average performance on recent validation folds is lower than

the average performance of the current best algorithm. This way, many candidate configurations can already be dropped after a single validation fold at the risk of occasionally eliminating a superior candidate solution.

Evaluation using a lower budget

One way to reduce the training time is by estimating the performance using a reduced training budget (Mohr and van Rijn, 2022). This can be achieved in various ways: one can subsample the given set of data points, decrease the number of attributes, and lower image resolution in computer vision tasks (Chrabaszcz et al., 2017), or limit the training process to a few iterations (e.g., train a neural network with a given number of epochs) or up to a given cutoff in running time. The latter approach is sometimes called low-fidelity learning (Binder et al., 2020), or multi-fidelity in cases where multiple training budgets are used (Hu et al., 2019).

When evaluating on a lower budget, one may wonder at which budget to sample. Obvious answers to this could be using a percentage of the data set, e.g., 10% or 50% of the train data. Provost et al. (1999) addressed this problem by proposing *efficient progressive sampling*, which has later been used in various other approaches, such as the learning curve method by Leite and Brazdil (2010) and successive halving (Jamieson and Talwalkar, 2016). They motivate their approach by stating that when having access to a large data set, not all data points need to be utilised, and propose a technique that detects an appropriate number of data points that should be used for a given configuration. They propose to evaluate configurations using multiple budgets, using a geometrical schedule, e.g., using a data set of size $n = \{64, 128, 256, 512, \ldots\}$, until convergence is detected. Here, convergence refers to the fact that the learning curve is saturated, and performance does not further improve when more data is provided. They prove that this type of schedule is asymptotically optimal in terms of computation time. In other words, the authors claim that this procedure will select a data set size that obtains maximised performance, utilising at most a constant factor more running time than when training the algorithm on all available data. However, often this procedure will actually be faster than training the algorithm on all data available, presumably in cases when an algorithm obtains a saturated performance with fewer data than all available data.

Successive halving (Jamieson and Talwalkar, 2016) is a multi-armed bandit method that aims to reduce training time by allocating resources more efficiently. This method also uses the geometrical schedule as proposed by Provost et al. (1999). Initially, a random set of configurations is sampled and evaluated at a certain budget (e.g., data

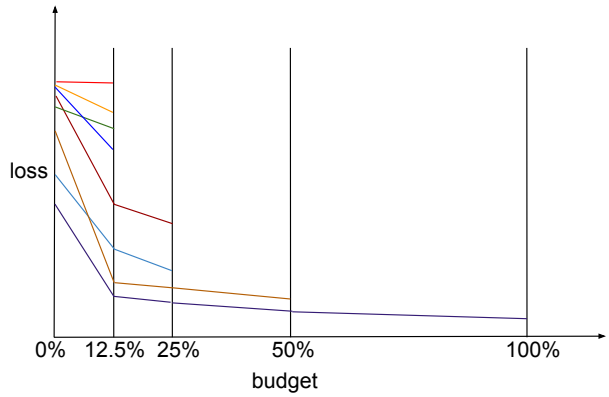

Figure 3.3: Illustration of the sample schedule of successive halving. The x-axis represents the budget in terms of data points used to evaluate a given configuration, whereas the y-axis represents the performance in terms of loss (lower is better). Initially, eight configurations are evaluated on 12.5% of the maximum number of data points, after which the worst-performing four configurations are dropped. The remaining four configurations are evaluated on 25% of the maximum number of data points, after which the worst-performing two configurations are dropped. This procedure is repeated until only one configuration is evaluated on 100% of the data points (Feurer and Hutter, 2019).

samples). This budget represents only a fraction of the normally required training time of the candidate configurations, ensuring that less time is taken. The performance of the selected configurations is evaluated, and only the top percentage (e.g., 50%) of the configurations with the highest performance are selected for the next round, where they are evaluated on a larger data sample. This process is repeated until only one configuration is left, as illustrated in Figure 3.3.

Successive halving has several hyperparameters that also need to be determined. For example, the minimal budget and the initial number of configurations are important hyperparameters that can significantly influence performance. Setting the minimal budget too low might exclude certain configurations from being considered since some configurations may require a high budget to excel and might thus be dropped too early when the initial budget is too low. Li et al. (2017) proposed hyperband, an extension to successive halving that aims to dynamically balance the number of configurations and the initial budget allocated for evaluating the configurations. Hyperband is essentially a loop around successive halving, invoking it multiple times with a different minimal budget and number of configurations. The maximum budget per learning algorithm is fixed. Each iteration of successive halving is called a bracket. Generally, the configurations per successive halving bracket are sampled completely at

random from a larger configuration space. Hyperband starts with a bracket that evaluates a high number of configurations with a low budget; in each subsequent bracket, the number of initial configurations is decreased while the initial budget is increased. Effectively, each subsequent bracket of successive halving will explore the same sample sizes as the previous bracket, except for the first. As an edge case, the final bracket of successive halving is run with only a few configurations, and the initial budget is the same as the maximum budget. Using this property, Li et al. (2017) proved that hyperband is never more than a log factor slower than random search.

Successive halving and hyperband are performance evaluation methods that work well in combination with random search but can also be naturally integrated into other search strategies. Baker et al. (2018) propose fast-hyperband, a method that employs a machine learning model to predict whether an evaluated configuration can improve over the best configuration found so far. Indeed, successive halving and hyperband are quite static in the way they drop candidate configurations, and by employing a model, better-informed decisions can be made.

Various methods have been proposed that directly combine multi-fidelity methods with Bayesian optimisation. While most of these methods can be used 'out of the box' in combination with most search algorithms described in Section 3.2.2, Bayesian optimisation is more complex, as it trains an internal surrogate model. Falkner et al. (2018) combine hyperband with Bayesian optimisation to select new candidate configurations. As hyperband usually samples uniformly at random, it can greatly benefit from focusing on good regions on the search space. Specifically, Falkner et al. (2018) use the TPE as a surrogate model (Bergstra et al., 2011). There is empirical evidence from the work of Zela et al. (2018) that the correlation between the performance with a low training budget and high training budget is weak. In light of this, Li et al. (2017) and Falkner et al. (2018) suggest increasing the sample size gradually.

Early stopping by learning curve extrapolation

Learning curves describe the performance of learning algorithms as a function of a given resource, e.g., the number of training iterations or the number of training examples, and are commonly used to extrapolate to the performance on the full budget (Mohr and van Rijn, 2022). Various resources exist to obtain historic learning curves on many data sets, such as OpenML (Vanschoren et al., 2013), LCBench (Zimmer et al., 2021) and LCDB (Mohr et al., 2022).

Leite and Brazdil (2005) proposed a method that leverages similarities in the learning curves observed for different data sets. Their work builds upon the assumption

that if the data sets are similar, the configurations will also perform or rank similarly. The method requires access to a set of learning curves on historic data sets; it uses a distance function between learning curves and a k-NN-based algorithm to determine to which historic data sets a given data set is most similar. Utilising this distance function, learning curves of the same configuration on different data sets are being identified based on their shape similarity, assuming that similar data sets will lead to similar learning curves. After identifying similar data sets, knowledge of configurations that worked well on these is applied to the current data set. Later, Leite and Brazdil (2010) extended this work by also taking into consideration the so-called meta-features, and van Rijn et al. (2015) further extended the approach to also take into consideration a measure of running time. *Freeze-thaw Bayesian optimisation* allows to dynamically stop (freeze) and restart (thaw) the training procedure (Swersky et al., 2014b). The optimisation of hyperparameters stops when it seems unlikely that it will lead to finding a model with a small loss. Then, another hyperparameter configuration will be evaluated. In case the chances for finding a small loss for a stopped HPO process have increased, that process can be resumed.

Domhan et al. (2015) proposed a technique for the early termination of unpromising configurations using a probabilistic model that predicts the performance distribution based on the first part of a learning curve. The partially observed learning curve is modelled using a set of 11 parametric curve models. In order to yield accurate predictions, this method usually requires a relatively long partial learning curve. Later, Klein et al. (2017c) improved this idea by proposing a neural network-based method, incorporating specific learning curve operators to learn the prediction model across different learning curves of various algorithms on the same data set. Both Domhan et al. (2015) and Klein et al. (2017c) assume that it is possible to model the learning curve by using a set of function families, such as the inverse power law (Brumen et al., 2021).

Klein et al. (2017a) proposed FABOLAS, a Bayesian optimisation method that also models the improvement over various amounts of budget and uses this model to select a configuration.

Most early-stopping approaches require a validation set to estimate the performance of the ongoing training process. There are two drawbacks to this approach. Firstly, the evaluation of the model on the validation set at different intervals is computationally expensive. Secondly, it requires making a choice on the size of the validation set, considering the trade-off between low generalisation error and the use of sufficient amounts of training data. To address these problems, Mahsereci et al. (2017)

proposed using an early stopping strategy for gradient-optimisation tasks without a validation set. For this purpose, the information on local statistics of the computed gradients is used. Without a need for a hold-out validation set, this method allows the optimiser to use all available training data.

Mohr and van Rijn (2021) introduced *learning curve-based cross-validation (LCCV)*, an extension to cross-validation that takes into account the evaluation of the learning curve of a given hyperparameter configuration. LCCV considers all configurations in order and works with the concept of the best configuration encountered so far. The main assumption of their work is that learning curves are convex and provide empirical evidence that this holds for observation-based learning curves of many algorithms on most data sets. Using this convexity assumption, they make an optimistic estimation of what the maximum performance of a given configuration at a certain budget can be, similar to the formulations of Sabharwal et al. (2016). Using this optimistic estimation, LCCV determines whether a given configuration will still be able to improve over the currently best configuration. If this is not the case, then the configuration can be discarded prematurely. Thus, similar to racing, LCCV takes a rather conservative approach and only discards a configuration when it is rather certain that it will not improve over the currently best configuration.

3.2.4 HPO systems and libraries

In this section, we review various well-known systems and libraries that can be used for HPO. There are no HPO-systems specifically tailored for machine learning on time-series data. The techniques can be extended for time-series analysis. We note that there is some overlap with the works that are described in the previous sections; here, our focus is on the implementation of the underlying methods and practical considerations regarding their use. Our descriptions are based on those given in scientific publications. We note that in some cases, development efforts may have continued, leading to improvements in functionality and usability.

SMAC (Hutter et al., 2014; Lindauer et al., 2022) is a general-purpose algorithm configurator and HPO system based on Bayesian optimisation. One of the distinguishing features of SMAC is its use of a random forest as the underlying surrogate model, rendering the Bayesian optimisation procedure broadly applicable to various types of search spaces. SMAC is used at the core of various widely used AutoML systems, including AutoWEKA (Thornton et al., 2013) and Auto-sklearn (Feurer et al., 2015).

HyperOpt (Bergstra et al., 2013) implements various optimisation algorithms,

including random search, TPE (Bergstra et al., 2011) and adaptive TPE. It can be parallelised using Apache Spark and MongoDB.

Spearmint (Snoek et al., 2012) is a Bayesian optimisation system. It uses Gaussian processes as a surrogate model and expected improvement as an acquisition function. Compared to vanilla Bayesian optimisation, Spearmint allows for effective parallelisation across multiple cores. Results of an empirical study comparing TPE, SMAC and Spearmint suggest that it is preferable to use SMAC and TPE when dealing with large and conditional search spaces, whereas Spearmint is recommended for low-dimensional and continuous problems (Eggensperger et al., 2013).

Optuna (Akiba et al., 2019) is an HPO system built upon TPE. Earlier optimisation frameworks, such as SMAC and HyperOpt, only allow a static definition of the hyperparameter space and cannot be used when no full description of the hyperparameter space is given by the user. Optuna, however, provides a define-by-run API that allows users to dynamically define and modify the search space. It also includes multi-fidelity strategies to speed up the optimisation process. Optuna covers several multi-fidelity strategies for performance evaluation, e.g., the asynchronous successive halving algorithm (Li et al., 2020a) and hyperband (Li et al., 2017).

Bayesopt (Martinez-Cantin, 2014) is a flexible framework that supports the optimisation of continuous, discrete and categorical hyperparameters. It allows users to select from many components relevant to Bayesian optimisation, such as the initialisation procedure, the acquisition function, the optimisation of the acquisition function, the surrogate model and the evaluation metric. Furthermore, Bayesopt implements an improvement to calculate the covariance matrix of Gaussian processes that, at each iteration, determines how many new elements will appear in the covariance matrix and how many will remain the same. This reduces the computational complexity of computing this matrix from $O(n^3)$ to $O(n^2)$.

RoBO (Klein et al., 2017b) is a package that implements various HPO algorithms based on Bayesian optimisation, including several methods focusing on multi-fidelity and auxiliary tasks, such as multi-task Bayesian optimisation (Swersky et al., 2013), Bohamian (Springenberg et al., 2016), and Fabolos (Klein et al., 2017a).

BOHB (Falkner et al., 2018) implements, in addition to the equally named BOHB algorithm, various relevant baseline methods, such as successive halving and hyperband. The BOHB package supports parallel computing and aims to address various practical problems that arise when running HPO algorithms in parallel on multiple CPUs.

MOE (Yelp, 2014) (metric optimisation engine) is an HPO package based on

Bayesian optimisation implemented in Python. It uses expected improvement as an acquisition function and Gaussian processes as a surrogate model.

