Thomas Absenger, BSc

Growing Decision Trees with Reinforcement Learning

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Roman Kern

Institute for Interactive Systems and Data Science
Head: Univ.-Prof. Dr. Stefanie Lindstaedt

Graz, December 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

Decision trees are one of the most intuitive models for decision making used in machine learning. However, the greedy nature of state of the art decision tree building algorithms can lead to subpar results. This thesis aimed to use the non-greedy nature of reinforcement learning to overcome this limitation. The novel approach of using reinforcement learning to grow decision trees for classification tasks resulted in a new algorithm that is competitive with state of the art methods and is able to produce optimal trees for simple problems requiring a non-greedy solution. We argue that it is well suited for data exploration purposes due to diverse results and direct influence on the trade-off between tree size and performance.

Contents

List of Figures

1 Introduction

Decision trees are a well-understood research topic with wide-spread applications in data science and machine learning. The benefits of using decision trees for machine learning tasks such as classification are numerous. One of the most vital benefits is that decision trees are intuitive and easy to understand, especially when compared to other prominent models like neural networks. The white box model offered by decision trees visualizes the decision process, which is likely close to the way humans reach decisions when faced with various alternatives. There are, however, several downsides to decision trees, first and foremost among which is the fact that building optimal decision trees is an NP-hard problem. Traditional decision tree building algorithms circumvent this problem with a non-optimal greedy approach.

Using reinforcement learning, which is non-greedy by nature, as a tool to grow decision trees may create a synergy between the method and the outcome which leads to overall better results. While decision trees (once built) are easily understood and offer insights about the data, reinforcement learning may be able to offer a very human-like approach to how decision trees are built. Therefore, we can identify three main reasons of combining these methods.

- The black box model used by reinforcement learning can be alleviated by producing an intuitive and easy to understand outcome.
- The process of building decision trees is done in a human-like fashion (as offered by the naturally inspired reinforcement learning).
- The non-greedy nature of reinforcement learning offers an advantage over current greedy methods, potentially resulting in better performing decision trees.

The combination of using reinforcement learning to build decision trees is, to the best of our knowledge, a novel approach, which poses the following leading research questions.

- *Is it possible to build decision trees with reinforcement learning?*
- *What is a suitable approach (algorithm) to build decision trees with reinforcement learning?*
- *Can such an approach produce non-greedy results?*

These research questions emphasize the scientific nature of the thesis. Therefore, the goal is not to develop a tool which can be used productively but to explore the possibilities of this novel approach and evaluate its feasibility compared to other state of the art methods. A successfully designed algorithm which can compete with state of the art approaches would open up research for further improvements, utilizing the recently discovered power of deep learning and reinforcement learning.

To answer these research questions this thesis first discusses the background in chapter 2 Related Work. Having formed a basis of definitions and mutual understanding, state of the art research results for reinforcement learning and decision trees are discussed. The following chapter 3 Method introduces the methodology of how the new algorithm was designed and the concepts and numerous possibilities offered by the combination of reinforcement learning and decision trees. Subsequently, section 3.2 System describes the actual implementation, its parameters and their connection to the concepts introduced in the Method chapter. In chapter 4 Evaluation various parameter configurations are tested to assess the hypotheses stated by the research questions and the ideas introduced in the Method chapter. Afterwards, the results and their implications for the feasibility of the approach are discussed. Finally, chapter 5 Conclusion sums up the approach, reflects on the research questions and sheds light on possible future work.

2 Related Work

This thesis mainly touches two large topics: decision trees and reinforcement learning. While understanding these terms in a broader scope across multiple research fields can be beneficial, the focus lies on using and defining them within the context of computer science and, more specifically, machine learning. Both methods can be viewed as subtopics of machine learning and share certain ideas, which makes combining them an interesting endeavour. To define these terms, general concepts of machine learning will be explained first to provide a common basis. This basis will then be used to define decision trees and reinforcement learning. Once the background has been established, state of the art research will be examined briefly.

2.1 Machine Learning

In general it can be stated that machine learning is concerned with detecting patterns in data automatically. There is a wide area of application for machine learning and it already performs a multitude of meaningful tasks in our everyday lives (like spam filtering for emails or automatic face recognition). What sets it apart from traditional computer software is that the performed tasks usually are of such high complexity that manually defining a set of rules (or similar mechanisms) would be unfeasible. However, if a task cannot be programmed explicitly how is a computer able to perform it? The key is to empower a program to learn and adapt by itself, which is a key characteristic of machine learning [44].

The following sections provide understanding of key aspects of (machine) learning and explain its connection to the prominent terms *artificial intelligence* and *deep learning* which are particularly relevant for reinforcement learning. A more

technical perspective reveals a categorization of machine learning methods in supervised and unsupervised learning. This distinction is necessary to understand how input data shapes machine learning approaches and where reinforcement learning and decision trees fit in. Additionally, the machine learning application *classification* will be discussed as it is the practical task used in this thesis.

2.1.1 Learning

This section is mainly based on [44], where Shalev-Shwartz and Ben-David highlight the importance of understanding what learning encompasses. Its importance quickly becomes apparent, as the ability to learn and adapt is one of the most fundamental aspects of life in general. Learning enables humans (and, generally, animals) to adapt to changes on an individual basis. While some machine learning approaches try to replicate the way humans learn, others employ more artificial methods. However, certain fundamentals remain the same and they are especially interesting concerning reinforcement learning. For this reason, understanding the term *learning* is necessary.

A basic example of learning for animals is bait shyness (or poison shyness). Rats encountering unknown food will first consume only a small amount of this food to check whether it will have bad effects. If bad effects are experienced the rat will avoid food with the same taste and smell in the future. A similar machine learning task is avoiding spam emails. In this case the machine memorizes a large amount of spam emails (labeled by humans as such) and is able to compare new emails to the memorized ones. However, the spam scenario is an example of *learning by memorization* and lacks the ability to generalize. If the approach is not able to generalize from previous experience, it will not be able to correctly predict new unseen data, which is a vital ability of a learning agent.

To illustrate another essential ability, the rat example is extended. An experiment performed by Garcia and Koelling [14] showed that rats avoid food causing typical symptoms of poisoning but do not avoid food associated with electrical shocks (or other stimuli which are not logically related to food). Rats apparently have prior knowledge telling them that electrical shocks cannot be a consequence of consuming a certain food item. Such prior knowledge is essential for making correct decisions and dramatically impacts performance of machine learning algorithms.

In conclusion, two vital components of learning can be determined: generalization and prior knowledge. Prior knowledge should be viewed with caution as (in machine learning) it typically introduces restrictions and makes the application of an algorithm less flexible [44].

2.1.2 Categorization of Machine Learning Methods

To get a better understanding of what kind of learning task has to be performed in which context, a rough categorization of methods is provided in this section. Machine learning methods are able to solve a wide variety of tasks. It is logical to base the categorization of machine learning methods on the type of problem which has to be solved, or rather how the agent solving a task can interact with the environment in the most efficient way. A first categorization yields the following distinction:

- Supervised learning
- Unsupervised learning
- Semi-Supervised learning

Most of the machine learning algorithms can be assigned to one of these categories and are in fact built upon the principles they represent, as available data often dictates what is or is not possible (in terms of desired outcome and methodology) [44].

Supervised Learning

In an abstract way, in supervised learning the process of learning is aided by a supervisor which tells the agent whether the learned model fits the desired outcome [44]. For the example of a classification task, the supervisor can tell whether a sample has been classified correctly and give the agent corresponding feedback (aiding it in its learning). As it would be extremely inefficient to have a human supervisor, information about correct classification (labels) is contained within the training and test data.

Zhu [54] describes some of the pros and cons of supervised learning. If a dataset is labeled in a task-relevant way, choosing a supervised approach will generally

be advisable as it leads to better performance. But while the task of supervising the learning agent cannot realistically be performed by a human, labeling the initial test and training data is often done by humans. As this is tedious and time-consuming work or might not even be possible, using supervised learning is not always feasible.

Unsupervised Learning

As described in [19], unsupervised learning lacks any supervisors which can tell the agent whether the learned model fits the desired outcome. This typically means that datasets are not labeled. Without clear feedback, unsupervised approaches have to produce internal measures to gauge the learning progress. However, the lack of a feature to predict leads to the question of what is the desired outcome in the first place. Typically, unsupervised learning methods are used to discover interesting characteristics of a dataset, for pre-processing or visualizations.

A typical use case of unsupervised learning is clustering of data. Similar data points are clustered together which can lead to the discovery of different classes of data points. These clusters and their boundaries could then be used to perform classification on unseen data. Assigning meaningful labels to the clusters is not trivial, but possible (with limited outcomes). However, the results are more accurate with data labeled by external agents (for example humans). But simply clustering the data and performing a rough analysis of the outcome can be enough to gain new insights, making it useful for data analysis.

Due to these restrictions, some tasks cannot be solved with unsupervised learning alone.

Semi-Supervised Learning

While labeled data (in combination with supervised learning) is often necessary to solve certain tasks it is also hard to obtain [44]. Semi-supervised learning uses both labeled and unlabeled data to reduce the amount of necessary labeled data and to provide more data in general, which can lead to better generalization. However, using a semi-supervised approach (as opposed to supervised learning alone) does not always translate to better performance.

Semi-supervised approaches are a prominent example for *active learning* as well. In active learning, the agent does not rely purely on the data provided but may also ”ask” a supervisor for additional information [44]. Optimally, this bit of information improves performance significantly as it represents an especially tough situation or data point. In the case of unlabeled data, the agent may ask for labels.

This basic categorization has been widely adopted in the machine learning community and serves as a first decision point when considering which algorithm to use. However, not all methods can be assigned to a category cleanly as can be seen in the section about reinforcement learning.

2.1.3 Classification

Two of the most prominent and (arguably) important machine learning applications are *regression* and *classification*. As the practical examples in this thesis use classification as a task, it will be discussed briefly. Both regression and classification share the concept of mapping an input to an output, mostly via supervised learning [2]. While the goal of regression is to predict a quantitative (numeric) output, classification aims to predict to which class the input belongs (nominal). The predictor used in this thesis (decision trees) can be used for regression and classification tasks [19]. However, classification is more natural as a class can be assigned to each leaf of the tree (as opposed to assigning numbers of continuous nature to each leaf which naturally lacks precision).

There are numerous widely used applications for classification [2]. Some of them simply perform binary classification, where samples have to be assigned to one of two classes. An often occurring example is spam filtering, where an email is an input-sample and the classifier predicts whether it belongs to the class *spam* or *no spam*. A more business oriented use case is credit scoring, where potential customers are classified into high-risk and low-risk groups (in respect to offering them credits).

A more complex example is the classification of handwritten letters. The input for such a predictor are images (of handwritten letters) and the output predicts which letter a given image represents. Not only are there at least as many classes as letters in the alphabet, but each letter exists in an infinite amount of variations (due to

different handwriting). Not all of these variations can be present in the training data, hence the need for generalization. While it is not always possible to assess whether machine learning or humans perform better, generally no machine has managed to outperform humans in handwritten letter recognition yet [2].

These examples and various others like speech recognition or medical diagnoses highlight the usefulness but also the complexity of classification.

2.1.4 Deep Learning and Artificial Intelligence

Deep learning and *artificial intelligence* are terms which have gained popularity in the last few years due to breakthroughs in research (several among which happened around 2012 in object recognition, first among those the winner of the ImageNet competition 2012, Krizhevsky et al. [25] which is believed to have had the largest impact on the rise of popularity of deep learning [34]). The field of machine learning as a whole has been deeply impacted by this rise of popularity and significantly boosted the state-of-the-art in several areas. This leads to some confusion about the terms machine learning, deep learning and artificial intelligence, especially for readers not familiar with the field. Therefore, a distinction between them has to be drawn.

Deep Learning

Even though deep learning as an acknowledged research term has been around since 1986 [10], its viability relies heavily on high processing power. Due to this reliance it has only become popular more recently (with the rise of GPU calculations for deep learning). The reliance on processing power stems from the basic idea behind deep learning: Hierarchical representation of processing. Deep learning - also known as hierarchical learning - aims to process data in multiple layers, each layer responsible for extracting specific information [27]. A hierarchy of layers, each representing a simple concept, can be built this way. To model complex dependencies with such a hierarchy, it has to be several layers deep (hence deep learning) [16]. As each layer has multiple connections to other layers, the spanned network can become complex, and computationally expensive to learn. In summary, deep learning can be viewed as a machine learning technique which relies on deeply layered representations of simple concepts.

Artificial Intelligence

Artificial intelligence (AI) is a much more loosely defined term which constantly changes with the rise and fall of specific technologies. In fact, the definition of what is or is not artificially intelligent changes so often that this change in definition has its own name: *AI effect* [31]. This phenomenon describes the trend that AI problems are dubbed as *only computations* or *not intelligent* after they have been solved.

Even one of the most highly cited references on artificial intelligence by Russell and Norvig [43] does not define artificial intelligence clearly. Instead, they list multiple definitions across two dimensions: thinking versus acting and being human versus rationally ideal performance. These dimensions encompass the different understandings and expectations of artificial intelligence. For example, definitions which reside in the being human area focus on the concept of an artificial intelligence which models human thinking, acting, learning and behaviour. In contrast, a rationally ideal agent learns and performs the same tasks, but does not model humans. By definition, both agents are intelligent and artificially created.

Even though no clear definition can be determined, artificial intelligence research usually involves creating *intelligent agents*. An intelligent agent is an entity (for example a program) which perceives the environment and acts based on its observations [43]. This idea is directly mirrored in reinforcement learning, a prime example of artificial intelligence research.

2.2 Reinforcement Learning

The main machine learning method used in this thesis is reinforcement learning. The main ideas behind reinforcement learning originate in psychology research and were used computationally in several early works starting in the late 1950s. Research was soon neglected due to a lack of feasibility mainly caused by low computational power and confusion about the definition of reinforcement learning [48]. However, in recent years reinforcement learning was rediscovered and lead to some of the most impressive achievements in the field of computational intelligence. Examples include learning how to play ATARI games with human performance levels [33] and beating a Go world champion in a game of Go [45].

The idea behind reinforcement learning is based on learning as found in nature and humans: learning through interaction, trial and error. Interacting with our environment is one of the most fundamental and also most important ways to discover consequences and learn about what causes which effects [48]. More precisely, by performing different actions and observing the response of the environment, animals and humans can learn without the need for teachers. As a consequence, a mapping - called policy - between environment states, actions and corresponding responses of the environment is learned. If such a response is desirable, it is associated with a (positive) reward to strengthen (reinforce) the tie between state and action. A set of vital elements can be extracted from this basic idea which will be explained in the next section.

The following sections will explore the necessities, possibilities and limitations of reinforcement learning to give a comprehensive overview over the main ideas.

2.2.1 Elements

According to Sutton and Barto [48], several elements can be identified. These elements play a major role when viewing learning through interaction as a computational approach and form the basis of reinforcement learning. The elements are listed here and will be explained in more detail in the following paragraphs, mostly based on [48].

- An agent
- The environment
- Actions
- Rewards
- A value function
- A policy

Agent

The agent is the entity which performs the actions and influences the environment by doing so. In typical reinforcement learning problems, the agent does not possess prior knowledge about the problem or environment which marks its main difference to learning in nature.

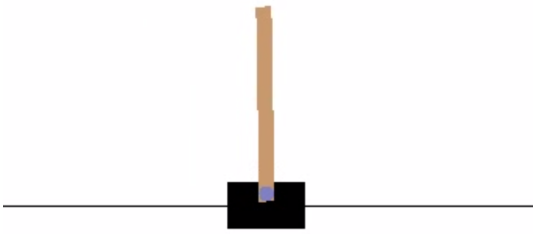

Figure 2.1: Cartpole. A pole balanced on a cart which has to be kept upright by applying force (left or right) to the cart. Based on the OpenAI learning framework [6].

Environment

The environment represents the environment the agent acts in and is a collection of all possible states the observed environment can take. Depending on the problem it can take many forms and might expose very little or an overwhelming amount of information. To emphasize the different forms environment can take, two polar examples are discussed: Cartpole and real life environments as viewed by humans.

A classical control problem in machine learning is Cartpole [3]. The goal of Cartpole is to keep a pendulum, which is balanced on a cart, upright while keeping the cart approximately centered. A graphical representation as provided by the learning framework OpenAI [6] can be seen in figure 2.1. The cart (black box) can be pushed left or right to influence the movement of the pendulum. In the classical form of the problem as described in [3] the agent only receives four pieces of information per time-step (representing the current state): position of the cart, velocity of the cart, angle of the pendulum and angular velocity of the pendulum.

Due to the continuous nature of each variable, the amount of different states is infinite, making the environment infinitely large. However, each state only consists of four meaningful one-dimensional variables, which will mostly stay within the same boundaries making generalizing over them fairly easy (as discussed by Geva and Sitte in [15]).

In contrast, real life environments as viewed by humans are very complex. Information humans receive about the environment is very detailed and consists

of multiple variables: high resolution vision, hearing, smelling, touching and others. Additionally, each of these variables is multi-dimensional or cannot even be modeled properly. If one wants to build machines with human-like decisions, all of these factors may play a role, highlighting the significance of this seemingly non-relevant example.

Both discussed environments have an infinite amount of states, but the composition of each state differs drastically.

Actions

The possible actions an agent can perform in an environment are typically referred to as action space and correspond to the decisions an agent can make. In the previous example of Cartpole, the action space is limited to two actions (apply force to the cart from the left or the right). While having a low dimensional action space can be helpful for the performance of an algorithm, it does not imply that the underlying problem can be solved easily. However, having a large action space typically makes learning more difficult, as exploration of different state-action pairs is more costly. Results from the ATARI games solved by Mnih et al. in [33] support this.

Rewards

The rewards or reward signals are provided by the environment to define how beneficial an action is. As the goal of a reinforcement learning agent is to reach the maximum (accumulated) reward, the reward signals define which goal should be achieved. For humans, examples for rewards are feeling pleasure (positive) or feeling pain (negative). It is important to note that the reward signal alone does not ensure the long-term success of the agent as a reward is always associated with one specific state-action pair. If an agent always chose to perform the most rewarding action (at the current moment), it could potentially end up in a sub-optimal position. Instead, it might be beneficial to accept some negative consequences early on to perform better in the long run. Such behaviour is called *non-greedy* and is a common characteristic of more refined reinforcement learning methods.

