Alma Mater Studiorum · Università di Bologna

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Artificial Intelligence
MASTER THESIS
in Combinatorial Decision Making and Optimization

Automated Feature Extraction for Algorithm Selection in Combinatorial Optimization

Candidate:
Alessio Pellegrino

Supervisor:
Prof. Zeynep Kiziltan
Co-Supervisors:
Dr. Özgür Akgün
Dr. Nguyen Dang
Prof. Ian Miguel

IV Session
Academic year 2023/24

Abstract

Given a combinatorial problem, there could be multiple ways to model it into a constraint optimization model that could be solved by a solver. Choosing the right combination of a model and a target solver can have significant impact on the effectiveness of the solving process. Furthermore, the choice of the best combination of constraint model and solver can be instance-dependent, i.e., there may not exist a single combination that works best for all instances of the same problem. In this thesis, we consider the task of building machine learning models to automatically select the best combination for a problem instance. A critical part of the learning process is to define *instance features*, which serve as input to the selection model. The choice of the feature set can widely impact the final performance of the machine learning model. During the year, most of the work has focussed on building the feature set of an instance starting from a low-level representation and a brief execution of the instance on a solver. Here, we aim to learn the instance features directly from the high-level representation of a problem instance using a transformer encoder. This approach not only allows us to incorporate high-level semantics which could be lost in the low-level representation but it also removes the necessity of running the instance saving time. Furthermore, using a transformer encoder to learn the features eliminates the need to hand-craft the feature set, a process which may be very long and error-prone. We evaluate the performance of our approach using the Essence modelling language with a case study involving three different problem classes.

Acknowledgements

I would like to thank my parents for all the support they gave me during the years and especially for not mocking me too hard when I decided to give up a well-paid job in favour of another degree in something they don't even know how to pronounce. I would also like to thank my sister for being the fun one in the family, leaving me only the job of being the smart one. Thank you, Viola this would never have been possible without you.

Another special thanks goes to my Supervisor, Prof. Zeynep Kiziltan, who believed in me and allowed me to develop this work both here and in Scotland at the University of St. Andrews. I would also like to thank Doc. Özgür Akgün and Prof. Ian Miguel for all the help he provided when I was in St. Andrews and Dott. Nguyen Dang for taking an interest in our work and for joining us to help. Finally, I would like to thank the University of St. Andrews and all the staff from the School of Computer Science for being so welcoming to me and giving me all the resources I needed.

But the most important thanks goes to me for developing this thesis and everything related to it. Thank you, Alessio.

This work was supported by the European Union's Justice programme, under GA No 101087342, POLINE (Principles Of Law In National and European VAT) and by a scholarship from the Department of Computer Science and Engineering of the University of Bologna.

Contents

Chapter 1

Introduction

The following chapter serves as an introduction to this thesis. It will start with a general introduction to the research field in Section 1.1. Section 1.2 illustrates the reasons behind the choose of this thesis topic. Section 1.3 will, then, focus on the main contributions of our research and, finally, Section 1.4 will describe the organization of this document.

1.1 Research Area

The field of Combinatorial Optimization [31] is a well-established and extensively studied area that lies at the intersection of mathematics and computer science. Its primary objective is to determine the optimal solution for a given combinatorial problem. Over the years, numerous techniques have been developed to express and model combinatorial problems, each requiring varying levels of expertise to be effectively utilized. Alongside these modelling techniques, such as SAT and Constraint Programming, a variety of programs capable of interpreting the resulting models and automatically computing solutions have been created. These programs are commonly referred to as solvers.

To facilitate the modelling phase of a combinatorial problem, the researchers have developed many high-level languages such as Essence and Minizinc. These languages abstract much of the complexity of the modelling process and also introduce automatic reformulations that enhance the final model performances. Furthermore, they make it very easy to reuse the same model over multiple solvers as their formulation is solver-independent.

Despite the significant progress made in developing such techniques and solvers, none have been able to entirely overcome the inherent complexity of combinatorial optimization problems. Indeed, it has been formally proven that many of these problems, such as the well-known Traveling Salesman Problem (which involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting point) [30], are NP-complete. Consequently, any solver capable of solving such problems must also be NP-complete. However, in practice, it has long been observed that there is no single algorithm that performs best on all problems or even on all instances of the same problem [39, 44, 66]. To solve difficult problems, it has been often proven effective to employ a portfolio of algorithms with complementary strengths.

The area of research dedicated to identifying the best algorithm of a portfolio for solving a specific problem is known as Algorithm Selection. This field had, during the years, been successfully applied to many research domains including Boolean Satisfiability [80], Constraint Programming [62, 52], AI planning [75], and combinatorial optimisation [45]. Since it is often difficult to determine which solver will perform best on a given instance, the authors of SATzilla [80] (along with many subsequent researchers) have employed machine learning algorithms to establish a correlation between solvers and their performance on different problem instances.

Most existing Algorithm Selection tools for Combinatorial Optimization problems, such as SATzilla and SunnyCP [8], rely on traditional machine Learning algorithms, such as Random Forest [13]. These algorithms typically require numerical features that effectively represent the input in order to make accurate predictions. In the context of Algorithm Selection, the input to the Machine Learning algorithm is a problem instance. However, identifying appropriate features to represent problem instances is a challenging task. Different tools adopt different sets of features, which are generally derived by partially executing the instance on a solver and then collecting relevant statistics from the execution process.

1.2 Motivations

As previously discussed, determining the most effective set of features to use for Algorithm Selection in combinatorial optimization is far from straightforward. Moreover, to the best of our knowledge, no prior research has focused exclusively on the feature extraction process in this specific context.

The standard approach used to extract instance features is to convert the instance to a low-level representation and run it on a solver. The final result is a combination of running statistics gathered from the solver execution and some static features gathered from the low-level instance. Although this process has been proven effective in multiple works [80, 8], it also presents some drawbacks: Firstly, converting the instance and running it takes a non-negligible amount of time limiting the possible time savings of the algorithm selection process. Furthermore, during the conversion from a high to a low-level representation, some relations in the instance can be lost. Relations that could, instead, be really useful during the selection process. Finally, the features extracted using this methodology had to be manually hand-crafted and selected by some field experts. This process can be long and error-prone and could, ultimately, decrease the effectiveness of the final result.

1.3 Contributions

In this work, we introduce a novel approach for feature extraction based on a Transformer Encoder, which directly processes the textual representation of instances without requiring either parsing or partial execution by a solver. This offers three different advantages: Working at the high-level instance offers both a lower computational cost and captures the semantics that may get lost at a lower-level representation. Finally, our approach removes the need to hand-crafting the features to use. Using a transformer encoder also allows to automatically learn the features that better suit the task at hand. Although our methodology was empirically tested on ESSENCE instances, it remains highly general and applicable to a wide range of combinatorial optimization problems, without imposing any assumptions on the underlying solving strategies.

Furthermore, for our work, we used a portfolio of models as well as a portfolio of solvers to choose from. The final algorithm, as we intend in this work, is in fact the result of a combination of a model and an algorithm.

To the best of our knowledge, there is only one prior work based on Transformer Encoders used in a similar context but it was applied to a much more limited scope and it was limited to only one problem class [68]. We will present the neural network architecture employed in our feature extraction process and describe the training methodology designed to capture the semantics of the input instances. Following this, we will evaluate the extracted features across various algorithm selection approaches and provide a comparative analysis of our results against those obtained using an existing feature set developed for similar purposes.

1.4 Organization

This thesis is structured as follows:

Chapter 2 provides a comprehensive overview of the foundational technologies and concepts necessary to understand the work presented in this thesis, along with a discussion of prior research closely related to our contributions. Specifically, this chapter delves into the fields of Combinatorial Decision Making and Optimization, Machine Learning, and Deep Learning. In addition, it offers an in-depth exploration of the field of Algorithm Selection.

Chapter 3 describes in detail the methodology employed to achieve the results presented in this thesis. It includes a thorough discussion of the deep neural network architectures utilized, explaining the rationale behind their selection and the way in which they are trained to extract meaningful features from problem instances. Moreover, this chapter outlines how the extracted features are integrated into various algorithm selection algorithms to enhance solver performance.

Chapter 4 focuses on the different problem classes selected for evaluation, as well as the datasets constructed based on these classes. A detailed description is provided regarding the modelling of each problem class into combinatorial optimization models, the subsequent translation of these models into lower-level representations, and the solvers employed for executing those models. Additionally, this chapter specifies the number of instances generated for each problem class, explaining the process followed to gather and organize them into datasets suitable for experimentation.

Chapter 5 begins by detailing the training procedures followed for the deep neural networks used in this study, including the hyper-parameters and configurations adopted. It then presents a systematic outline of the experiments conducted to assess the efficacy of our feature extraction methodology. This chapter provides a detailed analysis of the experimental results, comparing the performance of our approach against baseline results obtained using alternative features drawn from existing literature.

Finally, Chapter 6 concludes the thesis by summarizing the main findings and contributions of our work. It discusses both the strengths and limitations of the proposed approach, providing a critical evaluation of its practical applicability. The chapter also outlines potential directions for future research, suggesting possible improvements and extensions that could further enhance the results obtained.

Chapter 2

Background

This chapter offers an overview of all the technologies and concepts needed to develop this project as well as a view of the current landscape of the related fields.

Section 2.1 will focus on the fields of combinatorial decision-making and Optimization which is the field we will apply our research on. Section 2.2 will give an overview of the field of Machine Learning while Section 2.3 mainly focuses on neural networks. Section 2.4 will focus on algorithm selection and its application to constraint programming and related fields. Finally, Section 2.5 will give an overview of the relevant related topics and research.

2.1 Combinatorial Decision Making and Optimization

This section will focus on discussing the field of combinatorial decision-making and optimization by, firstly, giving a formal definition of it. Then, Section 2.1.2 will give an overview of the exact methods to solve combinatorial problems while 2.1.1 will focus on approximate methods.

A combinatorial problem is a problem whose solution consists of an assignment of values to a set of variables with, typically, a discrete domain such that a set of *constraints* are satisfied. More formally, we can define a combinatorial problem with a tuple (V, C, D) where V is a set of variables, C a set of constraints over those variables and D a set of domains for the variables such that D_v is the domain of the variable v. The goal is to find an assignment for all variables such that all constraints hold true. Combinatorial problems come in two forms: (i) satisfaction problems where any valid assignment is a satisfactory solution. For example, the N-queen problem [40] is a satisfaction problem where we want to find a position for N different queens in an N $\times$ N chess board in such a way that all queens do not attach each other. Chess rules apply. (ii) Optimization problems where, on top of finding a feasible solution, the problems also require minimising/maximising an objective variable. For optimization problems, a scoring function F is also needed to assess the quality of the solution. An example of an optimization problem is the travelling salesman problem (TSP) [30]. The TSP problem is the problem of finding the circuit with the smallest length in a graph. The study of algorithms and techniques for solving combinatorial problems is called Combinatorial Decision-Making and Optimization which we will call Combinatorial Optimization from now on. Given a combinatorial problem, there are several ways we can use to find a feasible solution for it. Among the possible categorizations of the solving methods, we will focus on the approximate methods vs exact methods categorization.

2.1.1 Approximate Methods

Approximate methods are fast algorithms that find solutions for combinatorial problems. They can generally find nearly optimal solutions but often get stuck on local minima or maxima and never converge to an optimal solution. These methods are generally meant for optimization problems but can be easily adapted for decision ones.

Local Search Local search methods are iterative optimization techniques that start with an initial solution and iteratively move to a neighbouring solution with the aim of improving an objective function. Unlike global search methods, local search restricts the exploration to a neighbourhood of the current solution, which makes it computationally efficient but prone to getting stuck in local optima. Prominent examples of local search include hill climbing, simulated annealing [43], and variable neighbourhood search [29]. These methods are widely used in practice due to their simplicity, speed, and effectiveness for large combinatorial search spaces.

In general, local search relies on the concept of *neighbourhood structures*, which define the set of possible moves from one solution to another. The choice of the neighbourhood structure plays a crucial role in the algorithm's performance, as it affects the search space's connectivity and the ability to escape local optima. While basic local search can terminate at a local optimum, more advanced variants, such as simulated annealing, introduce mechanisms to accept worse solutions with a certain probability, promoting exploration and improving the likelihood of finding better solutions.

Meta-Heuristic Methods Meta-heuristic methods are higher-level strategies designed to explore the search space more broadly, making them effective at solving complex Combinatorial Optimization problems where traditional methods struggle. Unlike local search methods, which primarily rely on neighbourhood structures, meta-heuristics incorporate additional mechanisms for escaping local optima and balancing exploration (searching new areas) and exploitation (refining known good solutions). Popular meta-heuristic algorithms include genetic algorithms [47], ant colony optimization [23] or particle swarm optimization [24].

These methods typically use a population-based or memory-driven approach. For instance, genetic algorithms simulate the process of natural selection by iteratively evolving a population of solutions using operators like selection, crossover, and mutation. Ant colony optimization uses a decentralized agent-based approach where artificial ”ants” traverse the problem space and deposit pheromones to guide future searches. Meta-heuristics are highly adaptable and can be tailored to specific problem domains, making them suitable for applications ranging from logistics and scheduling to Machine Learning and network optimization.

2.1.2 Exact Methods

Exact methods will always find the optimal solution for a Combinatorial Optimization problem given enough compute time. However, in order to prove optimality, the algorithm may have to navigate the entire search space making exact methods much slower in contrast to approximate ones. Generally, these algorithms build the solution by navigating a search tree where each node corresponds to a sub-problem derived from the original problem. A naive attempt at solving combinatorial optimization problems could be to navigate the search tree until all search variables have been assigned and then verify the correctness of the solution. This is called a “pure search” algorithm. While correct, this algorithm is exponential in time complexity making it very hard to converge to a solution. Different approaches have found different ways to cut down the search space, for example, many OR-solvers algorithms use a process called *“relaxation“* which aims at solving a simplified version of the problem to obtain a bound to the actual problem.

Branch and Bound A better approach to pure search is to use the branch and bound paradigm [48]. At each step of the algorithm, a selected node from the search tree is expanded (or “branched”) into smaller sub-problems by partitioning the decision space. For each sub-problem, the algorithm computes a bound—either an upper or lower bound on the optimal solution that could be obtained from that sub-problem. If this bound indicates that the sub-problem cannot lead to an improvement over the current best solution, it is discarded (pruned) from further consideration. This process continues iteratively, focusing only on sub-problems that have the potential to improve the current solution until the optimal solution is found or no further sub-problems remain.

Constraint Propagation [11] is a popular technique to cut the search space used in constraint programming. Constraint propagation exploits the set of constraints to reduce the domain of unassigned variables based on the assigned variables in the current solution. A common method to propagate constraint is General Arc Consistency (GAC) which ensures that every value in the domain of each variable is consistent with the constraints involving that variable. Formally, a variable v_i with a domain D_{v_i} is considered GAC if, for every constraint C involving v_i, each value $d_i \in D_{v_i}$ can be extended to a complete set of assignments that satisfies C. This process iteratively removes inconsistent values from the domains of the variables until a fixed point is reached, significantly reducing the search space for Combinatorial Optimization problems.

To further improve constraint propagation, many solvers have also included dedicated algorithms for some common constraints. These constraints are called global constraints [76]. Unlike basic constraints that operate on individual pairs of variables, global constraints involve a larger set of variables and exploit their interrelationships to achieve stronger propagation. Examples of commonly used global constraints include `AllDifferent`, which ensures that a set of variables take unique values, and `Cumulative`, which enforces resource constraints in scheduling problems.

Even if each of the strategies discussed above has been proven effective [48, 54], many combinatorial problems have been proven to be NP-Complete and NP-hard [41] and, therefore, no polynomial algorithm exists to solve these problems.

2.1.3 Modelling

The process of solving a Combinatorial Problem comprises two fundamental steps: (i) constructing an appropriate model, and (ii) instantiating and solving the model using a solver. Both steps are critical to obtaining feasible solutions effectively; a well-structured model can significantly improve solver performance and the likelihood of finding an optimal solution. This section will focus specifically on the modelling phase.

There is no single correct approach to writing a combinatorial model [70], but the objective is generally to maximize the search efficiency. Some general techniques to design a good model are:

1. **Minimize the Number of Variables**: Reducing the number of variables in the model helps simplify the search space and reduces computational overhead. Where feasible, leverage auxiliary variables only when they significantly enhance model expressiveness or propagation.

2. **Constrain Variable Domains**: Limit the domains of variables as tightly as possible. Narrowing domains through upper and lower bounds or other

constraints can avoid unnecessary exploration of infeasible regions in the solution space, expediting convergence to optimal solutions.

Despite these general principles, determining the most effective model can be challenging. Often, different models are more suitable for distinct problem instances, and achieving an optimal formulation may require iterative refinement. To aid in this process, researchers have developed high-level modelling languages that can be transpiled automatically to lower-level representations. This transpilation enhances model flexibility and allows solvers to make use of specialized propagation algorithms.

Modelling Languages Over the years, a variety of modelling languages have been developed to support the modelling process. These modelling languages often rely on constraint programming techniques due to the expressiveness and flexibility of this paradigm. each offering different levels of abstraction and functionality. These languages simplify the process by allowing users to express constraints more intuitively while enabling advanced solver optimizations. Key languages include:

- **Minizinc** [58]: A high-level language that includes support for numerous global constraints and search heuristics, offering a rich modelling framework. Models in MINIZINC are typically transpiled to Flatzinc, making them compatible with a wide array of CP solvers.

- **Essence Prime** [61]: Similar to MINIZINC, ESSENCE PRIME is a high-level language designed to simplify constraint modelling. It employs a different reformulation pipeline compared to MINIZINC that allows for different refomualtions.