Scikit-Optimize (Head et al., 2017) implements a sequential model-based approach to optimisation. It supports several methods, including sequential optimisation with decision trees/gradient boosted trees and Bayesian optimisation with Gaussian processes. For acquisition functions, it supports expected improvement, lower confidence bound, and probability of improvement.

Optunity (Claesen et al., 2014) covers several of the previously mentioned optimisers for HPO, including grid search, random search, particle swarm optimisation and CMA-ES.

Syne Tune (Salinas et al., 2022) is a package for distributed HPO. It includes a range of optimisers (including random search, Bayesian optimisation and evolutionary search) and multi-fidelity approaches (including BOHB and hyperband). In addition, it also supports more advanced methods for hyperparameter transfer learning, constrained hyperparameter optimisation and multi-objective optimisation.

3.3 Neural architecture search

In this section, we turn our attention to AutoML systems for automatically designing deep neural network architectures. Conventionally, neural networks are represented in the form of computational graphs (Goodfellow et al., 2016) of nodes that perform operations (e.g., addition, convolution, pooling, activation) on the input they receive from their parent nodes. The architecture of a neural network represents the parents of each node (i.e., structure or node connections), as well as the operations performed by the nodes.

Training a neural network requires setting two sets of hyperparameters. The first group are *training hyperparameters* that mainly affect the training process. These are hyperparameters such as the learning rate, optimiser type or batch size. The second group, however, are *architectural hyperparameters* that define the network architecture, such as the number, sizes and operations of layers. Neural Architecture Search (NAS) research mainly considers optimising the latter category of hyperparameters. Therefore, in the rest of this section, the term hyperparameter mainly denotes architectural hyperparameter.

Network architecture engineering is still a time-consuming and expensive task that needs to be performed manually by experts. NAS methods aim at finding good architectures by optimising architectural hyperparameters. Conceptually, NAS is a sub-field

of AutoML that builds, to a great extent, on the hyperparameter optimisation (HPO) techniques discussed in Section 3.2. We can frame NAS as an optimisation problem with the goal of finding an architecture that achieves the best possible performance in the target task within a predefined search space. We note that each layer within the network architecture defines new hyperparameters to be set for its operations, which leads to a tree-structured search space. This makes NAS more complex than HPO for classic machine learning algorithms without conditional hyperparameters.

Neuro-evolution of augmenting topologies (NEAT) (Stanley and Miikkulainen, 2002b) is one of the earlier approaches proposed in the early 2000s, aiming at automating the process of designing neural network architectures by jointly optimising network topology and parameters. NEAT is based on the idea of evolving the neural network architecture (structure and connection weights) using a genetic algorithm. However, being designed to work on the level of single neurons, this approach does not scale to deep network architectures with millions of neurons and different layer types. Zoph and Le (2017) were among the first who considered the idea of 'neural architecture search' with the goal of automating the design of modern deep neural networks. This initial attempt at NAS relied on the availability of very substantial computational resources (800 GPUs for four weeks). Many different approaches have been proposed to make NAS more computationally efficient while still attaining high performance.

NAS methods can be described in terms of the three components introduced in the context of HPO (Elsken et al., 2019b). In NAS, *search spaces* can comprise architectural and training hyperparameters. The efficient design of a search space using prior expert knowledge on the type of networks that perform best for a given class of problems can largely improve the efficiency of the search. The *search strategy* determines how to find good hyperparameter configurations (and hence, a good architecture) within a given, possibly vast, search space. *Performance estimation approaches* are used to decide which configurations will achieve high performance on a given data set without the need to perform potentially very time-consuming, full training and validation.

In the following, we review prominent and important NAS methods focusing on the underlying search space design in Section 3.3.1, and search strategy, in Section 3.3.2 and Section 3.3.3 covers various performance evaluation approaches proposed for NAS. Finally, we will discuss available NAS libraries and benchmarks in Sections 3.3.4 and 3.3.5, respectively.

Table 3.1: An overview of NAS methods categorised based on the search strategy and the search space.

Search Space	Search Strategy	Name
Micro-level	Reinforcement Learning	path-level transformation (Cai et al., 2018b), NASNet (Zoph et al., 2018), ENAS (Pham et al., 2018), Block-QNN (Zhong et al., 2018), RENAS (Chen et al., 2019c), (Chen et al., 2019d), FPNAS (Cui et al., 2019)
	Bayesian Optimisation	PNAS (Liu et al., 2018a), BOGCN-NAS(Shi et al., 2020), NAS-BOWL (Ru et al., 2020a), Auto-pytorch (Zimmer et al., 2021) CSNAS (Nguyen and Chang, 2022), BANANAS (White et al., 2021a),
	Evolutionary Algorithm	CoDeepNEAT (Miikkulainen et al., 2019), AmoebaNet-A (Real et al., 2019), AmoebaNet-B (Real et al., 2019), Lemonade (Elsken et al., 2019a), RENAS (Chen et al., 2019c), MONCAE (Dimanov et al., 2021), OSNAS (Zhang et al., 2022a), DFG-NAS(Zhang et al., 2022b)
	Monte Carlo Tree Search	AlphaX (Wang et al., 2020)

	Gradient Descent	MaskConnect (Ahmed and Torresani, 2018), GHN (Zhang et al., 2019) NAO (Luo et al., 2018) DARTS (Liu et al., 2019b), Proxyless (Cai et al., 2019), sharpDARTS (Hundt et al., 2019), PDARTS (Chen et al., 2019b), DARTS+ (Liang et al., 2019), PCDARTS(Xu et al., 2020), SNAS (Xie et al., 2019), FBNET (Wu et al., 2019), FAIR DARTS (Chu et al., 2020), DROP NAS (Hong et al., 2020), GDAS(Zhang et al., 2020),DOTS (Gu et al., 2021), IDARTS (Xue et al., 2021), EC-DARTS (Zhou et al., 2021), TNASP(Lu et al., 2021), MR-DARTS (Gao et al., 2022), B-DARTS (Ye et al., 2022)
	Random Search	RS NAS (Li and Talwalkar, 2019), RandomNAS(Zhang et al., 2020)
Macro-level	Reinforcement Learning	AutoNET (Mendoza et al., 2016), RL NAS (Zoph and Le, 2017), MetaQNN (Baker et al., 2017), DeepArchitect (Negrinho and Gordon, 2017), ENAS (Pham et al., 2018), Net Transformation (Cai et al., 2018a), PROXYLESSNAS(Cai et al., 2019)
	Bayesian Optimisation	DeepArchitect (Negrinho and Gordon, 2017), Auto-Keras (Jin et al., 2019), NASBOT (Kandasamy et al., 2018), CSNAS (Nguyen and Chang, 2022) (Nguyen et al., 2021)
	Evolutionary Algorithm	GeNet (Xie and Yuille, 2017), (Real et al., 2017), CGP-CNN (Suganuma et al., 2018), NASH (Elsken et al., 2018), Neuro-Cell-Based Evolution (Wistuba, 2018), Lemonade (Elsken et al., 2019a) (Irwin-Harris et al., 2019)
	Monte Carlo Tree Search	Monte Carlo planning (Wistuba, 2017)

	Gradient descent	Smash (Brock et al., 2018), TAS(Dong and Yang, 2019)
Hierarchical	Reinforcement Learning	MnasNet (Tan et al., 2019)
	Evolutionary algorithm	Hierarchical (Liu et al., 2018b)
	Gradient Descent	(Shin et al., 2018), Auto-DeepLab (Liu et al., 2019a)
	Bayesian Optimisation	NAGO (Ru et al., 2020b)

Table 3.1 gives an overview of prominent NAS methods based on their underlying search space and search strategy. As seen in the table, we distinguish micro-level, macro-level, and hierarchical search space design approaches. Widely used search strategies include methods based on reinforcement learning, Bayesian optimisation, gradient-based and evolutionary algorithms. We note that there are methods such as Lemonade (Elsken et al., 2019a) and RENAS (Chen et al., 2019c) that appear under more than one category in our table as their search strategy or search space covers more that one approach.

In what follows, we will describe the NAS techniques listed in Table 3.1 in more detail.

3.3.1 Search space

The search space of a given NAS method represents the space of all possible neural network architectures. However, searching within a vast space of all possible hyperparameter settings is an extremely computationally expensive task. Most research in search space design has focused on imposing simplifying constraints to reduce the number of architectural hyperparameter configurations. Two aspects have been considered in designing the search space of neural network architectures: (1) the design of the global architecture of the network, and (2) the design of sub-architectures that can be repeated to create a full neural network in a modular fashion. This directly leads to the concepts of macro-level and micro-level searches. The macro-level search considers optimising the entire network by searching for the operations and connections between layers (see, e.g., Brock et al., 2018; Kandasamy et al., 2018; Pham et al., 2018; Zoph and Le, 2017). The micro-level search focuses on optimising cells or blocks (see, e.g.,

Liu et al., 2018a, 2019b) that will be stacked to construct the final network. There are also approaches that consider a hierarchical search space (see, e.g., Liu et al., 2019a, 2018b; Tan et al., 2019; Zhong et al., 2018) by making use of a combination of the two previously mentioned approaches that leads to a hierarchically structured search space. In the following sections, we elaborate on these approaches.

Macro-level search spaces

Macro-level search considers generating the entire structure of the network. Designing the architecture in a layer-wise manner greatly reduces the degrees of freedom within the global search space; at the same time, it typically still allows for an arbitrary sequence of layers, each with its own architectural hyperparameters. Macro-level search spaces typically include hyperparameters such as the number of layers, conditional hyperparameters, such as the type of each layer (e.g., convolution, pooling) and the hyperparameters of these operations (e.g., filter size and stride of a convectional layer).

Previously, Baker et al. (2017); Kandasamy et al. (2018); Zoph and Le (2017) and Brock et al. (2018) have considered this type of search space by generating new networks (with a pre-defined maximum number of layers) in each iteration of the search process. Baker et al. (2017) generated the network architecture by iteratively searching within the space of architectural hyperparameters for each layer (e.g., number of filters, filter height, and filter width). Zoph and Le (2017) further proposed to widen the search space by allowing skip connections and branching layers. Figure 3.4 shows two different examples of chain-structured networks that make up this type of search space. The network shown on the left is a simple example where every layer receives its input from the previous layer and transfers its outputs to the next. The network shown on the right has a more complex structure, including multiple branches and skip connections. In a layer-wise architecture, the space of possible networks increases exponentially with the possible number of layers. Xie and Yuille (2017) imposed further limitations on the design of layer-wise architectures by defining a search space of networks with a limited number of stages, where each stage is composed of a number of pre-defined layers.

Micro-level search spaces

A macro-level search space provides flexibility in terms of defining a network by considering a large set of architectural hyperparameters. However, by having a large search space, the task of finding a good architecture will become computationally ex-

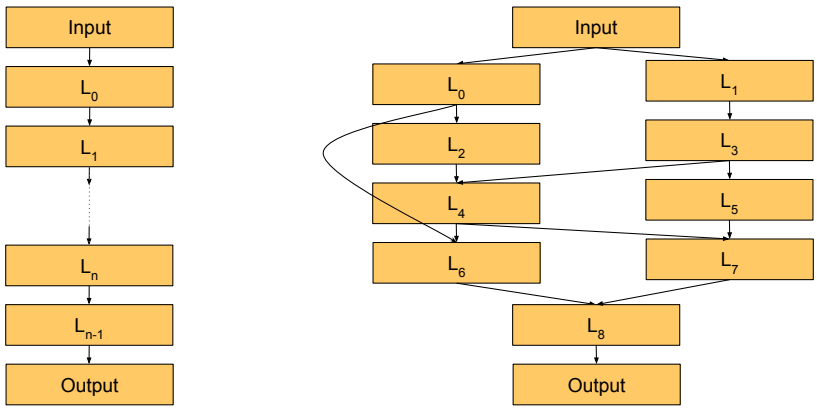

Figure 3.4: Examples of chain-structure neural networks. Each L_i in the graph indicates a layer with a specific operation (Elsken et al., 2019b).

pensive in terms of training time and other resources. Inspired by the observation that well-known convolutional neural network (CNN) architectures such as ResNet and Inception include repeated motifs (Zoph et al., 2018), the second group of algorithms has considered a micro-level approach defined based on cell or block structures.

Cell-based search: Cell structures are, in essence, mini-architectures composed of a number of layers and operations. These cells will be iteratively stacked to create a larger architecture. Searching for the best architecture will then be reduced to searching for the best cell structure (Zoph et al., 2018); the cells creating a larger network will all have the same architecture but different weights. In principle, by imposing additional structure on the search space, searching within a cell-based space is much simpler than searching within the space of all possible network structures. This structure will impose a limit on the maximum achievable performance by cell-based approaches. However, as shown by Zoph et al. (2018), the micro-level search can still achieve higher accuracy than macro-level search in a much shorter amount of time by using better initial models for the cell search. The cell-based architecture can also potentially generalise better to other problems and thus allow better transfer of knowledge across data sets. The full networks designed using NAS in macro-level search spaces are task- and data-specific, and they are typically difficult to transfer

to other data sets when the input data sizes are different. In a cell-based structure, better transferability can be achieved by adding more downsampling operations before cells (Zhong et al., 2018) or by adding more copies of the cell (Zoph et al., 2018).

In micro-search, it is common to formalise the NAS search space of a cell as a directed acyclic graph (DAG) where nodes represent local operations, and directed edges represent the flow of information (see, e.g., Liu et al., 2018a; Luo et al., 2018; Pham et al., 2018). Cells typically have two inputs and a single output node. In convolutional cells, input nodes of the cell are acquired from the previous two layers of the network. In recurrent cells, the input nodes are defined based on the current state and the previous one (Liu et al., 2019b). Figure 3.5 represents an example of a recurrent cell with four computational nodes.