Value Functions

While rewards specify how beneficial a state-action pair is at a given moment, value functions hold information about the long-term impact of states. A loose description of the value of a state is how much future rewards can be expected after reaching this state. An agent should seek to reach high-value states to maximize the long-term benefit of actions. However, learning a value function can be highly difficult and is the main challenge of reinforcement learning.

Policy

Policies define the behaviour of the agent. They basically are a mapping from states to actions and provide an action for each state (the choice of the action can be stochastic). Policies can be considered the outcome of a reinforcement learning algorithm as they are the only element needed to determine the agent's behaviour.

2.2.2 Possibilities and Limitations

Reinforcement learning is one of the more general and intuitive approaches to learning. It offers a vast amount of possibilities and advantages over other methods. The most important ones, extracted from [48], are briefly discussed.

- Reinforcement learning is an intutive approach to learning as it is most likely similar to the way animals and humans learn (by trial-and-error).
- The distinction from supervised and unsupervised learning opens up new possibilities for solving challenging problems.
- It is possible to provide end-to-end solutions as opposed to only viewing isolated sub-problems, like most other approaches do.
- Reinforcement learning has a lot of natural synergies to other research fields like neuro-science or psychology. Both sides of the synergies have benefited greatly from each other so far.
- Due to the incorporation of temporal difference learning, reinforcement learning is generally non-greedy and can learn globally optimal policies.

Unfortunately, reinforcement learning also comes with limitations, first among which is the complexity of solving reinforcement learning problems itself. A brief discussion of limitations, based on [48], follows.

- Reinforcement learning algorithms and models usually come with numerous hyper-parameters which have to be fine-tuned.
- Reinforcement learning suffers from the difficult trade-off between exploration and exploitation. It is necessary to explore a great proportion of state-action pairs to gain a powerful value function. However, without performing already known beneficial actions, later states cannot be reached.
- The model of the value function is usually abstract and hard to interpret.
- The need for exploration usually causes comparatively long training times.
- Reinforcement learning could greatly benefit from integrating prior knowledge. While this is true for most machine learning methods, it is still hard to achieve.

It has to be noted that both reinforcement learning and decision trees are intuitive in their own way. While reinforcement learning closely models the way humans learn, decision trees are an intuitive representation of how to make decisions.

2.2.3 Supervised Versus Unsupervised

In the previous sections a distinction between different categories of machine learning methods was drawn. As already implied, assigning reinforcement learning to one of these sections is subject to some discussion. The general consensus is that reinforcement learning resides in its own category [44, 48]. Why is that?

- The goal of *supervised learning* mostly lies in building a model which directly fits the training data based on clearly defined labels. Generalization and recognizing patterns directly in the data are key characteristics of supervised learning. On the other hand, reinforcement learning learns a model which represents the value function, a mapping from states to actions based on rewards and trial-and-error. As rewards can be seen as a form of supervising, the confusion present especially in early days of reinforcement learning research [48] is understandable.
- The goal of *unsupervised learning* is to discover new patterns or structures in the input data without any kind of external supervisor (like labels or reward).

The distinction to reinforcement learning which relies on rewards is clear in this case.

2.3 Decision Trees

Decision trees are most commonly used for classification problems [44]. Other uses like regression are possible and some more creative use cases will be mentioned in section 2.4 State Of The Art. As stated in [19], in general, a decision tree splits the input space into different segments. Each of these segments contains data points matching a chain of decision criteria. These segments can be given meaning by assigning certain classes to them and stating: samples in this segment belong to class X. In case of regression, the mean (or other measures) of all samples in a segment can be seen as the regression output.

A simple decision tree for binary classification with two-dimensional data can be seen in figure 2.2. The accompanying data is plotted in figure 2.3. These figures show a simple example of how to classify mangoes into two categories: ripe and not ripe. While the chosen features and decision criteria are fictional, the logic behind the decision process can be easily understood just by looking at the decision tree in figure 2.2. A mango can be classified as ripe if it weighs more than 425 grams and is softer than a specific threshold. This example highlights one of the most important perks of decision trees (especially in the context of this thesis), the ease of interpreting them [44]. This ease of interpretation stands in direct contrast to most of the reinforcement learning models, which are black boxes and hence not easily understandable.

Unlike reinforcement learning, decision trees generally cannot be seen as a machine learning method. Rather, they are modelling the data, much like neural networks do in reinforcement learning. *Decision tree learning* on the other hand is a term which encompasses various methods and algorithms for building decision trees. Both sides, decision trees as models and decision tree learning, will be elaborated in the following sections.

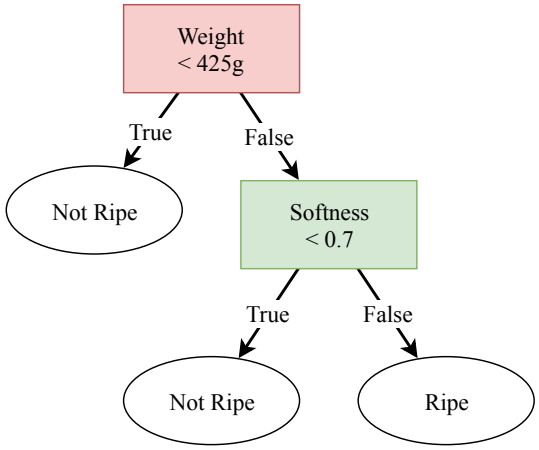

Figure 2.2: A decision tree for classifying mangoes. A simple decision tree with two splitting criteria which decides whether mangoes are ripe. The rectangular shapes show decision nodes, the elliptical shapes show leaves of the tree (representing classes). At each decision node, a sample (mango) either fulfills the criterion and follows the True path or does not fulfill it and follows the False path until a leaf (and hence decision) is reached. See figure 2.3 for data and the resulting regions and decision boundaries.

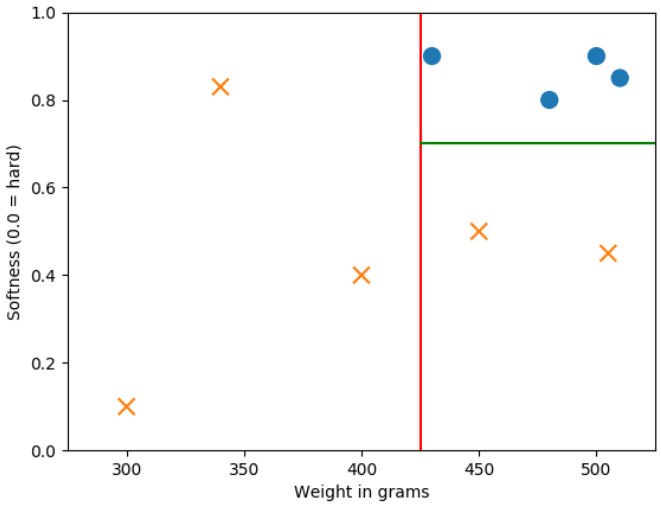

Figure 2.3: Mango data with decision boundaries. Nine mangoes are plotted according to softness and weight. Blue circles represent ripe mangoes, orange X's represent unripe mangoes. The decision boundaries segment the data into three regions corresponding to the leaves in the decision tree which can be seen in figure 2.2.

2.3.1 Decision Trees as Models

Decision trees do not originate from computer science and have been used for decision tasks (especially for business decisions) for a long time. In 1964, Magee [30] praised decision trees as an up-and-coming tool for business decisions. He highlights their power in complex decisions involving multiple consecutive decisions or events, stating that they illustrate alternative outcomes (as well as their likelihood) in an easily understandable manner. Even today, decision trees and tree structures in general are often used in planning problems in decision theory [26, p. 268].

In the context of computer science, decision trees as a model are considered one of the most useful predictive models for interpretation [19]. They have relevance in multiple fields, but most prominently machine learning and data mining. In contrast to decision theory, in the context of data mining and machine learning, decision trees do not have to be used for *making decisions*. Rather, they are used for modelling data. The resulting model can be used for making decisions (for example deciding whether a mail is spam or not) but can also be used for several other purposes like information extraction or simply to give interpretable insights into a data mining process [42].

2.3.2 Decision Tree Learning

Stating that decision trees can be used as (predictive) models raises the question of how such a model can be built. In the context of machine learning and classification and regression tasks, *decision tree learning* answers this question as it encompasses methods and algorithms used for building decision trees as predictive models. Shalev-Shwartz and Ben-David [44] describe the building of decision trees as a trade-off between training risk (performance) and size of the tree. Intuitively, large trees can easily reach high training performance as a large quantity of decision criteria can split the sample space into many small regions/clusters. The most extreme example maps every training sample to a leaf of a tree. In this example, the decision tree loses most of its advantages (like ease of interpretation) and most likely overfits. It is also easily contrived that small decision trees may not produce enough decision boundaries to effectively segment the sample space. However, finding an optimal decision tree (with minimum size) is an NP-complete problem. As a consequence, most methods rely on heuristics, mostly greedy learning, to build

decision trees. While greedy algorithms can produce good results the limitation of relying on local optima can lead to bad results in certain scenarios [7].

2.3.3 Possibilities and Limitations

According to [19] decision trees have several advantages opposed to other methods:

- As already mentioned, a key advantage is that they are easy to understand and interpret. This is especially useful in classification or regression tasks where it is necessary to offer transparency or to discover new facts about the data (for example when classifying real estate by value, it is vital to know why an object belongs to a certain class).
- Visualizing decision trees is straight-forward. The resulting visualization (especially for smaller trees) is easy to understand and offers quick insights.
- Decision trees work with both categorical and numerical input features. Additionally, data usually does not have to be normalized or prepared in other ways. As a consequence, only little preprocessing is necessary.
- Decisions based on decision trees feel more natural to some people, as they seem to model human decision making. This is particularly apparent when comparing them to other machine learning methods. In [44] it is also stated that human programmers produce code similar to the logic of decision trees when writing a predictor.
- Decision trees as models are efficient at handling large amounts of data (when already built).

On the other hand, decision trees have significant limitations as well:

- Decision trees typically perform worse than other regression and classification methods [19]. However, decision trees can also be uniquely suited for specific problems as can be seen in figure 2.4. While reasonably small decision trees fail to find a good decision boundary for linearly separated data, they outperform other methods in circumstances like the bottom right non-linear example.
- Robustness. Small changes in the training data can lead to significantly different outcomes [19].

- Decision tree learning is an NP-complete problem and has to rely on heuristics to achieve results in a reasonable amount of time [44].
- Decision trees are prone to overfit. Several countermeasures exist, but their performance is debatable [18, 46].

Due to these limitations, decision trees often cannot compete with the results of other methods. However, some of the limitations can be overcome by employing methods like bagging. Several methods will be explained in the following section.

2.3.4 Overcoming Limitations

Some of the most common strategies to boost the performance of decision trees include *tree pruning*, *bagging*, *random forests* and *boosting*. The idea behind these methods will be explained based on James et al. [19]. These methods are relevant for this thesis as they can likely be utilized well when growing decision trees with reinforcement learning.

Tree Pruning

Tree pruning is mostly used to counter overfitting. After initial training, decision trees are often large, complex and tailored to closely to a training set. The idea behind tree pruning is to reduce the size of an initially large tree by taking a subtree (or pruning away other branches) with similarly good performance but greater generalization. As examining each possible subtree is computationally unfeasible, a more sophisticated method has to be employed, the most popular of which is cost complexity pruning, originally introduced 1984 by Breiman et al. [4]. Tree pruning not only increases performance on unseen samples but also leads to better readability of the trees (due to their smaller size).

Bagging

Bagging is not exclusive to decision trees but describes a general method of reducing variance between outcomes of multiple training runs (with differing training data). Decision trees are fairly unstable and prone to changing significantly based on the selection of training data (and hence having a high variance). This limitation makes

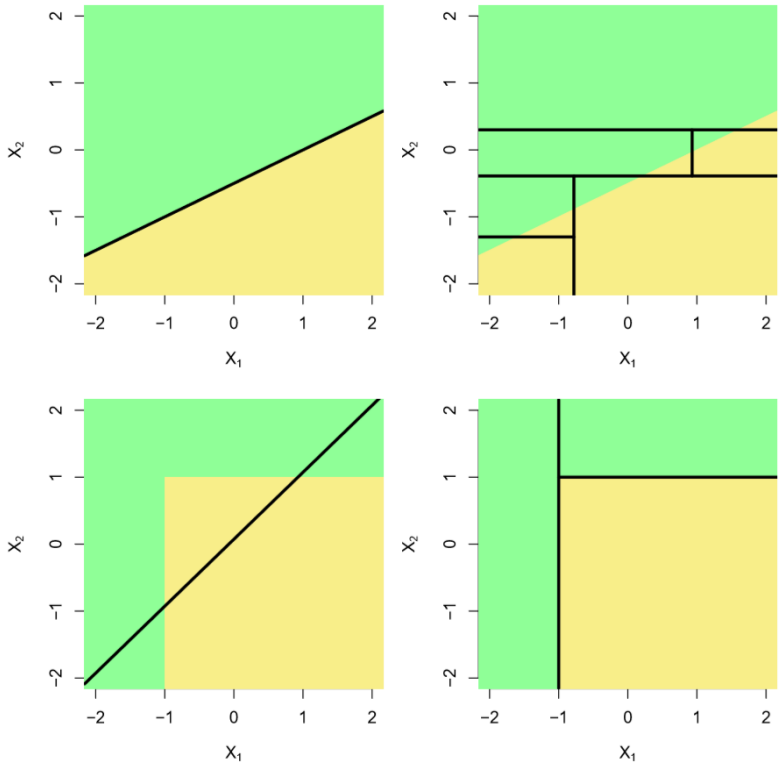

Figure 2.4: Linear versus decision tree boundaries. Two dimensional data with perfect linear separation (top row) and non-linear separation (bottom row). The black lines show exemplary decision boundaries for linear estimators (left column) and decision trees (right column). Both linear and decision tree approaches perform well in one scenario, but badly in the other. [Image extracted from [19] under © James, Witten, Hastie & Tibshirani. Used under the terms of Austrian copyright law: §42f]

the application of bagging ideal for decision trees. The idea is to reduce the variance by averaging over multiple outcomes produced by different training sets. As the amount of training data is usually limited, choosing incomparable (non-overlapping) training sets is often not possible. To counter this problem, bootstrapping is used to produce several training sets which can contain duplicate samples from the training data. While the decision trees resulting from these training sets have high variance among each other, the average over the trees (in case of regression) or resulting majority vote (in case of classification) is typically fairly low. However, it has to be noted that bagging comes at the expense of interpretability as not only one tree is produced but multiple (typically more complex) trees.

Random Forests

Random forests build on the same principles as bagging. Several decision trees are trained based on different bootstrapped training sets and the average over these trees produces the prediction. However, random forests not only reduce the overall variance but also decorrelate the trees. In the scenario of a small percentage of dominant features (for example three out of twenty features), trees produced by bagging will be highly similar (correlated) as choosing the dominant features as splitting criteria is the preferred option for most training sets. Hence, all trees will be biased towards the dominant features. Random forests aim to eliminate correlation between trees by restricting the choice of splitting criteria for each tree node. When building a tree, for each split, the algorithm randomly chooses a subset of features to be considered, eliminating most of the features as possible choices (among which might be dominant features). When building a sufficiently large amount of trees, this restriction benefits the overall performance.

Boosting

Boosting describes a procedure which is not unique to decision trees. However, for decision trees, it is once again based on the idea of producing multiple trees and combining them to a final predictive model. However, in contrast to bagging and random forests, no bootstrapping is used to produced different training sets. In fact, only one training set is used which is modified incrementally. The basic idea behind boosting is to build on previously learned decision trees and modify the

training data in such a way that problematic splits are brought to the foreground. The process is artificially slowed down to allow multiple trees to fit similar data, resulting in another difference to bagging and random forests: the size of the trees. It is often sufficient to produce trees with a small amount of splits (sometimes just one). Unfortunately, small trees do not stop boosting from suffering from the same disadvantage as bagging and random forests: a lack of interpretability.

2.4 State Of The Art

The current state of the art in the research fields reinforcement learning and decision trees will be discussed in the following sections. Some historically significant publications will be examined as well to further the understanding of the topics. To the best of our knowledge, the main topic of this thesis - using reinforcement learning to grow decision trees - has not been discussed in any research and is a novel idea.

2.4.1 Reinforcement Learning

Publications in the field reinforcement learning are frequent due to its recent rise in popularity. Simply performing a search on Google Scholar[1] using the keyword *reinforcement learning* yields hundreds of results per year. This section provides an overview over historically important and more recent state of the art publications.

Skipping publications from the earliest stages of reinforcement learning, the first paper in this section by Sutton [47] highlights the formal introduction of one of the key success factors of reinforcement learning: temporal difference (TD) learning. In contrast to traditional learning methods which only try to learn by comparing final predictions and outcomes, TD learning learns by comparing successive predictions along the timeline to the outcome. Sutton shows that TD learning generally produces more accurate results with less computational effort in multi-step prediction scenarios (prediction and outcome are separated by multiple steps). The learning rule $TD(\lambda)$, where $\lambda \in [0, 1]$ is introduced. λ controls how significant temporally far removed states are. $TD(1)$ is identical (in outcome) to supervised learning and

[1] https://scholar.google.com/ (Accessed on: 2018-10-13)

$TD(0)$ is very short-sighted and only considers the next state within a sequence of states. Sutton proves that $TD(0)$ and $TD(1)$ asymptotically converge to the minimal error with a rigorous mathematical proof which goes beyond the scope of this section. The ability to reach optimal results with TD learning is proven mathematically and empirical results show it reaches these results faster than traditional methods (in the examined scenarios). In conclusion, TD learning is an important foundation for modern reinforcement learning, as it models the temporal connection between far removed states.

