design of learning algorithms for belief networks is also a very active area of research, and a brief sketch is provided in Section 19.6.

The choice of representation for the desired function is probably the most important issue facing the designer of a learning agent. As well as affecting the nature of the learning algorithm, it can affect whether the problem is feasible at all. As with reasoning, in learning there is a fundamental trade-off between expressiveness—is the desired function representable in the representation language—and efficiency—is the learning problem going to be tractable for a given choice of representation language. If one chooses to learn sentences in a nice, expressive language such as first-order logic, then one will probably have to pay a heavy penalty in terms of both computation time and the number of examples required to learn a good set of sentences.

By “a good set of sentences,” we mean a set that not only correctly reflects the experiences the agent has already had, but also one that correctly predicts its future experiences. Therein lies one of the most vexing philosophical problems of all time. How can one possibly know that one’s learning algorithm has produced a theory that will correctly predict the future? And if one does not, then how can one say that the algorithm is any good? Certainly, if one cannot say for sure that an algorithm is any good, then one cannot hope to design good learning algorithms! In Section 18.6, we discuss a mathematical approach to the study of induction algorithms that provides tentative answers to these questions, and also sheds considerable light on the complexity of learning different kinds of function representations.

18.3 LEARNING DECISION TREES

Decision tree induction is one of the simplest and yet most successful forms of learning algorithm. It serves as a good introduction to the area of inductive learning, and is easy to implement. We first describe the performance element, and then show how to learn it. Along the way, we will introduce many of the ideas and terms that appear in all areas of inductive learning.

Decision trees as performance elements

A decision tree takes as input an object or situation described by a set of properties, and outputs a yes/no “decision.” Decision trees therefore represent Boolean functions. Functions with a larger range of outputs can also be represented, but for simplicity we will usually stick to the Boolean case. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labelled with the possible values of the test. Each leaf node in the tree specifies the Boolean value to be returned if that leaf is reached.

As an example, consider the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate\(^4\) `WillWait`, where the definition is expressed as a

\(^4\) The term goal concept is often used. Unfortunately, the word “concept” has been used in so many different ways in machine learning that we think it best to avoid it for a few years.
decision tree. In setting this up as a learning problem, we first have to decide what properties or attributes are available to describe examples in the domain. Suppose we decide on the following list of attributes:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant’s price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai, or Burger).
10. WaitEstimate: the wait estimated by the host (0–10 minutes, 10–30, 30–60, >60).

The decision tree usually used by the first author for this domain is shown in Figure 18.4. Notice that the tree does not use the Price and Type attributes, considering these to be irrelevant given the data it has seen. Logically, the tree can be expressed as a conjunction of individual implications corresponding to the paths through the tree ending in Yes nodes. For example, the path for a restaurant full of patrons, with an estimated wait of 10–30 minutes when the agent is not hungry is expressed by the logical sentence

\[ \forall r \ Patrons(r, \text{Full}) \land \text{WaitEstimate}(r, 0-10) \land \text{Hungry}(r, N) \Rightarrow \text{WillWait}(r) \]

Expressiveness of decision trees

If decision trees correspond to sets of implication sentences, a natural question is whether they can represent any set. The answer is no, because decision trees are implicitly limited to talking about a single object. That is, the decision tree language is essentially propositional, with each attribute test being a proposition. We cannot use decision trees to represent tests that refer to two or more different objects, for example,

\[ \exists r_2 \ Nearby(r_2, r) \land \text{Price}(r, p) \land \text{Price}(r_2, p_2) \land \text{Cheaper}(p_2, p) \]

(is there a cheaper restaurant nearby). Obviously, we could add another Boolean attribute with the name CheaperRestaurantNearby, but it is intractable to add all such attributes.

Decision trees are fully expressive within the class of propositional languages, that is, any Boolean function can be written as a decision tree. This can be done trivially by having each row in the truth table for the function correspond to a path in the tree. This would not necessarily be a good way to represent the function, because the truth table is exponentially large in the number of attributes. Clearly, decision trees can represent many functions with much smaller trees.

5 Meanwhile, the automated taxi is learning whether to wait for the passengers in case they give up waiting for a table and want to go on to another restaurant.

6 One might ask why this isn’t the job of the learning program. In fact, it is, but we will not be able to explain how it is done until Chapter 21.
For some kinds of functions, however, this is a real problem. For example, if the function is the parity function, which returns 1 if and only if an even number of inputs are 1, then an exponentially large decision tree will be needed. It is also difficult to use a decision tree to represent a majority function, which returns 1 if more than half of its inputs are 1.

In other words, decision trees are good for some kinds of functions, and bad for others. Is there any kind of representation that is efficient for all kinds of functions? Unfortunately, the answer is no. We can show this in a very general way. Consider the set of all Boolean functions on \( n \) attributes. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table. The truth table has \( 2^n \) rows, because each input case is described by \( n \) attributes. We can consider the “answer” column of the table as a \( 2^n \) bit number that defines the function. No matter what representation we use for functions, some of the functions (almost all of them, in fact) are going to require at least this many bits to represent.

If it takes \( 2^n \) bits to define the function, this means that there are \( 2^{2^n} \) different functions on \( n \) attributes. This is a scary number. For example, with just six Boolean attributes, there are about \( 2 \times 10^{19} \) different functions to choose from. We will need some ingenious algorithms to find consistent hypotheses in such a large space.