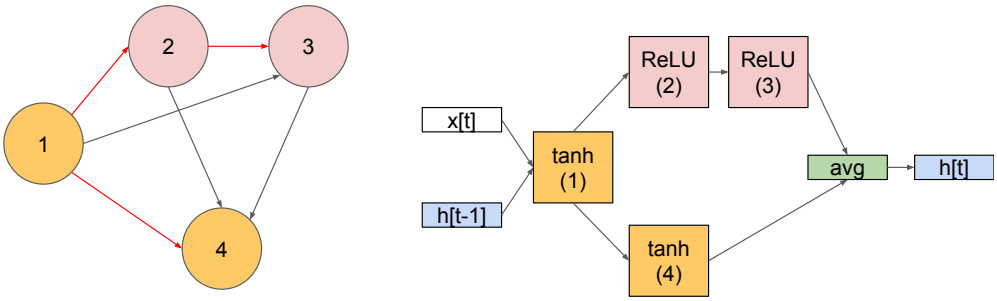

Figure 3.5: An example of a recurrent cell in a micro-level search space. Left: The search space is represented in the form of a DAG. The order of operations is represented by the red arrows. Right: the recurrent cell created by taking a subgraph of the DAG presented in the right by taking the red edges (Pham et al., 2018). In Section 3.3.3 we explain how these subgraphs are used for improving the performance evaluation.

NASNET (Zoph et al., 2018), ENAS (Pham et al., 2018), DARTS (Liu et al., 2019b), SNAS (Xie et al., 2019), and PNAS (Liu et al., 2018a) are examples of NAS approaches based on cell-level search spaces. NASNET (Zoph et al., 2018) is one of the first approaches in the design of cell-based NAS. NASNET focuses on NAS in image classification by making use of a search space based on modular convolutional cells; *normal cells* (with output and input in the same dimension), and *reduction cells* (with the output dimension being half of the input dimension, in both height and width). Based on the combination of these two types of cells, architectures can be built for processing images of any size. The structures of these cells can be searched to identify which operations are applicable to hidden states within cells and how to combine the outputs of pairs of hidden states into a new one. The 'normal cell, reduction cell' structure has been adopted by other researchers for CNNs (Liu et al., 2018a,

2019b; Real et al., 2019), along with additional simplifications to the cell structure. For instance, PNAS (Liu et al., 2018a) formalises cell-structures for CNN NAS that can implicitly emulate the normal and reduction cells mentioned earlier; by pruning a number of operations that were not selected from the search space considered by Zoph et al. (2018), a much smaller search space is obtained that can be searched more efficiently.

Efficient neural architecture search (ENAS) (Pham et al., 2018) is another impactful search space that formalises both CNN and RNN cells along with an approach for weight sharing to speed up the performance evaluation (explained in Section 3.3.3). More generic computational cells have been proposed by Liu et al. (2019b).

Some of the micro-level search spaces have used cell-level search within spaces of predefined chain-structured architectures (see, e.g., Liu et al., 2018a, 2019b; Pham et al., 2018). This approach initially tries to find a good cell structure by considering the connection topology and operations tied to each connection. Next, a fixed number of such cells is stacked in a chain-structured network. In order to go beyond simple chain structures and benefit from multi-path structures (which are commonly used in the state-of-the-art CNNs) Cai et al. (2018b) introduced a path-level network transformation operation permitting modifications of the path topology of a given network that defines the connection paths between layers. Using the path-level operations, Cai et al. (2018b) constructed a generalised multi-branch tree-structured search space that can encode predefined multi-branch structures based on advanced human-designed architectures (e.g., ResNets (He et al., 2016), DenseNets (Huang et al., 2017), PyramidNet (Han et al., 2017))). This approach significantly improves the efficiency of the cell design compared to a simple chain structure.

Block-based search: The networks obtained by repeatedly reusing optimised cells are of limited diversity. To overcome this limitation, blocks (i.e., cells with diverse structures) can be optimised. FPNAS (Cui et al., 2019) optimises various different blocks to generate the full network. This is done using a bi-level optimisation problem, where each block is optimised separately, while keeping all other blocks fixed. Similarly, FBNET (Wu et al., 2019) and ProxylessNAS (Cai et al., 2019) support layer-wise search for blocks to increase the diversity of networks found. Chen et al. (2019d) proposed to use eight different types of blocks based on well-known network architectures such as ResBlocks and Inception.

Hierarchical search spaces

Hierarchical search spaces combine micro-level and macro-level approaches to improve layer diversity. One way to create a hierarchical search space is by means of recursion. For instance, Liu et al. (2018b) proposed an approach starting from lower-level motifs as a small set of primitive operations (convolution, depth-wise convolution, separable convolution, max-pooling) at the bottom level of the hierarchy, from which higher-level motifs are then built recursively, such that the highest-level motif corresponds to the full architecture. At each level of this hierarchy, motifs are represented in the form of DAGs. A similar approach has been proposed by Ru et al. (2020b) to create a three-level hierarchy: At the top level, there is a graph of cells. At the middle level, each cell is represented as a graph of nodes. At the bottom level, each node is represented as a graph of operations. By varying the graph generator hyperparameters at each level, a diverse range of architectures can be obtained.

MnasNet (Tan et al., 2019) makes use of a different hierarchical search space. In this approach, a full CNN architecture is factorised into different segments, each comprising a number of identical layers. For each segment, the operations and connections of a single layer, as well as the number of layers, are optimised. The optimised layer will be repeated to create a full segment. This approach was inspired by the idea that different parts of the network should be treated differently, as they play different roles in the overall accuracy and inference latency (e.g., in a CNN architecture, earlier blocks impact the inference latency more as they process larger input sizes). Liu et al. (2019a) took a different strategy in proposing a bi-level hierarchical search space that allows selecting the network-level structure by searching within a space of popular network designs.

3.3.2 Search strategy

After the search space has been decided, a search strategy is needed to find the best architecture within this space. In the following, we give an overview of search strategies used in NAS.

Reinforcement learning

NAS can be addressed using reinforcement learning. In this case, an agent's goal is to generate a network from the action space of the search space. As outlined in Section 3.2, the policy-gradient approach is a well-known reinforcement learning technique

that relies on optimising policies with respect to rewards. Using a policy-gradient-based method for NAS to design CNNs and RNNs was first proposed by Zoph and Le (2017). In this approach, the structure and connectivity of the elements of the neural network are encoded in the form of an architectural string (as illustrated in Figure 3.6). These architectural strings are composed of a set of tokens (for CNNs, these are

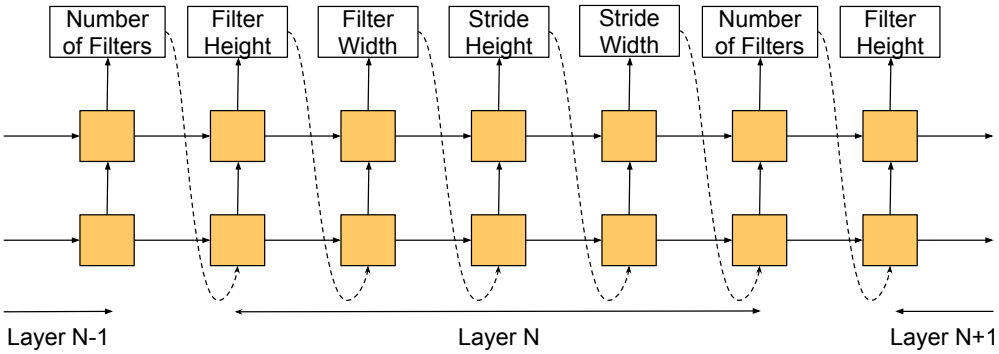

Figure 3.6: The controller is implemented in the form of an RNN. Here, this controller samples a feedforward neural network with only convolutional layers (empty squares) and predicts an architectural string composed of the hyperparameters of the convolutional layers (filter height, filter width, stride height, stride width and the number of filters) (Zoph and Le, 2017).

architectural hyperparameters such as filter height, filter width, stride height, stride width, and the number of filters for one layer). The controller is implemented as an RNN that will generate a child network by predicting the hyperparameters of such an architectural string one by one and stopping when a certain pre-defined number of layers has been generated. The hyperparameters that are predicted by the controller can be considered as a list of actions performed in the design of an architecture, and the reward corresponds to the accuracy achieved by the child network. The policy gradient is calculated to update the controller using this reward signal with the aim of maximising the expected accuracy of the generated architectures. The REINFORCE algorithm (Williams, 1992) mentioned earlier in Section 3.2.2 is used to optimise the parameters of the controller.

MetaQNN (Baker et al., 2017) and BlockQNN (Zhong et al., 2018) use Q-Learning, the other popular reinforcement learning algorithm. MetaQNN (Baker et al., 2017) defines states as a group of hyperparameters and uses a learning agent to generate layers one by one, until the entire network has been generated. This approach assumes that a well-performing layer in one network will also perform well in another one. The generated network is then trained, and its accuracy is used as a reward for the agent.

BlockQNN (Zhong et al., 2018) defines an action space that allows block-wise (as opposed to layer-wise) network generation with an agent that is trained to choose layers within a block.

ENAS (Pham et al., 2018) and NASNet (Zoph et al., 2018) are examples of reinforcement learning NAS approaches with an action space that allows generating architectural cells (explained earlier in Section 3.3.1).

Defining an action space based on function-preserving transformations allows to generate new architectures by transferring knowledge from previously trained networks and thus speeding up the evaluation process. Layer-level architecture transformations (Cai et al., 2018a) allow actions such as adding filters or layers, while path-level transformations (Cai et al., 2018b) allow actions that modify the path topology of the network.

Bayesian optimisation

As outlined in Section 3.2.2, Bayesian optimisation employs a surrogate model and an acquisition function in the optimisation process. For hyperparameter optimisation, Gaussian processes and random forests are the two commonly used surrogate models and expected improvement is a popular acquisition function.

Some of the design choices in the NAS search space cannot be encoded in the form of continuous variables (e.g., the number of layers or types of activation functions). While Gaussian processes in their basic form are only applicable to continuous variables, by using newly developed kernel functions, they can also handle categorical and conditional hyperparameters (Swersky et al., 2014a). For instance, to address the NAS problem using Gaussian processes new distance metrics have been proposed for measuring the similarity between pairs of network architectures (see, e.g., Jin et al., 2019; Kandasamy et al., 2018). Typically, these metrics are implemented in the form of edit distance between architecture encoding. For instance, Auto-Keras (Jin et al., 2019) proposes a Bayesian optimisation approach that uses Gaussian processes as a surrogate model combined with a distance metric based on network morphism (Wei et al., 2016), which allows to morphologically transform the architecture of a neural network while keeping its functionality. The Auto-Keras framework uses edit-distance neural network kernels to estimate how many operations need to be performed to change (morph) one neural network into another. GP-NAS (Li et al., 2020d) is another Gaussian process-based technique that uses a customised kernel function to measure the similarity between architecture encodings. Since relevant operations (e.g., the number of channels, kernel size) considered for creating these encodings are diverse in

type and incomparable, the kernel function used in GP-NAS works on groups of operations rather than on individual operations. NASBOT (Kandasamy et al., 2018) defines new kernel functions, referred to as optimal transport metrics for architectures of neural networks (OTMANN), to measure the similarity between two neural networks. OTMANN measures the distance between neural networks using three factors that determine the performance of a neural network: (1) the operations performed at each layer, (2) the types of these operations, and (3) how the layers are connected. The OTMANN distance can be computed by solving an optimal transport problem, a well-studied optimisation problem for which several effective solvers exist (Peyré and Cuturi, 2019). Ru et al. (2020a) proposed Gaussian process-based Bayesian optimisation using graph kernels (Weisfeiler-Lehman subtree graph kernel) that can be applied to ENAS (Pham et al., 2018) cells (explained earlier in Section 3.3.1).

Neural predictor surrogate models have also been used as the surrogate model of Bayesian optimisation for NAS; these are neural networks that are repeatedly trained on the architectures under evaluation to predict the accuracy achieved by previously unseen architectures. BANANAS (White et al., 2021a) uses neural predictors in combination with Bayesian optimisation. To make the Bayesian optimisation process more efficient, a variant of Thompson sampling (Thompson, 1933) is used as the acquisition function; Thompson sampling is a heuristic technique for balancing exploration and exploitation, which is a crucial element of Bayesian optimisation. White et al. (2021a) propose a new variant of Thompson sampling called *independent Thompson sampling*, which allows parallel Bayesian optimisation runs. Two options have been used for implementing neural predictors: MLPs and graph neural networks. Compared to MLPs, graph neural networks, as studied by Shi et al. (2020), are better suited to capture the topological structure of neural networks represented in the form of DAGs, and they can handle a variable number of nodes and scale to larger input architectures. They, however, require large amounts of training data (in the form of empirically evaluated architectures).