Another influential paper by Watkins [51] describes Q-Learning, which was originally formulated in his PhD thesis [52] in 1989. Q-Learning forms the basis of numerous modern reinforcement learning algorithms (for example Deep Q-Networks [33]). Building on the idea of TD learning (described in the previous paragraph), Q-Learning describes an algorithm which learns an optimal policy by estimating Q values. Q values represent the value of an action a given the state x and are defined as

$$Q^{\pi}(x,a) = R_x(a) + \gamma \sum_{y} P_{xy}[\pi(x)]V^{\pi}(y) \tag{2.1}$$

where π is the policy, $R_x(a)$ is the reward of performing action a in state x and γ is the weight for the sum of future rewards when following policy π. The sum over all states y consists of the probability P_{xy}, going from state x to state y based on the action chosen by the policy times the value $V^{\pi}(y)$ of the following state y. Since the current reward and future rewards are considered, choosing the action a which maximizes $Q^{\pi}(x,a)$ is the optimal action to perform in state x. The challenge lies in learning a model for $Q^{\pi}(x,a)$, which is solved by a straightforward algorithm described in the paper.

Up to the point of the publication of [33] by Mnih et al. in 2015, reinforcement learning could only be used reliably for problems with small states, eliminating most real world applications. The introduction of Deep Q-Networks (DQN) revitalized reinforcement learning research by proving its sole prowess to solve complex problems in the form of ATARI 2600 games and can be considered a state of the art algorithm. Mnih et al. present several novel ideas to address the shortcomings of previous reinforcement learning algorithms. To handle complex high-dimensional states, deep neural networks are employed. To handle possible divergence and instability of neural networks, two adaptions to standard Q-Learning are employed:

Experience replay to battle correlation among state sequences, and a target network as source for error calculations, once again reducing correlation within the error calculation. Further techniques used to increase performance are minimal preprocessing of states (images from the games), states consisting of four consecutive frames and minibatch-learning. In conclusion, DQN manages to learn ATARI games at human-level performance solely based on the pixel values of the game with an end-to-end reinforcement learning solution without injecting any prior knowledge.

So far only reinforcement learning approaches for discrete action spaces have been discussed. Gu et al. [17] introduce a state of the art reinforcement learning algorithm for continuous action spaces. The algorithm, normalized advantage functions (NAF), introduces two key novelties. Firstly, Q-Learning is modified to produce continuous actions. Secondly, the representative power of model-based learning is used to accelerate generalization of the model-free approach. Traditionally, reinforcement learning algorithms for continuous action spaces require a dedicated model for the policy (policy gradient) and, if the value functions should be integrated, a second model for the value function (actor-critic). NAF represents an approach with only one network which represents value function and policy. This approach is more elegant, easier to implement and generally more efficient. The adaption to Q-Learning is the splitting of the Q values into two summands

$$Q(x, a) = A(x, a) + V(x) \tag{2.2}$$

where x is the current state, a is the performed action, $A(x, a)$ the *advantage* and $V(x)$ the value of the state x. The advantage can be viewed as the value the action a has compared to other actions a^0. This representation allows for analytical solutions for maximizing continuous actions. Learning can be further accelerated by incorporating knowledge from external models, which inject complete state-action-reward triples into the replay memory with so-called imagination rollouts. In conclusion, [17] introduces NAF, a Q-Learning based algorithm for continuous action spaces which is elegant and performs well.

The final paper [13] of this section combines trees and reinforcement learning, but in an entirely different way than this thesis. Farquhar et al. present the aptly named algorithms TreeQN and ATreeC (word plays on the popular DQN [33] and A3C [32] algorithms). The goal of the incorporation of trees is to refine conventional estimates of the value function and Q values (via neural networks) with look-ahead

trees. To achieve this, at each step at state s, a look-ahead tree of configurable depth is used which considers future state-action-reward triples. Predicting such triples makes the estimates for the current state more accurate and meaningful. For the adaption of DQN, this enhanced estimate of the Q values directly replaces the traditional deep neural network. For A3C, the tree architecture is used for predicting Q values in the policy (actor) network with the same implementation. In conclusion, enhancing DQN and A3C with on-line look-ahead trees for more accurate predictions of Q values boosts their performance in selected ATARI games and other discrete action space tasks.

2.4.2 Decision Trees

Decision trees are an empirically well understood topic with large amounts of publications. However, the current frequency of publications is not on the same level as reinforcement learning as decision tree learning has reached a high degree of maturity. Some of the most influential publications will be discussed here as well as state of the art algorithms.

Using decision trees for regression or classification was an idea which came up surprisingly late (in the 1970s [9]) and was spearheaded by Leo Breiman who published his findings in 1984 [5]. This book extensively covers decision trees and produced some key ideas about building decision trees, resulting in the basic algorithm now called CART (Classification and Regression Trees). Like most methods for building decision trees it is greedy and relies on recursive binary splitting. Each split is optimal in respect to information gain. However, due to the greedy nature, the resulting tree may not be optimal. The key ideas introduced are:

- The choosing of splits is based on probability theory. In case of classification, gini impurity can be used as a measure. Regression requires other measures like the sum squared error. The gini impurity of a node N from a dataset with C classes is given by

$$g(N) = 1 \quad \sum_{i=1}^{C} \left(\frac{jsamples\ 2\ C_i j}{jsamples\ 2\ Nj} \right)^2 \tag{2.3}$$

  where the sum represents the fraction of samples belonging to class C_i across all samples in node N squared.

- A variety of stopping criteria are discussed and their impact highlighted. A simple but powerful criterion is a minimum number of samples per leaf.
- Trees can be built large and then pruned to be smaller and more readable. The key factor is to use cross validation for performing tree pruning.

An algorithm introduced shortly after CART and based on very similar ideas is Iterative Dichotomiser 3 (ID3) [36]. Like CART, ID3 performs recursive binary splitting with various small differences. The measure to evaluate splits is information gain and therefore based on entropy (however, there is little to no theoretical and practical difference between using information gain and gini impurity [38]). The information gain in a node T according to split a is given by

$$IG(T, a) = H(T) \quad H(Tja) \tag{2.4}$$

where H is the entropy. While information gain is used as the measure to decide how to perform splits, an additional restriction is employed: In any given path in the tree, each feature may only occur once. This severely limits the size but also the expressive power of the tree.

Due to severe limitations of ID3 (like inability to split continuous features) Quinlan developed successors called C4.5 [37] and C5.0, both of which are distributed under a commercial license. Even though C4.5 was published in 1993, it still offers state of the art performance and is one of the most widely used algorithms for building decision trees [53]. Quinlan provides several improvements over ID3 in [37]. Most noticeable among which are:

- Handling of continuous features by dividing them into discrete intervals. Due to this division, they can be used like discrete features internally.
- The restriction of only selecting a feature once per path in the tree is lifted. To counter overfitting and large trees, pruning is used.
- Features can be preprocessed in two particular ways. Certain features can be treated as high-cost features, indicating that they should preferably be selected at deeper nodes. Additionally, values can be marked as not available which leads to disregarding them for calculations. This significantly boosts performance for incomplete or noisy data.

Possibly one of the highest impact changes is the adaption of the feature selection for splits. The selection in ID3 suffers from a bias towards features with a wide diversity of discrete values, which would require a split for each value. As such splitting is not

beneficial for the overall outcome, it has to be avoided. C4.5 introduces weighting to counteract such values

$$gainratio(T, a) = \frac{IG(T, a)}{\sum_i^n H(\frac{jT_ij}{jTj})} \tag{2.5}$$

where T is the node split by a and the sum represents the entropy of splitting T into n (number of classes) parts.

While there have been only few developments for building single decision trees, ensemble methods such as boosting have made progress more recently. The final paper in this section concerns such a method: rotation forests [41]. Rodríguez, Kuncheva and Alonso address the trade-off between accuracy and diversity which is present in nearly all ensemble method. It describes the common problem that overall performance of ensemble methods relies on diverse and accurate predictors. However, diversity implies straying from optimal solutions and hence hinders accuracy. The concepts behind bagging and random forests have already been explained previously. Rotation forests build on both ideas and add Principal Component Analysis (PCA) as means for achieving higher diversity and accuracy. It has been shown that PCA (for dimensionality reduction) is not well suited for classification in any part of the pipeline. In rotation forests, an ensemble method, PCA is used for the sole reason of creating greater diversity. PCA is performed on a randomly selected subset of features and samples to extract the principal components used for transforming the data. The transformed data is then used as training data for building decision trees. The produced trees generally have higher accuracy and diversity than comparable methods which translates into slightly better overall performance. This does, however, come with the cost of even further reduced readability of the trees.

3 Method

This chapter consists of two main parts. First, the methodology which discusses the theory and ideas behind the design of the algorithm and how these ideas were born is explained. The Methodology section covers all the components of the algorithm and discusses various approaches to each component and the algorithm as a whole. After the overall components are defined, the System section describes how they were implemented and how the various ideas can be mapped to actual testable parameters.

3.1 Methodology

In this section, the methodology of how the algorithm introduced in this thesis was developed is covered. After explaining the basic idea, the various sub-elements and their challenges are discussed, finally resulting in a complete algorithm. As the development of the algorithm was accompanied by trial and error, not only successful but also failed ideas are mentioned.

The idea of combining decision trees and reinforcement learning to perform classification is lead by two main motivators. Numerous state-of-the-art methods for classification use black-box models to perform predictions. As transparency is an important aspect of classifiers, the intuitive interpretability offered by decision trees is an attractive approach to classification. However, state-of-the-art decision tree building algorithms conquer the NP-hard problem employing greedy methods. Additionally, peak performance is only reached when using methods like bagging, which considerably reduce the interpretability of decision trees. In summary, the two main issues with decision tree classifiers are greediness and lack of interpretability.

Both of these issues can be conquered with reinforcement learning. As reinforcement learning is - by its nature - non-greedy, decision trees can be built in a non-greedy manner. This non-greediness, in turn, leads to more optimal trees which offer higher performance without the need for bagging, maintaining the interpretability of a single decision tree. Additionally the black-box reinforcement learning model is used to build a white box model. So in theory, the combination of decision trees and reinforcement learning offers results greater than the sum of its parts.

This reasoning (and simple curiosity about whether it could work) lead to the idea of combining decision trees and reinforcement learning. However, the novelty of this approach provides a huge number of possibilities which need to be explored. To provide the development workflow and this thesis structure, the main elements of the algorithm (and the challenges and possibilities they offer) are explained in the following sections. It is important to note that not all mentioned possibilities can be explored in great depth, as the scope of such an endeavor would be too big.

For all the elements, the basic scientific approach for designing and assessing them is as follows.

1. Make reasonable hypotheses and test them. It is vital that results can be compared easily and objectively.
2. Start with the simplest possible implementation and dataset to assess the hypothesis.
3. If possible make switching between different variants possible via parameters.
4. Reflect, evaluate, make conclusions and adapt where necessary.
5. Increase the complexity of the problem.
6. Got to step 2.

3.1.1 The Overall Algorithm

The overall algorithm has the largest impact as it describes the structure of the algorithm. While specific elements of the algorithm (like the reward function) can impact the performance strongly, the general structure answers important fundamental questions by making certain assumptions. The main questions are mainly concerned with which part of the algorithm handles which problem and

generally define how the tree is built. A brief listing of the most important aspects of the overall structure reads as follows.

- How is the tree built?
- Is the tree built iteratively, recursively or in one step?
- If it is not built in one step, where are new nodes attached?
- What are the responsibilities of the reinforcement learning output?
- What are the responsibilities of the environment?

These fundamental questions shape the structure of the algorithm. As each variation in the overall algorithm changes the other elements in a significant way, only a small amount of overall variations are examined. For example, the decisive power of the reinforcement learning algorithm is tried with different variants while the way in which the tree is built (iteratively) does not change. Making different assumptions here (for example generating trees in a single step) would result in such a distinctly different algorithm that it would form an additional dedicated research topic.

The most basic version of the algorithm is given in algorithm 3.1. This representation specifically leaves out details and could be applied to any kind of assumed answers of the above questions. It only defines the following aspects:

- A dataset is the input of the algorithm.
- A decision tree should be built to classify this data.
- The environment is defined by the dataset and some parameters θ_E. It provides states (s), rewards (r) and whether the episode is done ($done$) for a given action a.
- A reinforcement learning algorithm is used to built the decision tree. It is defined by its parameters θ_{RL} and learns a policy π which is responsible for choosing actions (a) based on the current state (s).
- Learning happens for a set number of episodes N_{max}.

Building The Tree

The algorithm can be extended with the following assumptions, which shape the structure of actions and states.

- The tree is a binary tree.
- The tree is built iteratively.

Algorithm 3.1 Basic Algorithm

Require: Dataset D
Ensure: Decision tree classifier over D
  Initialize Environment E ((D, θ_E)
  Initialize reinforcement learning algorithm with parameters θ_{RL}
  n_{ep} (0
  while n_{ep} N_{max} **do**
    s (empty state
    while stopping criterion not reached **do**
      a ($\pi(s)$
      $(s, r, done)$ ($E(a)$
      Perform the reinforcement learning algorithm learning procedure to update the policy π
    end while
  end while

- The environment decides where new nodes are attached.

As a consequence, the following can be stated about actions and states: An action is responsible for appending a single node to the tree built in a given episode at the position provided by the environment. States have to represent the current position in the tree, be it via structural information about the tree, data available at this point or any other representation.

Since the environment provides the position of where new nodes should be attached to the tree, a mechanism for doing just that has to be defined. If possible, this mechanism should not influence the greediness of the algorithm or worsen the interpretability of the tree while still providing sensible locations. Various possibilities can be identified:

- Choosing new node positions at random.
- Building a full binary tree (for example by building level after level, from left to right).
- Choosing positions based on a measure.

Obviously, building a full binary tree is not optimal, as the assumption of optimal decision trees being full binary trees intuitively does not hold. The underlying problem is illustrated by the decision tree seen in figure 3.1. If we assume that this

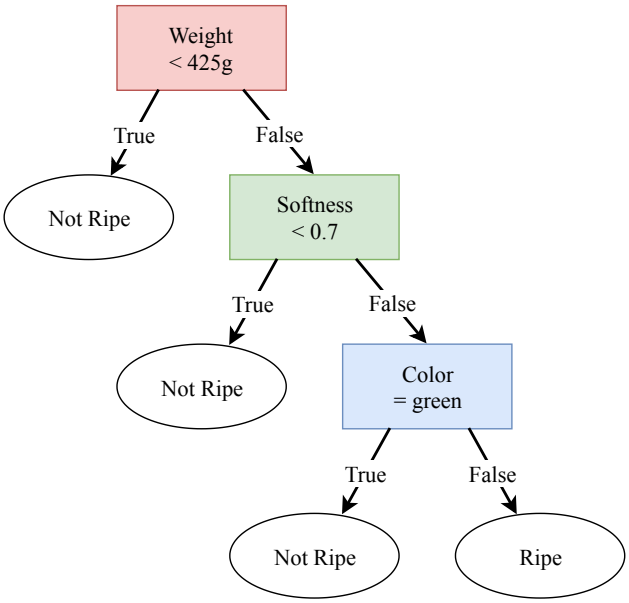

Figure 3.1: A (fictional) deeper decision tree for classifying mangoes. Rectangular shapes are nodes, elliptical shapes represent leaves.

tree is close to optimal (with a node count of three, excluding leaves), a full binary tree with comparable performance would have $n_{nodes} = 2^{depth} \quad 1 = 2^3 \quad 1 = 7$ nodes. A significantly larger tree. For trees with more depth, the disparity becomes even greater.

Choosing new node positions randomly would probably work but could result in unnecessarily large trees as well. However, the third alternative - choosing positions based on some measure - offers the benefit of choosing consistently meaningful positions. Additionally, an appropriate measure does not even influence the greediness of the algorithm.

Fortunately, traditional decision tree algorithms provide metrics which fulfill these requirements. CART [5] uses a metric called gini impurity (or gini score) to evaluate the information gain provided by a split. In CART, it is used to choose the split offering the highest information gain and is mainly responsible for the greedy behaviour of the algorithm. However, it not only describes information gain but can also be used to the represent the purity (with respect to class distribution) of the data present in a node (after filtering the data through the tree up to this node). ID3

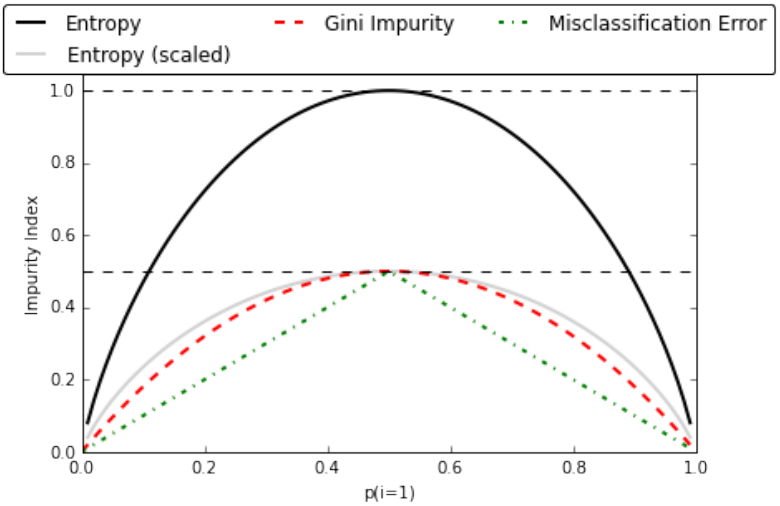

Figure 3.2: Comparison of decision tree information metrics. Gini impurity and scaled Entropy are nearly identical and are maximized at highest uncertainty ($p = 0.5$). Missclassification error is of limited use due to a bias towards unbalanced datasets. [Image extracted from [39] under MIT License, published in [40].]

and its successor C4.5 use a similar metric simply called information gain which is based on entropy. As there is no practical difference between gini impurity and entropy (in the context of decision trees, see figure 3.2) but gini impurity is slightly faster to calculate [38], gini impurity is used in this thesis.