- **Essence** [27]: ESSENCE aims to provide maximum abstraction in modelling, enabling highly general and compact models. It supports a broader range of combinatorial constructs, making it suitable for modelling complex problems with minimal effort from the user.

These languages facilitate experimentation with different model formulations and enable modellers to leverage built-in heuristics and optimizations, ultimately improving the efficiency and flexibility of constraint programming approaches. ESSENCE PRIME and MINIZINC offer a similar level of abstraction, ESSENCE, on the other hand, offers higher abstraction compared to other languages. To translate the model into a lower lever it is necessary to translate the model into an ESSENCE PRIME one and the translation must be assisted by the user whenever a non-trivial transformation must be done.

2.1.4 The Essence Pipeline

While also MiniZinc has a translation and reformulation pipeline that results in an enhanced model, this section will only focus on the Essence pipeline since it is the one used in this project.

The Essence pipeline is designed to systematically transform a high-level, abstract Essence model into a concrete form that can be efficiently solved. This transformation is accomplished in two main stages, utilizing the tools Conjure [2] and Savile Row [60].

Conjure—Translating Essence to Essence Prime Starting from a high-level Essence model, a user can employ Conjure to translate it into Essence Prime. This translation step converts abstract modelling constructs into a more concrete form that is compatible with solving engines, bridging the gap between the user's conceptual model and the solver's operational model. The translation process in Conjure is driven by rule-based heuristics and includes the following key features:

- **Automated Decision-Making**: Conjure can assist in making non-trivial modelling decisions by querying the user when needed. Alternatively, it can autonomously select variable representations (e.g., compact vs. sparse) using predefined heuristics. The last option available is to produce all possible Essence Prime models. This option can be helpful to create a portfolio of models or to manually test all the available options.

- **Symmetry Recognition and Breaking**: The rule-based translation approach employed by Conjure not only refines the model iteratively but also identifies and breaks symmetries within the model. Symmetry breaking is essential in reducing the search space, which accelerates the solving process.

Savile Row—Refinement and Encoding for Solvers Once the Essence Prime model is produced, it is further processed by Savile Row, a modelling assistant tool that performs additional refinement and translates the model into a syntax compatible with the chosen solver. The refinement process applied by Savile Row involves a series of transformations that enhance the model's computational efficiency:

- **Abstract Syntax Tree (AST) Transformation**: Initially, Savile Row converts the model into an AST representation. Various transformation rules are then applied to optimize the AST, such as flattening nested expressions, unrolling loops and quantifiers, and breaking down complex expressions into simpler components.

- **Sub-expression Elimination**: Savile Row performs sub-expression elimination to identify and consolidate repeated expressions within the model. For instance, given the constraints:

   1. $w + x + y + z = 6$
   2. $z + y + w = 5$

   the common sub-expression $z + y + w$ is replaced with a new variable a, simplifying the model and revealing that $x = 1$ (since $x + a = 6$ and $a = 5$). This process reduces redundancy and improves solver efficiency by simplifying the model structure.

- **Tabulation of Constraints**: Savile Row supports the aggregation of multiple constraints into a single *table constraint*, an operation known as tabulation [3]. By organizing constraints in tabular form, the solver can process them as a unified entity, enhancing propagation and reducing computational overhead.

- **SAT Encoding Option**: Savile Row provides the flexibility to translate CP models into SAT (Boolean satisfiability problem) formulations, enabling the use of SAT solvers as a back end for constraint programming. This approach leverages the efficiency of SAT solvers for certain types of combinatorial problems and allows the CP model to be evaluated within the SAT framework.

After the Savile Row's reformulation has been completed, the model passes to the chosen solver which finds a solution. The solution can then be fed back to Savile Row and Conjure to be consistent with the original Essence model.

2.2 Machine Learning

In this section, we will explore the field of Machine Learning (ML), explain the differences between supervised and unsupervised techniques, and provide an overview of the models used in this project. In particular, Section 2.2.1 defines supervised methods, the decision tree model and its implementation and gives an overview of the random forest model as well. Section 2.2.2 gives a detailed description of unsupervised machine learning and of the K-means and hierarchical clustering algorithms.

ML is a subfield of artificial intelligence (AI) that focuses on creating algorithms and statistical models that enable computers to improve performance on tasks through experience, rather than relying on explicit programming [81]. Essentially, it is the process of automatically identifying patterns in data and using those patterns to make predictions or decisions. In practice, an ML algorithm takes in a dataset (referred to as training data) and generates a model: a mathematical function that, when presented with new data, can make predictions based on patterns found in the training set.

Rather than using raw data as input, ML models typically rely on specific measurable properties of the input, known as features. For example, features that describe a person might include attributes like height, gender (if applicable), and hair colour. These features are carefully selected to align with the specific task the model is designed to perform.

ML algorithms can be classified into three main categories: supervised, unsupervised and mixed. While is it also possible to make this distinction basing it on the need for the algorithm to receive data (supervised) or being able to generate it itself (unsupervised), here we will base our distinction on the presence of labels for the data, for supervised learning that will be discussed in Section 2.2.1, or the absence of labels in the case of unsupervised learning that will be discussed in Section 2.2.2. In this case, we define mixed approaches as those methods that rely on both labelled and unlabelled data in different steps of the training process. Neural Networks can often be classified as mixed approaches methods. Also notable is the existence of ML algorithms that can automatically generate data to be trained on, Reinforcement learning approaches [36] appertain in this category because their training process implies a simulation step where the actions taken by the model are simulated and then the ML algorithm is updated based on the outcome of the simulation step.

The most common tasks in ML include:

- **Classification**: A supervised task where the model assigns discrete labels (e.g., spam or not spam) to data points.
- **Regression**: Another supervised task similar to classification, but instead of predicting a category, the model outputs continuous values (e.g., predicting house prices).
- **Clustering**: An unsupervised task that involves grouping data points into clusters based on shared features, without pre-existing labels (e.g., segmenting customers into groups with similar behaviours).

Every ML algorithm undergoes a *tuning phase*, during which it processes the training data to learn how to label or predict future data points. The tuning method varies for each algorithm and plays a crucial role in the model's effectiveness. The quality and quantity of training data are particularly important; models trained on data that closely resembles the real-world distribution are more likely to perform well in real-world applications.

A key challenge in ML is achieving the right balance between generalization and specificity. If a model is too simple (underfitting), it will not capture the underlying patterns in the data, resulting in poor performance. Conversely, if the model is overly complex and fits the training data too closely (overfitting), it may capture noise or irrelevant details, leading to poor generalization and performance on new, unseen data. Finding the right balance between these two extremes is critical to building robust ML models. To allow the user to avoid under or overfitting, most ML algorithms offer a number of hyper-parameters: values that will be used to guide the training of the model. To measure the quality of the trained model before going through some test data which is data we already know how to categorize (similarly to the training data) but we do not show to the model during training. It is also common to use a *validation set* which is a dataset the model is not trained on and that we can use to evaluate the model.

2.2.1 Supervised Machine Learning

A supervised ML algorithm requires labelled data, meaning the input data comes with known outcomes (labels). The algorithm learns by mapping inputs to the correct outputs, building a model that can predict labels for new, unseen data.

Decision Trees A decision tree [14] is a classification and regression model structured as a tree, where each leaf represents a label or prediction, and each internal node represents a decision or condition that splits the data. For classification tasks, each leaf contains a class label, while for regression tasks, each leaf contains the average of the training values assigned to that leaf. To make a prediction, the decision tree starts at the root and traverses down by selecting child nodes based on the condition at each node, eventually arriving at a leaf with the predicted label or value.

The training process of a decision tree is a recursive algorithm that splits the data at each node based on selected features. The base case occurs when the data at a node belongs to a single class (for classification) or is *small* enough, in which case the node becomes a leaf. Otherwise, the algorithm selects a condition to split the data into child nodes and continues the process recursively.

For classification trees, the criteria for selecting the splitting condition include:

- **Information gain**: Information Gain measures the reduction in uncertainty or entropy after a split. For a node T it is defined as $IG(T) = H(T) - H(T|c)$ where $H(T)$ is the *entropy* of the node T, and $H(T|c)$ is the weighted sum of the entropies of its child nodes. Entropy $H(X)$ quantifies the uncertainty in the dataset and is calculated as $H(X) = -\sum_j p_j log_2(p_j)$ where p_j is the probability of class j. A split with high information gain effectively reduces the uncertainty in the data.

- **Gini index**: The Gini Index measures node impurity, evaluating how often a randomly chosen element would be incorrectly classified. It is computed as $I_G(T) = 1 - \sum_i p_i^2$ where p_i is the probability of class i at node T. A lower Gini Index indicates a purer node, meaning most data points belong to a single class.

For regression trees, the criteria for choosing splits are based on minimizing error:

- **Mean squared Error (MSE)**: MSE quantifies the difference between predicted and true values. It is calculated as $MSE = \frac{1}{N}\sum_i^N(\hat{y_i} - y_i)^2$ where N is the number of elements, $\hat{y_i}$ is the predicted value for element i and y_i the true value of element i.

- **Residual Sum of Squares (RSS)**: RSS measures the total squared difference between the actual values and the model predictions. It is computed as $\sum_p^P \sum_i^{N_p}(y_i - \hat{y_p})^2$ where P is the number of children resulting from the split, y_i it the ground truth of element i and $haty_p$ is the average value in each child node.

- **Variance**: Variance reduction measures how much splitting a node reduces the variability in the target variable. A successful split creates child nodes where the target values are more similar, indicating a more accurate prediction.

Additionally, decision trees include hyper-parameters that influence their structure and complexity, such as the maximum depth of the tree, the minimum number of samples required to split a node, and the minimum number of samples in a leaf. Proper tuning of these hyper-parameters is essential to balance model complexity and prevent overfitting or underfitting.

To prevent overfitting, decision trees often incorporate pruning techniques. Pruning reduces tree complexity by removing branches that do not provide significant predictive power. Pre-pruning stops the growth of the tree during training, whereas post-pruning removes branches after the tree is fully grown, often based on performance on a validation set.

Random Forest Random Forest is a robust machine-learning ensemble method known for its high accuracy and versatility. Unlike traditional ML approaches that rely on a single predictive model, ensemble methods combine the outputs of multiple sub-models to improve predictive performance. In a random forest, these sub-models are individual decision trees 2.2.1. The final prediction in a random forest is typically the majority vote for classification tasks or the average of predictions for regression tasks, effectively reducing variance and enhancing model stability.

First introduced by Breiman in 2001 [13], random forests work by constructing multiple decision trees, each trained on a different subset of the training data (a technique known as bootstrap sampling) and a random subset of features at each split. This approach results in an aggregate model that significantly improves prediction accuracy compared to individual trees, as each tree is less prone to overfitting due to its exposure to varied training data and feature sets.

A notable variant within ensemble learning is Tree Boosting [26], which uses an iterative approach to refine predictions. Unlike random forests, where trees are built independently, tree boosting sequentially builds trees, with each new tree learning from the errors of the previous ones. This process is guided by the gradient of a loss function, a measure of the difference between the predicted and true labels. At each iteration, the algorithm adds a tree that reduces the loss, thus enhancing prediction accuracy. Each tree's contribution is scaled by a learning rate hyper-parameter, which controls the model's sensitivity to new trees and helps manage overfitting.

A popular implementation of tree boosting is XGBoost [16], which efficiently builds shallow trees (typically one node and two leaves) to correct residuals from prior trees. XGBoost is recognized for its speed and performance, particularly on large datasets, due to optimized regularization and a scalable training algorithm.

2.2.2 Unsupervised Machine Learning

An unsupervised ML algorithm works with unlabelled data, where no predefined labels or outcomes are provided. The algorithm must discover patterns or structures in the data on its own.

K means Clustering The K-means algorithm [1] is one of the most popular and widely used unsupervised machine-learning techniques for clustering. It is based on the concept of *centroids*, which represent the centres of data clusters. The algorithm aims to partition the dataset into k clusters, each defined by a centroid, by iteratively refining the centroids to minimize within-cluster variance.

In each training iteration, the algorithm assigns each data point to the nearest centroid based on a distance function. After all points have been assigned, each centroid is recalculated as the mean of the points in its cluster, shifting toward the central position of its assigned points. This process repeats until the centroids stabilize, or until a set number of iterations is reached. The need for a distance function is a strong limitation of this algorithm, however, it can be neglected in most of the ML applications since they rely on a feature vector for which it is possible to use the Euclidean distance, Cosine similarity or Manhattan distance.

K-means can also be formulated to minimize the distortion function, which measures the total squared distance between each data point and its nearest centroid. Formally, this function can be written as:

$$D = \sum_{1}^{N} (x_i - \text{decode}(\text{encode}(x_i))^2$$

where the encode function maps each data point to its nearest centroid, and the decode function maps each centroid back to its vector representation. Minimizing the distortion function helps ensure that centroids represent their assigned points accurately. To optimize this function, K-means often relies on algorithms like gradient descent [73] or, in some cases, genetic algorithms [47].

A known limitation of K-means is its sensitivity to the initial placement of centroids. If a centroid is initialized far from any data point, it may result in an unpopulated cluster, reducing the algorithm's effectiveness. To address this, K-means algorithms may reinitialize any centroid without assigned points by positioning it near the cluster with the highest distortion, ensuring it participates in clustering.

Additionally, K-means++ initialization [38] has been developed to improve the placement of initial centroids. This technique places centroids iteratively with a probability proportional to the squared distance from existing centroids, increasing the likelihood of well-distributed initial centroids and enhancing convergence speed and clustering quality.

K-means clustering is best suited for datasets where clusters are spherical and well-separated, as it relies on the assumption that clusters are isotropic (having similar spread in all directions). This reliance on Euclidean distance as a similarity measure makes it less effective for identifying clusters with irregular shapes, varying densities, or overlapping regions. For example, in cases where clusters are elongated, concave, or intertwined, K-means may fail to distinguish between them and could assign points inaccurately. This limitation can result in suboptimal clustering performance, particularly for data with complex structures, where algorithms like DBSCAN [20] or spectral clustering [59] might provide better results. Recognizing these shape and density limitations is essential when considering K-means for any real-world application to ensure appropriate clustering outcomes.

The algorithm always converges to a local optimum and it is impossible to get stuck on a loop since the number of possible states is finite given a fixed amount of training points and a fixed amount of centroids.

Hierarchical Clustering Hierarchical clustering [67] is another prominent unsupervised machine learning algorithm that organizes data into a tree-like structure of nested clusters, known as a dendrogram. This method is based on the principle of connectivity, grouping data points based on their relative proximity.

There are two main approaches to hierarchical clustering: *agglomerative* and *divisive*. The agglomerative approach, also known as ”bottom-up” clustering, starts with each data point as its own cluster and successively merges the closest pairs of clusters based on a linkage criterion, such as single linkage (minimum distance), complete linkage (maximum distance), or average linkage. Conversely, divisive clustering, or ”top-down” clustering, begins with all data points in a single cluster and recursively splits clusters until each point forms its own cluster.

The resulting dendrogram provides a comprehensive visualization of the cluster hierarchy, allowing users to identify meaningful clusters by ”cutting” the tree at different levels. This flexibility enables the algorithm to handle datasets with complex cluster shapes and varying densities effectively. Moreover, the distance metric and linkage criterion can be customized to suit specific data characteristics, such as using Euclidean distance for continuous data or Jaccard similarity for binary data.

However, hierarchical clustering has notable limitations. Its computational complexity, typically $O(n^3)$, makes it less suitable for large datasets compared to other clustering algorithms. Additionally, once clusters are merged or split, the algorithm cannot revise these decisions, potentially leading to suboptimal clustering outcomes. Despite these challenges, hierarchical clustering remains a valuable tool for exploratory data analysis, particularly when the underlying structure of the data is unknown or complex.

2.3 Neural Networks

This section will discuss neural networks. Section 2.3.1 details the components of neural networks, then, Section 2.3.2 describes the principal steps of training a neural network. Section 2.3.3 makes an important distinction between feed-forward and recurrent models. Finally, 2.3.4 describes the Transformer architecture and its use.

A Neural Network (NN) or Deep Neural Network (DNN) is a specialized type of supervised ML algorithm grounded in the universal approximation theorem [55]. Structurally, a NN is composed of a sequence of layers, each followed by an activation function. Networks with more than two layers are classified as *deep* neural networks, whereas networks with two or fewer layers are called *shallow*. A visual representation of a NN can be seen in figure 2.1

A particular advantage of NNs with respect to other ML approaches is their ability to be able to work with raw data by automatically generating the needed features. This removes the complex process of feature engineering and extraction.

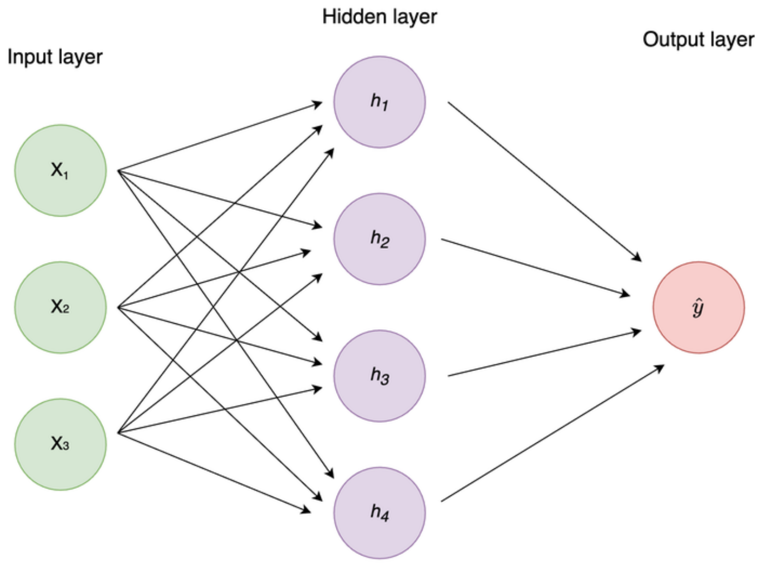


Figure 2.1: Visual representation of a neural network
Source:
https://lassehansen.me/post/neural-networks-step-by-step/

2.3.1 Neural Networks Architecture

The architecture of a NN can strongly impact its performance. Therefore, designing a model is a crucial part of the development of a NN. This section will cover the key components of each NN that shape its architecture.