Another approach to using Bayesian optimisation for NAS is using tree-based models (see Section 3.2.2), which do not require defining a distance function and can easily handle conditional and categorical hyperparameters. Auto-Net 1.0 and Auto-Net 2.0 (Mendoza et al., 2016) are two NAS approaches based on this idea. Auto-Net 1.0 uses the random-forest-based Bayesian optimisation system SMAC (Hutter et al., 2011) as an optimiser, and Lasagne (LasagneContributors, 2022) as a deep-learning library. Auto-Net 1.0 is an extension of auto-sklearn (which will be introduced in Section 3.4) (Feurer et al., 2015). Like auto-sklearn, it supports feature extraction, data prepro-

cessing, and ensemble modelling. However, fully-connected feed-forward neural networks are the only machine learning models supported by Auto-Net 1.0. In Auto-Net 2.0, BOHB (Falkner et al., 2018), a combination of TPE and Hyperband (previously described in Section 3.2), is used as an optimiser, and PyTorch (LinuxFoundation, 2022) is used instead of Lasagne. Similar to Auto-Net 1.0, Auto-Net 2.0 covers all the preprocessing algorithms of auto-sklearn. Auto-pytorch tabular (Zimmer et al., 2021) further extends Auto-Net 2.0, targeting deep learning on tabular data sets by incorporating meta-learning and efficient micro-level search space.

Evolutionary algorithms

Evolutionary algorithms (see Section 3.2.2) have been used for evolving neural networks (a.k.a. neuro-evolution) for over two decades (see, e.g., Angeline et al., 1994; Stanley and Miikkulainen, 2002a,b; Yao, 1999). Neuro-evolution focuses on jointly optimising the weights and hyperparameters (architectural and training hyperparameters) using evolutionary algorithms. This approach only scales to small and medium-scale neural networks. Modern evolutionary NAS (EvNAS) research separates the two optimisation problems by employing evolutionary algorithms only for hyperparameter optimisation and leaving the optimisation of network weights to gradient-based optimisation, which is the recommended approach for training deep models (Bengio, 2012). In the following, we briefly discuss relevant aspects of both of these research lines. For a comprehensive survey, we refer interested readers to the survey by Liu et al. (2023).

Before adding a network to the population of candidate solutions, we need to decide on genome encoding. The evolutionary algorithm will further modify the genome of individuals through mutation and crossover operations. There are two types of encoding approaches, direct and indirect. In direct encodings, parameters to be optimised are directly presented in a genome. In indirect encodings, a transformation is used to interpret the neural network from a genome (Templier et al., 2021). NEAT (Stanley and Miikkulainen, 2002b) is an example of a neuro-evolutionary approach that uses the direct encoding approach to design multilayer perceptrons (MLPs). In NEAT, both node and connection genes are directly encoded into genomes as a mixture of binary, discrete or continuous variables. The connection genes specify for each node the in- and out-node connections, the weight of those connections, whether the connection is expressed, and the innovation number (which is meant to help keep topological innovations protected for a few generations before they disappear again from the population). The genome can be modified through three types of mutation

operations: (1) changing the weights of nodes, (2) adding connections between nodes, and (3) inserting a new node between a connection. By working on the level of single neurons, the direct encoding approach cannot scale to automatically design deep neural network architectures.

Indirect encoding schemes are later proposed to address this issue, using transformations or generation rules for creating architectures in a more compact manner. Miikkulainen et al. (2019) proposed an extension of NEAT for deep networks using an indirect encoding that allows each node in a genome to represent an entire layer rather than a single neuron. Similarly, HyperNEAT (Stanley et al., 2009) proposes an indirect encoding approach called connective compositional pattern-producing networks (CPPN), to create repeating motifs that represent spatial connectivity patterns as functions in Cartesian space.

One of the first EvNAS approaches proposed by Real et al. (2017) uses an indirect encoding scheme for representing simple single-layered networks with no convolutions, which are evolved into a far more complex network with high performance. Compared to NEAT, each mutation instead of changing one node can insert/remove layers of hundreds of nodes. Initiating the evolutionary process with trivial networks, as proposed by Real et al. (2017) makes the process of finding a good network slow. To speed up the search for better architectures, Liu et al. (2018b) proposed a diversification scheme based on design patterns defined by human experts to initialise the search. They further proposed a hierarchical encoding scheme and an action space for mutating hierarchical genotypes, where lower-level motifs are used as building blocks for constructing higher-level motifs.

After defining the genome encoding, effective evolutionary operators have to be chosen. As described in Section 3.2, evolutionary algorithms iteratively select a number of parent individuals based on a fitness function and generate offspring by using crossover and mutation steps. Xie and Yuille (2017) proposed using the classic roulette wheel selection approach, where the fitness of an individual is determined by training the corresponding neural network on a reference data set, and its selection probability equals its fitness divided by the sum of all fitness values in the population. In this manner, better individuals have a higher chance to participate in reproduction. Tournament selection (Liu et al., 2018b; Real et al., 2017, 2019) is another parent selection operator, in which p randomly selected individuals from the entire population (with or without replacement) participate in a tournament. The best individual in this group is selected as a parent (Goldberg and Deb, 1990) and this process is repeated for a number of rounds until the mating pool is filled. Real et al. (2017) used pairwise

tournament selection ($p = 2$, which is a common choice). In pairwise selection, the performance of only two randomly selected individuals are evaluated in each iteration, which makes the selection operator substantially more efficient than when a larger p is used. Real et al. (2019) introduced an ageing process by associating an age parameter with each individual in order to track how long an individual has been within a population. This allows biasing the tournament selection for the next generation towards younger genotypes. The oldest individuals are removed at each cycle to keep the population size fixed. Discarding the oldest individuals rather than the worst ones allows for exploring more of the search space.

Mutation and crossover operators can be used to generate the individuals in the next generation of the population (offspring networks). In general, the aim of the mutation is to search for the best individual around a single individual (Liu et al., 2023). For CNNs, mutation can involve adding/removing convolution operations, adding/removing skip connections or changing filter size/learning rate/weights (Real et al., 2017). For long short-term memory (LSTM) networks, mutation can involve adding/removing a connection of two LSTM layers, or adding/removing a skip connection of two LSTM nodes (Miikkulainen et al., 2019). Suganuma et al. (2018) used point mutations that randomly change both the type and connections of a layer. Lorenzo and Nalepa (2018) used a Gaussian process-based mutation operator that progressively refines the individuals. Elsken et al. (2019a) used network morphism and approximate network morphism as mutation operators. Network morphism extends the network (e.g., add skip connections, add more filters in a convolution), while approximate network morphism reduces its size (e.g., remove skip connections, remove some filters of a convolution). Network morphism only allows operators in the space of neural network architectures that preserve the function of the network. This removes the need to train offspring networks from scratch.

Next to mutation, the crossover operation can also be used in generating individuals for the next generation. The mutation operation modifies a single parent, resulting in offspring that are somewhat similar to their parent. Crossover, in contrast, combines features from two different individuals and can thus create offspring that differ substantially from their parents. One-point crossover is an example of an operator that has been used in EvNAS for combining two parents of the same length into an offspring (Ahmed et al., 2019). This requires that the length of the two individuals (e.g., the depth of a network) has been defined beforehand. The variable-length encoding strategy proposed by Sun et al. (2020) supports more effective architecture design processes by performing crossover on individuals with different lengths. A number

of EvNAS approaches do not make use of crossover operators, opting for a simpler approach (see, e.g., Liu et al., 2018b; Real et al., 2017). Similar to earlier evolutionary approaches (see, e.g., Suganuma et al., 2018; Xie and Yuille, 2017) that predefine the depth of the network, the approach by Real et al. (2017) removed the crossover operation and thus limits the search space to networks with a predefined depth. Other NAS approaches based on evolutionary algorithms include crossover operators (Irwin-Harris et al., 2019; Lu et al., 2019; Xie and Yuille, 2017; Zhang et al., 2022a). Irwin-Harris et al. (2019) have proposed an encoding strategy based on DAGs that uses crossover and mutation to construct CNN architectures of variable depth and with arbitrary graph structure. Xie and Yuille (2017) made use of crossover and mutation operations on architectures represented in multiple stages (where in each stage, the convolutional operators have a similar number of filters or channels). This structure allows encoding architectures into a fixed-length binary string. The mutation operation is performed by flipping a bit to preserve the quality of good individuals by slightly modifying them. The crossover operator applies to each stage in order to preserve the local structure within stages.

Monte Carlo tree search (MCTS)

Optimisation of neural architectures can also be viewed as a sequential decision process that can be solved using MCTS. The upper confidence bounds applied to trees (UCT) algorithm is a well-known Monte Carlo tree planning algorithm for tree-structured state-action spaces with a built-in exploration-exploitation mechanism that has been commonly used to search the space of network architectures (Negrinho and Gordon, 2017; Wang et al., 2020; Wistuba, 2017). MCTS works by defining a search tree, representing the search space that is traversed by visiting nodes based on a tree policy and a rollout policy. In the context of NAS, the internal nodes of this tree can represent hyperparameter values. A model is defined when reaching a leaf node of the tree. Originally, in MCTS, when visiting a node in the tree, all children of that node need to be expanded before any of their children is expanded. This is not ideal for nodes that represent numerical hyperparameters with a large range of values that lead to a similar performance. To address this issue, Negrinho and Gordon (2017) proposed combining UCT with a bisection approach that restructures the branches of the tree for such hyperparameters. At each node, instead of committing to a specific value of a hyperparameter, a sequential committing process is followed. It is first decided if the chosen hyperparameter value is in the first or second half of the set of hyperparameters. The bisection structure implicitly allows sharing information among similar paths over

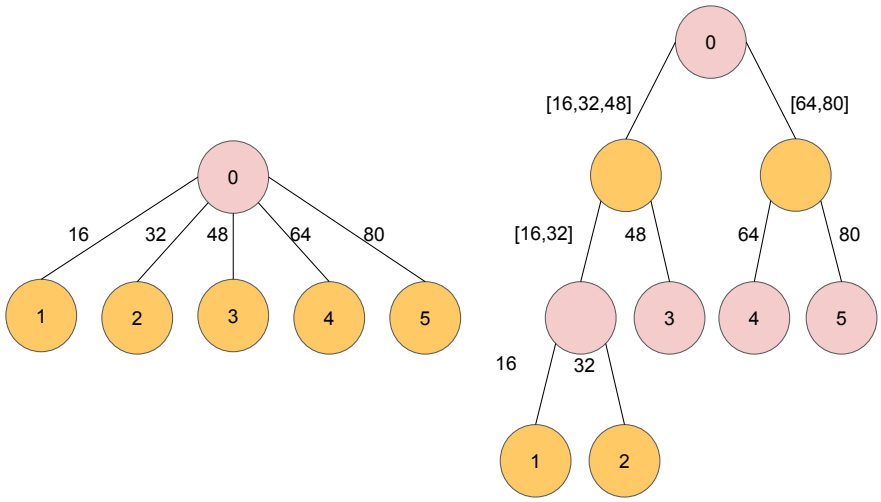

Figure 3.7: (a) A tree that encodes one hyperparameter and its possible values (16, 32, 48, 64, 80) (b) The same tree restructured with bisection. This allows more information to be shared between possible paths (e.g., sampling path to node 1 provides partial information about nodes 2 and 3) (Negrinho and Gordon, 2017).

the search tree, enabling search in a large search space (see Figure 3.7).

Techniques based on information sharing in different ways have been used to improve the performance of MCTS. Wistuba (2017) defines the NAS problem as a Markov decision process where each state describes the current network architecture, and each action adds a layer to the network. To address this problem with MCTS, Wistuba (2017) proposed two other variants of UCT that allow sharing information between branches of the search tree with the focus on reducing the time required for finding a good architecture through information sharing: the first of these shares information for the same action in similar states, while the second shares information between similar actions. This is achieved based on predictions of the final reward from previously selected actions. To improve the performance of MCTS, AlphaX (Wang et al., 2020) proposes to use a meta-deep neural network trained to estimate the accuracy achieved by candidate architectures.

Gradient-based methods

All the methods mentioned so far consider neural architecture search as a black-box optimisation problem over a discrete search space. This approach involves the evalua-

tion of the neural architectures with a large set of parameters that takes a significant amount of time and computational resources. Another category of approaches considers optimising the architecture using gradient descent. This way, much fewer data is needed for optimising the hyperparameters compared to the black-box optimisation approach (Liu et al., 2019b).

Since gradient-based optimisation is only applicable to continuous search spaces, continuous relaxation approaches are used to transform the NAS search space to a continuous one. Some of the earlier approaches to creating continuous search spaces, such as (Ahmed and Torresani, 2018; Saxena and Verbeek, 2016; Shin et al., 2018), mainly focused on optimising a limited set of architectural hyperparameters. For instance, MaskConnect (Ahmed and Torresani, 2018) focuses on optimising connectivity patterns by defining learnable masks in the form of binary vectors that are learnt jointly with network parameters, and Shin et al. (2018) focused on optimising a number of CNN hyperparameters, such as filter size, number of channels and group convolution, by defining a continuous function based on these hyperparameters.

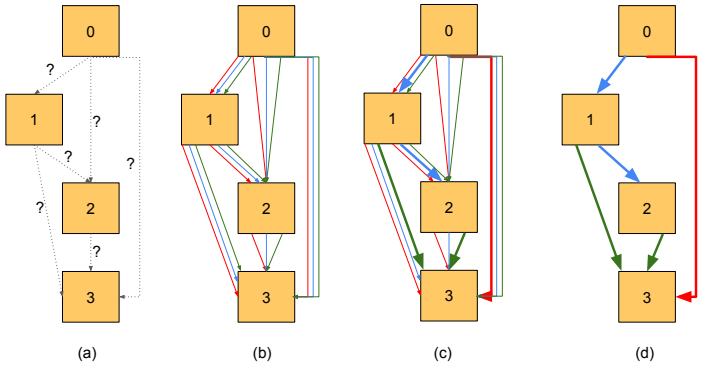

Figure 3.8: (a) Operations on the edges of the DAG are unknown. (b) The continuous relaxation approach of DARTS (Equation 3.6) creates a mixture of operations. (c) Solving a bilevel optimisation problem allows jointly optimising of the mixing weights and the network weights. (d) The final architecture is determined from the learned mixing weights (Liu et al., 2019b).