Gini impurity in general calculates as given in equation (3.1) where C represents the number of classes, C_i class i and s is a data sample. The probability $P[s \ 2 \ C_i]$ describes the probability of an arbitrary sample s belonging to class C_i.

$$g = \sum_{i=1}^{C} P[s \ 2 \ C_i](1 \quad P[s \ 2 \ C_i]) = 1 \quad \sum_{i=1}^{C} P[s \ 2 \ C_i]^2 \tag{3.1}$$

So this formula describes the probability of a sample belonging to class C_i being wrongly labeled as any other class C_j where j 6= i, summed over all classes. It has several beneficial properties [40, 5].

- It is zero when all samples belong to a class C_i.

- It is independent of the number of samples (it is normalized by the use of probabilities).
- The function is nearly identical to scaled entropy (see figure 3.2) and has similar properties like a maximum function value for maximum uncertainty.
- It is not affected by unbalanced datasets.

In the context of decision trees and the data available at specific nodes, the formula can be adapted as seen in equation (3.2). For a given node N the formula is identical to equation (3.1) with the restriction that only samples available in this node are considered. Additionaly the probability $P[s \ 2 \ C_i]$ is given as the fraction of the number of samples belonging to class C_i and the total number of samples in node N.

$$g(N) = 1 \quad \sum_{i=1}^{C} \left(\frac{jsamples \ 2 \ C_i j}{jsamples \ 2 \ Nj} \right)^2 \tag{3.2}$$

Due to the properties of gini impurity, selecting where to attach new nodes to the tree is straightforward. An algorithm describing the procedure is given in algorithm 3.2. The idea is to choose the leaf with the highest uncertainty (which is equal to highest gini impurity). As, most of the time, it does not make sense to introduce a split when there is a small number of samples present, an additional condition d_{min} (minimum samples per leaf) is introduced. This approach does not introduce greediness as it does not influence the decision of the split performed at the chosen position. Furthermore, the node with maximum impurity causes bad performance so it has to be split anyway. Since leaves do not influence each other (the lie on different paths in the tree), the order in which new nodes are attached to leaves is irrelevant.

Balance Between Reinforcement Learning and Environment

So far, the questions *Is the tree built iteratively, recursively or in one step?* and *Where are new nodes attached?* have been answered. This leaves the question about the balance of responsibilities between environment and reinforcement learning. Since the reinforcement learning part is responsible for the actual learning, this question of balance also determines the amount of freedom the learning agent is given. Responsibilities of the environment are restricting the freedom of the learning

Algorithm 3.2 New Node Position

Require: Dataset D, Tree T
Ensure: New node gets added where most beneficial.

```
d_min ( appropriate minimum samples per leaf
if T is empty then
  return empty
else
  Classify D on T
  l_max ( arbitrary leaf(T)
  for l ( leaf(T) do
    if gini(l) > gini(l_max) and j(d  D) 2 lj  d_min then
      l_max ( l
    end if
  end for
  return l_max
end if
```

agent by making certain assumptions about the way the tree is built or the nature and amount of information given to the learning agent.

In theory, the more power and information the learning agent has, the better the outcome since the whole picture is seen and can be modeled by the agent. However, in practice, more information and more power also mean higher complexity, more difficulty in learning (and therefore longer learning times) and possible divergence of the learning process. To achieve good results in a reasonable amount of time, trade-offs have to be made.

Two assumptions which restrict the learning agent have already been mentioned (building the tree iteratively and leaving node attachment position to the environment). Further restrictions are discussed in the following sections.

Other parts of the algorithm are discussed in their respective subsection. The structure of states and the amount of information they contain are discussed in subsection 3.1.2 State Representation. Reward functions and their impact and implications are explained in subsection 3.1.3 Reward Function. The various challenges and adjustable parameters of reinforcement learning algorithms (in the context of

this thesis) are described in subsection 3.1.4 Reinforcement Learning. In subsection 3.1.5 Definition of Done, conditions for terminating the algorithm are discussed. And finally, subsection 3.1.6 Post Processing elaborates methods of post-processing the results obtained from the basic algorithm to modify performance or interpretability.

3.1.2 State Representation

Choosing appropriate state representations is a challenging task as the number of possible representations is high and the implications they produce are extensive. As the state is the input of the reinforcement learning policy which produces actions $\pi(s) = a$, the state representation controls the nature and amount of information available to the reinforcement learning agent.

In this thesis, three categories of state representations were identified. While combinations of the three categories are possible, doing so was deemed out of scope for this thesis.

- Data states. States directly contain samples from the dataset.
- Extracted feature states. States contain metrics calculated from features or samples in the dataset.
- Tree states. States represent the current structure of the tree which is being built or the node which should be attached.

The following subsections will examine each category in more detail, describing the idea and potential problems, advantages and disadvantages.

Data States

In case of data states, a state only consists of samples taken directly from the dataset. The idea is that the learning agent sees the data present at a given node and chooses a new split based on it. As the data is naturally filtered through the tree, the samples present at a node are an accurate representation of the current state, as they implicitly contain information about previous splits. This approach can be compared to a human given a number of samples and choosing an appropriate split

based on it. From a human perspective, it is already apparent that this task is not trivial.

In more detail, given a dataset D with n samples and m features (including labels), a state containing all samples and features simply is an $n \quad m$ matrix with each row representing one sample and each column one feature. Consequently, the state has a size of $s_{size} = nm$. For large datasets this can be problematic.

The state contains the full dataset in the root node only. The first split happening in the root node splits the data into two parts. For a new node attached as the left child of the root (binary tree), the state only contains samples which fulfill the splitting criterion in the root node. Consequently, the number of samples present in this node is smaller than n. From here on, the number of samples present in a node is denoted as n_i where i corresponds to the index of the node starting with $i = 0$ for the root node. Consequent indices calculate as $i = 2i_{parent} + (1 \text{ if } left \text{ else } 2)$ and correspond to a breadth first node counting of a full binary tree. An examplary tree augmented with indices and data counts (based on table 3.1) can be seen in figure 3.3.
Since $n_i < n \; 8 \; i > 0$, but the state (input of the neural network used as the reinforcement learning model) must always have the same size, a way to generate states from samples which do not fit the fixed state size has to be provided. There are several options to do this.

- Always provide all samples but provide an additional column which represents whether a sample is active at the current node. For the data in table 3.1 this would mean adding an extra column with the value 1 if the sample is active and 0 otherwise.
- Always provide all samples, but fill values of samples which are not active at the current node with a set value. For the data in table 3.1 this would mean setting all values of currently inactive samples to (for example) 1.
- Set the state size to any value greater than zero but smaller or equal to n and only provide currently active samples. As the number of samples may be smaller than the chosen state size, the remaining slots have to be filled. This can be done either by one of the above two options (padding or contained column), or by randomly drawing samples from the pool of active samples and putting them in the remaining slots (potentially resulting in samples being present multiple times in a single state).

Age	Weight [kg]	Height [cm]
15	40	163
25	97	170
32	73	180
10	18	120

Table 3.1: Fictional data showing age, weight and height of humans.

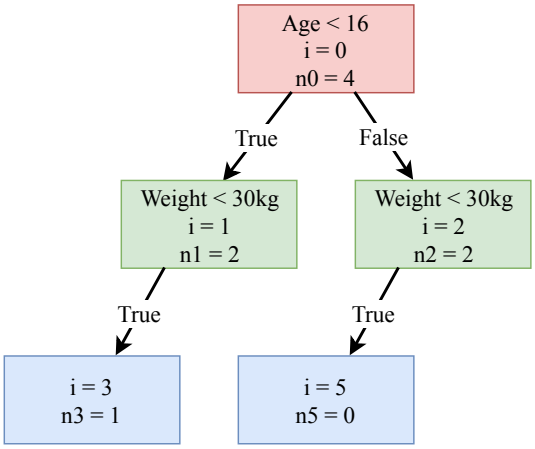

Figure 3.3: An incomplete decision tree for classifying obesity. Each node contains the index i of this node and the number of samples n_i which are active. The sample count is based on the data given in table 3.1.

- With networks like recurrent neural networks it is possible to feed samples one by one, eliminating the need for any of the above methods.

All of these options have potential advantages and disadvantages. Including all samples in the state leads to states of size $s_{size} = nm$ or in case of the additional column $s_{size} = n(m + 1)$. As already mentioned, this is unfeasible for large datasets. However, the natural counter-measure of reducing the state size to not include all samples limits the information the learning agent sees, potentially removing vital information. But learning in batches, relying on random sampling and good generalization might eliminate this issue.

Data states can be modified in several other ways. For example, the state matrix can be flattened from its natural $n \quad m$ shape to a vector of shape $nm \quad 1$, simplifying the input dimensionality at the cost of potentially losing structural inter-dimensional

correlations. Another item which has not been discussed so far is whether the labels should be included in states. In theory, labels can be excluded as the model can learn to assign actions to certain data states without the need for labels. This has the advantage of removing a column, making the state more manageable. However, labels provide vital information, probably outweighing the upside of leaving them out.

Some potential advantages of data states are:

- The reinforcement learning agent almost sees the full picture.
- The learned model may be applicable to other datasets with reduced learning effort due to pattern recognition directly in the data.

On the other hand, several potential disadvantages can be identified:

- High state complexity. Large networks with comparatively long training times might be a consequence.
- Some trade-offs between state size and information contained have to be made.
- The state of the tree is not explicitly known to the learning agent.

Extracted Feature States

In the case of extracted feature states, states consist of metrics calculated from the active samples of a node. For each feature multiple metrics can be calculated. Examples for such metrics are mean or variance but could also include more sophisticated measures (like principal component analysis) tailored to the classification task. Additionally, metrics across multiple features can be included (like the node purity). This approach, when transferring it to a human perspective, is analogous to a human only being given metrics like mean and variance about some data samples and choosing a beneficial split based on it. This comparison highlights the potential difficulty of this task, as the given information is very limited.

More specifically, for a dataset D with n samples and m features (including labels), state size is independent of n as no samples or information per sample are included. When calculating t metrics per feature and u additional metrics, the state has a size of $s_{size} = mt + u$. The state size being independent of n of is a useful property as it

prevents the issue of dealing with different numbers of active samples per node (as discussed in subsection 3.1.2 State Representation).

Potential advantages of using extracted feature states are.

- Low state complexity.

The potential disadvantages outweigh the advantages heavily.

- The learning agent does not see the full picture. It is constrained by external preprocessing.
- The simplification of samples produces the risk of losing the representative power of states necessary to map actions to states.
- The learning agent is constrained in its freedom. The agent should be able to find its own internal metrics (in the model) instead of relying on external sources to choose them.
- The state of the tree is not explicitly known to the learning agent.

Tree States

So far, approaches relying directly on the data samples or metrics about the data have been discussed. Data states allow the learning agent to see the full picture at the cost of large states with possibly complex inter-dimensional dependencies while extracted feature states reduce the state complexity at the cost of external preprocessing, which hides information from the learning agent. Tree states represent an alternative approach which do not explicitly include information about the active samples. Instead, only information about the currently built tree is present in the state. The most obvious choice of information about the tree is the structure of the tree itself.

This approach can be compared to a human being presented with a partially built decision tree with the task of choosing the next split at a given position. Without access to the dataset, this task seems impossible at the first glance. However, suppose this human has seen this partially built tree (or fairly similar ones) a thousand times before and knows from experience that splitting based on one particular feature leads to a good end result. Now, performing splits is straight-forward and simply based on experience. This idea of choosing actions based on experience and trial

and error is what reinforcement learning is built for. Therefore, theoretically, this approach should work.

To be more specific, for a dataset D with n samples and m features (including labels), the state size is independent of n and m as the state contains information about the tree, not the data. As the state is independent of the dataset, other measures have to defined.

Building states representing trees offers a lot of freedom about the structure of the state and the information contained in the state. However, two fundamental aspects of reinforcement learning states in general have to be considered:

- A state should uniquely represent a given situation. As the policy chooses actions based on states, identically looking states will result in identical actions. If those states represent different situations requiring different actions, the idea of a policy based on states does not work.
- States representing similar situations should look similar. If this is not the case, generalization cannot work.

In conclusion, the structure and information of a state should be unique per unique situation but also similar to similar situations. To make this work, a way of uniquely describing a (partially built) decision tree has to be found. Additionally, information about where the next node should be attached has to be included in the state (see section 3.1.1 Building The Tree).

Generally, the act of serializing a binary tree is not a new problem and is traditionally done either via a linked list or a simple array representation. The more concise representation (in the context of reinforcement learning) is the array representation. It is based on the index i of a node, which starts with $i = 0$ for the root node. Consequent indices calculate as $i = 2i_{parent} + (1 \text{ if } left \text{ else } 2)$ and correspond to a breadth first node counting of a full binary tree. The array representation of the structure of a binary tree is an array with entries set to 0 or 1. The array is 1 for all node indices of the tree and 0 otherwise.

This representation only contains the structure of the tree and is not unique for decision trees. Two decision trees can have the same structure but with different splits applied at each node. Consequently, additional information has to be provided. For each node, this information is:

- The feature this node splits.

- The criterion by which the feature is split.

The 1s in the array representation are replaced with the splitting feature and criterion and the 0s with two padding values. While this representation fulfills the state criteria, the size of this representation is problematic. For example a binary tree of depth d (with any number of nodes larger than d) would lead to a state of size $s_{size}(d) = 2(2^d - 1)$. If the state should be able to represent all trees with a maximum depth of $d = 10$, this leads to $s_{size}(10) = 2046$. This problem can be circumvented by making one array entry per node and simply including the index of this node. While a full binary tree of depth d would still result in a similar state size, representing deep but narrow trees is not problematic anymore. As this scenario is far more likely for decision trees, the representation favoring deep trees should be preferred.

While representing the full tree is a suitable approach, an even simpler and arguably better approach exists. The full tree representation based on indices suffers from too high similarity of states as two different trees can only be distinguished by node indices, which are close to each other. Additionally, the full tree representation is not even necessary to provide a unique and meaningful state for a new split at a given position. Instead, it is enough to only look at the path through the tree leading to the new node. Why is that?
As data is filtered through the tree, only samples fulfilling all splitting criteria along the path leading to the new node actually reach the new node. All the other samples end up at other nodes. As the split introduced at the new node only affects samples which actually reach it, only the characteristics of these samples have to be considered. Samples reaching the new node are in turn characterized by the path they took through the tree. Therefore, all nodes of the tree, apart from the path to the new node, can be ignored as they have no impact on the data reaching the new node.
To illustrate this statement, consider the decision tree in figure 3.4. Node 1 contains samples with characteristics age smaller than sixteen and body weight smaller than thirty kilograms. The split introduced at Node 1 only impacts samples which actually fulfill the criteria of the path leading to it. On the other hand, the rest of the tree has no influence on this new split at all as the relevant samples never visited the other nodes.
Taking it one step further, it is even beneficial to leave out the rest of the tree as it represents additional irrelevant information which only confuses the learning agent. Looking at the tree in figure 3.4 again illustrates this example. The split

introduced in Node 1 should be the same, independent of the information present in the other path (for example whether Node 2 was already added). Consequently, only including the path to the new node in the state should speed up the learning process as the additional rule of ignoring irrelevant parts of the tree does not have to be learned.

Serializing a single path through the tree is trivial. One can simply put the nodes on the path in an array. To ensure the uniqueness of states, the following information has to be included per node:

- The feature this node splits.
- The criterion by which the feature is split.
- Whether the path fulfilling the splitting criterion is taken (for each node).

The third item is necessary as can be illustrated in the following example. Figure 3.4 shows an incomplete decision tree which could be used to decide whether a person is obese. A path state only containing information about features and splitting criteria would look identical for Node 1 and Node 2. When the agent is given the task to add a new node to the position of Node 1 or Node 2, the information it sees is identical resulting in the same action. However, Node 1 and Node 2 contain vastly different data. A person younger than sixteen years with a body weight under thirty kilograms (Node 1) can be considered normal in most cases while a person older than sixteen years with a body weight smaller than thirty kilograms (Node 2) would likely be underweight. Adding information about which direction is taken at each node makes the states for Node 1 and Node 2 unique, giving the agent a solid basis to decide on.

An even more simplified tree state could omit the path to the new node and instead only include the parent of the node and its depth in the tree. By leaving out the path to the node, the state becomes even smaller. However, this representation once again cannot guarantee the uniqueness of states, as the path to two identical nodes at the same depth can be vastly different. In conclusion, path states offer a compromise between full tree states and smaller states which still guarantees the desired properties.

So far basic concepts behind tree based states have been discussed. Implementing these basic concepts still offers a lot of freedom for various options with possible implications. These options will be discussed in section 3.2.3 Tree Environment when a broad overview over the whole system is given.

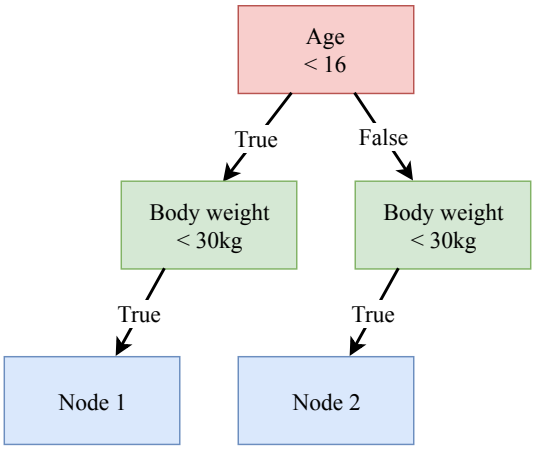

Figure 3.4: An incomplete decision tree for classifying obesity to highlight states for new nodes and the necessary information they should contain. Given states which only contain information about features and splitting criteria, the paths to Node 1 and Node 2 are identical. However, the data context is vastly different.

Tree based states offer various potential advantages.

- States have a low complexity independent of the size of the dataset.
- The tree is explicitly represented in the state, resulting in unique but generalizable states.
- Tree states are an elegant approach circumventing many of the problems of data based states.

However, potential disadvantages can be identified as well.

- The reinforcement learning agent only implicitly sees the full picture.
- Tree states rely heavily on exploration.
- The model learned based on tree states is highly connected to the dataset it was learned from. Adapting such a model to other datasets is most likely not feasible.