Types of Layers There are multiple types of layers, each composed of a set of neurons which compute a linear function. The primary distinction between layer types is the linear function each neuron computes. Regardless of layer type, each layer has a set of weights, which are the actual parameters of the linear function. Some of the most common types of layers are:

- **Linear Layer**: In a linear layer, each neuron computes an affine function represented as:

  $$f(x_1, \dots, x_n) = w_1x_1 + \dots + w_nx_n + \beta.$$

  This type of layer is widely used due to its capacity to approximate diverse functions. The output of the entire layer can be obtained through matrix multiplication as follows

  $$I_{n,m} \cdot W_{k,n} + B,$$

where I is the input matrix with each row representing an n-dimensional input vector, W is the weight matrix where each column represents one of k neurons' affine functions (excluding bias), and B is the bias vector. Computing the output always in this way allowed the hardware maker to create specialized chips that made the computation extremely efficient.

- **Convolutional layer**: Convolutional layers are particularly effective for image-processing tasks. The convolution operation, represented here by $o_{i,j}$, is influenced by a central input value $i_{i,j}$ and its neighbouring values. This spatially-aware approach makes it well-suited for images, where neighbouring pixels often carry relevant contextual information. Convolutional layers share weights across neurons, making them computationally feasible even with high-dimensional inputs. The convolution operation is defined as:

$$(S * W)_{i,j} = \sum_{m=-\lfloor M/2 \rfloor}^{\lfloor M/2 \rfloor} \sum_{n=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} S_{i+m,j+n} \cdot W_{m,n},$$

  where S represents the input matrix, and W is the weight matrix of size $M \times N$.

- **Attention layer**: The attention layer is particularly prominent in Natural Language Processing due to its ability to handle long input sequences by selectively focusing on relevant parts of the data. This selective focus is achieved by weighting input features according to their interdependencies and relevance to the current processing task. The attention mechanism operates on three primary vectors:

  - **Query** (Q): Represents the current processing state.
  - **Key** (K): Encodes representations of all positions in the input sequence.
  - **Value** (V): Contains the actual information that needs to be aggregated.

  The attention scores, which determine the alignment between different parts of the input and the processing state, are computed by taking the dot product between the query and each key, followed by scaling and normalization into a probability distribution. The attention computation is given by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

  where softmax is the normalization function explained in the following paragraph, and d_k denotes the dimensionality of the key vector.

Activation Functions After each layer, usually, there is an activation function. The primary purpose of the activation function is to introduce non-linearity between layers, which enhances the network's capacity to generalize and approximate complex target functions using fewer parameters. Additionally, activation functions must be differentiable to enable gradient-based optimization techniques during training.

Some widely used activation functions include:

- **Sigmoid**: This function constrains the input to the range 0 to 1, calculated as:
$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$
Due to its output range, Sigmoid is often selected as the final activation in NNs, particularly when the output represents a probability.

- **ReLU** (Rectified Linear Unit): Defined as $\text{ReLU}(x) = \max(x, 0)$, this function is both simple and effective. Although ReLU is technically non-differentiable at $x = 0$, this issue is typically resolved by assigning its derivative at this point to 1. ReLU gained popularity through the work of Krizhevsky [46] as its straightforward formulation helps mitigate the vanishing gradient problem that affects traditional activation functions.

- **Tanh**: it is similar to the Sigmoid activation but squeezes the input between -1 and 1. Its formula is:
$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

- **Softmax**: Like Sigmoid, Softmax transforms the input values to fall within the range 0 to 1. However, unlike Sigmoid, Softmax considers the entire set of inputs, ensuring that the sum of the output values is 1. This property makes Softmax particularly useful for representing categorical distributions. The Softmax function is defined as:
$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$
Similar to Sigmoid, Softmax is frequently used as the final activation layer, especially in classification tasks where the output can be interpreted as a probability distribution.

2.3.2 Training a Neural Network

Training an NN involves optimizing the weights of its layers to approximate a target function as accurately as possible.

Each NN can be conceptualized as a composition of simpler, differentiable functions. This compositional structure allows us to apply the chain rule to compute the partial derivatives of the loss with respect to each layer's input and parameters. These partial derivatives, which represent the gradients of the loss function with respect to the weights, are then used to update the weights during training.

To formally express this, let us denote a NN as a sequence of functions $f_1, f_2, \ldots, f_n$, where each function f_i represents the transformations applied in a given layer. Given an input x, the output of the network $f(x)$ can be written as:

$$f(x) = f_n(f_{n-1}(\ldots f_2(f_1(x))\ldots)).$$

Since each function f_i is differentiable, we can apply the chain rule to compute the gradient of the loss L with respect to any intermediate variable z_i at layer i. Specifically, if $z_i = f_i(z_{i-1})$, then the gradient of the loss with respect to z_i can be written as:

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial z_n} \prod_{j=i+1}^{n} \frac{\partial f_j}{\partial z_{j-1}}.$$

This recursive application of the chain rule is known as backpropagation, and it enables the efficient computation of gradients throughout the network.

To adjust the weight, the gradient descent algorithm is employed, with backpropagation serving as its computational backbone. Gradient descent leverages the gradient of the loss function, which indicates the direction and rate of steepest increase, to adjust weights in the opposite direction—toward a local minimum. This iterative process helps to reduce the error over time by taking steps proportional to the negative gradient of the loss with respect to each weight. The gradient descent update rule is generally expressed as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \nabla L(NN(x, \Theta_t), y).$$

Where Θ are the NN's weights, L is a special function called loss function that we want to minimize, x is the input, y is the target output and α is a hyper-parameter called learning rate used to set the amount of tweaking done to the parameters.

Loss functions The loss function, or simply "loss", is a crucial component of the NN training process. The loss quantifies the discrepancy between the target results (ground truth) and the predictions produced by the model. By minimizing the loss through backpropagation, the network effectively reduces incorrect predictions and improves its ability to approximate the target function accurately. Selecting an appropriate loss function is crucial, as each type is tailored to specific tasks and data characteristics.

Some commonly used loss functions include:

- **Cross-Entropy Loss**: Widely used for multi-class classification tasks, Cross-Entropy loss is well-suited for scenarios where each instance belongs to one class out of multiple possible categories. Often combined with the Softmax activation function in the output layer, Cross-Entropy measures the difference between the true label distribution y and the predicted distribution $\hat{y}$. It penalizes misclassified predictions more heavily, leading to faster convergence in classification tasks. The formula for Cross-Entropy loss is:

$$\text{CrossEntropy}(y, \hat{y}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i).$$

- **Binary Cross-Entropy Loss**: A specialized variant of Cross-Entropy loss, Binary Cross-Entropy is typically used for binary classification tasks (i.e., two classes) or for multilabel classification problems where each instance may belong to multiple classes. It is particularly effective when paired with the Sigmoid activation function in the output layer, as it helps produce outputs in the range $[0, 1]$, interpretable as probabilities. This loss function measures the error between the binary ground truth y and the predicted probability $\hat{y}$, with the following formula:

$$\text{BinaryCrossEntropy}(y, \hat{y}) = -\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right).$$

- **Mean Squared Error (MSE) Loss**: Also known as squared loss, MSE is commonly used for regression tasks, where predictions are continuous values. MSE computes the average of the squared differences between the target values y and the predicted values $\hat{y}$, effectively penalizing larger errors more than smaller ones. While useful, MSE can be sensitive to data with large variance, leading to potentially high loss values if the target values vary significantly. The MSE loss is defined as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

Optimizer Training a NN requires a systematic approach to update its weights, reducing the error between predictions and actual values as calculated by the loss function. However, computing the loss on the entire dataset at once is often infeasible, both in terms of computational resources and memory requirements.

For this reason, *Stochastic Gradient Descent* (SGD) is commonly employed as a practical solution.

Stochastic Gradient Descent (SGD) is an iterative optimization technique that updates the model weights by calculating the gradient of the loss function for a small subset of the training data, known as a *batch*. Each batch is used to compute an estimate of the gradient, and weights are updated accordingly. By iterating through all batches in the dataset, the model completes one *epoch*.

Many variations of SGD exist to improve convergence. One of the most famous is *Adam*, or Adaptive Moment Estimation [42], which combines the advantages of both Momentum [53] and RMSprop [71]. Momentum adds a sort of *"inertia"* to the gradient making it faster at first and slower at later epochs. RMSprop, instead, scales the gradient based on the frequency of its component making the updates larger for infrequent features and smaller for frequent ones. This dual approach is achieved by maintaining both an exponentially weighted average of past gradients (first moment) and squared gradients (second moment). This dual adaptation enables Adam to adjust the learning rate for each parameter while leveraging momentum to accelerate convergence. The Adam update rules are given by:

$$m^{(t+1)} = \beta_1 m^{(t)} + (1-\beta_1)\nabla L(w),$$

$$v^{(t+1)} = \beta_2 v^{(t)} + (1-\beta_2)\left(\nabla L(w)\right)^2,$$

$$\hat{m}^{(t+1)} = \frac{m^{(t+1)}}{1-\beta_1^t}, \quad \hat{v}^{(t+1)} = \frac{v^{(t+1)}}{1-\beta_2^t},$$

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t+1)}+\epsilon}}\hat{m}^{(t+1)},$$

where m and v are the first and second-moment estimates, and β_1, β_2 are decay rates, commonly set to 0.9 and 0.999, respectively. The adjustments $\hat{m}$ and $\hat{v}$ correct for bias in the initial estimates, enhancing stability. Adam's adaptive learning rates, combined with momentum, make it particularly effective for deep networks and models with large parameter spaces.

2.3.3 Feed-Forward and Recurrent Neural Networks

NN architectures can be categorized in several ways, but a foundational distinction exists between *feed-forward* and *recurrent* neural networks. This classification is based on how data flows through the network and how the network handles sequential dependencies within input data.

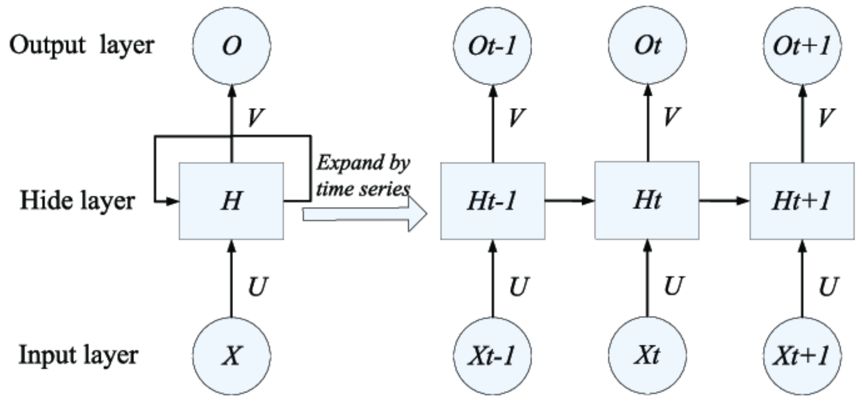


Figure 2.2: Visualization of a Recurrent Neural Network model
Source: `https://www.researchgate.net/figure/llustration-of-the-recurrent-neural-network-RNN-structure_fig2_346468428`

Feed-Forward Neural Networks (FNN) are the simplest form of neural network architecture. In these networks, information flows in one direction only—from input nodes, through hidden layers, to output nodes in one step. These kinds of networks are well-suited for any task with a fixed input size. FNNs are highly parallelizable both during training and at test time.

Recurrent Neural Networks (RNNs) are specifically designed to handle sequential data by introducing connections that form directed cycles within the network. These cycles enable the network to maintain a form of memory or *state*, which is updated at each time step. This internal state allows RNNs to process arbitrary-length sequences by retaining information about previous inputs in the sequence, making them well-suited for tasks involving inherently sequential data, such as natural language processing, time-series forecasting, and speech recognition. A visual representation of an RNN architecture can be seen in figure 2.2 Despite their utility in handling sequential data, recurrent NNs face two significant challenges:

1. **Limited Parallelization**: Training RNNs is inherently sequential, as each time step depends on the state from the previous steps. This dependency restricts the possibility of parallelizing the training process across different time steps, making RNN training significantly slower compared to feed-forward architectures.

2. **Vanishing Gradient Problem**: The gradient at any given time step t in an RNN depends on the gradients of all previous time steps. Consequently,

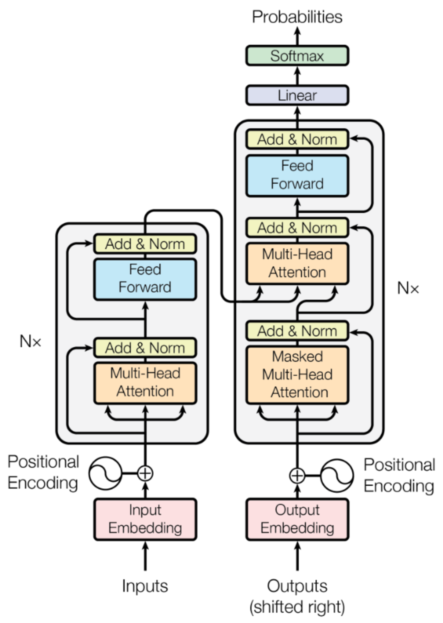


Figure 2.3: Visualization of a transformer model
Source: Attention is all you need, `https://arxiv.org/pdf/1706.03762`

when backpropagating through many time steps, gradients can diminish exponentially, especially for long sequences. This issue, known as the *vanishing gradient* problem, can prevent effective weight updates, particularly for earlier time steps in a sequence. As a result, RNNs may struggle to retain information over long sequences, making them less effective for tasks that require long-term dependencies.

2.3.4 The Transformer Architecture

The Transformer [77] architecture represents a major breakthrough in the NN world making a difference in all the related fields. The transformer was exceptionally revolutionary in the field of Natural Language Processing, fundamentally shifting how language models are designed. Unlike RNNs, which process sequences step-by-step, the Transformer uses an attention-based mechanism to process entire sequences simultaneously, enabling efficient parallel computation. Not only that. the attention mechanism allows each input token to have a dedicated state vector that could selectively choose what to focus on in the current sentence removing the limitation of a shared state that had to encode the whole semantic meaning. A visualization of the Transformer architecture can be seen in figure 2.3

The Transformer is based on an encoder-decoder structure, where both the encoder and decoder consist of multiple layers of self-attention and FNNs:

- **Self-Attention Mechanism**: At the core of the Transformer is the self-attention mechanism, which allows each position in the input sequence to focus on (or "attend to") other positions. This capability is crucial for capturing context over long distances in a sentence. Given an input sequence, the self-attention mechanism calculates three vectors—*query*, *key*, and *value*—for each word and computes a weighted sum based on their dot product.

- **Multi-Head Attention**: To capture different types of relationships between words, the Transformer uses multiple self-attention heads, which allows the model to focus on various parts of the sequence simultaneously. Each head computes its attention separately, and the outputs are concatenated and linearly transformed to form the final attention representation.

- **Position-wise Feed-Forward Networks**: After the attention layers, the output is passed through a FNN. This step applies a non-linearity and further refines the representation learned by the attention layers.

- **Positional Encoding**: Since the Transformer processes sequences in parallel, it lacks the inherent ordering found in RNNs. To incorporate information about the order of tokens, positional encodings are added to the input embeddings. These encodings are computed using sinusoidal functions that allow the model to differentiate between the positions of words in a sequence.

Variants and Extensions of the Transformer Since its introduction, several variants and extensions of the Transformer have been proposed, each designed to enhance its performance for specific tasks:

- **BERT (Bidirectional Encoder Representations from Transformers)** [22]: uses only the encoder part of the Transformer and trains it in a bidirectional manner. BERT has shown state-of-the-art results on tasks like question answering, sentiment analysis, and named entity recognition. Being the encoder part of the transformer architecture, BERT-like models are extremely good at encoding the semantics of the input and, thus, are especially good for classification problems.

- **GPT (Generative Pre-trained Transformer)** [64]: uses the decoder portion of the Transformer in an autoregressive fashion for text generation. GPT models have achieved remarkable success in tasks such as text completion, summarization, and dialogue generation. Using only the decoder part of the transformer, allows the model to become exceptionally good with tasks that do not involve a lot of context.

2.4 Algorithm Selection

This section details the field of Algorithm Selection (AS), starting from section 2.4.1 with some theoretical foundation and different approaches to AS. Section 2.4.2 describes some application of AS to Combinatorial optimization problems and the evaluation metrics used to develop the algorithms.

2.4.1 Foundation

AS is the process of identifying the most appropriate algorithm for solving a given problem or set of problem instances, with the objective of optimizing performance according to a given metric such as accuracy, computational efficiency, or robustness. This section explores the core aspects of algorithm selection, including the theoretical foundations and methods for selection.

AS stems from the no free lunch theorem [78], which posits that no single algorithm outperforms all others across all possible problems. This theorem suggests that the selection of an algorithm should be tailored to the characteristics of the problem at hand. As a result, the algorithm selection problem can itself be framed as an ML task, where we seek to learn a mapping from problem features to the most suitable algorithm.

Rice introduced the foundational framework for algorithm selection [66], who proposed a four-component model:

1. **Problem Space**: The space of instances P that the algorithm aims to solve, each instance being characterized by a set of features F describing the problem instance.