DARTS (Liu et al., 2019b) is a highly influential approach that considers a continuous relaxation of a cell-based search space applicable to both recurrent and convolutional networks. It aims to find optimal sub-architectures for the normal and reduction cells explained in Section 3.3.1. Consider a cell represented with a DAG composed

of a set of nodes and edges. Each specific node $x^{(i)}$ is a latent representation, and each edge (i, j) is associated with an operation $o^{(i,j)}$ that transforms the latent representation $x^{(i)}$ to $x^{(j)}$. An intermediate node is calculated from its predecessors $x^{(j)} = \sum_{i<j} \left(o^{(i,j)} \cdot x^{(i)}\right)$.

The DARTS approach, illustrated in Figure 3.8, relaxes the categorical choice of a particular operation o on node $x^{(i)}$ (e.g., convolution, max pooling) to a softmax over all possible operations $\mathcal{O}$. This will acquire a mixture of candidate operations for each edge denoted by $\bar{o}^{(i,j)}(x)$ (Liu et al., 2019b):

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \left(\frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o' \in \mathcal{O}} \exp\left(\alpha_{o'}^{(i,j)}\right)} \cdot o(x) \right) , \tag{3.6}$$

where $o(.)$ denotes an operation applied to node $x^{(i)}$, and the operation mixing weights of the pair of nodes $x^{(i)}$ and $x^{(j)}$ connected by the edge (i, j) is parameterised by a vector $\alpha^{(i,j)}$ with the size of $\mid \mathcal{O} \mid$. The goal of the architecture search process will be to identify an architecture encoding α in the form of a set of continuous variables $\alpha = \{\alpha_o^{(i,j)}\}$, each representing the weight of an operation. A discrete architecture will be produced by replacing mixed operations $\bar{o}^{(i,j)}(x)$ with a single operation that has the highest weight ($o^{(i,j)} \in \underset{o \in \mathcal{O}}{\operatorname{argmax}}\, \alpha_o^{(i,j)}$).

The architecture search process then aims to find α^* that minimises the validation loss $\mathcal{L}_{val}(w, \alpha^*)$, while the weights of the network w^* are determined by minimising the training loss $w^* = \underset{w}{\arg\min}\, \mathcal{L}_{train}(w, \alpha^*)$. This bi-level optimisation problem can be solved using gradient descent to jointly optimise w and α.

There are a number of issues with this continuous relaxation approach that have been addressed by follow-up research. Increasing the depth of the candidate networks considered in DARTS exponentially increases the size of the space to be searched and, consequently, the GPU memory overhead. In light of this, Liu et al. (2019b) originally restricted architecture search to a network of 8 cells and evaluated it on a network of 20 cells. Proxyless (Cai et al., 2019), addresses the memory inefficiency of DARTS (Liu et al., 2019b) by defining proxy tasks (i.e., training on smaller data sets). This is achieved through a path binarisation approach for reducing the memory footprint. During the training of an overparameterised network, many paths remain active in memory. By defining binary gates instead of continuous gates, only one path will be retained active in memory at run-time.

Another issue with DARTS is caused by the difference in the behaviour of shal-

low and deep networks (i.e., faster gradient descent by shallow networks versus higher performance with deeper networks), leading into a so-called depth gap between search and evaluation. The networks preferred in the search process will not necessarily be optimal for evaluation (Chen et al., 2019b). To address this issue, PDARTS progressively increases the depth of candidate architectures while approximating the search space to prevent memory issues. The search is divided into multiple stages. At the end of each stage, the network depth increases, while the number of candidate operations is reduced using their scores (calculated based on $\alpha^{(i,j)}$) in the search process so far as a criterion for selection. PCDARTS (Xu et al., 2020) uses another approximation scheme based on partial channel connection. During the continuous approximation, instead of sampling all channels connecting two connected nodes within a cell, PC-DARTS samples a random subset of channels and takes the computation on this subset as a surrogate for that on all channels.

Furthermore, the bi-level optimisation approach of DARTS suffers from so-called performance collapse (Chen et al., 2019b; Chu et al., 2020; Liang et al., 2019), i.e., performance decay of the discovered architecture as the number of search epochs increase, which is caused by the dramatic aggregation of skip-connections in the selected architecture. PDARTS (Chen et al., 2019b) and B-DARTS (Ye et al., 2022) address this issue by means of new regularisation strategies. PDARTS incorporates design search space regularisation to alleviate the dominance of skip-connections during the search, which leads to more hyperparameters that need to be defined by the user. The BetaDecay regularisation used by Ye et al. (2022) imposes constraints to prevent the value and variance of activated architecture parameters from getting overly large. A number of other studies have aimed to gain further insights into the cause of performance collapse. DARTS+ (Liang et al., 2019) shows that the number of skip-connections is linked to overfitting, which can be addressed by an early stopping strategy for the search process.

On the other hand, Chu et al. (2020) found evidence that the issue is caused by unfair competition among various operations in the bi-level optimisation process, which often leads to the dominance of skip-connections. They propose a cooperative mechanism implemented by additional activations for each parameter $\alpha^{(i,j)}$ to eliminate the competition by allowing operations to be switched on or off without being suppressed. Different forms of unfair competition between simultaneously trained operations in DARTS, leading to training stability issues, were identified and more generally studied by Gu et al. (2021); Hong et al. (2020). Hong et al. (2020) proposed to address this issue through the use of a new group drop-out operation, while (Gu et al., 2021)

proposed a new continuous relaxation approach that completely decouples operations and topology search in differentiable architecture search.

Neural Architecture Optimisation (NAO) Luo et al. (2018) is another gradient-based NAS approach. While DARTS makes use of a continuous representation of the architecture, including its weights, NAO is based on a continuous embedding of the architecture search space only. In this approach, an auto-encoder is used to learn a continuous representation of neural network architectures. A surrogate model is trained on this continuous representation to predict the performance of previously unseen candidate architectures. In each iteration of the search process, the surrogate model is used in gradient-based optimisation to select a new architecture to be evaluated. This new architecture is obtained by mapping the continuous representation back to a discrete one using a decoder learned as part of the autoencoder model. The surrogate model and autoencoder are updated at every iteration in order to minimise the encoder-decoder model loss and the performance prediction (surrogate) loss. While NAO uses gradient descent for hyperparameter optimisation, unlike DARTS, it does not consider a bi-level optimisation of weights and hyperparameters. However, NAO it can still use weight-sharing approaches (in Section 3.3.3) separately to speed up the evaluation of candidate networks.

Gradient-based approaches such as DARTS can be used in combination with gradient-based meta-learning approaches to further improve search efficiency. Recently, Elsken et al. (2020) have proposed MetaNAS, which integrates NAS with gradient-based meta-learning techniques. Their approach extends DARTS to optimise a meta-architecture with meta-weights during meta-learning; this facilitates the adaptation of the architecture to novel tasks.

Random search and grid search

As in the case of HPO, random search (Bergstra and Bengio, 2012; Li and Talwalkar, 2019) and grid search (Zagoruyko and Komodakis, 2016) have been used in NAS. Both approaches are very simple and easy to run in parallel, but they are usually not considered to be very efficient. Random search has been considered a competitive baseline for NAS (Li and Talwalkar, 2019; Yu et al., 2020). Comparing the performance of models on Penn Tree Band (PTB) (Marcus et al., 1994) language modelling and CIFAR-10 (Krizhevsky et al., 2010) image classification data sets, Yu et al. (2020) demonstrated that, on average, a number of state-of-the-art NAS algorithms (ENAS (Pham et al., 2018), DARTS (Liu et al., 2019b) and NAO (Luo et al., 2018)) have similar performance to random search when using the same search space for finding a

recurrent and convolutional cell.

Li and Talwalkar (2019) evaluated random search with early stopping and weight-sharing strategies (explained in Section 3.3.3) on the search space of DARTS and demonstrated that it achieves performance competitive with that of DARTS and ENAS. This is not due to the poor performance of the latter NAS algorithms but mainly the consequence of the constraints they have imposed on the search space. In a well-constrained search space, even random search can perform well. Another reason could be the weight-sharing strategy that negatively impacts architecture ranking during the search.

3.3.3 Performance evaluation in NAS

A major performance bottleneck of NAS systems arises from the need to train each candidate neural network. This is especially the case for deep neural networks, whose training takes substantial amounts of time. Therefore, performance evaluation is a much more important topic in the context of NAS than in HPO methods for classic machine learning algorithms. Early NAS systems, such as NASNet (Zoph et al., 2018) and AmobaNet (Real et al., 2019), trained every candidate architecture from scratch, racking up thousands of days of GPU time. The design of cell-based search spaces, as discussed in Section 3.3, can, to some extent, decrease the total cost of performance evaluation. The search process can also be sped up significantly using network morphisms and function-preserving transformations (Chen et al., 2016) that allow modifying the structure of a network without majorly changing its predictions. Furthermore, efficient performance estimation strategies have recently become a major focus for research on NAS methods. In the following subsections, we discuss approaches taken to speed up the performance evaluation in NAS.

Parameter sharing

Parameter (or weight) sharing or inheritance between network architectures is an approach that can reduce the computational demands of NAS methods. Parameter sharing can be performed by reusing the weights of previously optimised architectures or sharing computations between different but related architectures.

Defining the search space that allows sampling sub-architectures from super-networks facilitates parameter sharing (Cai et al., 2018a; Elsken et al., 2018; Pham et al., 2018). ENAS (Pham et al., 2018), SMASH (Brock et al., 2018) and convolutional fabrics (Saxena and Verbeek, 2016) use an over-parameterised super-network,

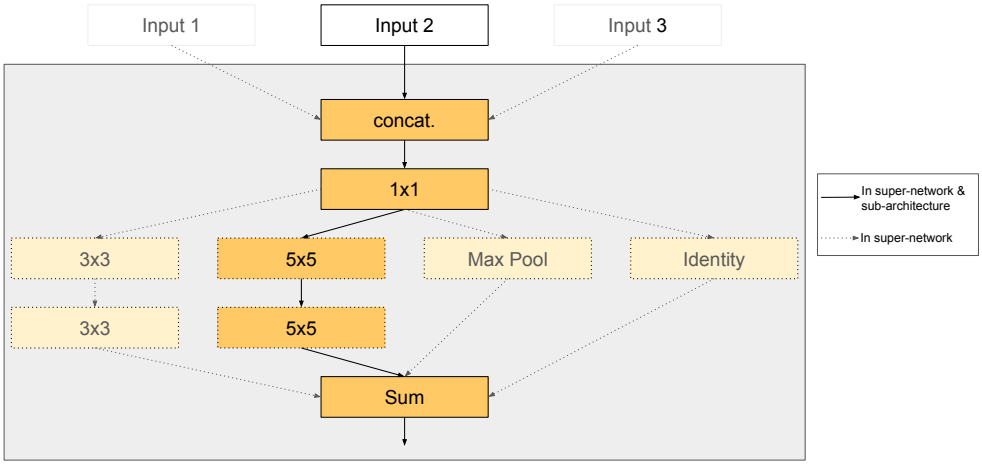

Figure 3.9: An example of a one-shot cell that receives three inputs concatenates them, applies a 1x1 convolution operation, followed by multiple other operations and finally sums the result together. A sub-architecture, denoted by solid edges, can be evaluated by disabling inputs and operations by zeroing them out (Bender et al., 2018).

defined in a discrete search space (also referred to as a one-shot model), that is a superposition of all possible sub-architectures, and enforce parameter sharing between sub-architectures. Once the weights of the super-network are trained, the performance of all sub-architectures on the validation set can be ranked and compared without training by enabling and disabling edges (see Figure 3.9). The most promising sub-architectures need to be further retrained. Generating and training a super-network for a given search space is itself a complicated task. Muñoz et al. (2022) proposed a further approach, dubbed BootstrapNAS, that automates the generation and training of super-networks from pre-trained models.

Creating differentiable search spaces (see, e.g., Liu et al., 2019a,b; Xu et al., 2020) using a continuous relaxation mechanism is another approach that implicitly allows parameter sharing. A differentiable search space allows parameters and hyperparameters to be jointly optimised using gradient descent without a need for a candidate architecture to be iteratively sampled and evaluated. The continuous representation proposed in DARTS (Liu et al., 2019b), for instance, benefits substantially from parameter sharing by defining a super-network that is differentiable in both network weights and architectural hyperparameters at the expense of high GPU memory con-

sumption.

Limits of parameter sharing strategies have been studied by Xie et al. (2022a); Yu et al. (2020). Yu et al. (2020) demonstrated that parameter sharing strategies degrade the ranking of candidate architectures. This is due the fact that estimating the performance of sub-architectures by copying their weights from a super-network may not reflect the true performance of candidate architectures. Xie et al. (2022a) identified an instability issue arising in a scenario where individual runs of the search process will lead to networks of different quality. This instability is due to the optimisation gap between the super-network and its sub-architectures. It is not guaranteed that an optimised super-network creates an optimised sub-architecture. Different reasons have been identified for this issue, including the unfair competition between parameters and hyperparameters, errors in calculating gradients in the continuous search and falling into local optima while optimising the super-network.