3.1.3 Reward Function

One of the most vital components of the algorithm is the reward function. It is responsible for assigning rewards to actions performed at a given state. As the

reinforcement learning agent tries to maximize the (accumulated) reward it receives, the reward function directly influences the agent and therefore the result. The reward function essentially shapes the result.

To illustrate the impact the reward function can have, chess is taken as an example. The ultimate goal of chess is to win by taking the opponent's king before losing your own. There are numerous ways and strategies to reach this goal. For a beginner, the most obvious way to success may be to try to eliminate as many of the opponent's chess pieces as possible to gain a numbers advantage. Ultimately, this advantage may lead to victory. A reward function reflecting this strategy could simply give a reward of $r = 1$ whenever a move (action) results in one of the opponent's chess pieces being taken and $r = 0$ otherwise. There are several obvious issues with this reward function:

- There is no punishment for losing your own chess pieces (especially your king).
- Taking the opponent's king and therefore winning the game gives a reward of $r = 1$, the same as taking any other piece.

Consequently, this naive reward function could result in an agent which mindlessly tries to take any opponent's piece, the more the better. Losing the game early or losing too many pieces would still be prevented as it cuts of any future rewards. However, winning a game in only a few moves would result in a lower accumulated reward than losing after a long game. So the true goal (winning the game) differs from the goal the reward function implies (eliminate the opponent's pieces). This difference between the true goal and the implied goal happens because of wrong assumptions. In this case the assumption was that removing the opponent's pieces leads to victory.

The Overall Goal

The chess example illustrates the pitfall (wrong or incomplete assumptions) of designing reward functions in an obvious way. However, not all problems are as intuitive as chess and offer a clear goal. In the case of decision trees questions like *What is an optimal tree?* and *What is the ultimate goal?* cannot simply be answered with *Winning the game!*.

Consequently, the first step is defining the goal. But the goal may vary depending on the use case the decision tree is built for. Some examples for goals are:

- A decision tree with maximum F1-score for high classification accuracy.
- A multitude of diverse decision trees (for an ensemble method).
- A small and easily interpretable decision tree (for illustration purposes).

The reward function can be modeled to represent these goals. This thesis mainly focuses on single decision trees (not ensemble methods). Therefore, catering to the goal of multiple diverse trees is not important. The remaining goals are both important. On the one hand, a tree should perform well (high classification accuracy or F1-score) but on the other hand it should still be small and interpretable. From this reasoning one fixed rule can be derived: If a smaller tree with similar or better performance exists, this tree should be found. In summary, the resulting goals are defined as:

- Find a decision tree with high performance.
- If a smaller tree with similar or better performance exists, find it.

In practice, these goals may oppose each other. Bigger trees will mostly have higher performance (at least for training data). The reward function, among other things, models this trade-off.

Performance Measures

Defining an appropriate reward function fulfilling the goal requires adequate performance measures to gauge the impact of an action. Again, several options can be identified.

A widely established performance measure for classification tasks is the F1-Score [44]. It is usually preferred over classification accuracy as it produces meaningful numbers for datasets with uneven class distribution due to the use of *precision* and *recall*. Precision (see equation (3.3)) measures the percentile of correct predictions of one class over all predictions of this class, so how many of the predicted class i samples actually belong to class i. Recall (see equation (3.4)) measures the percentile of correctly classified samples of class i over all samples of class i, so how many out of all samples were predicted correctly. Maximizing both precision and recall usually results in a trade-off between the two measures. This trade-off is measured

by the F1-score, which is defined in equation (3.5). In its basic form, F1-score is a measure for binary classification. It can be applied to multiclass problems by applying a weighted average across all classes.

$$precision = \frac{\text{True Positives}}{\text{Predicted Positives}} \tag{3.3}$$

$$recall = \frac{\text{True Positives}}{\text{Actual Positives}} \tag{3.4}$$

$$F_1 = 2 \frac{precision \quad recall}{precision + recall} \tag{3.5}$$

F1-score is a suitable measure to assess the performance of a (finished) decision tree for classification. However, it may not be suitable to evaluate the value of a single action while building the decision tree.

Another well established measure in the context of decision trees can be used to measure the performance of decision trees. Gini impurity has already been discussed in section 3.1.1 Building The Tree and is given in equation (3.2). It measures the purity of samples in a node and therefore provides a measure of how well a node or leaf has isolated samples of a single class. In contrast to F1-score it is more suitable for evaluating single nodes as opposed to the whole tree. This focus on single nodes makes it an ideal candidate to judge the isolated impact of appending a new node to the tree.

Reward Functions

Having a definite goal and performance measures to evaluate whether the goal was reached (at least partly) only leaves the actual definition of the reward function. As always, several options can be identified. Each option has implications about the resulting tree, mostly about greediness, size, classification performance and efficiency of the algorithm, most of which stand in contrast to each other. The optimal reward function would result in a small, non-greedy tree with high classification performance which is built with high efficiency.

Before defining concrete reward functions, the concept of *node rewards* and *tree rewards* is introduced. Node rewards focus on giving rewards which rate the impact of appending a single node to the tree. This can be achieved by an appropriate

performance measure or by taking the difference between the status before and status after appending a node. Tree rewards on the other hand rate the tree as a whole. This not only includes the classification performance of the tree but may also include the size or even more abstract measures like readability of the tree.

The first option focuses on the non-greediness of the tree by providing *sparse rewards*. While reinforcement learning itself is a non-greedy approach, this property can be strengthened (and must indeed to a certain extent be enforced) with an appropriate reward function. The idea is to give no rewards while building the tree but only when the tree is finished. The reward given when the tree is finished (see subsection 3.1.5 Definition of Done) has to be a tree reward to rate the whole tree. The function can be stated as seen in equation (3.6) where $r(s, a)$ is the function giving the reward based on the current state s and action a performed on it. State and action can be viewed as the current tree receiving a new node based on action a. The tree reward r_{tree} is then calculated for the resulting new tree.

$$r(s, a) = \begin{cases} r_{tree}, & \text{if } done \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

There are several options to calculate the tree reward r_{tree}. These options are calculated based on several measures including measures like the F1-score, gini impurity, tree size and maximum tree size. Numerous sensible combinations of these measures are possible and can be scaled with the likes of logarithmic functions. One such option can be seen in equation (3.7) where T is the tree. This option rates the tree based on the F1-score and the tree size. Smaller trees with similar F1-scores will yield higher rewards. However, significantly smaller trees with worse F1-scores may also produce higher rewards. Scaling the tree size fraction with a logarithmic function levels out the more extreme differences between tree sizes.

$$r_{tree}(T) = F1(T) + F1(T) \ \ log\left(\frac{\text{maximum tree size}}{size(T)}\right) \tag{3.7}$$

While sparse rewards theoretically produce great results, they can also significantly impede the learning process as they blur the cause effect relation. The chess example from the beginning of this section illustrates this problem. As a human without any knowledge about chess, learning from an entire game of chess solely based on the outcome (win or lose) seems nearly impossible. Unfortunately, introducing

intermediate rewards also means that new assumptions have to be made and forced on the learning agent. In the case of decision tree learning, intermediate rewards (in the form of *node rewards*) usually result in some sort of greediness.
The most basic approach is to define a reward like given in equation (3.8). It is based on the gini impurity of the newly added node N. If a newly added node N is pure, $g(N)$ will be small and hence close to the optimal reward of zero (other rewards are negative).

$$r_{node}(N) = \quad g(N) \tag{3.8}$$

This approach has a significant problem. Suppose two nodes $N_{i\ 1}$ and N_i are added to the tree, where N_i is a child of $N_{i\ 1}$. If $N_{i\ 1}$ introduced a highly beneficial split to the tree its gini impurity will be fairly low, resulting in a good reward. Its child N_i will most likely have low impurity as well, independent of the split it introduces as the active samples in the node were already separated by the parent node. So adding the node N_i produces a high reward without being a beneficial action.
Countering this problem is straight-forward and only requires to take the difference between the state before and after. The adapted formula can be seen in equation (3.9) where $g_{children}(N)$ represents the weighted sum of gini impurities of the children of node N.

$$r_{node}(N) = g(N) \quad g_{children}(N) \tag{3.9}$$

The gini impurity before, $g(N)$, examines all active samples at node N while the gini impurity after, $g_{children}(N)$, examines the children resulting from the split introduced by node N. This reward function rates local improvement and is therefore greedy. Similar reward functions based on other performance measures can be introduced. An example based on F1-score can be seen in equation (3.10) where $T_{i\ 1}$ is the tree before adding a new node and T_i the tree after adding it.

$$r_{node}(T_{i\ 1}, T_i) = F1(T_i) \quad F1(T_{i\ 1}) \tag{3.10}$$

So far, tree rewards and node rewards have been examined. Tree rewards potentially produce better results due to non-greediness while also making learning more difficult. Node rewards are greedy by nature but provide the learning agent with

immediate feedback, potentially speeding up the learning process significantly. To get the best of both worlds, a combination of the two seems to be the logical step. Equation (3.11) offers such a combination. While the tree is being built, node rewards are provided. The finished tree then receives a tree reward which has to balance out the greedy rewards by being more significant.

$$r(s,a) = \begin{cases} r_{tree}, & \text{if } done \\ r_{node}, & \text{otherwise} \end{cases} \tag{3.11}$$

Another option is to always provide tree rewards and node rewards but weight them. Equation (3.12) shows such an approach with w_{node} weighting node rewards and w_{tree} weighting tree rewards. The weights can be either fixed or chosen dynamically based on the progress of the building process (for example weight tree rewards higher when the tree size is larger).

$$r(s,a) = w_{node} \quad r_{node} + w_{tree} \quad r_{tree} \tag{3.12}$$

This weighting approach is more powerful and basically models all previously discussed reward functions. The weights simply have to be set accordingly. They also allow for a dynamic focus on the importance of tree size, performance and greediness as tree and node rewards represent different characteristics.

While putting all the discussed reward functions into one function is neat, the different choices for tree rewards, node rewards and their respective weights still provide a huge space of possibilities which have to be evaluated.

3.1.4 Reinforcement Learning

The overall algorithm examined the basic usage of reinforcement learning in the whole process. For reinforcement learning to function, an environment which provides states and rewards and accepts the execution of actions has to defined. All of these items have been discussed. This leaves reinforcement learning itself, the method which is responsible for learning to execute the correct action at a given moment based on the rewards it receives. This thesis mainly focuses on finding a new method for generating decision trees. Reinforcement learning is mainly used as a tool to fulfill this goal and not the focus of experimentation, examination

and evaluation. However, reinforcement learning methods have a large amount of parameters or even significantly different models and concepts which may have an impact on the overall concept. For this reason, a closer look at different concepts, models and parameter configurations has to be made.

Discrete and Continuous Actions

The first and most significant difference (in the context of this thesis) between reinforcement learning methods is the nature of actions they can predict. Originally, reinforcement learning was designed for taking discrete actions (choosing an action out of n possible actions). As building a decision tree requires a choice of features (discrete) but also splitting points (continuous), designing an optimal algorithm solely based on discrete actions is probably impossible. One possibility to deal with this limitation is to limit the reinforcement learning agent to only choosing features (which are discrete) and leaving the choice of splitting point to an external agent.

However, modified reinforcement learning methods can also deal with continuous action spaces. As the continuous action space is infinitely large, this problem is naturally harder. Methods like NAF [17] do not choose actions in the same way as discrete reinforcement learning methods do. As the action space is infinitely large, it is approximated quadratically with the turning point representing the set of action values with approximated maximum Q-value. This renders a discrete action based on Q-values infeasible. Instead, a continuous value for each feature (which can be loosely interpreted as a confidence measure) is responsible for making the discrete choice. The continous splitting point comes more naturally.

In conclusion, the methods for discrete and continuous actions offer two interesting options.

1. The (discrete) reinforcement learning agent only makes discrete choices (it chooses which feature to split on). An external agent is responsible for choosing where to split this feature. Traditional decision tree building algorithms offer (greedy) options for designing such an agent.
2. The (continuous) reinforcement learning agent chooses the feature and the splitting point. This approach is much harder but also provides the learning agent with more freedom, making the results potentially less greedy and therefore theoretically better.

Other options like stacks of agents or ensemble methods exist but are deemed out of scope of this thesis.

Model

The model (typically some form of a neural network) takes states as inputs and produces measures (for example Q-values) which allow for choosing good actions. The choice of which model to use is mainly influenced by the state representation. The following models are commonly used.

- Feed forward neural networks (can be applied to all kinds of problems as they are very general)
- Convolutional neural networks (typically used for processing images due to their suitability for pattern recognition)
- Recurrent neural networks (typically used for processing time series)

For the different proposed states (see subsection 3.1.2 State Representation), arguments can be made for all three model architectures. Tree based and extracted features states are small and most suited to feed forward neural networks. For data states, convolutional neural networks are an interesting choice due to the ability to recognize patterns in the data. For data states arranged like a series of individual samples, recurrent neural networks are intriguing. This thesis only examines feed forward neural networks, leaving the possibility of utilizing convolutional or recurrent neural networks (or other architectures) for future examination.

Parameters

Modern reinforcement learning methods come with numerous tunable parameters which are mostly interconnected. Parameters like learning rate or target network update frequency have a high impact on performance but little to no implications for the outcome. These parameters simply have to be optimized. Other parameters like γ, which weights the significance of future rewards have implications for the outcome and will be briefly discussed here.

- Gamma (γ): Controls how much influence the value of future states has on the Q-value of the current state based on the action taken. High values rate the future higher and implicate a far look-ahead (provided the future can

be predicted fairly accurately). Low values are short-sighted and focused on locally optimal actions. This directly influences the greediness of the tree building process.

- State history size: Depending on the chosen state representation it may be necessary to not only look at the current state but also consider previous states to make meaningful decisions. This parameter controls how many past states should be provided to the learning agent. For decision trees this is the equivalent of not only considering the current status of the tree when making a decision but also considering previous states. Intuitively, actual tree states likely will not profit from a state history as it provides no additional information. For other representations, it remains interesting.
- Replay buffer: The replay buffer holds previously encountered state-action-reward triples and provides the samples for experience replay. The starting size and maximum size of the replay buffer controls the size and length of the experience the learning agent can use for learning. A small replay buffer indicates that previously encountered experiences are more important. Depending on the exploration and learning rate of the agent, it can have impact on the final outcome.

Exploration Strategy

Exploration is a vital part of reinforcement learning. It ensures that the learning agent does not get stuck in only locally optimal or even suboptimal positions. As reinforcement learning can be seen as a guided random exploration of possibilities (which becomes less and less random as time goes on), ensuring that this randomness does not end too soon but also considers all relevant possibilities is an interesting challenge (exploration versus exploitation). In this thesis a simple but well proven exploration strategy is used: epsilon greedy exploration, the same as used in various papers like DQN [33] or NAF [17]. The idea is simple. Epsilon (ϵ) represents the probability of choosing a random action. This idea is formulated in equation (3.13) where s represents the current state and a an action.

$$action = \begin{cases} \text{random}, & \text{with probability } \epsilon \\ \max_a Q(s,a), & \text{otherwise} \end{cases} \tag{3.13}$$

To reduce the amount of random choices as the agent learns, ϵ has to be adjusted. This adjustment is controlled by the parameter ϵ_{decay} which typically takes values

in the range $[0.95, 1)$. It is used as shown in equation (3.14) and gradually makes ϵ smaller in each step until it reaches a certain threshold.

$$\epsilon_{i+1} = \epsilon_i \quad \epsilon_{decay} \tag{3.14}$$

This strategy alone only works for discrete action spaces. Continuous actions require the exploration of a continuous space. Commonly used strategies include:

- Linear decay exploration
- Ornstein-Uhlenbeck exploration (based on the Ornstein-Uhlenbeck process [49] as used in [28])

All strategies work by adding noise to the original continuous values. The amount of noise is typically controlled by mean, variance and the length of the previous learning process (similar to ϵ decay). Further particulars of individual strategies are not discussed at this point.

3.1.5 Definition of Done

An essential question has not been answered yet: When is the tree which is being built considered to be finished? This question is not be confused with the end of the learning procedure, it is solely concerned with individual trees. In this thesis, whenever the keyword *done* is used it refers to one episode being finished which is the equivalent of a tree being completed.

The methods to determine when a tree is finished are simple and utilize the following questions.

- Has the tree size (number of nodes) exceeded a certain threshold?
- Do the performance measures (for example F1 score) indicate a sufficient level of performance?
- Do the current leaves contain enough samples to justify adding another node to the tree?

In other words, a tree is considered to be finished if it has reached a certain size or if its performance is sufficiently good. Additionally, the requirement of a minimum number of samples per node has to be met. A leaf which only contains a small

amount (for example less than ten) active samples is considered to not have a large enough sample size to justify adding another split as the low number of samples simply make the decision too much of a guess (the samples may not be representative).

3.1.6 Post Processing

Traditional decision tree building algorithms have a post processing step which reduces the usually large trees to a manageable size. This step is called pruning and is necessary to make the tree smaller, interpretable and to avoid overfitting. The characteristic of producing large trees stems from the greedy nature of the algorithms. As most greedy decision tree building algorithms are deterministic, they run only once and cannot change the structure of the tree to incorporate decisions at another point.
Due to the non-greedy nature of reinforcement learning, pruning is not necessary for the trees produced by this algorithm. In fact, no post processing is needed whatsoever.

Another additional method to boost the performance of decision trees is to apply ensemble methods like bagging or random forests. In this thesis, none of these methods are applied. However, implementing them would be straight-forward and complemented by the nature of the algorithm, which relies on a stochastic process to build decision trees.
Decision tree ensemble methods rely on the diversity of trees to produce good results. The process by which trees are built in this thesis offers various possibilities to produce diverse trees which will be discussed later in section 5.1 Future Work.

3.2 System

This section deals with the implementation of the algorithm. First, the overall architecture is described with focus shifting to more detailed components in the later stages of this section. All components of this system were implemented in the following environment.