2. **Algorithm Space**: The set of candidate algorithms A that can be applied to the problem space.

3. **Performance Space**: A set of performance metrics M (e.g., runtime, solution quality) which quantify the effectiveness of applying an algorithm to a given problem instance.

4. **Selection Mapping**: A mapping $S : P \to A$ that selects an algorithm $a \in A$ for a problem $p \in P$ based on maximizing performance metrics in M.

Several methods have been developed to address algorithm selection, leveraging statistical, ML, and meta-heuristic approaches. Broadly, these methods can be divided into two categories: *deterministic selection* and *predictive selection.*

Deterministic Selection Methods Deterministic methods rely on fixed rules or heuristics for algorithm selection, often informed by domain knowledge. These methods are simple and interpretable but may lack flexibility. Deterministic methods include: (i) **Rule-Based Selection**: Uses predefined rules to map problem features to algorithms. (ii) **Hierarchical Decision Trees**: Use a series of decisions based on instance features to narrow down the choice of algorithms.

Predictive Selection Methods Predictive methods employ ML models trained on past performance data to predict the best algorithm for new problem instances. They can leverage any kind of ML algorithm but usually, the AS problem is intended as a classification problem that selects the appropriate algorithm given a problem instance or a regression problem where, for each algorithm, the ML model tries to predict the future performance of such algorithm on the given problem instance. Other possible approaches also leverage clustering techniques to cluster together problem instances with similar characteristics that make them suitable to be solved with an algorithm.

2.4.2 Applications in Combinatorial Optimization

There have been multiple applications of algorithm selection in the field of combinatorial optimization. The most popular is probably SATzilla [80] which won multiple competitions as the best SAT solver. This tool extracts some key properties of the instance such as the number of variables or connectivity of the graph and then uses a regression model to predict the performance of each solver in its portfolio. Afterwards, SATzilla chooses the solver with the best-predicted performance to solve the instance.

In the Constraint programming world, a notable algorithm selection tool is Sunny CP [52]. This tool extracts a set of features from the instance and uses them to compute a similarity measure with previously seen instances. It then selects the algorithm or portfolio of algorithms to use in a scheduled manner. If the system is allowed to run multiple solvers concurrently, it also implements information sharing between solvers to guide the search. The tool can also be updated by constantly adding the newly solved instance to the historical data.

Furthermore, Autofolio [51] is a general algorithm selection approach that can work for any combinatorial optimization problem due to its generality. It takes as input, for each instance, a feature vector and a score associated with each algorithm in the portfolio. Then the model is trained to maximise/minimize the given score. It is also possible to add the time necessary to compute the features, this way, the model can decide when computing the features could improve performance or if it would be enough to just choose the single best option of the portfolio.

[44] presents a survey on general AAS techniques and how they can tackle combinatorial problems resulting in better performances. It shows different techniques that can be used to optimize the final runtime such as directly selecting one algorithm or predicting a schedule of algorithms to run for a finite amount of time.

The effectiveness of algorithm selection strategies is measured by evaluating improvements in performance metrics, often through cross-validation on benchmark datasets. In particular, the best overall algorithm (Single Best) is selected as a baseline and the most optimal combination of all algorithms (Virtual best) is selected as a lower bound. While the goal is to get as close as possible to the virtual best, the overall performances are measured as improvements over the single best.

For robust evaluation, a standard approach involves using benchmark problem sets or cross-validation to assess generalizability. Additionally, *ablation studies* are often conducted to evaluate the impact of individual instance features and model components on algorithm selection accuracy.

2.5 Related Work

In examining the field of combinatorial optimization and algorithm selection, this project stands out due to its focus on feature extraction rather than introducing new selection methods. Most related work presents innovative approaches to algorithm selection, where feature extraction is only one component of a broader pipeline. In this project, however, feature extraction is the primary focus. In a similar way, NNs have previously been applied to combinatorial optimization tasks, but generally with goals that differ from those addressed here.

This section reviews the state of the art in feature extraction for combinatorial optimization problems, in Section 2.5.1, and then explores the application of NNs in combinatorial optimization research in Section 2.5.2.

2.5.1 Feature Extraction for Algorithm Selection in Combinatorial Optimization

Feature extraction processes for algorithm selection across combinatorial optimization applications often follow a consistent framework. Initially, low-level features are extracted automatically from the instance, including attributes such as the maximum constraint arity, average and maximum domain size, and counts of variables and constraints. These features can often be gathered by analysing the instance's structure and properties alone. Following this, a brief preliminary run of the instance on a solver is performed to gather additional information, such as the solver's choice patterns and search statistics. All the tools that will be presented will share this approach.

Notable algorithm selection tools especially tailored at constraint programming include SUNNY [52] and CPHydra [15], which employ a k-Nearest Neighbors approach to create a solver schedule aimed at maximizing the probability of solving a given instance within a specified time constraint. In contrast, Proteus [32] uses a hierarchical, portfolio-based approach that could solve the combinatorial problem using both CP and SAT. Tools initially developed for SAT problems can often be adapted to other paradigms, and vice versa. For example, [6] and [8] present empirical comparisons of SUNNY with other algorithm selection tools originally developed for SAT scenarios, such as 3S [35] and SATzilla [80]. In general, it is not hard to translate a combinatorial problem instance written with a specific paradigm in mind to any other paradigm and, as such, it is easy to adapt any AAS algorithm to work with all the combinatorial optimization solving strategies.

2.5.2 Neural Networks in Combinatorial Optimization

There are several uses for NNs in the Combinatorial Optimization space. Here, we address the three possible directions one could take to intersect the two fields. In particular, we will focus on assisted modelling via Language Models, using NNs to improve the search process of a solver and using NNs for AS within the Combinatorial Optimization world.

The modelling phase of Combinatorial problems can be assisted by Language Models to make the process easier and faster. One line of research focuses on using language models to generate CP models from natural language descriptions of problems, as seen in works like [74] and [5]. Chatbots Like ChatGPT [79] can be used to automatically model a problem starting from natural language even without any specific training on the subject as demonstrated, for example, by [57].

Including an NN in the search process of a solver can enhance the search. Some studies have demonstrated the use of NNs for feature extraction from search algorithm trajectories, which can then assist in heuristic algorithm selection for specific problem domains such as, for example, bin packing [4]. [56], Instead, offers an overview on how to apply ML algorithms, including NN, into the search process of a solver. Other studies have focussed on the use of NNs to directly generate a solution for a given instance problem. For example, [10] uses reinforcement learning techniques to generate a solution from an instance. This can be extremely helpful to find an initial solution to feed to a solver and improve upon as done by [25] who proposes a mixed approach to solve scheduling jobs by leveraging graph NNs to find initial solutions that are then refined via a CP solver. This dual method allows them to leverage the speed of NNs while also solving the limitations of their stochastic approach by using CP to prove correctness and optimality.

NNs in the field of AS have been applied in a similar manner to our intents, notably, transformers have been successfully applied to specific combinatorial problems for feature learning; for instance, [68] demonstrates how transformer architectures can effectively learn instance features for the Traveling Salesman Problem. The current work differs in that it focuses on general instance feature extraction across a broad range of Combinatorial Optimization problems, specifically for models expressed in the Essence language, broadening the potential application to any instance encoded within this specification framework. Furthermore, our approach, while tailored to Essence instances, is completely agnostic to the underlying algorithms making it suitable for all possible solvers without focusing on a specific solving paradigm.

Chapter 3

Methodology

This section outlines the methodology utilized in this work, detailing the processes of feature learning, extraction, and application. Section 3.1 examines the feature learning procedures and the diverse strategies applied, while Section 3.2 introduces different AS methods developed to exploit these features effectively.

Recall that, for the purposes of this study, an algorithm is defined as a pairing of an Essence Prime model with a solver. Using a set of solvers alongside an Essence model, we employ Conjure to create a portfolio of algorithms available for each instance. The objective of an AS task is to select the optimal algorithm for a given instance from this portfolio. While there are various definitions of "optimal", we define it here as the algorithm that yields the shortest runtime. Addressing the task through machine learning involves two key steps: (i) defining a set of features that accurately characterize an instance, and (ii) using these features to predict the best-performing algorithm.

In practice, we often impose a cut-off time on every algorithm. When an algorithm fails to solve an instance within the cut-off, we penalise the run using the Penalised Average Runtime (PAR) method [51], where the runtime of a failed run is recorded as k times the cut-off time. Following the existing AS literature [49], we set $k = 10$.

For the first step, we employed an NN to identify the features that best represent each instance. The input to this model is the raw text of an Essence instance. Transformer Encoders are particularly well-suited for this purpose, as they efficiently capture complex linguistic correlations. Using a transformer encoder like BERT [22] has multiple advantages: BERT-like architectures are known for their ability to capture high-level features and relationships, enabling automated feature learning without requiring domain-specific, handcrafted inputs.

For the second step, there are two primary approaches: integrating feature learning and algorithm selection within a single neural model or utilizing an ex-

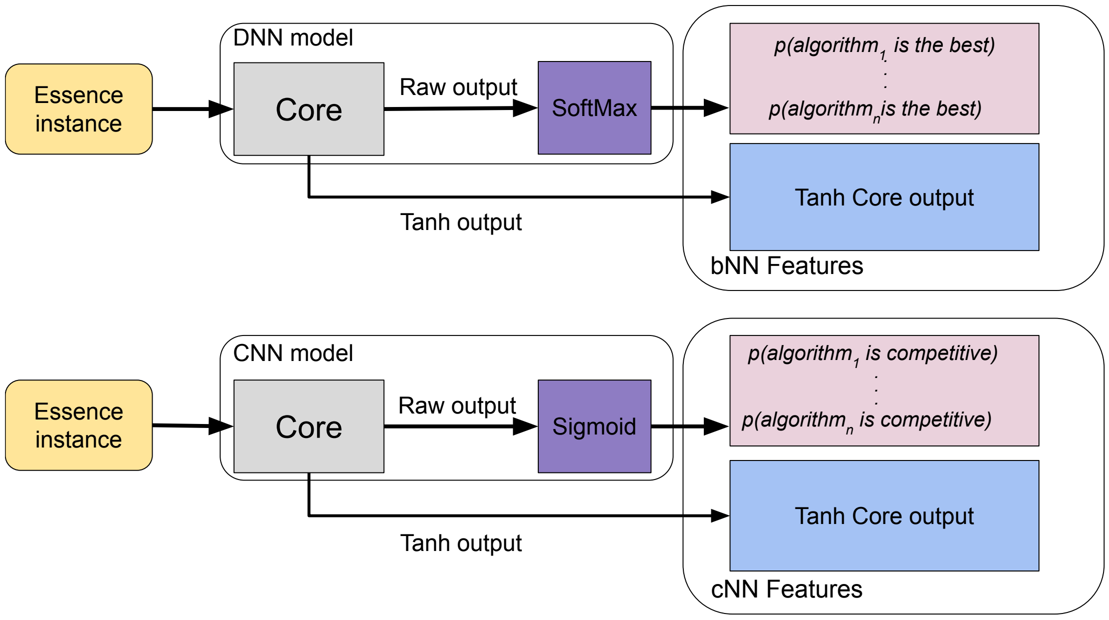


Figure 3.1: Different ways of performing AS starting from the core architecture.

ternal ML algorithm to perform AS on the learned features. While the first approach seems natural and more straightforward, the latter approach allows us to incorporate state-of-the-art AS tools from existing literature and experiment with alternative AS strategies.

3.1 Feature Learning Using a Transformer Encoder

To address the first point, we propose employing an NN that encapsulates a transformer encoder to deal with textual input. This approach has many advantages. First, transformer encoders such as BERT have been proven to be effective in capturing high-level language features [22], eliminating the need to run a solver to extract the necessary features. Second, an NN model can automatically generate the necessary features starting from the raw input. This eliminates the need for handcrafting an effective feature set. We build two NN models, called BNN and CNN, both of which receive as input the raw text of the Essence instance in tokenized form (where each input word and symbol are transformed into a number).

BNN Model This model learns the best algorithm for a given instance. The learning is modelled as a multi-label, single-class classification task where the as-

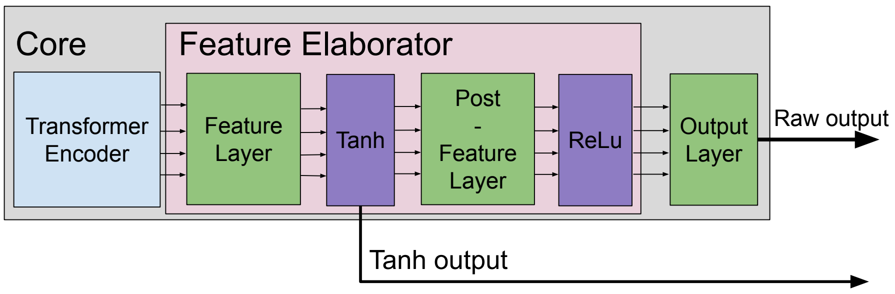


Figure 3.2: The two neural network architectures' common part (core).

signed class is the best algorithm. The final activation function is SoftMax, which generates a probability distribution over all the algorithms in the portfolio, where a higher probability indicates a higher likelihood to be the best. The SoftMax activation is well-suited for this purpose since the best algorithm is just one and we would like the highest probability to be associated with that particular algorithm.

CNN Model Learning one algorithm as the best may be restrictive and error-prone when multiple algorithms exhibit similarly good performance. The CNN model thus learns the *competitivenes* of the algorithms in the portfolio. We consider an algorithm to be competitive if it solves an instance in less than ten seconds or in less than double the time taken by the best algorithm for that instance. Multiple algorithms can be marked as competitive for a given instance. For example, if the best algorithm takes 15 seconds, any algorithm that completes the task in under 30 seconds is deemed competitive.

The learning is modelled as a multi-label, multi-class classification task where each algorithm is associated with a competitiveness fraction. The final activation function Sigmoid, which yields a probability for each algorithm, is well-suited for this purpose. While we still want a probability value as in the BNN model, there could be multiple equally competitive algorithms and the competitiveness probability of one algorithm is uncorrelated with that of the other algorithms.

As shown in Figure 3.1, the two models share the same architecture with the only difference being in their final activation function. The probability values in the output of the models will be part of the extracted features for AS, as we detail in Section 3.2. The common part of the models (referred to as Core) is composed of three components: (i) Transformer Encoder, (ii) Feature Elaborator, and (iii) Output Layer. The core architecture can be seen in Figure 3.2 and, more in details, it is composed of:

1. **Transformer Encoder**: Based on the BERT-base-uncased architecture but, unlike the base BERT model, it accepts inputs up to 2048 tokens, allow-

ing it to process extensive ESSENCE instances with numerous parameters. The BERT transformer architecture has been successfully applied to text classification tasks before [63].

2. **Feature Elaborator** This component is further divided in four sub-components with the purpose of projecting the output features of the Encoder into the desired dimensions. The sub-components are applied in order and are:

   (a) **Feature Layer**: A linear layer that condenses the output of the transformer to a smaller feature vector, mitigating the risk of dimensionality issues in subsequent ML applications [18].

   (b) **Tanh Activation**: The output is passed through a Tanh activation to limit values between -1 and 1. Its output will be part of the extracted features for AS, as we detail in Section 3.2.

   (c) **Post-Features Layer**: A linear layer up-projecting the feature vector. This allows the NN to further elaborate the features and avoid underfitting [65].

   (d) **Relu Activation**: Non-linearity is introduced through a ReLU activation function applied to the post-feature layer's output.

3. **Output Layer**: The final common linear layer, assigns a value to each algorithm in the portfolio, which will later be transformed into probabilities.

3.2 Algorithm Selection Using the Learnt Features

To address the second point of our ML-based AS, which is using the extracted features to predict the best algorithm for an instance, we propose two approaches: integrating feature learning and AS within a single NN model (referred to as *fully neural*), or exploiting an external ML-based AS algorithm using the extracted features (referred to as *hybrid*). While the first approach seems natural and more straightforward, the latter allows exploiting state-of-the-art AS tools and experimenting with alternative AS strategies.

Once the NN models are trained, we can use the BNN model alone as a fully neural approach to predict the best algorithm. To do so, we simply use the probability values in the output as features and pick the algorithm associated with the highest probability. For the hybrid approach, we can extract features from each NN model by combining the probability values in the output with the output of Tanh in a single feature vector. This combined vector integrates the encoder-derived semantic representation with informative prediction. We refer to such combined feature vectors as bNN (when extracted from the BNN models) and cNN (when extracted from the CNN models) features, as depicted in Figure 3.1.

Subsequently, any ML-based AS algorithm can utilize the extracted features. Here, we explore two approaches: a state-of-the-art AS model from literature and a simpler alternative inspired by AS principles.

Autofolio[51] is a state-of-the-art AS algorithm able to perform both classification and regression tasks. It is possible to tune it using SMAC[50]. The base model is a standard random forest model, however, the tuning process can choose between random forest for classification, random forest for regression and XGboost[16]. The ASs that use Autofolio will be denoted by A.

As an alternative AS algorithm, we opted to use K-means clustering to cluster the instances based on their similarities and assign a different algorithm to each cluster. To optimize the K-means clusters, we pre-define a set of hyper-parameter configurations to configure the K-means algorithm. After clustering, each cluster is assigned the algorithm with the lowest PAR10 score within that cluster's instances. This assignment is validated on evaluation data to select the configuration with the lowest cumulative PAR10 score. The K-means approach draws on the proven effectiveness of clustering for AS[9] and the spatial characteristics of the transformer encoder embeddings[28], where semantically similar instances are represented by similar vectors. At test time, an unseen instance is clustered according to the best-fit configuration, with the best algorithm for that cluster deemed as the best. The ASs that use the K-means-based approach will be denoted by K.