Performance predictors

Another line of work targeting more efficient performance evaluation in NAS aims at building models to predict the performance of the neural networks in terms of the final accuracy or ranking of candidate architectures. The performance predictor is once initialised at the start of the NAS process and subsequently queried with many architectures within the inner NAS optimisation loop. Benchmarks (extensively covered in Section 3.3.5) such as NAS-Bench-101 (Ying et al., 2019), NAS-Bench-201 (Dong and Yang, 2020) and NATS-Bench (Dong et al., 2021) that include a large number of evaluated architectures provide great opportunities for creating performance prediction models. The methods mentioned earlier in Section 3.2.3 for accuracy prediction through learning curve extrapolation do not have an initialisation phase. However, when used in NAS they still require the expensive process of training various candidate architectures.

One research direction to speed up the performance evaluation in NAS is to reduce the number of expensive training steps in the inner optimisation loop to zero. Deng et al. (2017); Istrate et al. (2019) proposed model-based approaches for the estimation of the accuracy achieved by a given network by analysing its structure. TAPAS (Istrate et al., 2019) initialises a predictive model for the performance of architectures based on data set characteristics and a lifelong database of experiments on previously trained neural networks. An accuracy predictor uses this model to predict the peak accuracy of each queried architecture after training.

Recently, several zero-cost methods (see, e.g., Abdelfattah et al., 2021; Mellor et al.,

2021) have been proposed that further reduce both the initialisation and query cost of predictors by exploiting more fundamental architectural properties of a network from its initial state by just a single forward/backward propagation pass on a single mini-batch of data. For instance, Mellor et al. (2021) proposed a scoring method that predicts the performance of an untrained architecture by estimating the overlap of activations of rectifier linear units between data points in untrained networks and previously trained networks. This approach is motivated by the idea that the more similar the activations between two inputs, the more difficult it is for the network to learn to separate them. Abdelfattah et al. (2021) proposed a number of zero-cost proxies using network pruning as an initialisation strategy. These proxies can be used to rank network architectures in NAS.

White et al. (2021b) performed an empirical comparison of performance predictors in the context of different NAS frameworks. Their results suggest that even a simple combination of strategies mentioned before (zero-cost, model-based and learning curve methods) can substantially improve performance by exploiting the complementary power of different strategies.

3.3.4 NAS systems and libraries

In this section, we review important systems and libraries for NAS that can be useful to practitioners and researchers. Some of these (e.g., AutoKeras (Jin et al., 2019), Auto-PyTorch (Zimmer et al., 2021)) are essentially an implementation of a specific NAS approach, which can be used by practitioners or further extended by researchers. These systems, in principle, follow a similar purpose to the AutoML systems covered in Section 3.4. However, since they address the NAS problem as opposed to the CASH problem, we cover them here. Another group of libraries (e.g., NNI (Microsoft, 2021), NASlib (White et al., 2021b)) focuses on providing a homogeneous codebase that can also support further NAS research. These libraries empower researchers to customise and build upon available methods, as well as to evaluate new NAS methods against important baselines. Prominent NAS systems and libraries include the following:

AutoKeras (Jin et al., 2019) is a NAS system built upon Keras supporting regression and classification tasks on image, text and tabular data sets. The proposed search strategy of AutoKeras is a combination of Bayesian optimisation and network morphism (Jin et al., 2019) (explained in Section 3.3.2). However, in the default setting, this system uses an optimisation method based on random search. AutoKeras supports optimising a macro search space and allows building networks using several

block types, such as Xception and ResNet. AutoKeras has a component called "TimeSeriesForecaster" which is used for time-series forecasting tasks. However, the users have to determine a hyperparameter "lookback", which means how much historical data are used during the forecasting.

Auto-PyTorch (Zimmer et al., 2021) is a NAS system built on top of the PyTorch framework. As a search strategy, auto-PyTorch uses a combination of Bayesian optimisation and hyperband to speed up the training process (explained in Section 3.2.2). It also uses meta-learning and ensembling to produce stronger models. Auto-PyTorch focuses on optimising the full machine learning pipeline, including preprocessing, NAS, network training and regularisation. It supports two macro-scale search spaces: a smaller one for optimising MLPs, and a bigger one for optimising deeper architectures that can include residual blocks. Originally, the main focus of Auto-PyTorch was to achieve state-of-the-art performance on tabular data sets by incorporating relevant pre-processing tasks into the search space. However, its usefulness has also been demonstrated in object detection tasks. Auto-PyTorch supports time series forecasting.

NNI (Microsoft, 2021) is a Python library for HPO and NAS which facilitates users in developing and using NAS techniques. It includes implementations of many HPO techniques (e.g., random and grid search, Bayesian optimisation, evolutionary algorithms, and Hyperband) as well as state-of-the-art one-shot NAS methods (e.g., DARTS, ENAS, FBNET, ProxlessNAS). The deep-learning framework supported by NNI is PyTorch, and APIs are provided to construct and explore different search spaces. NNI also supports Kubernetes, an important system for cloud-native deployments. Other than HPO and NAS, NNI covers model compression and feature engineering as well.

NASLib (White et al., 2021b) is another Python library based on PyTorch and mainly focused on supporting NAS researchers. It systematically follows a modular and flexible approach that facilitates the reuse of search spaces and evaluation pipelines. NASLib covers cell-based and hierarchical search spaces, various search strategies (e.g., random search, Bayesian optimisation, evolutionary, and gradient-based search), and performance evaluation approaches (e.g., one-shot, zero-cost, weight-sharing, learning curve-based). Furthermore, it includes an extensive collection of NAS benchmarks that can help researchers evaluate and further develop performance prediction approaches.

Katib (George et al., 2020) is a library for HPO and NAS, mainly aimed at providing a scalable, cloud-native and production-ready system. It supports Kuber-

netes and is agnostic to the underlying machine learning framework, supporting codes written by users in different frameworks (e.g., PyTorch, TensorFlow, ApacheMXNET, XGBoost). Katib includes implementations of various search strategies (e.g., random search, grid search, Bayesian Optimisation, TPPE, multivariate TPE, CMA-ES, hyperband and evolutionary algorithms); in terms of NAS approaches, only implementations of ENAS and DARTS are included.

Hypernets (DataCanvas, 2021) is a library supporting various deep-learning frameworks (Tensorflow, Keras, PyTorch). It supports macro search spaces and the ENAS micro-search space, as well as Monte-Carlo tree search (MCTS), evolutionary search and random search strategies.

3.3.5 NAS benchmarks

While NAS-generated architectures have shown promising performance across different domains, performing thorough and fair comparisons against state-of-the-art NAS approaches still remains an issue. To some extent, this issue can be alleviated by following the suggested best practices (see, e.g., Lindauer and Hutter, 2020) for making NAS codes and models available. However, even when the code is available, large computational resources are needed to reproduce the results of NAS experiments, and those resources are not broadly available to NAS researchers. A common solution to the challenges arising from this situation has been to directly use results reported in the literature as the basis for comparative evaluations. However, this approach can be misleading since those results can be strongly influenced by factors such as the type of GPUs or the parallelisation strategies employed.

To address this reproducibility issue, in the past few years, there has been a number of efforts to create benchmarks for facilitating NAS research. These benchmarks were initially created in the form of tabular data sets, whose entries provide relevant information about fully trained neural network models within a predefined search space. Having access to information such as running time and training/validation/test accuracy of all networks within a given search space, NAS researchers can faithfully simulate runs of a NAS procedure. This leads to tremendous speed-ups in the overall process of developing NAS algorithms. Naturally, the creation of these benchmarks tends to be computationally expensive; it involves the following steps: (1) definition of the search space, (2) sampling candidate architectures over this search space using a well-defined strategy and removing duplicates, (3) training and evaluation of the architectures from the resulting set of architectures using different objective functions

(or possibly even in the context of downstream tasks).

HPO-Bench (Klein and Hutter, 2019) and NAS-bench-101 (Ying et al., 2019) are among the first tabular benchmark data sets created in this manner. HPO-Bench covers 144 candidate architectures of a feedforward neural network and can be used for empirical evaluation of HPO methods. However, it is insufficient to evaluate NAS algorithms that focus on automatic design of advanced deep-learning architectures. NAS-Bench-101 is the first public data set suitable for benchmarking NAS research. It covers all 423 000 unique convolutional architectures of a cell-based search space defined based on a DAG of seven nodes and three operations. Each architecture was trained multiple times on CIFAR-10 leading to a large data set of over 5 million trained models. This benchmark can be used for comparing HPO methods as well as certain NAS algorithms that do not use parameter sharing or network morphisms.

NAS-Bench-201 (Dong and Yang, 2020) is tailored towards the evaluation of more NAS algorithms on more image data sets but within a smaller space based on a DAG of 4 nodes and 5 operations, resulting in 15 625 neural cell candidates in total. NAS-Bench-1Shot1 (Zela et al., 2020) reuses the NAS-Bench-101 data set with some modifications tailored to the evaluation of one-shot NAS methods. While benchmarks such as NAS-Bench-101, HPO-Bench, NAS-Bench-1Shot1 focus on solely architecture topology without considering the architecture size, NATS-Bench (Dong et al., 2021) covers both these factors by creating a benchmark of two sets of architectures with 15 625 neural cell candidates (4 nodes and 5 operations) to evaluate the architecture topology and 32 768 candidates for architecture sizes that can be used to evaluate all NAS methods. There have also been efforts to create more specialised benchmarks to evaluate more properties of a specific group of NAS systems. For instance, BenchENAS (Xie et al., 2022b) focuses on evaluating EvNAS algorithms along with providing a platform for running them in a fair manner, including a special parallel processing component for evaluating populations of candidate architectures. This benchmark currently covers 9 EvNAS algorithms that cover fixed-length encoding and variable-length encoding strategies, as well as single- versus multi-objective optimisation. NAS-Bench-NLP (Klyuchnikov et al., 2022) focuses on benchmarking NAS for natural language processing by using RNN cells as opposed to the convolutional cells used in the previous NAS benchmarks. It covers 14 000 trained architectures along with evaluation results using metrics from language modelling and relevant downstream tasks (e.g., sentence tasks, similarity and paraphrase tasks).

The above-mentioned tabular NAS benchmarks rely on the computationally expensive task of an exhaustive evaluation of all architectures within a given search

space. Therefore, to ensure feasibility, their search spaces are limited to very small architectural spaces compared to those usually considered in the NAS literature. This might compromise the generalisability of the models evaluated using existing tabular NAS benchmarks. To enlarge this search space, NAS-Bench-301 (Zela et al., 2022) provides a surrogate NAS benchmark that covers 10^8 architectures and is thus much larger than previous benchmarks. The underlying idea is to estimate the performance of candidate architectures using a surrogate model rather than actual evaluations. The authors of NAS-Bench-301 then created a new benchmark by sampling 60 000 architectures from a much more complex search space over which they had trained the surrogate model. While the degree to which the resulting benchmark is realistic depends on the accuracy of the surrogate model, empirical results by Zela et al. (2022) demonstrate that a surrogate model can yield a better model of the architecture performance when compared to a tabular benchmark. The reason for this is that tabular benchmarks tend to present few training runs for each architecture acquired using a mini-batch procedure, which itself can have a variance. Thus, a tabular benchmark gives only a simple estimate of the true performance of an architecture. A surrogate model trained on a smaller subset of networks has the potential to yield a much more accurate estimation by taking into account extra information, such as the similarity between architectures.

3.4 Broad-spectrum automated machine learning systems

In Section 3.3.4, we covered AutoML systems such as AutoKeras (Jin et al., 2019) and Auto-PyTorch (Zimmer et al., 2021) that address the NAS problem. In this section, we review a number of broadly known and widely used AutoML systems based on classic machine learning algorithms realised through addressing the CASH problem.

Although deep learning for certain data forms, such as natural images, has facilitated automated feature learning, most machine learning models still crucially rely on some degree of feature engineering and preprocessing (Chen et al., 2019a). Among the systems we study, only AutoGluon supports time-series forecasting. Therefore, many AutoML systems do not only focus on hyperparameter optimisation but on constructing full machine learning pipelines that include data preprocessors and feature extractors, as well as different machine learning models. To achieve this goal, a common approach (see, e.g., Feurer et al., 2015; Thornton et al., 2013) is combining all

design choices into a single large search space and then using HPO methods described in Section 3.2 (see, e.g., Bergstra et al., 2011, 2013; Hutter et al., 2011) to search for the best parameter setting. It should, however, be noted that not all HPO methods can scale to large hyperparameter spaces or correctly handle all relevant types of hyperparameters, notably the conditional hyperparameters, that are relevant in the context of pipeline generation. In this case, the search space usually includes two types of information: the choice of algorithms and their hyperparameters. The idea mentioned above was used by Thornton et al. (2013) to formally define the *combined algorithm selection and hyperparameter optimisation (CASH) problem* as a single optimisation problem, as per Definition 3.3.