- *Programming language*: Python 3.5
- *Machine learning framework*: Tensorflow r1.2 [1]
- *Utility functions*: Scikit-learn 0.20 [35]

3.2.1 Components

The algorithm is implemented with various components. An overview can be seen in figure 3.5. A brief description of each component reads as follows.

- *Main*: Controls the overall execution of the algorithm by setting configuration parameters and triggering evaluation.
- *Configuration*: Holds all configurable parameters for the whole algorithm (excluding externals like which dataset to process).
- *Environment*: A representation of the reinforcement learning environment which supports taking steps and returning states and rewards. It is responsible for assembling the tree based on the actions provided by the agent. Multiple environments exist, modelling different state representations.
- *Solver*: Responsible for solving the environment by providing appropriate actions for given states based on rewards. Multiple solvers were implemented representing different reinforcement learning algorithms.
- *Trees*: This component is a binary decision tree implementation which supports various operations necessary for execution of the algorithm.
- *Analyzer*: Responsible for evaluating the performance of the algorithm and of the outcome with various plots and textual summaries.

The following sections describe each component in more detail.

3.2.2 Configuration

The configuration component is responsible for providing a single access point for all changeable parameters. Different presets can be loaded to examine specific parameter sets. The parameters it controls are described in the following subsections where they belong semantically. The configuration also contains two top-level parameters.

- *Environment*: The environment to be used.

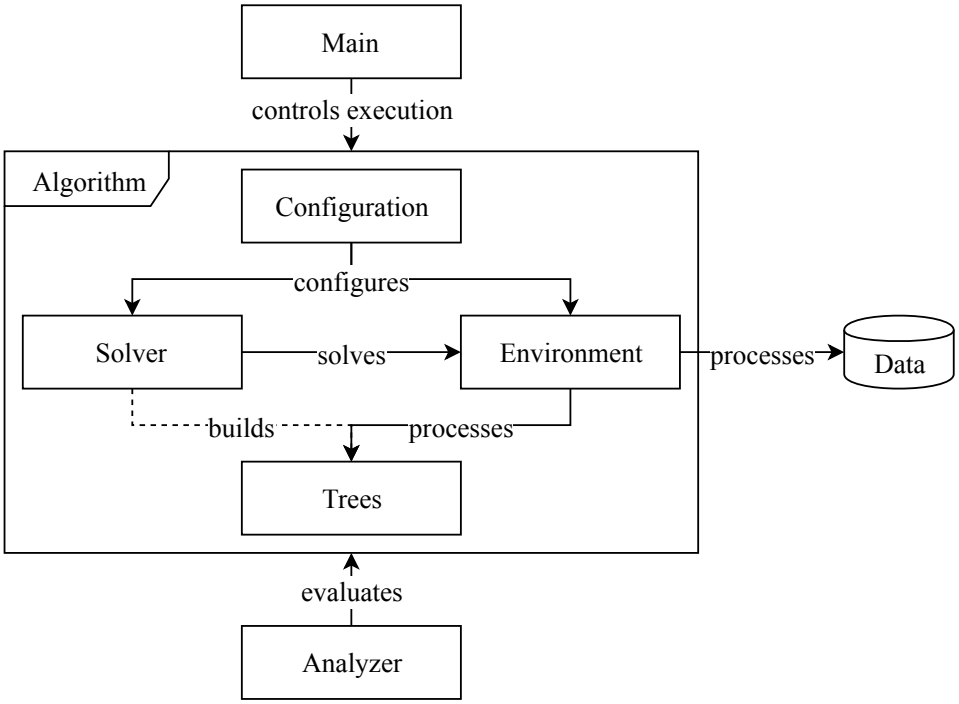

Figure 3.5: The basic components and their connection. The implementation of the algorithm is shaped by the components in the algorithm frame which implement the various options. Each of these options is controlled via parameters in the configuration component. Based on the configuration, the solver is responsible for solving the environment, which provides states and rewards and processes actions to build trees. The solver indirectly builds trees via the environment. Finally, the analyzer component evaluates the performance of algorithm and outcome.

- *Solver*: The selected solver.

3.2.3 Environments

The environments component is responsible for executing steps and providing resulting states and rewards. It assembles the currently built decision tree and accesses the dataset which is being solved.

The environment implementation has a high overall impact as it controls several key factors.

- The state representation.
- It decides where new nodes will be attached to the tree (see algorithm 3.2).
- It gives rewards.
- It decides when building the tree is finished ($done$).
- Depending on the nature of the reinforcement learning algorithm (discrete or continuous), it is responsible for deciding where to split a given feature (base on traditional decision tree building algorithms).

As different state representations require vastly different handling, an environment for each state representation was implemented. Before examining these specific environments, the parameters they share will be discussed.

- *Maximum tree size*: The maximum size (number of nodes excluding leaves) the tree can have (Default: 11). This parameter also controls the maximum number of steps the learning agent takes in each episode (as each step corresponds to adding a node to the tree).
- *Maximum tree depth*: The maximum depth the tree can reach (excluding leaves) (Default: 11).
- *Minimum samples per leaf*: The minimum number of active samples each node or leaf can contain (Default: 5).
- *Node reward*: The chosen node reward function (Default: gini impurity difference).
- *Tree reward*: The chosen tree reward function (Default: F1 score normalized by tree size).
- *Node punishment*: The punishment given for adding a new node to control the size of the tree (Default: 0).

- *Empty leaf punishment*: If the reinforcement learning agent decides to add a node which has a leaf with zero active samples, this punishment is administered (Default: 0).
- *Scale gini leaf sum*: Indicates whether the sum of gini impurities across all leaves should be scaled by the respective number of active samples in each leaf (Default: true). The gini sum is used as a performance measure for rewards and the $done$ check.
- *Normalize gini leaf sum*: Indicates whether the sum of gini impurities across all leaves should be normalized by the total sample count (Default: true).
- *Stopping criterion gini sum*: A threshold based on the sum of gini impurities across all leaves which decides whether the tree is completed (Default: 0.1).
- *Stopping criterion F1 score*: A threshold based on the F1 score which decides whether the tree is completed (Default: 0.9).
- *Normalize data*: Indicates whether data should be normalized (Default: False).

Data Environment

Data environments are characterized by directly including data samples in the states. As datasets can be big, a way of reducing the state size has to be introduced. This is simply done by only including a fixed amount of samples per state. If the number of active samples for a state is larger or equal to the chosen state size, samples are randomly picked from the active samples. In case of too few active samples, samples are drawn from the active sample pool repeatedly.

The parameters specific to the data environment are the following.

- *Flatten states*: Indicates whether states should be flattened from $n \quad m$ to $1 \quad nm$ (Default: false), where n is the number of samples and m the number of features.
- *Remove labels*: Indicates whether labels should be removed from the data when passing it to the reinforcement learning agent (Default: false).
- *State size*: The number of samples included per state. If the dataset is large, it is impractical to pass all samples to the learning agent as the state size would be too big.

Tree Environment

Tree environments produce states which directly model the tree without including any data samples. The two main ways of doing this are either via states including information about the whole tree or via states including only information about the path from the root node to the new node which should be added. Building these states offers a lot of freedom about what information to include in which way, which is reflected by the large number of parameters for this environment.

The parameters specific to the tree environment are the following.

- *State structure*: Controls whether states should encompass the whole tree or only the path leading to the node which should be added next (Default: path only).
- *State ordering*: Controls whether characteristics of one node should be grouped together (called alternating) or if the characteristics should be grouped together (called group). If one node has the information (*index*, *feature*, *split*), alternating states would include one node after each other, resulting in an alternating repetition of (*index*, *feature*, *split*). Groups would put all *indices* together, followed by all *features* et cetera (Default: alternating).
- *Inner state ordering*: Controls the ordering of node characteristics of above representation (for example whether the order should be (*index*, *feature*, *split*) or (*index*, *split*, *feature*)) (Default: (*index*, *direction*, *feature*, *split*)).
- *Full tree structure*: If the state structure represents whole trees, this parameter controls whether the default array serialization (called full tree) or the compact serialization relying on indices should be used (called compact) (Default: compact).
- *Include indices*: Indicates whether node indices should be included in the state for each node (Default: false).
- *Include splitting criteria*: Indicates whether node splitting criteria should be included in the state for each node (Default: true).
- *Include directions*: Indicates whether node directions (left or right) should be included in the state for each node (Default: true).
- *Include feature dimensions*: Indicates whether node feature dimensions (the feature to split on) should be included in the state for each node (Default: true).

- *Bottom to top*: Indicates whether trees and paths should be serialized from bottom to top (leaf to root) or starting from the root down to the leaves (Default: top to bottom).

3.2.4 Solvers

The solver component is in general responsible for choosing actions based on previously encountered states and rewards. Reinforcement learning was chosen as the method to implemented solvers. Three different reinforcement learning algorthims were implemented, each with their own solver. They share various parameters which are listed here.

- *Training episodes*: The number of episodes the agent trains (Default: 500). Each episode corresponds to one tree being built.
- *Testing episodes*: The number of episodes used for testing the learned model (Default: 50).
- *Hidden layer neurons*: The number of hidden layers and the amount of neurons used in them (in a feed forward neural network) (Default: Two layers with three hundred neurons each).
- *Learning rate*: The learning rate of the agent (Default: 0.005).
- *Learning rate decay*: The factor by which learning rate decays each step (Default: 1.0, no decay).
- *Gamma*: Gamma (γ) controls how much influence the value of future states has on the Q-value of the current state based on the action taken. A high value indicates a far look-ahead and little greediness (Default: 0.995).
- *Epsilon*: The starting value for Epsilon (ϵ) used in the epsilon greedy exploration (Default: 1.0, fully random).
- *Epsilon decay*: The rate at which ϵ decays in each step (Default: 0.9995).
- *Epsilon final*: The final ϵ value. Lower values are not possible during training (Default: 0.08).
- *Minibatch size*: The size of the batches used for learning (Default: 25).
- *Replay buffer start size*: The amount of steps taken to fill the replay memory before learning starts (Default: 500).
- *Replay buffer maximum size*: The maximum number of steps stored in the replay memory (Default: 100000).

- *State history size*: The amount of preceding steps appended to the current state. This state history gives the agent a small look in the past (Default: 1).
- *Train every n steps*: If steps are very small and do not change the state significantly, it might make sense to only train every n steps to get greater stability and less bias. As this is not true for the algorithm proposed in this thesis, the parameter can be left at its default (Default: 1, train every step).

By default, all solvers use feed forward neural networks as models. As the state representation is customizable, the default parameters and network structure may change according to the state representation or task at hand. This in turn has an impact on the other parameters. For example, changing the state representation from a small tree state to data states results in larger, more complex states that require a larger network to model the mappings. A larger network possibly requires more training to achieve good results without overfitting, resulting in a smaller learning rate. Another factor is the reinforcement learning algorithm itself. The indivdual implementations will be briefly examined.

DQN Solver

Deep Q Networks (DQN), published by Mnih et al. in 2015 [33] is used as a benchmark for most reinforcement learning algorithms. Some of the most important parameters discussed as general reinforcement learning parameters originated from this paper. The algorithm was implemented like described in the paper with minor adjustments for reward handling. A parameter which is specific to DQN and DDQN is the target network update frequency.

- *Update frequency*: The target network (used for error calculation) is updated every n steps to be identical to the main network (Default: 100).

DDQN Solver

Double DQN (DDQN) [50] is an improved version of DQN which tries to improve a common issue of reinforcement learning methods: overestimation of specific action values. The overestimation happens because standard DQN evaluates and selects actions based on the same measure. If this measure is inaccurate, an inappropriate action is selected with the evaluation of the action not raising any warnings. DDQN

separates selection and evaluation to reduce bias. This modification is surprisingly simple in its idea. Instead of using only the target network, the current and target network are used for selection and evaluation respectively. No new parameters are introduced for DDQN as it uses the same parameters as DQN.

NAF Solver

Normalized Advantage Functions (NAF) [17] is a state of the art algorithm for continuous action spaces. The core is based on Q learning, but modified to handle continuous actions. The implementation is based on the paper, excluding model based acceleration (as it requires external models). New parameters introduced by NAF are the following.

- *Tau*: Tau (τ) is used for a soft update of the target network. Instead of making the target network identical to the current network every n steps, the target network is continuously udated by the factor τ (Default: 0.001).
- *Updates per step*: To accelerate learning and reduce the amount of (possibly expensive) steps in the environment, several learning updates per step are performed (Default: 5).

3.2.5 Trees

The trees component is an enhanced binary tree implementation. It is enhanced with various functions needed for decision trees and decision tree learning. Some of the more prominent additions are listed here.

- Classification ability based on splits in each node.
- Efficient calculations of various metrics like gini impurity or F1 score while building the tree.
- Memory- and performance-efficient handling of data passing through the tree.
- Tree plotting.

3.2.6 Analyzer

The analyzer component evaluates the performance of both algorithm and the outcome of the algorithm. Various plots and textual outputs give an overview over the performance. The most important aspect of this component is that it documents configurations and according outputs in such a manner that they can analyzed and reproduced easily.

In summary, this chapter introduced the methodology and ideas used for designing a new decision tree building algorithm based on reinforcement learning. The need for assumptions which enforce restrictions on the learning agent has been discussed as well as the various components of the algorithm. Each component offers several possibilities of how it can be implemented. Theoretical arguments for and against these possibilities were discussed to form a concise outline for the algorithm. Finally, these ideas and possibilities got mapped to a concrete implementation, with changeable parameters representing the various ideas and possibilities mentioned earlier. The most important decisions for the methodology and implementation are shown in figure 3.6, where the bold path represents the default configuration. The next chapter will evaluate the algorithm and the decision trees it produces to get a clear and objective picture about the ideas and theories proposed in this chapter.

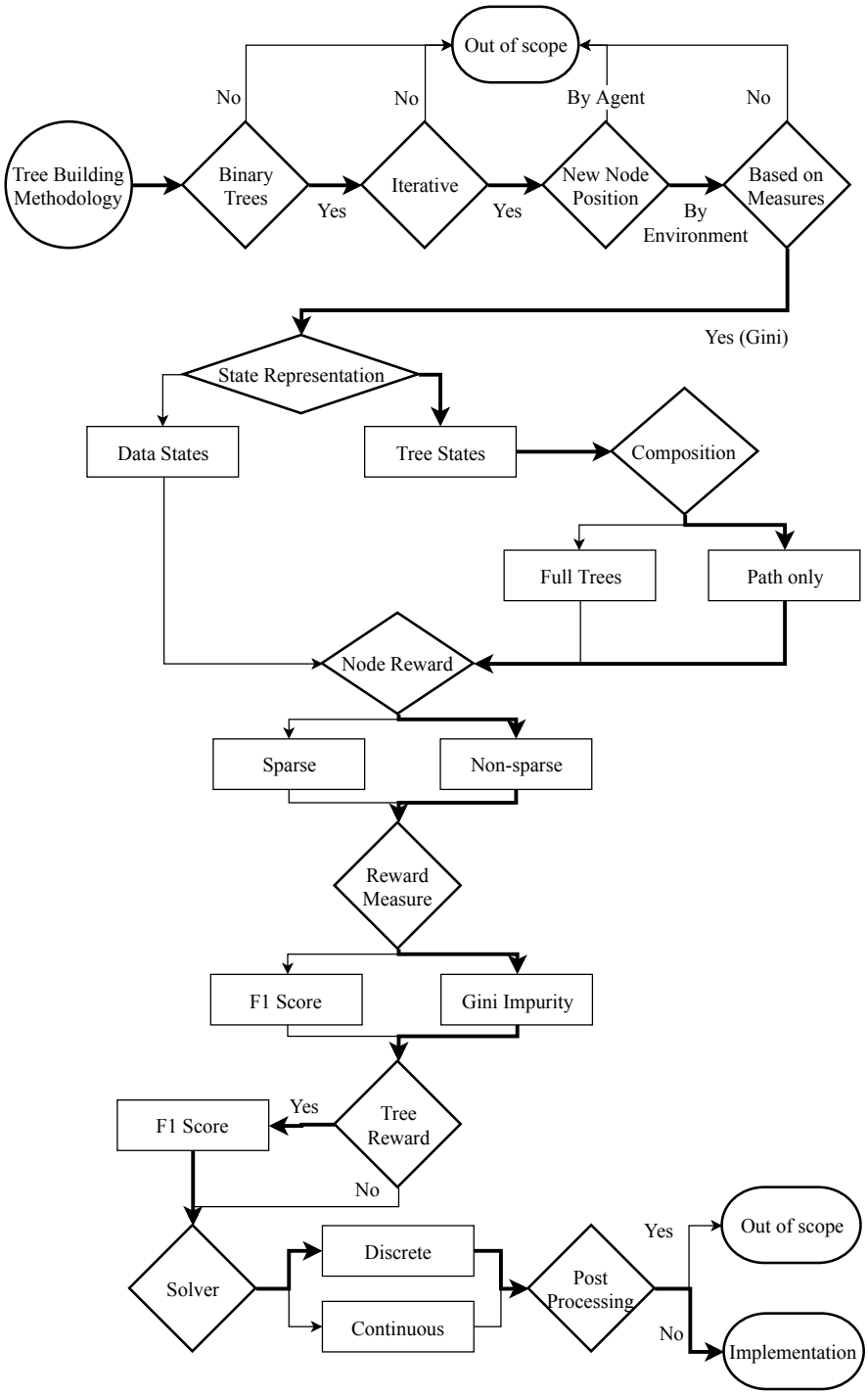

Figure 3.6: A flow chart showing the most important decisions in methodology and implementation. The bold path represents the default configuration.

4 Evaluation

In this chapter the proposed algorithm is evaluated in various configurations to answer the main research questions.

- Is it possible to build decision trees with reinforcement learning?
- What is a suitable way of building decision trees with reinforcement learning?
- Can the algorithm produce non-greedy results?

Additionally, the performance of the algorithm and the outcome is tested and compared to state of the art decision tree building algorithms. Before presenting the results, the evaluation methodology is explained followed by a closer look on the used data sets. Based on this methodology the results for the data sets are presented and finally discussed.