In summary, we have five ML-based AS approaches: fully neural `BNN` and four hybrids `(bNN,A)`, `(bNN,K)`, `(cNN,A)`, `(cNN,K)` (`A`, `K` denote Autofolio and K-means).

Chapter 4

Datasets

This section provides an in-depth exploration of the datasets used to train the model. The discussion begins with Section 4.1, which details the problem classes, their corresponding instances, and the methods used to gather these datasets. Section 4.2 examines the extraction and differentiation of models and solvers employed across the datasets. Finally, Section 4.3 discusses the complementary nature of the algorithms included in the datasets.

It is noteworthy that all datasets utilized in this study are publicly accessible on GitHub[1].

4.1 Problem Description and Instance Set

This section outlines the problem classes represented in our datasets and describes how they are formalized using the ESSENCE modelling language. All three problems—Car sequencing, Covering array, and Social golfers—are known to be computationally challenging, making them valuable case studies for algorithm selection. They also cover a broad spectrum of ESSENCE features (such as matrix, function, and relation variables, plus unnamed types) and vary in dataset size: Car sequencing includes a large number of parameters, whereas Covering array and Social golfers have only integer parameters. Moreover, as we show in Section 4.3 the different solver/model combinations exhibit complementary performance on these problems, further motivating their use as benchmarks.

For all three problems, we use the instances from [69], generated using the AutoIG framework [19]. AutoIG systematically varies problem parameters to produce a diverse and challenging set of instances. These instances we use are publicly available in the ESSENCE Catalogue [21].

[1]`https://github.com/SeppiaBrilla/EFE_project/tree/master/data/datasets`

```
given n_cars, n_classes, n_options : int(1..)
letting Slots  be domain int(1..n_cars),
        Class  be domain int(1..n_classes),
        Option be domain int(1..n_options)
given quantity : function (total) Class  --> int(1..),
      maxcars  : function (total) Option --> int(1..),
      blksize  : function (total) Option --> int(1..),
      usage    : relation of ( Class * Option )
find car : function (total) Slots --> Class
such that forAll c : Class . |preImage(car,c)| = quantity(c)
such that forAll opt : Option .
            forAll s : int(1..n_cars+1-blksize(opt)) .
              (sum i : int(s..s+blksize(opt)-1) .
                toInt(usage(car(i),opt))) <= maxcars(opt)
```

Figure 4.1: Essence model of the car sequencing problem.

4.1.1 Car Sequencing

The car sequencing problem involves sequencing a series of cars for production, where each car may require different optional features. The production line is structured into stations, each responsible for installing specific options, such as air conditioning or sunroofs. Each station has a limited capacity, processing only a fixed percentage of cars. To ensure an even workload distribution and prevent bottlenecks, cars requiring the same option must be distributed evenly along the sequence. For instance, if a station can handle a maximum of 50% of cars, the sequence must limit cars requiring the corresponding option to at most one in every two.

An Essence model is used to formally define the problem (Figure 4.1). This model defines three integer parameters: *n_cars*, *n_classes*, and *n_options*, representing the number of cars, classes, and options, respectively. From these, the domains *Slots*, *Class*, and *Option* are derived to define further parameters and decision variables. Three parameters with function domains are defined to represent the *quantity* of each class of car required, a maximum number of cars (*maxcars*) that can appear in any block of cars, and block size (*blksize*) for each option. The *usage* parameter is a relation that indicates which classes use which options.

The only decision variable (*car*) in the model is a mapping from car production slots to classes. The problem constraints are captured in two top-level constraints (denoted by the keywords *such that*). The first set of constraints ensures that the number of cars in each class matches the required quantity. The second set of constraints ensures that for each option, in any block of *blksize(opt)* consecutive cars, the number of cars requiring that option does not exceed *maxcars(opt)*.

The dataset consists of 10,214 instances.

```
given t : int(1..)
given k : int(1..)
given g : int(2..)
given b : int(1..)
where k>=t, b>=g**t
find CA: matrix indexed by [int(1..k), int(1..b)] of int(1..g)
such that
    forAll rows : sequence (size t) of int(1..k) .
        (forAll i : int(2..t) . rows(i-1) < rows(i)) ->
        forAll values : sequence (size t) of int(1..g) .
            exists column : int(1..b) .
                forAll i : int(1..t) .
                    CA[rows(i), column] = values(i)
such that forAll i : int(2..k) . CA[i-1,..] <=lex CA[i,..]
such that forAll i : int(2..b) . CA[..,i-1] <=lex CA[..,i]

```

Figure 4.2: ESSENCE model of the covering array problem.

4.1.2 Covering Array

The covering array problem is a well-known NP-complete problem [37], originating from applications in hardware design. A covering array $CA(t, k, g)$ of size b and strength t is defined as an array A of dimensions $k \times b$, whose elements are drawn from the set $Z_g = \{0, 1, 2, \ldots, g-1\}$. This array satisfies the following property: for any t distinct rows $r_1, r_2, \ldots, r_t$ $(1 \leq r_1 < r_2 < \cdots < r_t \leq k)$ and any tuple $(x_1, x_2, \ldots, x_t) \in Z_g^t$, there exists at least one column c such that $A[r_i, c] = x_i$ for all $1 \leq i \leq t$.

The smallest number of columns b for which such a $CA(t, k, g)$ exists is referred to as the covering array number, $CAN(t, k, g)$.

This research focuses on the decision version of the covering array problem, which determines whether a covering array $CA(t, k, g)$ can be constructed for a given number of columns b. The model used in this study is presented in Figure 4.2.

The input parameters for the model are: t, the strength, k, the rows, g the array's values domain, b the number of columns of the covering array. All the parameters are in the integer domain therefore, no new domain is needed.

The only decision variable is CA: an integer-indexed matrix of integer values in the range 1 to g. The matrix has size $k \times b$. The matrix is constrained by three high-level constraints that capture the problem specifications. The first constraint is enough to model the entire problem since it makes sure that, for every t distinct rows, for all integer values up to g, exists a column where the value indexed by the *i-th* row and that column has the *i-th* value. The last two constraints are symmetry-breaking constraints that enforce a lexicographic ordering of rows and columns.

The dataset used for this problem consists of 2,236 instances.

```
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched : set (size w) of
                partition (regular, numParts g, partSize s) from Golfers
such that
    forAll g1, g2 : Golfers, g1 != g2 .
        (sum week in sched . toInt(together({g1, g2}, week))) <= 1

```

Figure 4.3: Essence model of the social golfers' problem.

4.1.3 Social Golfers

The social golfers problem [72] was derived from a post by bigwind777@aol.com (Bigwind777) in 1998 on *sci.op-research*. The original problem required scheduling a set of golf games between 32 golfers each of whom plays once a week in groups of 4. The objective is to maximise the number of weeks so that no golfer plays in the same group as any other workers on more than one occasion. There are several possible variations such as to maximise the "socialization" (as few repeated pairs as possible) over a 10-week schedule or finding a schedule of minimum length such that each golfer plays with every other golfer at least once. The last variation is called "full socialization". This problem can be then easily generalized to m groups and n golfers over p weeks such that no golfer plays any other golfers twice.

The model employed for this research is shown in figure 4.3 and solves the decision version of the problem. In this version of the problem, we want to find a schedule over w weeks for g groups of s golfers each. Each input parameter has an integer domain. The schedule (*sched*), the only decision variable, is represented as a set of partitions of size w. Each partition has g elements. Each element is composed of s *Golfers*. *Golfers* is a new type declared for the model which represents all the golfers in the schedule and has size $g \times s$. The only constraint set on the *sched* variable, imposes that every pair of golfers meet at most one time for the full duration of the schedule.

We use a dataset of 1,039 instances for this problem.

4.2 Combinations of Models and Solvers

In this section, we will establish the process we used to obtain the different algorithms and the differences of the ESSENCE PRIME models generated for each problem class. Our portfolio of algorithms is composed of every possible combination of the model generated using CONJURE on the ESSENCE model and four solvers. The solvers are Kissat, Chuffed, CPLEX, and OR-Tools CP-SAT, each chosen for their potential complementary characteristics in combinatorial optimization.

Kissat [12] is a modern clause-learning Satisfiability (SAT) solver. Chuffed [17] is a Constraint Programming (CP) solver enhanced with clause learning. CPLEX [34] is a commercial Mixed-Integer Programming (MIP) solver that excels in solving problems that heavily use arithmetic constraints. OR-Tools CP-SAT [2] is a hybrid solver developed by Google that integrates clause learning, CP-style constraint propagation, and MIP-solving methods. We use Savile Row [60] to target these solvers. The Essence Prime models are obtained using Conjure in its portfolio mode, with variations arising from different representations of the variables and constraints formulations.

4.2.1 Car Sequencing

For the car sequencing problem, multiple possible translations arise from the representation of the *car* decision variable and the *usage* parameter, as well as the way problem constraints are formulated. The *car* decision variable has two possible representations. The first is a one-dimensional array indexed by cars, containing decision variables with integer domains, where each entry represents the class selected for that car. The other is a two-dimensional Boolean array, indexed by both cars and classes, where a true value indicates the assignment of a car to a class. The *usage* parameter also has two possible representations: a two-dimensional Boolean array or a set of tuples. The second problem constraint in the Essence model that refers to the *usage* parameter is refined with an *element* constraint when the Boolean array is chosen, instead with a *table* constraint when the set of tuples is chosen.

Using a combination of these model fragments, Conjure constructs three distinct Essence Prime models. The first model M_1 has a one-dimensional array of integer variables for *car* and a set of tuples with a *table* constraint for the *usage* parameter. The second model M_2 couples the same one-dimensional array for *car* with a Boolean array for *usage* and the *element* constraint. The third model M_3 uses a two-dimensional Boolean array for *car*, and a set of tuples and the *table* constraint for *usage*.

4.2.2 Covering Array

For the covering array problem, the Essence model generated only one Essence Prime version. Due to the simplicity of the parameters and the single variable, the final Essence Prime model shares them with the original version and no auxiliary variables are needed. Similarly, the lexicographic constraints can be easily

[2]`https://developers.google.com/optimization/cp/cp_solver`

translated using the same constraint. For the remaining constraint, CONJURE uses arrays to represent the sequences for the rows and values variables.

4.2.3 Social Golfers

For Social golfers, CONJURE produces four distinct ESSENCE PRIME models, each offering a different matrix-based encoding of the weekly partitions. The first model uses a single 3D matrix indexed by *weeks* $\times$ *groups* $\times$ *slots*, where each entry is an integer indicating which golfer is in a particular part of the partition for a particular group and week. The second model represents the same structure with a 3D Boolean matrix, this time indexed by *weeks*, *groups*, and *golfers*, storing *true* or *false* values to indicate whether a given golfer is in a specific part of the partition.

The third model splits the partitioning into multiple matrices, capturing information such as the number of parts, which golfer goes into each position, and how many slots are used in each group. By breaking down the partition into several matrices, this approach can exploit different constraint formulations within the same overall representation. Finally, the fourth model combines two matrix encodings: one storing explicit integer assignments for each slot and another tracking membership via Booleans.

These variations arise because CONJURE can refine the *partition* domain in multiple ways, translating high-level sets of groups into different low-level data structures. Each resulting ESSENCE PRIME model may be advantageous for certain solver strategies or types of instances, and all of them feed into our portfolio of solver-model combinations.

4.3 Dataset and Algorithm Complementarity

This section presents an analysis of the portfolio of algorithms employed for each problem class and evaluates the potential benefits of applying AAS to the selected problems. Each subsection provides a detailed discussion of a specific problem class. Before delving into the individual analyses, we establish the shared methodology for recording algorithm runtimes.

All experiments were conducted on a computational setup equipped with an AMD EPYC 7763 CPU. Each algorithm was allocated a single CPU core and subjected to a one-hour cut-off time per instance.

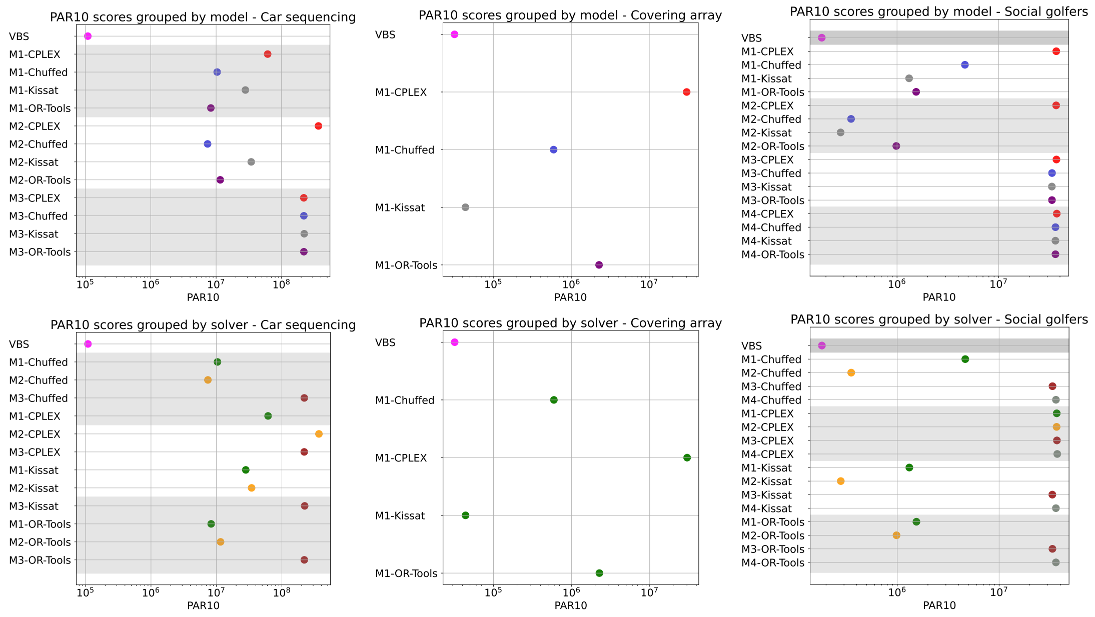


Figure 4.4: PAR10 value of each algorithm and the VB on the car sequencing (left), covering array (centre) and social golfers (right) instance sets (lower is better), where the algorithms are grouped by their models (top) or solvers (bottom). The results are in log scale.

4.3.1 Car Sequencing

The car sequencing problem class utilizes a portfolio of 12 algorithms derived from the combination of three Essence Prime models and four constraint solvers. The PAR10 scores for these algorithms, evaluated across the entire instance set, are shown in Figure 4.4. Additionally, the figure includes the Virtual Best Solver (VB), a theoretical construct representing the optimal algorithm selector that always identifies the best algorithm for each instance.

A key observation from Figure 4.4 is the lack of a dominant model or solver. The remaining algorithms, excluding those involving M_3, which always underperform no matter the solver, exhibit diverse performance characteristics. Notably, the gap between the VB and the best standalone algorithm, (M_2-Chuffed) is significant, with the SB achieving only 0.01% of the VB's performance. This underscores the potential for AAS to leverage complementary strengths among algorithms.

The complementarity of the algorithms in the portfolio can be further observed in Figure 4.5, where we plot on the left the average participation to VB (as the percentage of the instances where the algorithm is the best) and on the right the average competitiveness (as the percentage of the instances where the algorithm is competitive). We can see that even though M_2-Chuffed appears as the best overall

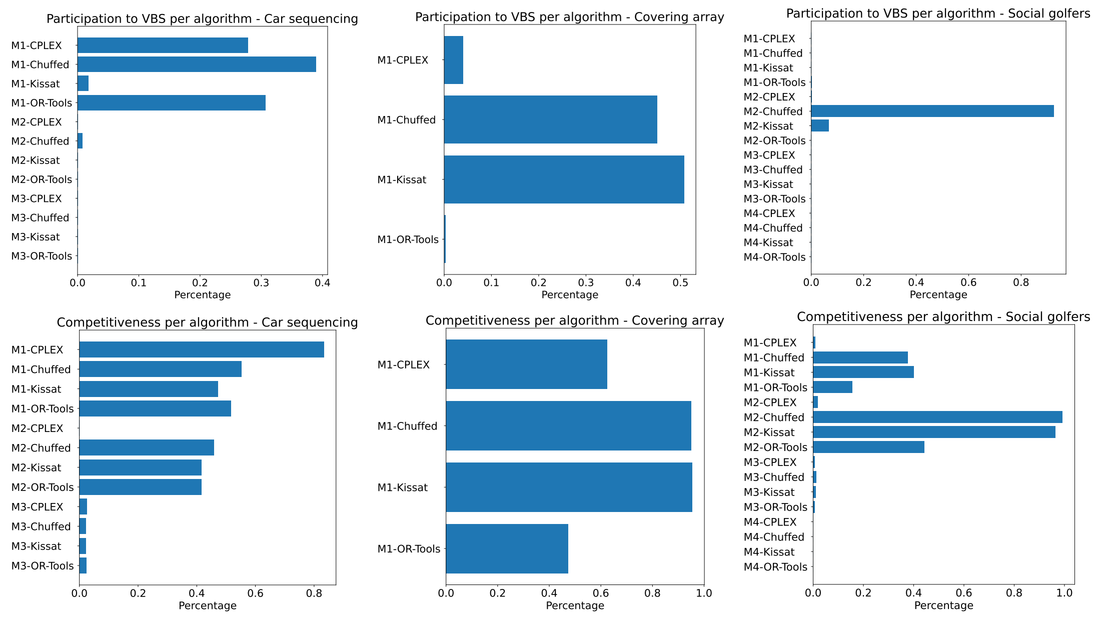


Figure 4.5: Participation, as a percentage, to VB (top) and competitiveness, as a percentage, (bottom) for the car sequencing (left), covering array (centre) and social golfers (right) instance sets.

algorithm in Figure 4.4, it is the winner on a fairly small number of instances according to the left plot of Figure 4.5. Instead, M_1-CPLEX, M_1-Chuffed and M_1-OR-Tools have significantly higher numbers of instances where they win. These three algorithms cover a significant part of the instance space.