Most AutoML systems address the CASH problem to automate supervised learning tasks by minimising the cross-validation loss based on classification or regression accuracy. By taking a different approach to validation, AutoML systems for other machine learning tasks can be realised. For instance, to address the scarcity of labelled data, Li et al. (2019) defined the automated semi-supervised learning (AutoSSL) problem. In AutoSSL, instead of using cross-validation to evaluate and optimise performance, the relative performance of semi-supervised learning algorithms compared to a baseline supervised learning algorithm is considered. Unsupervised learning tasks of descriptive nature (as opposed to predictive) that work on unlabelled instances are, however, not yet fully addressed by AutoML systems.

Existing AutoML systems can be differentiated with respect to the structure of the pipeline they construct. Some systems construct pipelines of a fixed and pre-specified structure, for example, one data preprocessor followed by a single machine learning model. Other systems do not rely on a fixed structure but can construct flexible pipelines, which can theoretically consist of an arbitrary number of preprocessors and models. Furthermore, many systems support the combination of multiple pipelines into an ensemble. Ensemble models can generally achieve higher performance and tend to be more robust against overfitting than single models (Guyon et al., 2010; Lacoste et al., 2014).

Meta-learning approaches can be used to predict the performance of models and hyperparameter settings for a given task based on experience on other tasks. A number of AutoML systems employ meta-learning as a strategy to speed up the search for performance-optimised machine learning pipelines.

Table 3.2 provides an overview of prominent AutoML systems. Other than the underlying search strategy and pipeline structure, this table lists the machine learning library that provides the basis for constructing the search space of the AutoML

system, along with an indication of whether an implementation is publicly available and whether meta-learning is employed. In the following subsections, we provide more detailed descriptions of the AutoML systems in Table 3.2, categorised based on the underlying optimisation technique.

Table 3.2: Overview of existing AutoML systems.

Name	**Search Strategy**	**Pipeline Structure**	**Library**	**Implementn. Available**	**Meta-learning**
ADMM AutoML (Liu et al., 2020)	alternating direction method of multipliers + Bayesian optimisation	fixed	Scikit-learn	no	no
Alpine Meadow (Shang et al., 2019)	multi-armed bandit + Bayesian optimisation	fixed	Northstar	yes (proprietary)	yes
ATM (Swearingen et al., 2017)	multi-armed bandit + Bayesian optimisation	fixed	Scikit-learn	yes	yes
Auto-sklearn (Feurer et al., 2015)	Bayesian optimisation	fixed + ensemble	Scikit-learn	yes	yes
Auto-WEKA (Thornton et al., 2013)	Bayesian optimisation	fixed	WEKA	yes	no
AutoCompete (Thakur and Krohn-Grimberghe, 2015)	random search & grid search	fixed	Scikit-learn	no	no
AutoGluon-Tabular (Erickson et al., 2020)	fixed order + Bayesian optimisation	flexible	Scikit-learn, XGBoost, LightGBM, CatBoost, self-implemented neural networks	yes	no
AutoML-DSGE (Assunção et al., 2020)	genetic programming	flexible	Scikit-learn	yes	no

Autostacker (Chen et al., 2018a)	evolutionary algorithm	flexible	Scikit-learn, XGBoost	no	no
FEDOT (Nikitin et al., 2022)	genetic programming + Bayesian optimisation	flexible	Scikit-learn, XGBoost, LightGBM, CatBoost	yes	no
FLAML (Wang et al., 2021)	estimated-cost-for-improvement-based sampling + direct search	fixed	Scikit-learn, XGBoost, LightGBM, CatBoost	yes	no
GAMA (Gijsbers and Vanschoren, 2020)	genetic programming	flexible + ensemble	Scikit-learn	yes	no
H2O AutoML (LeDell and Poirier, 2020)	fixed order + random search	fixed + ensemble	H2O	yes	no
Hyperopt-sklearn (Komer et al., 2014)	Bayesian optimisation	fixed	Scikit-learn	yes	no
ML-Plan (Mohr et al., 2018)	hierarchical planning	fixed	Scikit-learn, WEKA, MEKA	yes	no
MOSAIC (Rakotoarison et al., 2019)	Monte-Carlo tree search + Bayesian optimisation	fixed	Scikit-learn	yes	no
Naïve AutoML (Mohr and Wever, 2022)	fixed order + random search	fixed	Scikit-learn, WEKA	yes	no
Oboe (Yang et al., 2019)	meta-learning	fixed + ensemble	Scikit-learn	yes	yes
Oracle AutoML (Yakovlev et al., 2020)	meta-learning + gradient descent	fixed	Scikit-learn, XGBoost, LightGBM	yes	yes
RECIPE (de Sá et al., 2017)	genetic programming	flexible	Scikit-learn	yes	no
TPOT (Olson et al., 2016a)	genetic programming	flexible	Scikit-learn	yes	no

3.4.1 Systems based on random search and grid search

In Sections 3.2 and 3.3, we introduced the random and grid search approaches for HPO and NAS, respectively. Here, we briefly review AutoML systems that make use of these approaches. Since these search strategies do not scale to large search spaces, most research in this direction has worked on a strategy to improve search efficiency.

AutoCompete (Thakur and Krohn-Grimberghe, 2015) makes use of random search and grid search for the selection of a model, a corresponding hyperparameter setting and optional data preprocessing steps. It supports classification and regression tasks based on a search space defined over models and data preprocessors implemented in Scikit-learn (Pedregosa et al., 2011). In order to speed up the search process using these inefficient search strategies, their approach simplifies the CASH problem by limiting the search space based on the encountered data set type. Only specific algorithms are selected per data set type and certain hyperparameters of those are tuned.

H2O AutoML (LeDell and Poirier, 2020) is an AutoML system for the optimisation of machine learning pipelines, including a post-processing step for creating an ensemble of models. This system is implemented based on the H2O machine learning library (H2O.ai, 2017); while H2O AutoML offers the same automated preprocessing steps available in H2O, it does not select or optimise their hyperparameters within the full pipeline. H2O AutoML first evaluates user-selected models with pre-specified hyperparameters and in a pre-defined order, adding them to a leaderboard that keeps the ranking of models based on performance. Next, it tunes the hyperparameters of the models on the leaderboard using random search, where for some pre-specified models, more time is allocated than for others. The pre-specified models are those which are most promising in the developers' opinion. In the final step, H2O AutoML constructs a stacking ensemble using a pre-specified meta-model that is trained on the outputs of the optimised base models with the goal of finding the best combination of models.

Naïve AutoML (Mohr and Wever, 2022) is, according to its authors, a very simple solution to AutoML, which can be considered as a baseline for more complicated black-box solvers and even sometimes outperform them. The general idea of this system is to imitate the sequence and analytical process of optimising a pipeline by humans in different stages rather than creating a large search space of all design choices that can be optimised at the same time. It assumes machine learning pipelines consisting of a sequence of a fixed number of data transformers (that transform one

representation of data to another) followed by a predictor (that predicts the label of the input data). Pipelines are optimised in a series of optimisation stages, where each stage is responsible for constructing a certain part of the pipeline, e.g., one stage to select a predictor, a second stage to select a feature selector, and a third stage to set the hyperparameters of the predictor. Pipelines consisting of feature scaling, feature selection and a predictor are optimised in a specific series of stages based on the naïve assumption that each component can be optimised locally and independently. For instance, algorithm selection is performed on algorithms that are parameterised with their default hyperparameter setting. With the exception of the hyperparameter optimisation stage, which employs random search, all other stages evaluate solution candidates in a fixed order.

3.4.2 Systems based on Bayesian optimisation

Bayesian optimisation, introduced earlier in Section 3.2, is a popular method used for realising AutoML systems. Bayesian optimisation based on Gaussian process models is mainly applicable for low-dimensional problems with relatively few numerical hyperparameters. In contrast, Bayesian optimisation based on tree models is more suitable for high-dimensional, structured, and partially discrete problems, such as the CASH problem, and has been prominently used in AutoML systems (see, e.g., Thornton et al., 2013). In this section, we briefly describe a number of AutoML systems based on Bayesian optimisation.

Auto-WEKA (Thornton et al., 2013) is one of the earliest proposed AutoML systems. It tackles the CASH problem for a search space based on the algorithms available in the WEKA (Hall et al., 2009) machine learning environment, covering base classifiers, feature selection and meta-models for ensembling (voting, bagging, stacking). Auto-WEKA constructs machine learning pipelines consisting of an optional feature selector and a (meta-)model. Apart from feature selection, no further data preprocessing is supported. Auto-WEKA employs Bayesian optimisation based on SMAC (Hutter et al., 2011) (see Section 3.2.4) to optimise machine learning pipelines and their corresponding hyperparameters. Auto-WEKA 2.0 (Kotthoff et al., 2017) is the most recent version, in which supervised classification, as well as regression algorithms, are supported.

HyperOpt-sklearn (Komer et al., 2014) supports a search space based on the models and preprocessors in the Python library Scikit-learn (Buitinck et al., 2013). Series of multiple preprocessing steps are also supported by HyperOpt-sklearn, but

such series have to be explicitly specified in the search space. Hyperopt (Bergstra et al., 2013) (see Section 3.2.4) is used by HyperOpt-sklearn for the optimisation process.

Auto-sklearn (Feurer et al., 2015) is another AutoML system for generating machine learning pipelines by addressing the CASH problem using algorithms available in Scikit-learn (Buitinck et al., 2013). Auto-sklearn uses SMAC (Hutter et al., 2011) to build machine learning pipelines consisting of a data preprocessing step and a model. Compared to Auto-WEKA and HyperOpt-sklearn, auto-sklearn uses two additional techniques to increase the efficiency and accuracy: (1) Auto-sklearn supports the warm-starting of the Bayesian optimisation procedure from promising candidate solutions identified via meta-learning in order to accelerate the optimisation process. Meta-learning in Auto-sklearn is based on a set of simple information-theoretic and statistical meta-features. (2) Auto-sklearn also supports the construction of a voting ensemble consisting of several pipelines that were evaluated in the optimisation process. The ensemble construction used in auto-sklearn optimises the weights of the models in the ensemble in a greedy fashion, starting from an empty set and adding new models to it as long as it increases the validation accuracy.

PoSH (Portfolio Successive Halving) auto-sklearn (Feurer et al., 2018) is an extension of auto-sklearn with the aim of yielding good performance under tight time constraints. It introduces a more efficient meta-learning strategy and the option to use successive halving in the evaluation of pipelines in order to reduce the time spent in evaluating poorly performing candidate pipelines.

Auto-sklearn 2.0 (Feurer et al., 2022) further extends PoSH auto-sklearn by adding the possibility to automatically select the evaluation strategy (holdout or cross-validation, the number of folds in cross-validation, and whether to use successive halving or not) based on meta-learning.

Alternating direction method of multipliers (ADMM) AutoML (Liu et al., 2020) focuses on addressing the CASH problem to optimise pipelines composed of a data-preprocessor, feature selector and a machine learning model using ADMM (Boyd et al., 2011), an optimisation framework for solving complex constrained mixed integer problems based on decomposition. ADMM is employed to decompose and simplify the CASH problem into easier subproblems (e.g., differentiating between hyperparameter optimisation and algorithm selection) with a smaller number of hyperparameters that can be addressed using Bayesian optimisation. Reducing the number of hyperparameters can significantly speed up the convergence of the overall black-box optimisation process. The ADMM framework also supports constraints that, for

instance, restrict the prediction latency or the memory consumption of the machine learning pipeline.

AutoGluon-Tabular (Erickson et al., 2020) is an AutoML system with a focus on designing pipelines that generates complex model ensembles, excluding preprocessing steps. Its search space is defined over multiple models from Scikit-learn, XGBoost (Chen and Guestrin, 2016), LightGBM (Ke et al., 2017), CatBoost (Dorogush et al., 2018) and neural networks directly implemented in AutoGluon-Tabular. AutoGluon-Tabular takes a multi-layered ensembling approach, where multiple base models of the same type are first combined through a bagging ensembling approach. Multiple bagging ensembles can then be combined into a voting ensemble. The voting ensemble can be further extended by AutoGluon-Tabular by prepending one or multiple stacking layers consisting of multiple bagging ensembles each. The sequence in which models (base models and ensembles) are evaluated is fixed and predefined; essentially, more and more complex ensembles are constructed and evaluated during the search process. AutoGluon-Tabular provides different strategies for the optimisation of hyperparameters, but by default, Bayesian optimisation is employed. AutoGluon supports time-series forecasting.

3.4.3 Systems based on multi-armed bandits

Auto Tune Models (ATM) (Swearingen et al., 2017) is a distributed, collaborative, scalable AutoML system. ATM uses a hybrid Bayesian and multi-armed bandit optimisation method for optimising pipelines and offers model recommendations using meta-learning. It iteratively selects a model using a multi-armed bandit approach and executes one Bayesian opitmisation step on the hyperparameters of the selected model. ATM supports only pipelines consisting of single models without any data preprocessing. The focus of this system is on the support of multiple users in parallel, i.e., in the form of a cloud service.

Alpine Meadow (Shang et al., 2019) is an AutoML system integrated into the Northstar data science platform (Kraska, 2018). It optimises fixed pipelines consisting of data preprocessors and a model. This system focuses on providing interaction opportunities with the user. The system iteratively searches and recommends pipelines to the user and adapts its search space to the task using expert rules defined by the user. In each iteration of the search process, Alpine Meadow selects promising pipelines (i.e., steps of pipelines) through a multi-armed bandit approach and optimises the hyperparameters of these pipelines with Bayesian optimisation. Meta-learning is used

to warm-start the multi-armed bandit approach.