4.1 Evaluation Methodology

This section describes the evaluation methodology as well as the environment it is executed in.

4.1.1 Methodology

The questions which should be primarily answered by the evaluation are the research questions. To get meaningful answers to these questions several other sub-questions have to be answered as well. Several theories have been proposed in the chapter 3 Method. The most important ones will be examined in more detail. To answer such a question the following methodology is applied.

1. Formulate the question.

2. Apply a suitable parameter configuration.
3. Test this parameter configuration an adequate number of times by training with a training set and testing with a testing set with a random split of 75% training to 25% testing data.
4. Present the average results.

An interpretation of the results is then conducted in section 4.4 Discussion.

4.1.2 Environment

To make the results replicable and comprehensible, the environment in which the following results are obtained has to be recorded. All test are run on a middle-class desktop computer with the specifications which can be seen in table 4.1. All components are run at stock speed (no over- or underclocking). The machine learning framework (Tensorflow) supports GPU calculations. However, not using the GPU yields similar execution times as the learning procedure for problems used in this thesis does not take long and the overhead of using a GPU is noticeable.

Type	Component
CPU	Intel(R) Core(TM) i5-6600 @ 3.30 GHz
RAM	16 GB
GPU	GeForce(R) GTX 1060
VRAM	6 GB
OS	Ubuntu 16.10

Table 4.1: Hardware and operating system used for testing.

4.2 Data Sets

The data sets used for evaluating the algorithm were either generated or taken from the UCI Machine Learning Repository [11]. The chosen data sets are diverse in size, number of features and class distribution and are suitable for classification. First, generated data sets will be examined followed by real data sets in no particular order.

4.2.1 Generated Data Sets

Generating data sets allow for easier testing of specific hypotheses and are a great aid while developing an algorithm. Two types of data sets were generated and will be explained shortly.

Parametrized Data Set

The dataset mainly used for designing and testing new features is a parametrized generated dataset. It can be adapted easily via parameters to adapt the difficulty of solving it. The following parameters define the dataset.

- *Size*: The number of samples.
- *Features*: The number of features (including labels).
- *Classes*: The number of different classes the samples belong to (equally distributed).

The dataset is generated in such a way that the optimal tree is known.

Non-Greedy Data Set

This data set is generated in such a way that it cannot be solved optimally with a greedy algorithm. The purpose of this data set is to check whether the algorithm proposed in this thesis can solve it in a non-greedy and therefore optimal way. It has three binary features (taking the values 0 or 1). Feature 1 and 2 have a random uniform distribution of zeros and ones. The label is set to be the logical operation XOR (exclusive or) of feature 1 and 2. Feature 3 is equal to the label with a percentage higher than fifty percent. As splitting on feature 3 provides the highest gain, a greedy algorithm will split on feature 3 first and then on feature 1 and 2. An optimal algorithm will only split on feature 1 and 2 (to model the XOR relationship which defines the labels) and will therefore produce smaller trees.

4.2.2 Real Data Sets

Real data sets taken from the UCI Machine Learning Repository [11] are briefly described in the following sections.

Cryotherapy

The cryotherapy data set [24, 23] contains information about the treatment of warts using cryotherapy with the target being the success or failure of treatment. Its key characteristics can be seen in table 4.2. The characteristics suggest a small and well balanced data set with few attributes.

Property	Value
Samples	90
Features	7
Classes	2
Class distribution	(47%, 53%)

Table 4.2: Key characteristics of the cryotherapy data set.

Parkinsons

The parkinsons data set [29] contains processed voice recordings of people with and without parkinsons disease. Based on the different voice measures, the recordings should be assigned to healthy or parkinsons afflicted. Its key characteristics can be seen in table 4.3. The higher number of features and uneven class distribution set this data set apart.

Property	Value
Samples	195
Features	23
Classes	2
Class distribution	(25%, 75%)

Table 4.3: Key characteristics of the parkinsons data set.

User Knowledge Modeling

This data set [22] examines a users knowledge level based on several features like study time or exam performances. Table 4.4 shows the key characteristics of this data set. The higher number of fairly balanced classes make this data set an entry point for multi-class classification.

Property	Value
Samples	258
Features	6
Classes	4
Class distribution	(9%, 24%, 32%, 34%)

Table 4.4: Key characteristics of the user knowledge modeling data set.

Urban Land Cover

This data set [20, 21] contains preprocessed aerial images. Each image corresponds to one sample which should be classified as a type of urban land cover (for example trees or buildings). The key characteristics in table 4.5 show a very high feature count as well as multiple classes with fairly equal distribution.

Property	Value
Samples	168
Features	148
Classes	9
Class distribution	(9%, 14%, 10%, 15%, 8%, 17%, 10%, 8%, 9%)

Table 4.5: Key characteristics of the urban land cover data set.

Student Performance

The student performance data set [8] contains samples about students and their social background. Goal of the classification is to predict the final grade in a given subject (math). Table 4.6 shows the key characteristics. The high number of features

and classes as well as a very limited amount of samples per class make this data set very challenging.

Property	Value
Samples	395
Features	31
Classes	18
Class distribution	(10%, 1%, 2%, 4%, 2%, 8%, 7%, 14%, 12%, 8%, 8%, 7%, 8%, 4%, 2%, 3%, 1%, 1%)

Table 4.6: Key characteristics of the student performance data set.

4.3 Results

This section presents the results achieved with the algorithm proposed in this thesis. The following metrics are examined to gauge the performance.

- F1-score (of training and test set)
- Execution time of the algorithm
- Size of the resulting tree (always without leaves)

Multiple parameter configurations are tested on all data sets to find a suitable configuration which is then compared to other state of the art algorithms, including C4.5. As few changes as possible are applied to the default parameter configuration to keep the results as comparable as possible. However, not adapting the parameters significantly also leads to results which are not optimal (especially for different data sets) which has to be considered when viewing the results. Additionally, fundamental questions arising in the chapter 3 Method are examined. The discussion of the results happens in a later section 4.4 Discussion.

All results obtained with the algorithm feature two separate outcomes. On the one hand, the building process (and resulting tree) the reinforcement learning process converges to and on the other hand, the best tree which is created during the training procedure. As reinforcement learning is a stochastic method which relies on exploration, not every explored possibility is guaranteed to have a high impact on the converged outcome. Consequently, a tree with better performance

might be created during training than during testing (which works with the final reinforcement learning model and hence, policy).

To provide a baseline, partially random trees were built and tested. The default configuration runs for 500 episodes, resulting in 500 trees being built. To get comparable results, 500 partially random trees with a maximum of eleven nodes (excluding leaves) were generated. The trees are generated in a similar fashion to the one used in the algorithm for discrete actions. While the environment provides new node positions and calculates the locally optimal split for each node, the choice of feature per node is randomized. The results for this baseline can be seen in table 4.7.

Dataset	Best F1(test)	Tree size
Cryotherapy	0.788	10.8
Parkinsons	0.773	11
User Knowledge	0.664	11
Urban Land Cover	0.393	11
Student Performance	0.08	11
Average	0.540	10.96

Table 4.7: Average results over five runs (500 random trees per run) for random feature selection. New node positions and splitting points are not random. Results are given for each data set and averaged over all data sets in the final row. The F1 score shows the average best performing random tree on the test data and the tree size its size.

4.3.1 Reward Functions

So far reward functions have been treated as one of the deciding factors in the performance of the learning agent. This assumption is tested by comparing sparse and non-sparse reward functions, as well as two different reward measures (F1 score and gini impurity). Additionally, the impact of introducing a separate tree reward is examined.

Sparsity of Rewards

For sparse reward functions, rewards are only given once the tree is complete. The reward given is the negative sum of gini scores of all leaves. Results for all data sets averaged over five runs are visible in table 4.8.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	19	0.931	0.831	9.4	0.944
Parkinsons	45	0.898	0.839	7.4	0.947
User Knowledge	61	0.823	0.726	10.2	0.928
Urban Land Cover	110	0.429	0.481	11	0.726
Student Performance	44	0.131	0.212	11	0.202
Average	56	0.63	0.62	10	0.757

Table 4.8: Average results over five runs for sparse rewards. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

Non-sparse node rewards assign a meaningful reward to each individual action. Results based on the difference of gini impurity scores before and after adding a new node can be seen in table 4.9

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	18	0.897	0.911	6.6	0.947
Parkinsons	41	0.89	0.873	6.2	0.948
User Knowledge	58	0.79	0.78	10.8	0.934
Urban Land Cover	105	0.512	0.439	11	0.801
Student Performance	43	0.114	0.167	11	0.203
Average	53	0.64	0.63	9.12	0.767

Table 4.9: Average results over five runs for non-sparse rewards based on gini impurity. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

Overall, both results are very similar. Non-sparse rewards perform slightly better in all categories with the most significant difference visible in tree size, where non-sparse rewards result in trees which are on average one node smaller. Individual results and standard deviation (which are not listed here for clarity of results) also suggest that non-sparse rewards are considerably more stable. Purely based on these results, non-sparse rewards should be preferred.

Reward Measures

From the possible choices of reward measures, F1 score and gini impurity are examined as node rewards. Both rewards are given as non-sparse rewards. Results for gini impurity can be seen in table 4.9. When using F1 score as the reward measure, the results seen in table 4.10 are the outcome. When compared to gini impurity, no significant difference can be found. Due to the similar nature of the reward calculation this result is to be expected. However, the results for gini impurity are slightly better in most categories, making the use of gini impurity preferable.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	16	0.919	0.799	5.2	0.958
Parkinsons	48	0.889	0.842	8.8	0.949
User Knowledge	59	0.888	0.854	11	0.94
Urban Land Cover	111	0.453	0.423	11	0.764
Student Performance	43	0.123	0.168	11	0.205
Average	55	0.655	0.617	9.4	0.763

Table 4.10: Average results over five runs for non-sparse F1 score based rewards. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

Tree Rewards

The tests conducted so far all employed separate tree rewards. When only using node rewards (in the form of gini impurity), the results visible in table 4.11 are the consequence. Not using tree rewards has no significant impact on the performance of the algorithm, with one exception. The average tree size is considerably larger. The larger tree size does not seem to bring any benefits and can therefore be considered undesirable.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	12	0.919	0.822	10	0.95
Parkinsons	46	0.909	0.871	10.5	0.949
User Knowledge	56	0.823	0.801	11	0.938
Urban Land Cover	112	0.488	0.514	11	0.774
Student Performance	44	0.143	0.199	11	0.208
Average	55	0.657	0.641	10.7	0.764

Table 4.11: Average results over five runs when only using node rewards (no tree rewards). Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

4.3.2 State Representations

Several different state representations have been introduced in the chapter 3 Method. Results obtained from all three representations (tree paths, full trees and data states) are compared in this section. As tree paths are the default configuration, previous results all use tree paths with table 4.9 forming the baseline. Using full tree states in the compact form (the default binary tree serialization produces states which are too large to handle appropriately) results in table 4.12. The data state representation is limited to 20 samples per state to reduce the state size. Results can be seen in table 4.13.

The results put the default configuration with tree paths on top. Full tree states are a close contender but are noticably slower and produce bigger trees. Data states fall off in all categories. When using data states, trees take significantly longer to build, perform worse and are bigger.

4.3.3 Reinforcement Learning

While reinforcement learning has several adjustable parameters, the focus of this section lies on a comparison of continuous and discrete actions. Discrete actions only choose the feature to split while continuous actions choose the feature and the splitting point. In theory, the better, less-greedy version are continuous actions.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	18	0.897	0.888	7.8	0.939
Parkinsons	48	0.889	0.876	8.6	0.95
User Knowledge	61	0.72	0.702	11	0.926
Urban Land Cover	116	0.563	0.534	11	0.799
Student Performance	40	0.106	0.172	11	0.199
Average	57	0.635	0.635	9.8	0.762

Table 4.12: Average results over five runs when using full tree states. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	22	0.866	0.888	9.6	0.947
Parkinsons	58	0.907	0.844	10	0.95
User Knowledge	52	0.871	0.860	11	0.937
Urban Land Cover	192	0.576	0.499	11	0.792
Student Performance	58	0.086	0.085	11	0.201
Average	77	0.661	0.635	10.52	0.765

Table 4.13: Average results over five runs when using data states. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

However, the increased difficulty of choosing feature and splitting point makes it a hard task. The following results show an objective comparison of both approaches. So far, discrete actions have been used with the baseline formed by table 4.9. Using continuous actions gives results visible in table 4.14.

Compared to discrete actions, continuous actions clearly fall off in all categories. The execution time is the highest among all tested configurations by a large margin while also producing the worst results. The results presented in table 4.14 only show tests for real data sets. When running the same configuration for generated data sets with adaptive difficulty, this configuration does manage to find optimal results for easier problems. The most difficult generated data sets which can be

solved stably with this configuration has the following parameters.

- *Size*: 100 samples
- *Features*: 7 features (including labels).
- *Classes*: 3 classes (equally distributed).

While this configuration is similar to the cryotherapy data set, the generated data set is still less complex.

Dataset	Time[s]	F1(train)	F1(test)	Tree size	F1(best)
Cryotherapy	209	0.832	0.507	11	0.85
Parkinsons	377	0.712	0.580	11	0.743
User Knowledge	205	0.639	0.572	11	0.737
Urban Land Cover	3840	0.153	0.081	11	0.437
Student Performance	473	0.069	0.047	11	0.164
Average	1021	0.481	0.357	11	0.586

Table 4.14: Average results over five runs when using continuous actions. Results are given for each data set and averaged over all data sets in the final row. The F1 scores show the performance of trees produced by the final (converged) reinforcement learning policy for the training data set (train) and the testing data set (test) as well as the tree size. The final column shows the performance of the best trees found (based on F1 score) during training.

4.3.4 Greediness

One of the fundamental research questions of this thesis is: Can the algorithm produce non-greedy results? This question can be answered by looking at the performance for the non-greedy data set discussed earlier in this chapter. To get a greedy baseline, an implementation of C4.5 provided by the machine learning tool WEKA [12] was used. It produces the tree visible in figure 4.1 when outputting a tree without post-processing. This tree has a perfect F1-score (1.0) but is not optimal due to the first split on feature f3.

The new algorithm of this thesis produces trees as seen in figure 4.2. This tree also has a perfect F1-score but is smaller due to not considering feature f3 as a beneficial split. As the algorithm manages to produce this result stably, it can be considered to be able to produce non-greedy results.

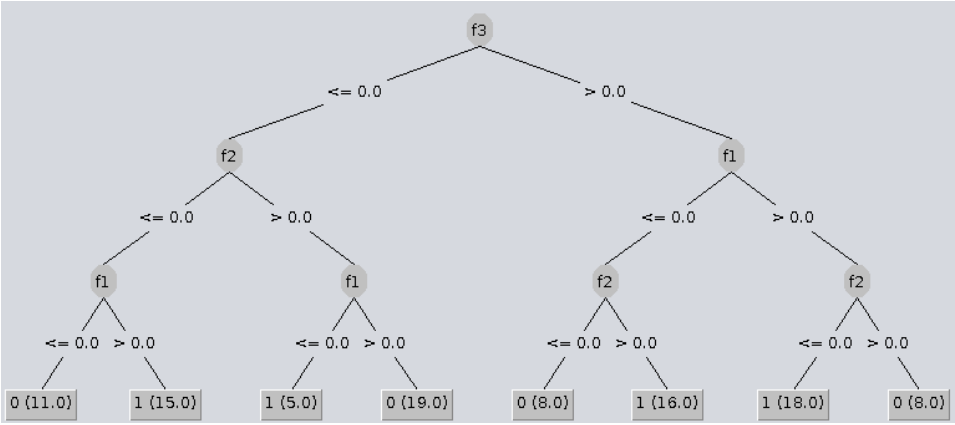

Figure 4.1: A greedy decision tree for the generated non-greedy data set produced by C4.5 (WEKA implementation) without post-processing.

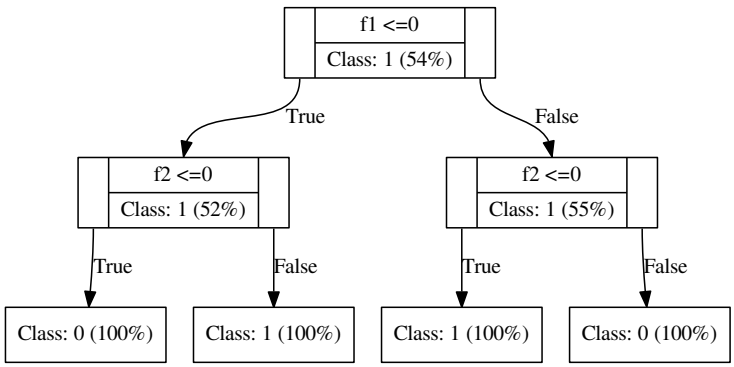

Figure 4.2: An optimal non-greedy decision tree for the generated non-greedy data set produced by the algorithm proposed in this thesis. Percentages represent the confidence of predicting the respective class in this leaf/node.

4.3.5 State of the Art

The results obtained so far compared internal configurations among each other. To determine whether the achieved results are competitive they have to be compared to other state of the art methods. The methods of choice are C4.5 and random forests with implementations provided by WEKA [12]. The default parameters from WEKA are used with the same training-test data split as used for this algorithm. When using default parameters for all three algorithms (with the exception of binary splits for C4.5), the results visible in table 4.15 show the performance measured by F1 score, averaged over 5 runs. Table 4.16 compares the average resulting tree sizes. For execution times, a comparison table is unnecessary as the WEKA implementations all finish in less than one second, making them faster by several orders of magnitude.

Apart from execution time, the decision trees produced by the algorithm proposed in this thesis perform noticeably better than C4.5 but also slightly worse than random forests. While performance is fairly similar, the tree sizes differ significantly. C4.5 always produces the same tree (as it is deterministic) and therefore a tree of the same size. The new algorithm, on the other hand, can produce substantially smaller trees and does so on average. Overall the comparison shows that traditional greedy algorithms execute faster but produce bigger trees and perform marginally better (random forests) or slightly worse (C4.5).