While many algorithms do not appear to participate at all to VB, they are all competitive in some instances (with varying percentages), as shown in the right plot of Figure 4.5. An exception is M_2-CPLEX which in fact resulted in the worst overall algorithm in Figure 4.4.

4.3.2 Covering Array

For this problem class, only one Essence Prime model is present but the solvers are still four, therefore, the final portfolio is composed of four algorithms. Figure 4.4 shows the PAR10 score of each algorithm in the full instance set with VB included as well. Contrary to the car sequence case, for this problem class the gap between SB and VB is much smaller with M_1-Kissat being the SB achieving a performance ratio of 0.72 compared to the VB.

Even though the gap between SB and VB is much smaller compared to the car sequencing case, figure 4.5, on the left side, shows that each algorithm contributes to VB, even M_1-Cplex which has the worst PAR10 score. M_1-OR-Tools is the least contributing algorithm while M_1-Chuffed and M_1-Kissat both contribute similarly to VB with M_1-Chuffed contributing slightly more than 50% of the time and M_1-Kissat roughly 45% of the time. A similar story can be told by watching the right side of figure 4.5 which shows the percentage of competitiveness of each algorithm in the instance set. We can see that M_1-Chuffed and M_1-Kissat are almost always competitive, M_1-Cplex is competitive around 60% of the time and M_1-OR-Tools roughy 50% of the time. This second picture of the datasets shows a different story compared to 4.4 where M_1-Kissat seems the clear winner. In fact, 4.5 suggests a much more complex instance set where AAS could take advantage of the different algorithms.

4.3.3 Social Golfers

The social golfers problem has four possible Essence Prime models, resulting in sixteen different algorithms in conjunction with the four solvers. The PAR10 scores for social golfers, shown in figure 4.4, are very different from the ones for the other classes: in this case, 6 algorithms dominate all the other with similar bad results. The algorithm including the models M_1 and M_2 have the best performance when coupled with Chuffed, Kissat and OR-Tools. The same cannot be said for CPLEX because it performs similarly to the other algorithms. Here the gap between SB and VB stands in the middle between car sequencing and covering array since VB as a percentage of SB is 0.65.

The participation to VB and the competitiveness of each algorithm, shown in figure 4.5 tell a similar story to figure 4.4, there are clear winners: VB is almost entirely represented by M_2-Chuffed with a very small inclusion of M_2-Kissat and, on the competitive side, the same two algorithms are overwhelmingly the best ones with M_1-Chuffed, M_1-Kissat, M_1-OR-Tools and M_2-OR-Tools being the only one being competitive more than 10% of the time. This may seem a less-than-ideal scenario, however, the still-interesting gap between SB VB shows that whenever M_2-Chuffed is not the best algorithm, it is worse by a considerable margin making an AAS approach more interesting.

Chapter 5

Experimental Study

Having established the methodology for our work and the potential advantages of AS for the datasets presented, we now experimentally evaluate the effectiveness of our approaches. As described in Chapter 3, given an Essence instance as input, we can either learn to directly predict the best algorithm for the given instance within a single BERT-like neural network (the fully neural approach, denoted as `BNN`, or we can split the learning into two phases (the hybrid approach). In the hybrid approach, the first phase focuses on feature learning using either the same BERT-like architecture (`bNN`) or a slightly modified version where algorithm competitiveness is used as the target (`cNN`). The second phase makes used of the learnt features as input and employs commonly-used machine learning models from AS literature, including K-means (`K`) and the well-known AS tool AutoFolio [51](`A`). In summary, we consider four hybrid variants, denoted as `(bNN,K)`, `(bNN,A)`, `(cNN,K)` and `(cNN,A)`. To guide this evaluation, we aim to answer the following research questions (RQs):

- **RQ1:** Can we use either the Tanh features or the output of models as features or do we need to combine them to obtain an effective feature set?
- **RQ2:** Can a single neural network effectively produce an AS model, or is it necessary to split the learning process into two phases?
- **RQ3:** How do the learned features compare to the existing FZN2FEAT features in terms of performance?
- **RQ4:** What is the computational cost of extracting the learned features, and how does this cost influence the final results compared to the FZN2FEAT extraction cost?

his chapter begins by detailing the experimental design and the training process for the neural networks. Subsequently, it addresses each of the research questions through a systematic experimental study.

5.1 Experimental Design

All experiments were conducted using Python 3.11 with PyTorch[1] and scikit-learn[2] for neural network implementation and K-means clustering, respectively. Python 3.6 was utilized for experiments involving AutoFolio.[3] The complete codebase supporting this work is publicly available in the project repository.[4]

Algorithm selection setup. As described in Section 3, we use the Penalised Average Runtime with a factor of 10 (PAR10) to measure the performance of an AS model on a set of problem instances. Each AS approach is evaluated using 10-fold cross-validation. During each fold, 10% of the training data is served as a validation set to support the neural network training process and hyper-parameter optimisation. All experiments are run on a computer with an AMD EPYC 7763 CPU where, each time, we limit the number of cores available for an algorithm depending on our necessities.

In the hybrid approaches, we make use of K-means and the AS tool AutoFolio for the AS task. AutoFolio offers a tuning mode using the hyper-parameter optimisation tool SMAC [33], which we employ in our experiments with a single CPU core and a tuning budget of 5 hours.

K-means Hyper-parameters In Chapter 3, we have introduced the K-means clustering algorithm and how we have used it as an AS approach. During the optimization phase of the clustering scheme, we use a set of hyper-parameters to combine to get the best clustering scheme for our purpose. In particular, the hyper-parameters we set are: the number of clusters which ranges between 2 and 21, the initialization method which could be random or using the K-means++ methodology, the number of maximum iterations which could be 100, 200 or 300, the tolerance which could be $10^{-3}, 10^{-4}$ or 10^{-5}. Finally, the last hyper-parameter we set is the number of possible initializations the algorithm can use which can be 5. 10, 15 or ”auto”. A detailed explanation of the use of each of these values can be found on the official Scikit-learn documentation.[5]

Normalised PAR10 scores. Due to the different scales of PAR10 across folds, following existing AS literature [51], we use a normalised version of the PAR10

[1]`https://pytorch.org/`

[2]`https://scikit-learn.org/stable/index.html`

[3]`https://github.com/automl/AutoFolio/tree/master`

[4]`https://github.com/SeppiaBrilla/EFE_project`

[5]`https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`

score in our comparison. The normalised PAR10 score is calculated as:

$$\text{score} = \frac{p(AS) - p(VB)}{p(SB) - p(VB)} \tag{5.1}$$

where $p(AS)$, $p(VB)$ and $p(SB)$ are the PAR10 scores of an AS approach, the VB, and the SB on the same fold, respectively. The VB has a score of 0 and the SB has a score of 1, and we want to minimise this normalised score. An AS approach is only considered effective if its normalise score is less than 1, i.e., it performs better than the SB (i.e., the AS approach without any learning required). Unless specified otherwise, reported prediction times include feature computation times.

5.2 Neural Network Training

For each dataset, the neural network models were trained from scratch to evaluate their learning capabilities independently, without leveraging any form of pretraining. The exploration of transfer learning and multi-problem feature learning is left for future work. All NN models were trained on a GPU with an NVIDIA A5000 accelerator.[6].

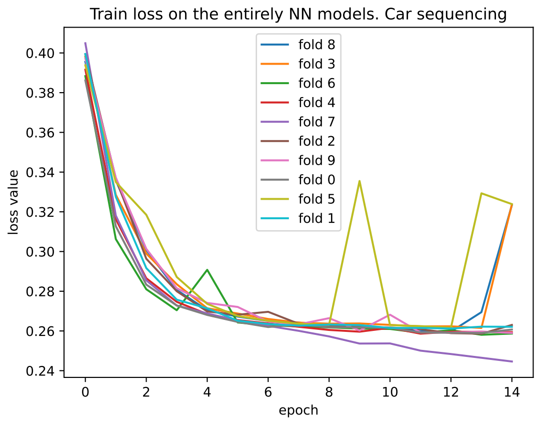
For approaches in which feature learning and algorithm selection were performed separately, the same data splits were used for the ML algorithm selector. Specifically, if an instance appeared in the test set of the NN, it was also included in the test set of the ML model using the extracted features.

Models hyper-parameters For both the BNN and CNN models, we need to set the sizes of the feature and post-feature layers as they are arbitrary and their values can vastly impact both the size of the models and their ability to learn the task effectively. After some manual tuning, we set the size of the feature layer to 100 neurons and the size of the post-feature layer to 200 neurons. These values are big enough to allow a lot of flexibility in the model's parameters while still being small enough not to cause over-parametrization problems (for the post-feature layer) or issues with the external ASs (for the feature layer).

5.2.1 BNN Models

The BNN models employed the Cross-Entropy loss function in conjunction with the Adam optimizer. For each dataset, the models were trained using distinct hyper-parameter configurations, including variations in learning rates, the total number of training epochs, and batch sizes.

[6]`https://www.nvidia.com/en-us/design-visualization/rtx-a5000/`



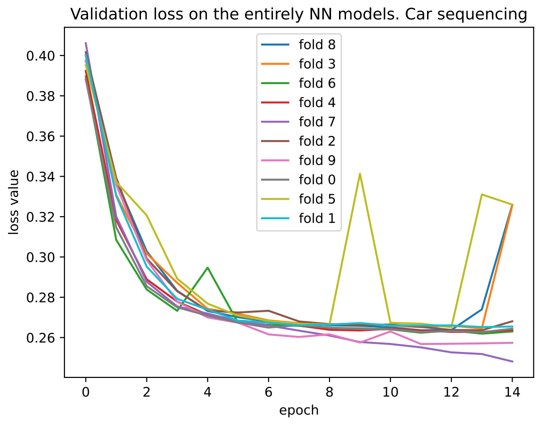


Figure 5.1: Loss values for the BNN models across all folds for the Car sequencing problem, shown for training (left) and validation (right) sets.

Figure 5.1 illustrates the evolution of the loss values for both the training and validation sets at the end of each epoch for the BNN models trained on the Car sequencing dataset. For this dataset, the models were trained for a total of 15 epochs using a learning rate of 7×10^{-6}. Due to memory constraints, the batch size was limited to only two elements per batch. The loss trends for the training and validation sets are generally consistent, with both following a similar overall trajectory. With the exception of a single case, the majority of networks exhibit diminishing improvements after the 6th epoch. In certain instances, the loss value increases during the final epochs, suggesting overfitting. Notably, the fifth fold displays the poorest performance, characterized by a more unstable trend and a substantially higher final loss value.

The BNN models trained on the Covering array problem were trained for a total of 100 epochs with a learning rate of 1×10^{-5} and a batch size of 32 items per batch. Their loss values during training can be seen in Figure 5.2. Among the datasets analysed, the models trained on the Covering array exhibited the least favourable trend. The loss curves reveal a pronounced tendency toward overfitting, as the loss for the training set continues to decrease while the validation loss stagnates or increases after approximately 25 epochs. This behaviour suggests that the models struggle to generalize effectively.

In Figure 5.3 we show the loss values for the training and validation sets for the BNN models when trained on the Social golfers dataset. The models for this dataset were trained for 50 epochs using a learning rate of 6×10^{-6} and a batch size of 32 items per batch. Similar to the trends observed in the Car sequencing models, the loss values for the training and validation sets follow comparable trajectories. However, unlike the Car sequencing models, the loss in the Social golfers

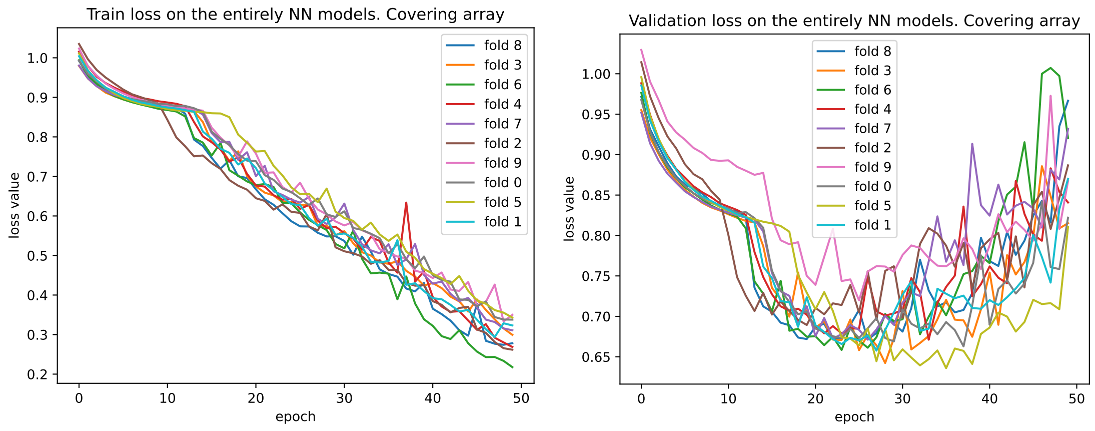


Figure 5.2: Loss values for the BNN models across all folds for the Covering array, shown for training (left) and validation (right) sets.

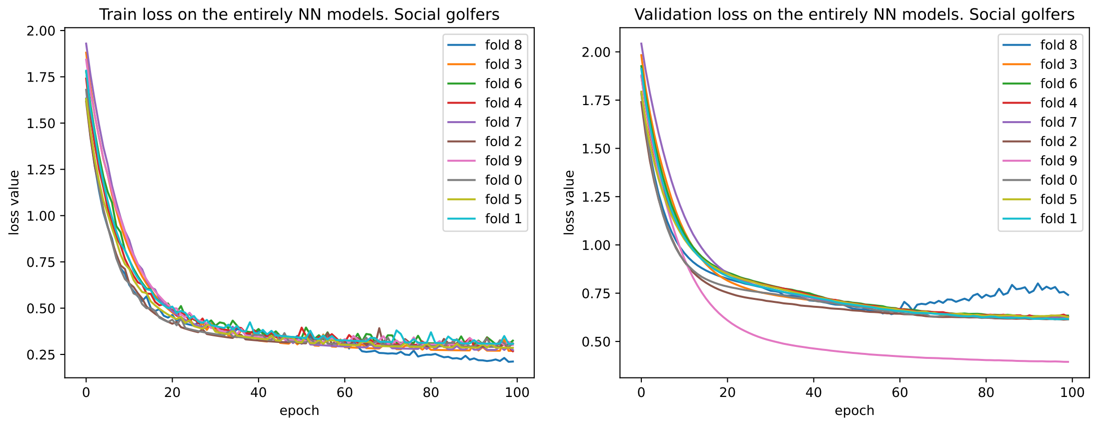


Figure 5.3: Loss values for the BNN models across all folds for the Social golfers, shown for training (left) and validation (right) sets.

models continues to improve, albeit at a slower rate, after stabilization around the 20th epoch. Two folds exhibit distinct behaviours worth highlighting: fold 9 achieves a notably lower validation loss compared to the other folds, while fold 8 demonstrates signs of overfitting beginning around the 60th epoch, with its validation loss subsequently increasing, ultimately leading to the worst performance among all folds.

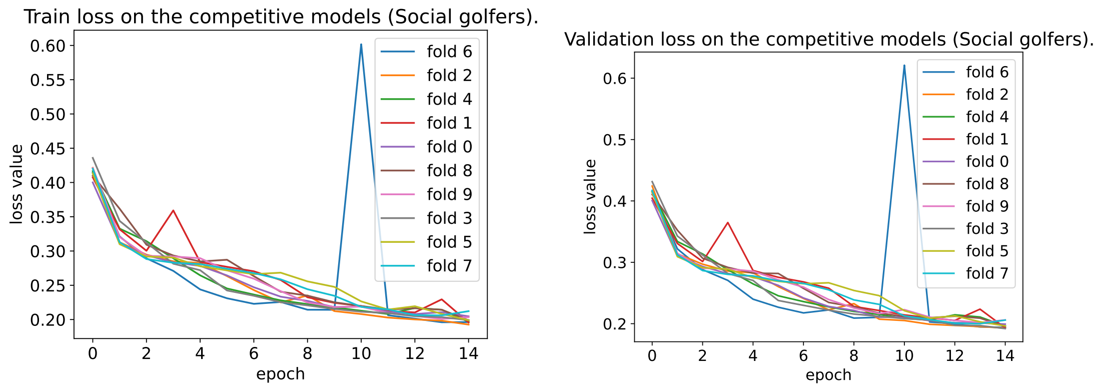


Figure 5.4: Loss values for the hybrid approach models across all folds for the Car sequencing problem class, shown for training (left) and validation (right) sets.

5.2.2 CNN models

The CNN models shared most of their characteristics with the BNN ones. However, instead of using the Cross-Entropy loss function, the Binary Cross-Entropy (BCE) loss function was employed due to the nature of the task, which involved multilabel classification. The optimizer remained the Adam optimizer. As with the BNN models, each dataset was trained using a distinct set of hyper-parameters. Furthermore, similar to the fully NN-based approach, model performance was evaluated on the validation set at the end of each epoch. To mitigate the risk of overfitting, the model with the lowest validation loss was saved during training.

