# Learning Agents

An agent is learning if it improves its performance on future tasks after making observations about the world.

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn’t the designers just program in that improvement to begin with? There are three main reasons:

1. The designers cannot anticipate all possible situations that the agent might find itself in
2. The designers cannot anticipate all changes over time
3. Sometimes human programmers have no idea how to program a solution themselves.

All components of an agent can be improved by learning.

Chapter 2 described several agent designs. The components of these agents include:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent’s utility.

Each of these components can be learned. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts “Brake!” the agent might learn a condition– action rule for when to brake (component 1); the agent also learns every time the instructor does not shout. By seeing many camera images that it is told contain buses, it can learn to recognize them (2). By trying actions and observing the results—for example, braking hard on a wet road—it can learn the effects of its actions (3). Then, when it receives no tip from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (4).

## Inductive Learning vs Deductive Learning

1. **Inductive Learning** consists of learning a function from a set of input-output pairs.
2. **Deductive Learning** consists of deriving a new rule from an old rule which is more efficient than the old one.

## Supervised Learning VS Unsupervised Learning VS Reinforcement Learning

In **SUPERVISED LEARNING,** the agent observes some example input–output pairs and learns a function that maps from input to output. In component 1 above, the inputs are percepts and the output are provided by a teacher who says “Brake!” or “Turn left.” In component 2, the inputs are camera images and the outputs again come from a teacher who says “that’s a bus.” In 3, the theory of braking is a function from states and braking actions to stopping distance in feet. In this case the output value is available directly from the agent’s percepts (after the fact); the environment is the teacher.

In **REINFORCEMENT LEARNING,** the agent learns from a series of reinforcements (rewards or punishments). Every time a reinforcement occurs, a reinforcement learning agent decides which of the actions prior to the reinforcement were most responsible for it.

In **UNSUPERVISED LEARNING,** the agent discovers patterns on data even though the data is not already labelled. The most common unsupervised learning task is **clustering**: detecting potentially useful clusters of input examples. For example, a taxi agent might gradually develop a concept of “good traffic days” and “bad traffic days” without ever being given labelled examples of each by a teacher.

In **SEMI-SUPERVISED LEARNING,** the agent learns from both labelled and unlabelled data at the same time. Semi-supervised learning surpasses the classification performance that can be obtained either by discarding the unlabelled data and doing supervised learning or by discarding the labels and doing unsupervised learning. In other words, it is a middle way between supervised and unsupervised learning.

## Supervised Learning

Text, letter

Description automatically generated

Here x and y can be any value; they need not be numbers. The function h is a **HYPOTHESIS**.

Learning in the context of supervised learning consists of searching through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis we give it a **TEST SET** of examples that are distinct from the training set.

An hypothesis function h is said to be **consistent** when it agrees with all training data.

An hypothesis function h is said to **generalise well** when it predicts the value of y for previously unseen data.

Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, **P(Y | x).**

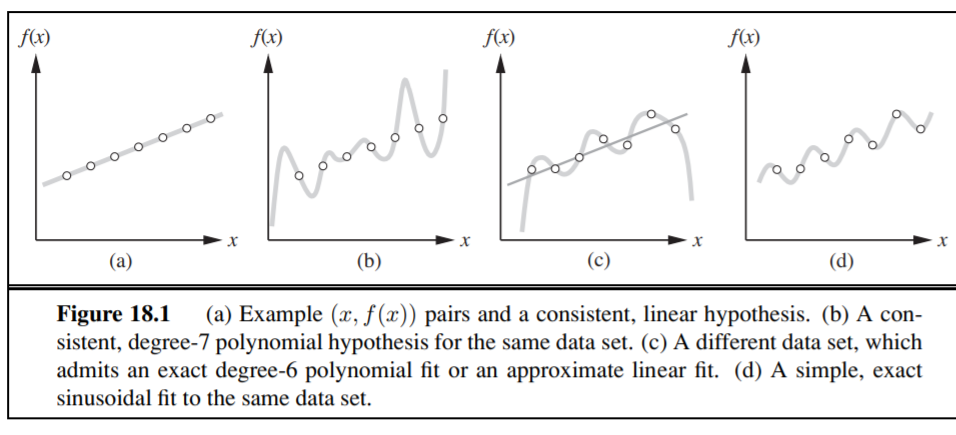
When the codomain of the unknown function f is a finite set of values (such as sunny, cloudy or rainy) then the learning problem is called **CLASSIFICATION.**

A **REGRESSION** problem occurs when the codomain is an infinite set of values (numbers). (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found exactly the right real-valued number for y is 0.)