	New		C4.5	Random forest
Dataset	F1(best)	F1(test)	F1(test)	F1(test)
Cryotherapy	0.947	0.901	0.863	0.908
Parkinsons	0.948	0.917	0.884	0.938
User Knowledge	0.934	0.824	0.877	0.828
Urban Land Cover	0.801	0.778	0.756	0.83
Student Performance	0.203	0.173	0.128	0.11
Average	0.767	0.719	0.702	0.723

Table 4.15: Comparison of average F1 score performance over five runs between this algorithm, C4.5 and random forests. For this algorithm the F1 training score is omitted as the best F1 result can be considered more relevant. Results are given for each data set and averaged over all data sets in the final row.

Dataset	New	C4.5	Random forest
Cryotherapy	6.6	3	-
Parkinsons	6.2	11	-
User Knowledge	10.8	13	-
Urban Land Cover	11	13	-
Student Performance	11	121	-
Average	9.12	32.2	-

Table 4.16: Comparison of average tree sizes over five runs between this algorithm, C4.5 and random forests. The individual random forest trees are not accessible in WEKA but have a maximum depth of 100 and can therefore be considered large. Results are given for each data set and averaged over all data sets in the final row.

4.4 Discussion

Having presented the results objectively, a discussion of the results is in order. This section views the results from other angles and evaluates them not in terms of numbers but from a more practical point of view. The implications of the results for real world applications are discussed as well as insights about why certain behaviours can be observed in the results.

4.4.1 Productive Application

An important question when proposing a new algorithm is whether it can be used in a meaningful way in a productive environment. Does the algorithm have any benefits already existing methods cannot offer? Are there any deal breakers which make the algorithm infeasible for real world applications? In the context of this thesis, the short answer to these questions is: It depends. A more elaborate answer consists of the various advantages and disadvantages this algorithm brings to the table.

Starting with the disadvantages, the list reads as follows.

- The algorithm is comparably slow. The average execution time of approximately 50 seconds stands in stark contrast to sub one second execution times of greedy algorithms. This makes the algorithm unsuitable for any kind of real time applications.

- The number of adjustable parameters is very high. While the same parameters have been used for all data sets, adjusting them could certainly lead to better results. In general, a high number of parameters is undesirable, even if they are robust (which can be argued in this case).
- The implementation requires the use of multiple components. Implementing a reinforcement learning algorithm is not a trivial task.
- The execution time for big data sets is problematic. If execution time is not an issue, the algorithm can certainly be applied to bigger problems as well. But the substantially faster greedy algorithms produce similar results, making them a more interesting alternative (depending on the use case).

However, the advantages which can be identified are certainly useful for many use cases and might give the algorithm an edge over the greedy competitors.

- The resulting decision trees require no post-processing (especially pruning).
- The single tree performance (no ensembles) is comparable or even slightly better than C4.5, a state of the art algorithm.
- The stochastic nature of the learning process leads to diverse trees which are produced during training or from the finished trained model with slight adjustments to the way the policy selects actions. The diversity of trees offers two key advantages over deterministic methods: Additional insights can be gained when interpreting multiple diverse trees as opposed to only a single tree. Moreover, this diversity can most likely be used for creating ensemble methods (even though no such attempt has been made in this thesis).
- The trees are smaller but have similar or better performance. This can be a huge bonus for the interpretability of the tree.
- When building decision trees, the trade-off between performance and size is always problematic. This trade-off can be directly influenced by the choice of reward signals.

These advantages and disadvantages suggest that there are use cases for which our algorithm can be used. The advantages suggest a focus on data exploration. The natural diversity of resulting trees as well as the influence on the size-performance trade-off make it an intriguing choice for exploring data.

That being said, the results presented in the previous section are not necessarily representative of a real world application. For scientific purposes the same parameter configurations were used for all data sets and as few adaptions as possible were made when changing specific parameters. While this suggest robustness, it

also means that the results can most likely be improved in all aspects. The learning procedure usually converges faster than the maximum amount of episodes set in the default parameters which means execution time can be reduced. Tailoring the parameters to data sets would most likely result in better performance (F1 score and tree size) as well.

4.4.2 Insights

The results show several peculiarities which have to be discussed. Understanding why a certain behaviour occurs is not always straight-forward and sometimes cannot even be proven objectively. Especially when using black box approaches like reinforcement learning, explaining an observation often relies on experience and understanding of the matter. Since the algorithm proposed in this thesis is a combination of reinforcement learning and deterministic approaches, some more light can be shed on observations. The following sections will discuss the observations which can give us more insight to the internal workings of the algorithm.

Overall Performance

When viewing the results, the relatively stable performance of F1 scores stands out. While execution times and tree sizes vary from configuration to configuration, F1 scores stay on a high level in most experiments. The only strong exception are the results from continuous actions (see table 4.14). This suggests that the fairly consistent F1 scores stem from the way splits are calculated. And indeed, in the discrete action case, splits are calculated deterministically and greedily. Consequently, the splits which are calculated will always yield at least some improvement. So as long as the choice of features to split is sensible, the greedy calculation of the splitting criterion will produce results. This statement is supported by the fact that for easier data sets (like cryotherapy) in the discrete action case, trees consistently produce high F1 scores, even during earlier stages of training because the choice of feature is not as important as long as the tree is sufficiently large.
What sets apart the different configurations is the resulting tree size. To produce small trees which perform well, the choice of features to split is important. During training, the F1 score will increase slowly and steadily but start out with acceptable values. On the other hand, trees are always big at the beginning of training and will

become drastically smaller as the model is trained. If the model cannot be trained appropriately, the trees will not become as small as possible. As a consequence, the main criterion for judging configurations is the average tree size (which sets the configuration relying on tree path states and tree rewards apart).

Performance Versus Freedom

While implementing the algorithm certain assumptions had to made. A prominent assumption is that the position of new nodes should be decided by the environment rather than the reinforcement learning agent. Relying on the environment to choose node positions restricts the freedom of the learning agent but also makes its job easier. In the case of node positions, the deterministic approach theoretically only has little negative impact on the greediness of the algorithm, making it a save assumption.

This trade-off between performance and freedom of the learning agent is consistently observable in the results for the different configurations. The strongest restriction of freedom - the choice of splitting criteria - also produces the highest difference in results. One only has to compare continuous actions (table 4.14) and discrete actions (table 4.9) to see the difference. Restricting the learning agent to only choosing the features to split instead of both the feature and the split dramatically boosts the performance but also relies on external greedy splits, the impact of which likely prevents the creation of optimal trees in complex scenarios.
On a smaller scale, the trade-off is also visible in most other configuration comparisons. For example, sparse rewards only give sparse but highly accurate rewards making overall rewards less greedy. However, the performance suffers slightly.

Reward Functions

The comparison of different reward functions yielded straight-forward results. Sparse rewards perform worse than non-sparse rewards. Additionally, node rewards based on gini impurity work better than node rewards based on F1 score. And finally, the introduction of tree rewards reduces the tree size significantly. However, simply stating the facts does not explain them.

The difference between sparse and non-sparse rewards has already been discussed and is a performance versus freedom trade-off. From a more technical perspective, the temporal difference learning employed by the reinforcement learning agent seems unable to accurately assign the final sparse tree rewards to the individual actions which produced the reward. In other words, the temporal difference between action and reward is too big or the connections too complex. This behaviour can be observed in the erratic error calculation which seems unable to accurately predict the impact of actions on future states.

Using node rewards based on F1 score suffers from other problems. The premise of basing rewards directly on the performance measure used for evaluating the whole tree seems promising. However, there seem to be two problems.
The first one can be stated with certainty: F1 score was originally used for evaluating binary classification problems. A suitable workaround for multi-class classification is to use the weighted average of individual binary classifications. However, this average is ill-defined if no sample is predicted to belong to a certain class. At the beginning of building the tree, the tree only consists of the root node and two leaves and can predict two classes at most, making the F1 score ill-defined for the arguably most important node of the tree. This problem persists until at least one leaf per class is added to the tree.
The second problem is not as tangible. Simply put, F1 score is not perfectly suited for valuing the addition of single nodes (even taking the difference between before and after adding a node). While adding a node might lead to better separation in this node, this separation may not be reflected in the leaves of this node. This cause-effect relation simply seems to be better modeled by gini impurity.

F1 score does however have a place in the reward function as it is perfectly suited for assessing the performance of a finished tree. Using F1 scores in addition to tree sizes in tree rewards proves to be driving factor in keeping trees small. The reason behind this is simple. Node rewards are based on the value a single node adds to the tree locally and are, as such, greedy. When only relying on node rewards, the best policy is likely to add as many nodes as possible, each with small improvements, to gain the maximum accumulated reward. Tree rewards counter this policy by giving a comparably significant reward to finished trees which scales based on the tree size. The accumulated reward now heavily relies on the tree reward which is smaller when the tree is big.

State Representations

Comparing the different state representations is difficult as the interpretation of states happens in the black box model of reinforcement learning. Still, using tree path states is the most successful approach. The reasons behind this cannot be proven, but strong statements about the reasons can be made.

For data states, the reasons of weaker performance are the most obvious. Large state sizes make an interpretation of the states more difficult and simply make learning much more complex. Higher execution times with worse performance make this assessment plausible. Another problem arises from the limited access to active samples. The number of samples has to be kept small so the state size does not explode even further. The more samples are included in the states, the harder learning gets and the more learning tends to diverge (as experienced while experimenting with the parameter configurations). As a consequence, states are much more confusable with each other, making a mapping from states to actions hard and cause-effect relations vague.

Full tree states are fairly similar to tree path states. Both use the tree as the basis for the reward and both contain similar information. However, the way this information is arranged is likely the key difference that leads to differences in performance. As stated in the Method chapter, full tree states have to be compressed to a compact representation to keep the state size manageable. As this compact representation relies on the inclusion of node indices, serializations of different trees might look fairly similar (only distinguishable by node indices that might be close to each other). Similar states for different trees leads to confusion as generalization is not as successful. Instead, states basically have to be learned by heart.

To summarize, this chapter introduced the evaluation methodology and data sets used for evaluating key performance aspects of the algorithm. Various parameter configurations were tested to highlight some of the fundamental assumptions made in the method chapter and to answer essential research questions. The results obtained via this evaluation suggest a viable alternative to state of the art algorithms for specific use cases. To shed some light on the internal workings of the algorithm, the key aspects of the algorithm were discussed, most important of which are the performance-freedom trade-off, the impact of reward functions and the role of the choice of splitting criteria calculation.

5 Conclusion

Designing a novel approach, such as described in this thesis, is not only about coming up with a solution with satisfactory results. The challenge has to be considered in its entirety to produce different angles and perspectives from which the problem can be viewed. As such, a review of the research questions is in order.

The first, most fundamental questions is: *Is it possible to build decision trees with reinforcement learning?* This question cannot simply be answered with a *yes*, but with: Yes, but not only is it possible, we also defined numerous aspects and perspectives which have to be considered when building decision trees with reinforcement learning. Instead of only looking for a solution we also defined the whole environment of which the solution is one possible manifestation. The chapter 3 Method described this environment, first in general terms and then with more specific assumptions and options to handle each of the identified components. In summary the solution environment is spanned by the following components.

- The overall algorithm, which defines the responsibilities of each component, especially the amount of freedom the reinforcement learning agent has.
- State representations, which control the nature and amount of information available to the learning agent.
- Reward functions, which directly impact the way decision trees are built by choosing which actions to reward in what way.
- Reinforcement learning itself. The various different algorithms, parameters and models they use have widespread implications.
- The definition of when the tree building process is done.
- Post processing, which takes the unmodified output of the algorithm and can elevate it or cater it to specific needs.

These components define the concepts which have to be considered when designing a decision tree building algorithm with reinforcement learning.

The second research question reads: *What is a suitable approach to build decision trees with reinforcement learning?* After laying out the components of a solution, designing an algorithm required finding suitable combinations of possibilities which work together. As the space of possibilities is huge, some assumptions and compromises had to be made. Ultimately, a suitable approach was found and tested in various (partly vastly different) configurations. The main innovations involved in the algorithm were the introduction of tree based states and the separation of rewards into tree rewards and node rewards to find a balance between performance and tree size.
Due to these innovations, the algorithm outperforms another state of the art algorithm (C4.5) in most data sets used for evaluation (while also producing smaller trees) and performs slightly worse than the random forest ensemble method. However, due to the comparably long execution times and complex implementation, it is not suitable for all use cases. But the diversity of results and direct influence on the size-performance trade-off make it an interesting choice for data exploration.

The algorithm developed in this thesis not only produces competitive results to other state of the art methods but also has a positive answer to the final research question: *Can the algorithm produce non-greedy results?* The results clearly showed that the algorithm stably finds optimal solutions for a data set which is solved in non-optimal way by greedy algorithms.

Ultimately, all research questions could be answered satisfyingly, producing results that look very promising for current and future work. After all, the premise of this work is too intriguing to not work out: *Using a decision-making method to make decision about how to make decisions.*

5.1 Future Work

The various possibilities and ideas introduced in the chapter 3 Method leave a lot of room for further experimentation and improvements. Without going into too much detail, several items for future work can be identified. Due to the novel nature of this thesis, this the following list could be expanded even further.

- One of the main drawbacks of data states is the huge state size. As already mentioned, a way of circumventing this would be to use a recurrent neural

network and feed the samples individually. This way, the state size would be limited to single samples and the varying amount of active samples per node.
- Sticking to data states as they are, convolutional neural networks could be used to handle large state sizes.
- In this thesis, three reinforcement learning algorithms were used. As new and improved methods are presented, the performance of this algorithm would benefit from adapting them. Especially for continuous action spaces, which produced sub-optimal results.
- The performance-freedom trade-off could be explored in both directions. Handing more power to the learning agent could lead to better results while restricting the agent even further could boost the problematic execution times.
- No individual parameter tuning was performed in this thesis. The automatic adaption of parameters based on data set characteristics would be an interesting approach to circumvent manual parameter search.
- The stochastic nature of reinforcement learning could be exploited to generate diverse trees for ensemble methods.
- A limited amount of reward functions was tested in this thesis. Experimenting with reward signals opens a lot of room for further experimentation.
- The state representations can be changed in many ways. Especially combinations of the three mentioned possibilities (tree states, data states, extracted feature states) could lead to interesting results. The more the learning agent sees, the better informed its choices are.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] E. Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[3] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[4] L. Breiman. *Classification and regression trees*. Routledge, 2017.

[5] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[8] P. Cortez and A. M. G. Silva. Using data mining to predict secondary school student performance. 2008.

[9] A. Cutler et al. Remembering leo breiman. *The Annals of Applied Statistics*, 4(4):1621–1633, 2010.

[10] R. Dechter. *Learning while searching in constraint-satisfaction problems*. University of California, Computer Science Department, Cognitive Systems Laboratory, 1986.

[11] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.

[12] F. Eibe, M. Hall, and I. Witten. The weka workbench. online appendix for "data mining: Practical machine learning tools and techniques". *Morgan Kaufmann*, 2016.

[13] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

[14] J. Garcia and R. A. Koelling. Relation of cue to consequence in avoidance learning. *Foundations of animal behavior: classic papers with commentaries*, 4:374, 1996.

[15] S. Geva and J. Sitte. A cartpole experiment benchmark for trainable controllers. *IEEE Control Systems*, 13(5):40–51, 1993.

[16] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[17] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[18] T. Hothorn, K. Hornik, and A. Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical statistics*, 15(3):651–674, 2006.

[19] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[20] B. Johnson and Z. Xie. Classifying a high resolution image of an urban area using super-object information. *ISPRS journal of photogrammetry and remote sensing*, 83:40–49, 2013.

[21] B. A. Johnson. High-resolution urban land-cover classification using a competitive multi-scale object-based approach. *Remote Sensing Letters*, 4(2):131–140, 2013.

[22] H. T. Kahraman, S. Sagiroglu, and I. Colak. The development of intuitive knowledge classifier and the modeling of domain dependent data. *Knowledge-Based Systems*, 37:283–295, 2013.

[23] F. Khozeimeh, R. Alizadehsani, M. Roshanzamir, A. Khosravi, P. Layegh, and S. Nahavandi. An expert system for selecting wart treatment method. *Computers in biology and medicine*, 81:167–175, 2017.

[24] F. Khozeimeh, F. Jabbari Azad, Y. Mahboubi Oskouei, M. Jafari, S. Tehranian, R. Alizadehsani, and P. Layegh. Intralesional immunotherapy compared to cryotherapy in the treatment of warts. *International journal of dermatology*, 56(4):474–478, 2017.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[26] H. Laux, R. M. Gillenkirch, and H. Y. Schenk-Mathes. *Entscheidungstheorie*. Springer-Verlag, 2012.

[27] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

[28] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[29] M. A. Little, P. E. McSharry, S. J. Roberts, D. A. Costello, and I. M. Moroz. Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *Biomedical engineering online*, 6(1):23, 2007.

[30] J. F. Magee. *Decision trees for decision making*. Harvard Business Review, 1964.

[31] P. McCorduck. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. AK Peters/CRC Press, 2009.

[32] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[34] R. Parloff. Why deep learning is suddenly changing your life. *Fortune. New York: Time Inc*, 2016.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[36] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[37] R. J. Quinlan. C4. 5: Programs for machine learning. 1993.

[38] L. E. Raileanu and K. Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[39] S. Raschka. Python machine learning. `https://github.com/rasbt/python-machine-learning-book`, 2015. Accessed on 27.12.2018.

[40] S. Raschka. *Python Machine Learning*. Packt Publishing, Birmingham, UK, 2015.

[41] J. J. Rodriguez, L. I. Kuncheva, and C. J. Alonso. Rotation forest: A new classifier ensemble method. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1619–1630, 2006.

[42] L. Rokach and O. Z. Maimon. *Data mining with decision trees: theory and applications*, volume 69. World scientific, 2008.

[43] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[44] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[45] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[46] C. Strobl, J. Malley, and G. Tutz. An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests. *Psychological methods*, 14(4):323, 2009.

[47] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[48] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[49] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[50] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[51] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[52] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[53] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[54] X. Zhu. Semi-supervised learning literature survey. 2005.