![Decision Tree Diagram](image)

Figure 18.4 A decision tree for deciding whether to wait for a table.
## Inducing decision trees from examples

An example is described by the values of the attributes and the value of the goal predicate. We call the value of the goal predicate the classification of the example. If the goal predicate is true for some example, we call it a positive example; otherwise we call it a negative example. A set of examples $X_1, \ldots, X_{12}$ for the restaurant domain is shown in Figure 18.5. The positive examples are ones where the goal WillWait is true ($X_1, X_3, \ldots$) and negative examples are ones where it is false ($X_2, X_5, \ldots$). The complete set of examples is called the training set.

<table>
<thead>
<tr>
<th>Example</th>
<th>Attributes</th>
<th>Goal</th>
<th>WillWait</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$X_2$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$X_3$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$X_4$</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$X_5$</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$X_6$</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$X_7$</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$X_8$</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$X_9$</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$X_{10}$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$X_{11}$</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$X_{12}$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Figure 18.5 Examples for the restaurant domain.

The problem of finding a decision tree that agrees with the training set might seem difficult, but in fact there is a trivial solution. We could simply construct a decision tree that has one path to a leaf for each example, where the path tests each attribute in turn and follows the value for the example, and the leaf has the classification of the example. When given the same example again, the decision tree will come up with the right classification. Unfortunately, it will not have much to say about any other cases!

The problem with this trivial tree is that it just memorizes the observations. It does not extract any pattern from the examples and so we cannot expect it to be able to extrapolate to examples it has not seen.

Extracting a pattern means being able to describe a large number of cases in a concise way. Rather than just trying to find a decision tree that agrees with the examples, we should try to find a concise one, too. This is an example of a general principle of inductive learning often called **Occam's razor**: The most likely hypothesis is the simplest one that is consistent with all observations. Some people interpret this as meaning “the world is inherently simple.” Even if the world is complex, however, Occam’s razor still makes sense. There are far fewer simple

---

7. The same example or an example with the same description—this distinction is very important and we will return to it in Chapter 21.

8. Sometimes spelled “Occam,” although the origin of this corruption is obscure.
hypotheses than complex ones, so that there is only a small chance that any simple hypothesis that is wildly incorrect will be consistent with all observations. Hence, other things being equal, a simple hypothesis that is consistent with the observations is more likely to be correct than a complex one. We discuss hypothesis quality further in Section 18.6.

Unfortunately, finding the smallest decision tree is an intractable problem, but with some simple heuristics, we can do a good job of finding a smallish one. The basic idea behind the DECISION-TREE-LEARNING algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. This way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

Figure 18.6 shows how the algorithm gets started. We are given 12 training examples, which we classify into positive and negative sets. We then decide which attribute to use as the first test in the tree. Figure 18.6(a) shows that Patrons is a fairly important attribute, because if the value is None or Some, then we are left with example sets for which we can answer definitively (No and Yes, respectively). (If the value is Full, we will need additional tests.) In Figure 18.6(b) we see that Type is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive and negative answers. We consider all possible attributes in this way, and choose the most important one as the root test. We leave the details of how importance is measured for Section 18.4, because it does not affect the basic algorithm. For now, assume the most important attribute is Patrons.

After the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one fewer attribute. There are four cases to consider for these recursive problems:

1. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.6(c) shows Hungry being used to split the remaining examples.
2. If all the remaining examples are positive (or all negative), then we are done: we can answer Yes or No. Figure 18.6(c) shows examples of this in the None and Some cases.
3. If there are no examples left, it means that no such example has been observed, and we return a default value calculated from the majority classification at the node's parent.
4. If there are no attributes left, but both positive and negative examples, we have a problem. It means that these examples have exactly the same description, but different classifications. This happens when some of the data are incorrect; we say there is noise in the data. It also happens when the attributes do not give enough information to fully describe the situation, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

We continue to apply the DECISION-TREE-LEARNING algorithm (Figure 18.7) until we get the tree shown in Figure 18.8. The tree is distinctly different from the original tree shown in Figure 18.4, despite the fact that the data were actually generated from an agent using the original tree.

One might conclude that the learning algorithm is not doing a very good job of learning the correct function. This would be the wrong conclusion to draw. The learning algorithm looks at the examples, not at the correct function, and in fact, its hypothesis (see Figure 18.8) not only agrees with all the examples, but is considerably simpler than the original tree. The learning algorithm has no reason to include tests for Raining and Reservation, because it can classify all
the examples without them. It has also detected an interesting regularity in the data (namely, that the first author will wait for Thai food on weekends) that was not even suspected. Many hours have been wasted by machine learning researchers trying to debug their learning algorithms when in fact the algorithm was behaving properly all along.

Of course, if we were to gather more examples, we might induce a tree more similar to the original. The tree in Figure 18.8 is bound to make a mistake; for example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. For a case where Hungry is false, the tree says not to wait, but the author would certainly wait. This raises an obvious question: if the algorithm induces a consistent but incorrect tree from the examples, how incorrect will the tree be? The next section shows how to analyze this experimentally.
function DECISION-TREE-LEARNING(examples, attributes, default) returns a decision tree
inputs: examples, set of examples
        attributes, set of attributes
        default, default value for the goal predicate
if examples is empty then return default
else if all examples have the same classification then return the classification
else if attributes is empty then return MAJORITY-VALUE(examples)
else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value v_i of best do
        examples_i ← {elements of examples with best = v_i}
        subtree ← DECISION-TREE-LEARNING(examples_i, attributes - best,
                                             MAJORITY-VALUE(examples_i))
        add a branch to tree with label v_i and subtree subtree
    end
return tree

Figure 18.7 The decision tree learning algorithm.

Figure 18.8 The decision tree induced from the 12-example training set.