3.4.4 Systems based on evolutionary algorithms

As described in Section 3.2.2, evolutionary algorithms are population-based optimisation algorithms. They work on sets (populations) of solution candidates (individuals). Genetic programming (Koza, 1994) is a particular evolutionary approach that is often employed as the search strategy in AutoML systems as it allows for a flexible description of machine learning pipelines (see Section 3.2.2). EAs are easy to parallelise, as the individuals in a population can be evaluated in parallel.

The Tree-based Pipeline Optimisation Tool (TPOT) (Olson et al., 2016a,b) uses genetic programming to optimise tree-structured machine learning pipelines based on the search space defined over Scikit-learn. The leaves of the TPOT's tree-structured pipelines represent hyperparameters and (copies of) the input data, while the inner nodes represent pipeline operators (preprocessors, decomposition, feature selection, models). Taking each operator in a pipeline as a primitive, TPOT can construct arbitrarily complex pipelines using genetic programming. However, this may also lead to infeasible pipelines (e.g., a classification pipeline that does not include a classifier). Furthermore, the complexity of the resulting pipelines increases the risk of overfitting. To address this latter issue, TPOT maintains a Pareto front of previously evaluated pipelines with respect to the two objectives of prediction accuracy and complexity (in terms of the number of pipeline operators), allowing the user to choose a reasonable trade-off.

Layered TPOT (Gijsbers et al., 2017) and **TPOT-SH (Parmentier et al., 2019)** are both extensions of TPOT that focus on accelerating the optimisation process on large data sets. This is achieved by first evaluating pipelines on smaller subsets of training data and selecting only the promising pipelines for training on larger subsets. Both of these approaches induce the risk of missing viable pipelines that perform poorly on a subset of the data but achieve high accuracy when trained on the entire data set.

RECIPE (REsilient ClassifIcation Pipeline Evolution) (de Sá et al., 2017) is an AutoML system focused on constructing classification pipelines that include preprocessing and classification models (from Scikit-learn) using grammar-based genetic programming (GGP) (McKay et al., 2010). Compared to TPOT's flexible genetic programming approach, that may lead to infeasible pipelines, using this grammar-based approach, RECIPE constrains the genetic programming process

(more specifically, the crossover and mutation operations) by employing background knowledge on the structure of feasible pipelines. Similar to TPOT, RECIPE does not rely on a fixed pipeline structure and can construct complex pipelines. In contrast to TPOT, RECIPE does not support the union of multiple preprocessing steps, but only sequences of preprocessing steps. On the other hand, it supports voting ensembles, which are not supported by TPOT.

Autostacker (Chen et al., 2018a) is an AutoML system that employs a basic evolutionary algorithm (different from genetic programming) to optimise machine learning pipelines with a search space composed of models (excluding preprocessors) from Scikit-learn (Buitinck et al., 2013) and XGBoost (Chen and Guestrin, 2016). In contrast to TPOT and RECIPE, which optimise single models, Autostacker constructs pipelines with a special stacking structure. These pipelines consist of multiple layers each, including multiple machine learning models. The models in the first layer take the raw input data as input, and models in subsequent layers can additionally make use of the outputs of models in preceding layers.

General Automated Machine learning Assistant (GAMA) (Gijsbers and Vanschoren, 2020) is an AutoML system that uses genetic programming to generate machine learning pipelines based on a given input data set. GAMA is implemented based on the search space of Scikit-learn and automatically constructs pipelines that include preprocessing and machine learning models. GAMA supports building an ensemble of the evaluated pipelines. The main distinguishing feature of GAMA compared to other AutoML systems, such as auto-sklearn and TPOT, is its focus on transparency to serve AutoML researchers by producing extensive log files about the behaviour of the population of pipelines during the optimisation process.

AutoML-DSGE (Assunção et al., 2020) is another AutoML system that employs grammar-based genetic programming to optimise classification pipelines consisting of Scikit-learn data preprocessors and models. AutoML-DSGE uses dynamic structured grammatical evolution (DSGE) (Lourenço et al., 2018), which is an extension of grammatical evolution (GE). GE uses a linear representation, i.e., individuals are represented in the form of lists of integers. Compared to tree-based representations, as used by RECIPE, a linear representation has the advantage of being easier to operate on and permitting the application of a wider range of evolutionary operators (McKay et al., 2010). DSGE uses an improved encoding with a higher locality (i.e., small changes in the genotype yield also only small changes in the phenotype) and a lower redundancy (i.e., different genotypes yield the same phenotype) compared to GE.

FEDOT (Nikitin et al., 2022) is an EA-based AutoML system for optimising pipelines, which operates in two phases: composition and hyperparameter tuning. During the composition phase, analogously to TPOT, FEDOT optimises the structure of machine learning pipelines represented as DAGs with preprocessors and models from Scikit-learn, XGBoost, LightGBM, and CatBoost, with the help of a tree-based genetic algorithm. However, in contrast to TPOT, it does not optimise the hyperparameters at this stage but considers only default hyperparameter settings. In a second phase, it tunes the hyperparameters of the best pipeline found by the genetic algorithm by means of Bayesian optimisation, using Hyperopt (Bergstra et al., 2013) (see also Section 3.2.4).

3.4.5 Systems based on Monte Carlo tree search

Monte Carlo tree search (MCTS) is a heuristic search approach that has been used widely in turn-based games, such as Chess or Go. As outlined in Sections 3.2 and 3.3, MCTS has also been used in hyperparameter optimisation and neural architecture search; furthermore, there are a few systems for the optimisation of machine learning pipelines based on MCTS.

MOSAIC (Rakotoarison et al., 2019) is an AutoML system designed for optimising machine learning pipelines with a predefined number of steps based on a search space defined over Scikit-learn (machine learning models and preprocessors). MOSAIC employs a hybrid optimisation approach combining MCTS and Bayesian optimisation. The authors of MOSAIC motivate this hybrid approach by pointing out that MCTS can efficiently solve sequential decision problems (in this case, the sequence of components that form a pipeline), whereas Bayesian optimisation is well-suited for solving expensive optimisation problems (in this case, hyperparameter settings of the components of the pipeline). The two extreme solutions one could consider for jointly optimising the structure and hyperparameters of a pipeline are: (i) to optimise hyperparameters of every possible pipeline structure or (ii) to estimate the performance of a pipeline structure by sampling a few hyperparameter settings of each pipeline structure. MOSAIC adopts an intermediate solution by coupling these two optimisation approaches tightly via a shared surrogate model, which, on the one hand, is incorporated in the MCTS approach by approximating the average performance of pipelines, and on the other, allows hyperparameter optimisation of given pipelines using Bayesian optimisation. Similar to Bayesian optimisation approaches, MOSAIC builds a surrogate model from machine learning pipelines and their hyperparameter

configurations evaluated earlier in the search process. Next, this surrogate model is used to derive a second surrogate model on pipeline structures from a number of hyperparameter configurations sampled for each structure. The second model is used to guide the MCTS.

Oracle AutoML (Yakovlev et al., 2020) is an iteration-free AutoML system for designing pipelines based on meta-learning and a gradient descent approach. Algorithm selection, adaptive data reduction, and hyperparameter optimisation are the three main components in the Oracle AutoML system. In the algorithm selection component, the Oracle AutoML system uses meta-learning to predict the relative performance of algorithms on subsets of the given data and selects the best algorithm according to these predictions. The adaptive data reduction component selects a representative subset of the data. On this subset, the hyperparameters of the selected algorithm are optimised in parallel, using a combination of gradient-based optimisation (for continuous and discrete parameters) and a brute-force approach (for categorial parameters).

3.4.6 Systems based on other methods

In this section, we discuss AutoML systems based on approaches other than those covered in previous subsections.

ML-Plan (Mohr et al., 2018) is an AutoML system for constructing fixed AutoML pipelines (composed of a preprocessor and a classifier) based on a search space defined over Weka and Scikit-learn. ML-Plan does not use any of the HPO approaches mentioned previously but rather regards AutoML pipeline construction as an AI planning task. Hierarchical planning can be used to organise the search space, and the resulting AI planning problem can be solved using the well-known approach of hierarchical task networks (HTNs) (Erol et al., 1994). In this case, AutoML tasks can be understood as graph search problems. A plan is a sequence of actions (e.g., selection of a specific algorithm) that can be organised in a hierarchically structured network that represents the dependencies of the actions. A lack of representative data for the search process leads to overfitting for most AutoML systems. ML-Plan addresses this problem by applying a two-phase search mechanism (search and selection). In the first phase, a global best-first graph search algorithm is used to identify good candidate pipelines on a part of the training data. In the second phase, the final pipeline is selected by applying Monte Carlo cross-validation on the full training data to the candidate pipelines identified in the first phase.

ML2-Plan (Wever et al., 2018) was introduced to extend ML-Plan for multi-label classification. While most AutoML systems focus on single-label classification or regression, ML2-plan builds upon the search space of MEKA (Read et al., 2016) – a multi-label extension of WEKA. Since the preprocessors in ML-Plan are for single-label classification, they were not included in ML2-Plan.

Oboe (Yang et al., 2019) is an AutoML system that optimises machine learning pipelines to create model ensembles with the help of meta-learning. Oboe employs collaborating filtering to select models for new data sets based on their performance on similar data sets. In an offline stage, it evaluates all candidate models (in the form of algorithms and corresponding hyperparameter settings) on a set of data sets and creates a matrix of cross-validation errors for those models. As opposed to explicitly extracting meta-features, fitting a low-rank model on this matrix allows learning latent meta-features for models and data sets that best describe the cross-validated error. To predict the performance of models on new data sets, Oboe introduces a time-constrained matrix completion method based on the optimal experiment design approach (Pukelsheim, 2006), a classic method to define optimal experiments according to statistical criteria. In this case, the optimal experiment is inferring the meta-features of the data sets, and the statistical criterium is minimising the covariance of the estimated meta-features.

TensorOboe (Yang et al., 2020) is an extension of Oboe that allows the optimisation of full pipelines consisting of data preprocessing steps, like imputation and scaling, and a model.

Fast and Lightweight AutoML (FLAML) (Wang et al., 2021) is an AutoML system that other than model accuracy, focuses on optimising computational resources for an efficient search process. FLAML optimises machine learning pipelines consisting of a single model without data preprocessing with a search space defined over models from Scikit-learn, XGBoost, LightGBM and CatBoost. For the hyperparameter optimisation, a direct search approach, as proposed by Wu et al. (2021), is employed. The goal of the search is to minimise the total CPU time for finding a model with a small error. To achieve this goal, in each search iteration, next to selecting a model and a corresponding hyperparameter configuration to evaluate, FLAML also determines the training sample size used in the evaluation as well as the evaluation strategy (hold-out or cross-validation). Based on previous evaluations, FLAML estimates for each model the CPU time of finding a configuration that results in an error lower than the currently best one. These estimations are used to select models and training sample sizes for each iteration of the search. This way, starting from cheap

trials and low-performing models, the search will gradually move to expensive trials with more accurate models.

3.4.7 Selecting an AutoML system

The choice between a large number of different AutoML systems can pose significant challenges for practitioners. The recent survey by Scriven et al. (2022) has compared some of the above-mentioned AutoML systems qualitatively, based on factors such as the effort and level of expertise needed to use them effectively. Additionally, the choice of the AutoML system to be used can be determined based on the nature of the data sets at hand, based on the following considerations:

- **Search space:** For practical reasons, a user might prefer to use certain machine learning algorithms or libraries. The type of data sets encountered in a specific use context also determines the preprocessing steps needed in the pipeline. In that case, it should be considered if the search space of the AutoML system contains the respective methods.
- **Ensembling strategy:** An advanced ensembling strategy can be expected to configure models with much higher accuracy than achievable by a single algorithm; however, this typically comes at the cost of additional computational resources.
- **Meta-learning:** A user should consider if there is meta-learning implemented in the AutoML system, but more importantly, if the data sets at hand are similar to the data sets considered for meta-learning by the AutoML system and are expected to benefit from meta-learning.
- **Interpretability:** A user should consider if the output of the AutoML system is interpretable for the data scientists operating it and any other stakeholder who needs to understand the output. In addition, an adequate visualisation of the AutoML optimisation process can offer more transparency and strengthens the user's trust in the AutoML system (Zöller et al., 2022).

A user might also consider efficiency in terms of running time and other resource requirements. This, however, requires a thorough and fair quantitative assessment of AutoML systems (e.g., by using an identical search space for all systems). A recent comparative study (Gijsbers et al., 2022) covering some of the previously mentioned AutoML systems provides some guidance in this respect.

A few AutoML systems support time-series forecasting (e.g., Auto-PyTorch, AutoKeras). The choice of the AutoML system to be used for time-series analysis tasks also can be determined based on the aforementioned considerations. However, currently, there are not so many options available for time-series analysis tasks. The AutoML systems that support the time-series forecasting we mentioned are all NAS-based systems.

3.5 Conclusion

In this chapter, we offered a comprehensive overview of the most important and impactful techniques in AutoML. We aimed to provide a strong basis for further developments in AutoML. Specifically, we provided the comprehensive background information needed to understand the advanced approaches taken in AutoML research; cover the topic of hyperparameter optimisation, a core technique used in most AutoML systems; provide an overview of the available state-of-the-art AutoML systems for classic machine learning and deep neural networks. However, there is very limited AutoML work specifically designed for time-series forecasting tasks. This thesis fills the gap between AutoML and time-series forecasting.