Figure 5.4 illustrates the evolution of the loss values for the training and validation sets for the CNN models trained on the Car sequencing dataset. The models were trained for 15 epochs with a learning rate of 7×10^{-6} and a batch size of 2 items per batch, a constraint imposed by memory limitations similar to the fully neural approach. The overall loss trend remains consistent across all folds, with a similar trajectory observed for both the training and validation sets. Unlike the BNN models, where the training process often stagnates or slows down after a certain number of epochs, the hybrid approach maintains a steady reduction in loss throughout all epochs. Two folds demonstrate particularly noteworthy behaviour: fold 8 experiences a small but noticeable loss spike at the 4th epoch, while fold 6 exhibits a more significant spike at the 11th epoch. Interestingly, in both cases, the loss subsequently returns to a downward trend, and the spikes are observed in both the training and validation sets.

The CNN models trained on the Covering array dataset (whose loss can be seen in figure 5.5) were trained for 100 epochs with a learning rate of 7×10^{-6} and a batch size of 32 items per batch. All folds exhibit a relatively slow start, with the loss decreasing at a sluggish pace during the initial epochs, followed by a

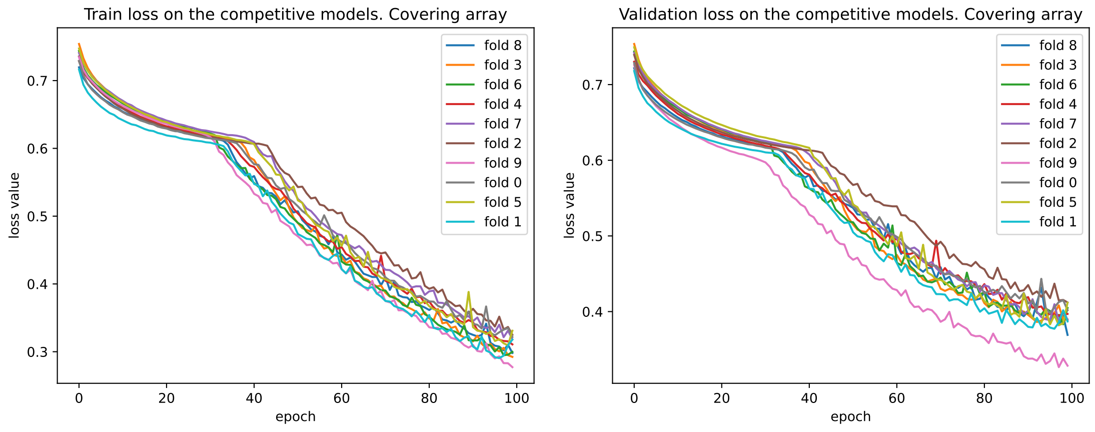


Figure 5.5: Loss values for the hybrid approach models across all folds for the Covering array problem class, shown for training (left) and validation (right) sets.

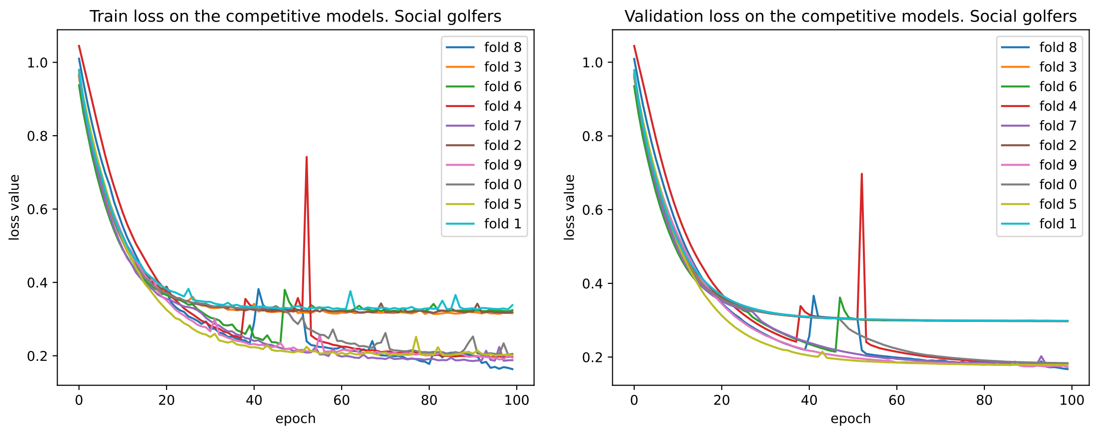


Figure 5.6: Loss values for the hybrid approach models across all folds for the Social golfers problem class, shown for training (left) and validation (right) sets.

more rapid convergence in subsequent epochs. This pattern is observed in both the training and validation sets, with all folds reaching a similar loss value by the end of training. Interestingly, only the 9th fold displays a slightly lower final validation loss than the other folds. This consistent convergence across folds suggests that the peculiar shape of the loss curves is likely a result of the inherent characteristics of the dataset or the loss landscape, rather than fold-specific anomalies.

Figure 5.6 illustrates the evolution of the loss values for the training and validation sets for the CNN models trained on the Social golfers dataset. The models were trained for 100 epochs with a learning rate of 9×10^{-6} and a batch size of 32 items per batch. The loss values for the Social golfers dataset follow a trend similar

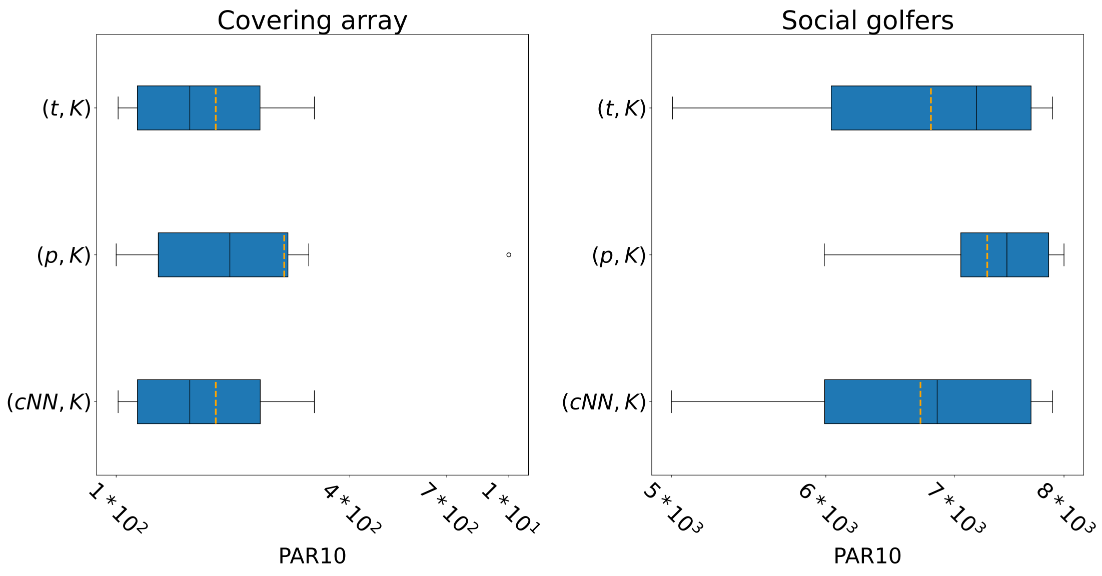


Figure 5.7: Comparison of the cNN features, the probabilities-only features (p) and the tanh-only features (t) for the covering array and social golfers problems.

to that observed in the BNN models. Specifically, there is a rapid convergence in the early epochs, after which the rate of improvement slows. Notably, the validation loss is more stable in the hybrid approach compared to its BNN counterpart. However, similar to the BNN models, the CNN models exhibit a few spikes in loss during training although, in this case, the spikes are at approximately the same epochs and across multiple folds, suggesting that they may be due to the structure of the loss landscape rather than the idiosyncratic behaviour of specific folds.

5.3 Ablation Study

In this section, we will answer **RQ1:** Can we use either the Tanh features or the output of models as features or do we need to combine them to obtain an effective feature set?

Figure 5.7 illustrates the benefits of combining the Tanh with the output probabilities. As shown, using only the output probabilities (p) in Figure 5.7 significantly hinders the learning process and leads to poorer performance for the K-means AS compared to the cNN features. In contrast, when only the Tanh features are used (t), the performances are more comparable to those obtained from the cNN features. Interestingly, for the covering array problem, both approaches achieve identical results. However, in the case of the social golfers problem, both the average and mean performance metrics indicate worse outcomes compared to the cNN

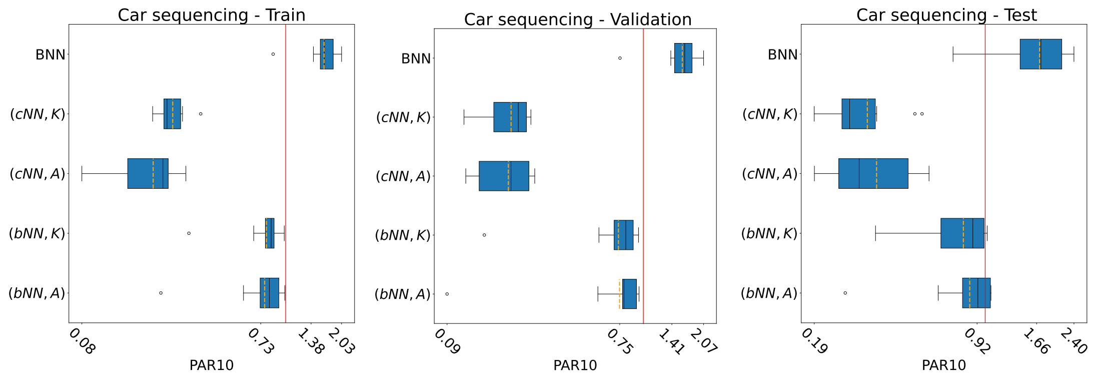


Figure 5.8: Normalized PAR10 scores of different AS approaches trained using neural features across 10 folds for the Car sequencing dataset. The red line indicates SB (1).

features, despite sharing the same total range. Although this part of the study is far from exhaustive, we still think these results are significant and we will only use the combined features (bNN and cNN) in the following sections.

5.4 Feature Learning and Algorithm Selection: Combining vs Splitting

In this section, we investigate RQ2: Can a single neural network effectively produce an AS model, or is it necessary to split the learning process into two phases? Furthermore, we aim to evaluate which training methodology yields the most effective semantic representation of problem instances, thereby enabling the accurate and consistent prediction of the best-performing algorithm.

Overall, the performance (Figures 5.8, 5.10 and 5.9) of the fully neural approach `BNN` is subpar compared to the hybrid ones. Even on the training set, a majority of `BNN` runs result in worse performance than the SB (which does not require any learning). Among the hybrid approaches, `(cNN, K)` consistently obtains the best overall performance across the three problem classes. Among the five studied approaches, `(cNN, K)` is the only one that consistently achieves better performance than SB.

A likely explanation for the underperformance of the fully neural approach lies in the inherent imbalance of the training data in the multi-class classification task. Algorithms that excel on only a small subset of instances are underrepresented, making them harder to predict correctly despite their significant impact on the overall PAR10 score. The competitive approach mitigates this imbalance by emphasizing performance differences across algorithms, allowing for better gen-

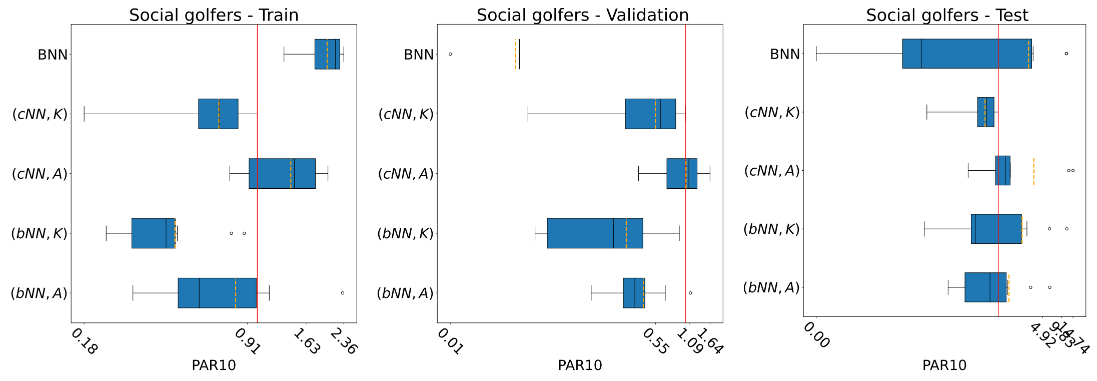


Figure 5.9: Normalized PAR10 scores of different AS approaches trained using neural features across 10 folds for the Social golfers dataset. The red line indicates SB (1).

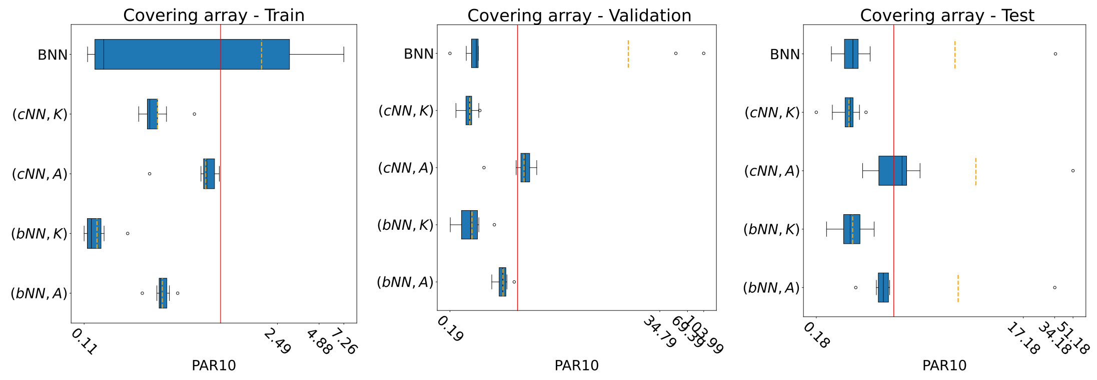


Figure 5.10: Normalized PAR10 scores of different AS approaches trained using neural features across 10 folds for the Covering array dataset. The red line indicates SB (1).

eralization and improved instance characterization. This generalization makes the learning process very effective resulting in a PAR10 score that significantly improves SB in every dataset.

It is also interesting to see that the combination of `cNN` features and the K-means AS approach results in much better performance compared to `(cNN,A)`, despite K-means' simplicity compared to a state-of-the-art AS framework like Autofolio. Our finding illustrates that simpler ML-based AS approaches can be quite effective for certain AS tasks compared to the more sophisticated approaches commonly used in AS literature.

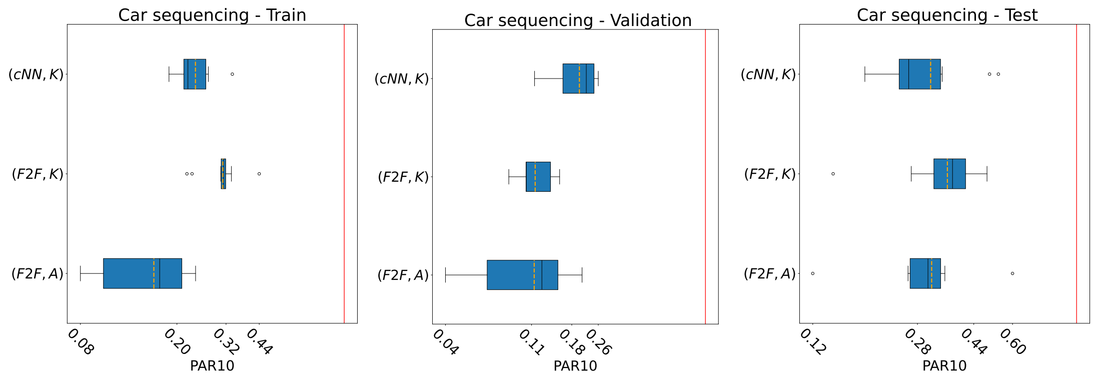


Figure 5.11: PAR10 scores of AS approaches cNN or F2F features across 10 folds for the Car sequencing dataset. The red line indicates SB (1)

5.5 Learnt features vs Fzn2feat

In this section, we address RQ3: How do the learned features compare to the existing FZN2FEAT features in terms of performance?

Building on the findings from Section 5.4, which demonstrated the superiority of the `(cNN, K)` approach, we now compare its performance against the FZN2FEAT (F2F) features [7]. These features have been chosen as a baseline as they have been used in literature before in AS tools to perform similar tasks. We will use the same AS algorithms as for the cNN features but train them using the F2F features instead. To extract the F2F features, it is necessary to run the instance on a solver and, therefore, convert the instance to FlatZinc [58]. To do so, we have used the default model output by CONJURE (M_1) and used SAVILLE ROW to output the FlatZinc instance to use with the feature extractor. All the instances have been computed using a machine with 2 CPU cores and 8GB of available RAM. On some instances, the extraction process crashed due to an excessive amount of RAM usage. While the feature extraction process for the bNN and cNN features requires dedicated hardware (a GPU), it's still noteworthy the fact that in no case the amount of used memory on the GPU surpassed a few GB.

Figure 5.11 shows the PAR10 scores for AS approaches trained on F2F features and the `(cNN, K)` approach for the Car sequencing dataset. `(cNN, K)` Outperforms `(F2F, K)` on 7 out of 10 folds.`(F2F,A)` demonstrates more consistent performance across folds, yielding a better mean PAR10 score compared to `(cNN, K)`. In contrast, `(cNN, K)` achieves a lower median PAR10 score, indicating greater robustness in capturing instance semantics and potential for better generalization.

For the Covering array dataset (Figure 5.12), the results reveal a clear advantage for the and the `(cNN, K)` approach. The system consistently outperforms SB across all folds and even approaches the performance of the VB in some cases.

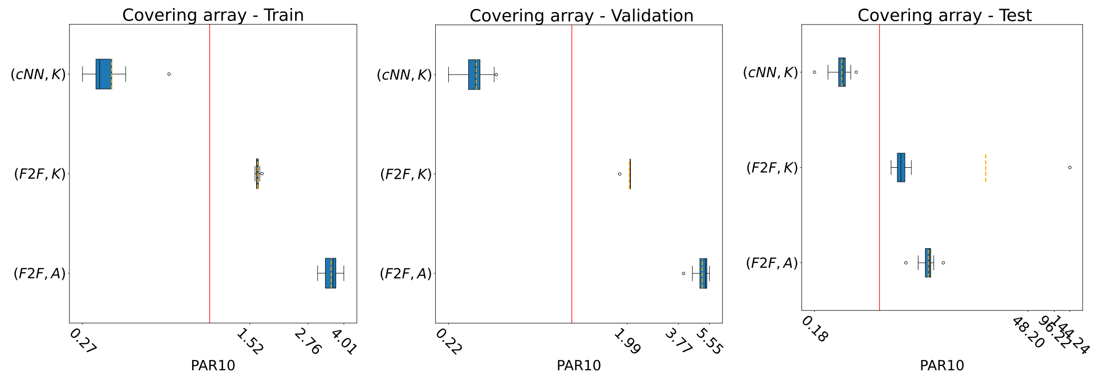


Figure 5.12: PAR10 scores of AS approaches cNN or F2F features across 10 folds for the Covering array dataset. The red line indicates SB (1)

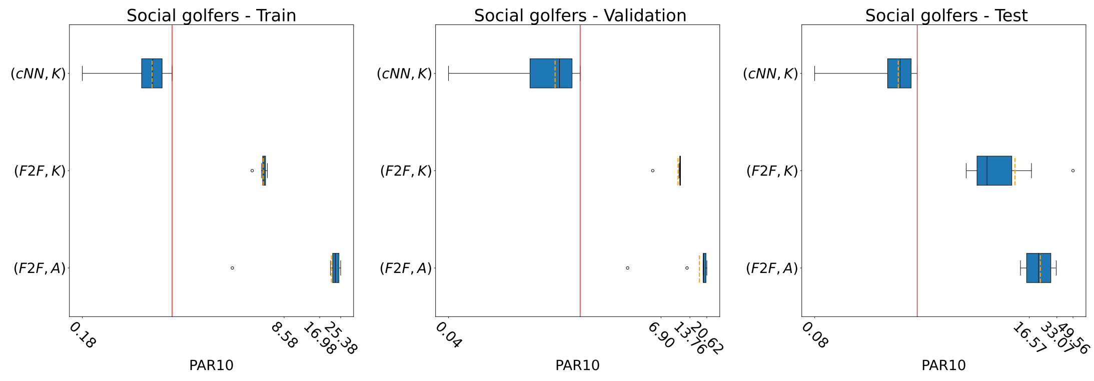


Figure 5.13: PAR10 scores of AS approaches cNN or F2F features across 10 folds for the Social golfers dataset. The red line indicates SB (1)

In contrast, AS systems trained on F2F features fail to surpass the performance of SB, even on the training set. This outcome persists even when feature extraction costs are excluded from the evaluation, suggesting that F2F features do not sufficiently capture the semantic properties of instances for this dataset.

The results for the Social golfers dataset, shown in Figure 5.13, exhibit a similar trend to that observed for the Covering array dataset. Here, `(cNN,K)` is the only approach that consistently outperforms SB. In contrast, AS models trained using F2F features consistently fail to outperform SB. However, in this dataset, the cost of feature extraction plays a more critical role. Excluding the feature extraction time, `(F2F,K)` does surpass SB algorithm in three different folds. Additionally, it is noteworthy that, for some problem instances, the F2F extraction process failed due to memory crashes. Although these crashes prevented the AS from making predictions, the system still incurred computational overhead, further impacting

fold	Car sequencing				Covering array				Social golfers			
	F2F + E	F2F	cNN + E	cNN	F2F + E	F2F	cNN + E	cNN	F2F + E	F2F	cNN + E	cNN
0	132.51	127.17	300.57	300.52	23.23	19.23	14.85	14.85	1,547.11	1,224.51	209.45	209.44
1	301.97	296.69	156.22	156.17	28.60	25.00	20.99	20.99	1,163.31	848.11	268.35	268.35
2	273.85	268.54	212.42	212.37	26.48	22.65	15.95	15.95	518.72	199.68	234.85	234.85
3	229.78	224.35	179.38	179.33	25.37	21.80	18.60	18.60	1,175.69	857.06	253.20	253.20
4	348.88	343.56	441.89	441.83	20.79	17.27	14.78	14.78	528.35	216.90	275.22	275.22
5	246.55	241.23	177.89	177.84	27.93	24.43	18.21	18.21	481.67	176.76	181.12	181.11
6	283.33	277.90	359.54	359.49	20.56	17.04	14.40	14.40	841.24	535.98	221.02	221.02
7	352.25	346.96	183.32	183.27	15.95	12.44	10.82	10.82	849.04	548.80	268.51	268.51
8	284.51	278.98	203.13	203.07	16.89	13.37	11.72	11.72	911.93	594.83	280.15	280.15
9	179.17	173.60	151.15	151.10	504.62	501.03	21.22	21.22	1,228.29	916.30	168.23	168.23

Table 5.1: Average AS predicted times with (+ E) and without feature extraction cost for each fold in the Car sequencing, Covering array, and Social golfers problem classes. The AS used for all the problems is K-means.

overall performance.

Across the three datasets, `(cNN, K)` consistently outperforms F2F-based approaches. The cNN-based approach achieves lower PAR10 scores and exhibits greater robustness across folds. In two datasets (Covering array and Social golfers), AS models trained on F2F features fail to outperform SB, even before considering feature extraction costs. These results suggest that cNN features are more effective in capturing the semantic properties of instances, leading to improved AS performance.

5.6 Feature Extraction Cost

In this section, we analyse and address RQ4: What is the computational cost associated with extracting the learned features, and how does this cost influence the final results compared to the F2F extraction cost?

The computational cost of feature extraction for all the problems is presented in Table 5.1. For the Car sequencing dataset, the feature extraction cost significantly affects the final AS score when F2F features are utilized. While the final results still outperform SB, the average instance time increases substantially. Conversely, the inclusion of the feature extraction cost has little to no impact on the final AS score for the cNN-based approach, as the computational time required for cNN feature extraction is minimal.

A similar pattern is observed in the Covering array problem. The inclusion of feature extraction costs causes the F2F-based AS to exhibit significantly worse average scores compared to when the extraction cost is excluded. Even though the F2F-based AS never outperforms SB, the detrimental impact of the extraction time remains consistent. In contrast, the cNN-based approach is unaffected, as the inclusion of feature extraction time does not change the AS results due to the relatively low computational cost of cNN feature extraction.

The Social golfers problem demonstrates the most severe impact of feature extraction time on the F2F approach. For 3 folds in particular, the extraction cost significantly negatively impacts the corresponding AS's overall performance and alters their ranking, while `(cNN,K)` achieves a better score on the remaining 7 folds no matter what. In some instances, the extraction process failed to complete, forcing the system to resort to the single best strategy while accounting for the wasted extraction time. As with the previous problem classes, the cNN-based AS remains unaffected, as its low computational overhead ensures that average scores do not change.

Chapter 6

Conclusions

This chapter serves as a concluding evaluation of our work, starting with a summary of the main contributions presented in this thesis and then proposing some possible future work to explore and improve on what we have presented.

6.1 Summary of Contributions

In this thesis, we have investigated a novel approach to feature extraction based on a Transformer Encoder within the context of algorithm selection for combinatorial optimization problems. Our proposed approach uniquely focuses on high-level instance characteristics, offering three significant advantages:

(i) The extracted features capture the high-level semantics of the problem instance, whereas existing methods predominantly rely on low-level details that can only be obtained after partially executing the instance using a solver. (ii) Our method enables faster feature extraction since it eliminates the need to execute the solver, potentially resulting in considerable time savings. (iii) Our features are automatically learned for the task eliminating the need for designing a rich and diverse feature set to characterize an instance.

We have introduced two distinct strategies for training a neural network to learn the desired features and have empirically validated the superiority of one training approach over the other.

Our methodology was evaluated on three diverse datasets, each representing different problem classes. Across all three datasets, our feature extraction approach yielded improvements over the single best solver strategy. Additionally, in two out of the three datasets, our approach outperformed the baseline FZN2FEAT features, while in the third dataset, our features achieved performance comparable to the baseline. Furthermore, we obtained competitive results, especially when our features were paired with the K-means strategy. These outcomes highlight both the effectiveness and generalizability of our approach, suggesting that it is a viable solution for integration into any algorithm selection tool designed to address combinatorial optimization problems.

6.2 Future Work

This research primarily aimed to demonstrate the feasibility of our approach by training several neural network models from scratch and evaluating the effectiveness of our training strategies and final feature extraction results. Consequently, the scalability of this approach across multiple problem classes remains an open question. A notable challenge lies in determining how to extract features for different problem classes, given that each class may be associated with a distinct set of algorithms. As a result, both the number of features and the architecture of the neural network may vary depending on the specific problem class.

Once the aforementioned scalability issue is addressed, we believe it will be essential to evaluate our features in conjunction with a state-of-the-art algorithm selection tool, such as SunnyCP [52]. Specifically, we propose testing our features as a direct replacement for the features currently employed by existing tools, as well as developing a specialized algorithm selection tool designed to fully leverage our proposed feature extraction methodology. Finally, all our methods used a simple AS strategy where the model predicts a single algorithm to use. In the literature, there exists a number of different approaches that also significantly improve the performance over the single best algorithm. As an example, many tools predict a schedule of algorithms to run ranging from the most to the least promising one [39].

Bibliography

[1] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8):1295, 2020.

[2] Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

[3] Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 3–12. Springer, 2018.

[4] Mohamad Alissa, Kevin Sim, and Emma Hart. Automated algorithm selection: from feature-based to feature-free approaches. *Journal of Heuristics*, 29(1):1–38, 2023.

[5] Boris Almonacid. Towards an automatic optimisation model generator assisted with generative pre-trained transformer. *arXiv preprint arXiv:2305.05811*, 2023.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, pages 316–324. Springer, 2013.

[7] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1357–1359, 2014.

[8] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

[9] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. Self-configuring cost-sensitive hierarchical clustering with recourse. In *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 524–534. Springer, 2018.

[10] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[11] Christian Bessiere. Constraint propagation. In *Foundations of Artificial Intelligence*, volume 2, pages 29–83. Elsevier, 2006.

[12] Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

[13] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[14] Leo Breiman. *Classification and regression trees.* Routledge, 2017.

[15] Derek Bridge, Eoin O'Mahony, and Barry O'Sullivan. Case-based reasoning for autonomous constraint solving. In *Autonomous search*, pages 73–95. Springer, 2012.

[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[17] Chuffed Developers. Chuffed, a lazy clause generation solver. `https://github.com/chuffed/chuffed`. Accessed: 2024-07-05.

[18] A. Crespo Márquez. The curse of dimensionality. In *Digital Maintenance Management: Guiding Digital Transformation in Maintenance*, pages 67–86. Springer, 2022.

[19] Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, and Peter Nightingale. A framework for generating informative benchmark instances. *arXiv preprint arXiv:2205.14753*, 2022.

[20] Dingsheng Deng. Dbscan clustering algorithm based on density. In *2020 7th international forum on electrical engineering and automation (IFEEA)*, pages 949–953. IEEE, 2020.

[21] Conjure developers. Essencecatalog: A collection of problem specifications in essence, 2024. Accessed: 2024-06-30. URL: `https://github.com/conjure-cp/EssenceCatalog`.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE transactions on systems, man, and cybernetics, part b (cybernetics)*, 26(1):29–41, 1996.

[24] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the sixth international symposium on micro machine and human science*, pages 39–43. Ieee, 1995.

[25] Imanol Echeverria, Maialen Murua, and Roberto Santana. Leveraging constraint programming in a deep learning approach for dynamically solving the flexible job-shop scheduling problem. *arXiv preprint arXiv:2403.09249*, 2024.

[26] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[27] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008. `doi:10.1007/s10601-008-9047-y`.

[28] Abram Handler. An empirical study of semantic similarity in wordnet and word2vec. 2014.

[29] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European journal of operational research*, 130(3):449–467, 2001.

[30] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. Traveling salesman problem. *Encyclopedia of operations research and management science*, 1:1573–1578, 2013.

[31] John N Hooker and W-J van Hoeve. Constraint programming and operations research. *Constraints*, 23:172–195, 2018.

[32] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In *Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings 11*, pages 301–317. Springer, 2014.

[33] F. Hutter, Holger H Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION 5*, pages 507–523. Springer, 2011.

[34] IBM. Ibm ilog cplex optimization studio: Cplex optimizer. `https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer`, 2022. Accessed: 2024-07-05.

[35] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17*, pages 454–469. Springer, 2011.

[36] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[37] Ludwig Kampel and Dimitris E Simos. A survey on the state of the art of complexity problems for covering arrays. *Theoretical Computer Science*, 800:107–124, 2019.

[38] Akanksha Kapoor and Abhishek Singhal. A comparative study of k-means, k-means++ and fuzzy c-means clustering algorithms. In *2017 3rd international conference on computational intelligence & communication technology (CICT)*, pages 1–6. IEEE, 2017.

[39] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

[40] Salabat Khan, Mohsin Bilal, M Sharif, Malik Sajid, and Rauf Baig. Solution of n-queen problem using aco. In *2009 IEEE 13th international multitopic conference*, pages 1–5. IEEE, 2009.

[41] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. An evolutionary approach to combinatorial optimization problems. In *ACM Conference on Computer Science*, pages 66–73, 1994.

[42] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[43] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[44] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190. Springer, 2016.

[45] Lars Kotthoff, Pascal Kerschke, Holger Hoos, and Heike Trautmann. Improving the state of the art in inexact tsp solving using per-instance algorithm selection. In *Learning and Intelligent Optimization: 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers 9*, pages 202–217. Springer, 2015.

[46] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[47] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm-a literature review. In *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pages 380–384. IEEE, 2019.

[48] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[49] M. Lindauer, Jan N van R., and L. K. The algorithm selection competitions 2015 and 2017. *AI*, 272:86–100, 2019.

[50] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL: `http://jmlr.org/papers/v23/21-0888.html`.

[51] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

[52] Tong Liu, Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. sunny-as2: Enhancing sunny for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021.

[53] Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum. *Advances in Neural Information Processing Systems*, 33:18261–18271, 2020.

[54] Luiz Antonio N Lorena and Marcelo G Narciso. Relaxation heuristics for a generalized assignment problem. *European Journal of Operational Research*, 91(3):600–610, 1996.

[55] Yulong Lu and Jianfeng Lu. A universal approximation theorem of deep neural networks for expressing probability distributions. *Advances in neural information processing systems*, 33:3094–3105, 2020.

[56] Jayanta Mandi, James Kotary, Senne Berden, Maxime Mulamba, Victor Bucarey, Tias Guns, and Ferdinando Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. *Journal of Artificial Intelligence Research*, 80:1623–1701, 2024.

[57] Kostis Michailidis, Dimos Tsouros, and Tias Guns. Constraint modelling with llms using in-context learning. In *30th International conference on principles and practice of constraint programming*, 2024.

[58] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[59] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[60] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

[61] Peter Nightingale and Andrea Rendl. Essence'description. *arXiv preprint arXiv:1601.02865*, 2016.

[62] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

[63] Rukhma Qasim, Waqas Haider Bangyal, Mohammed A Alqarni, and Abdulwahab Ali Almazroi. A fine-tuned bert-based transfer learning approach for text classification. *Journal of healthcare engineering*, 2022(1):3498123, 2022.

[64] Alec Radford. Improving language understanding by generative pre-training. 2018.

[65] A. Rangamani, M. Lindegaard, T. GA.ti, and T. A Poggio. Feature learning in deep classifiers through intermediate neural collapse. In *ICML*, pages 28729–28745. PMLR, 2023.

[66] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[67] Lior Rokach and Oded Maimon. Clustering methods. *Data mining and knowledge discovery handbook*, pages 321–352, 2005.

[68] Moritz Vinzent Seiler, Jeroen Rook, Jonathan Heins, Oliver Ludger Preuß, Jakob Bossek, and Heike Trautmann. Using reinforcement learning for per-instance algorithm configuration on the tsp. In *2023 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 361–368. IEEE, 2023.

[69] Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023.

[70] MiniZinc Development Team. Effective modelling practices in minizinc, 2024. Accessed: 2024-12-20. URL: `https://docs.minizinc.dev/en/stable/efficient.html`.

[71] Tijmen Tieleman and Geoffrey Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *COURSERA Neural Networks Mach. Learn*, 17, 2012.

[72] Markus Triska and Nysret Musliu. An effective greedy heuristic for the social golfer problem. *Annals of Operations Research*, 194(1):413–425, 2012.

[73] Ah Chung Tsoi and Ah Chung Tsoi. Gradient based learning methods. *International School on Neural Networks, Initiated by IIASS and EMFCSC*, pages 27–62, 1997.

[74] Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv preprint arXiv:2308.01589*, 2023.

[75] Mauro Vallati, Lukáš Chrpa, and Diane Kitchin. Asap: an automatic algorithm selection approach for planning. *International Journal on Artificial Intelligence Tools*, 23(06):1460032, 2014.

[76] Willem-Jan Van Hoeve and Irit Katriel. Global constraints. In *Foundations of Artificial Intelligence*, volume 2, pages 169–208. Elsevier, 2006.

[77] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[78] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[79] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[80] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[81] Zhi-Hua Zhou. *Machine learning*. Springer nature, 2021.