Given an Hypothesis Space H, how do we choose from among multiple consistent hypotheses?

One answer is to prefer the simplest hypothesis consistent with the data. This principle is called **OCKHAM’S RAZOR**, after the 14th-century English philosopher William of Ockham, who used it to argue sharply against all sorts of complications. Defining simplicity is not easy, but it seems clear that a degree-1 polynomial is simpler than a degree-7 polynomial.

## Trade-off Principle

In general, there is a trade-off between complex hypotheses that fit (consistent) the training data well and simpler hypotheses that may generalize better with previously unseen data.

In other words, it is not guaranteed that a consistent hypothesis generalises better than a non-consistent hypothesis. Therefore, the best hypothesis that can be chosen for a supervised learning problem may be one that is not consistent but generalises very well with previously unseen data.

We say that a learning problem is **REALIZABLE** if the hypothesis space contains the true function. Unfortunately, we cannot always tell whether a given learning problem is **realizable**, because the true function is not known.

Since it is not always possible to select the true function f because the learning problem may not be realisable then supervised learning consists of selecting the function which best approximate the true function that is the function that is most likely to be the true function.

## A screenshot of a newspaper Description automatically generatedTrade-off Expressiveness and Complexity

There is a trade-off between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space.

Indeed, the more expressive the hypothesis space is the more functions can be represented and so finding the hypothesis h that is most likely to be the true function f takes more time.

A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing h(x) when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate.

However, while an expressive language makes it possible for a simple concise hypothesis to fit the data, restricting the expressiveness of the language means that any consistent hypothesis must be very complex.

For a wide variety of problems, the decision tree format yields a nice, concise result. But some functions cannot be represented concisely. For example, the majority function, which returns true if and only if more than half of the inputs are true, requires an exponentially large decision tree.

Additionally, given an input vector of n Boolean attributes, the domain consists of 2n input vectors. If the codomain has just 2 values (TRUE and FALSE) then there are different functions that can be represented.

Given an input vector of n Boolean attributes then there are different functions that can be represented but even more decision trees to choose because a given function can be represented using multiple decision trees. Then, given a Boolean function that accepts an input vector of n Booleans attributes there are more than decision trees that we can choose from and consequently an efficient algorithm is needed otherwise the time to build a decision tree of a function would be EXPONENTIAL.

## Algorithm for computing decision tree given input-output pairs

Given a set of input-output pairs our aim is to build a decision tree that is consistent with the data and is as small as possible. However, finding the smallest consistent decision tree given a set of data is an intractable problem as we should search among more than decision trees.

Only a heuristic can be used to try to minimise the size of the decision tree. This heuristic consists of selecting at each branch of the tree the attribute that categorizes the data in the larger extent.

The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By “most important attribute,” we mean the one that makes the most difference to the classification of an example. That 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 shallow.

Figure 18.4(a) shows that Type is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples. On the other hand, in (b) we see 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 are left with a mixed set of examples. In general, 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 less attribute.

Diagram

Description automatically generated

Table

Description automatically generated

However, it is important to notice that splitting by the most important attribute is just a greedy heuristic and so it is not guaranteed to find the smallest consistent decision tree. Indeed, for example in fig. 18.4 even though splitting by the attribute Type may not seem a good split it may happen that it is possible to get all leaf nodes at the next level by using a certain attribute. On the other hand, it may happen that by using the more promising attribute Patrons leads to a decision tree that extends deeper.

The DECISION-TREE LEARNING algorithm pseudocode is provided below and also the obtained decision tree by applying the algorithm to the dataset provided in fig. 18.3

Diagram

Description automatically generatedThe algorithm works in the following way:

1. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node’s parent. (LINE 1 pesudocode)
2. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No. (LINE 2 pseudocode)
3. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or noise in the data; because the domain is nondeterministic; or because we can’t observe an attribute that would distinguish the examples. The best we can do is return the plurality classification of the remaining examples. (LINE 3 pseudocode)
4. If all cases above do not occur then choose the best attribute to split them (LINE 5 pseudocode)

It is important to notice that the DECISION-TREE is built from the dataset provided and not from the real function and so the tree built is not guaranteed to be equal to the real function. Therefore, the dataset provided plays an important role when creating decision trees, indeed, given two datasets extracted from the same problem it is very likely that the two decision trees built from either of the two datasets will look pretty different.

In the case of a stochastic or non-deterministic problem, multiple inputs in the data set provided may be the same but have different outputs. In these cases, the algorithm provided above will output the most frequent output for a given input.

Graphical user interface

Description automatically generated

## Entropy

Entropy is a measure of uncertainty (unpredictability) of a random variable. Lower entropy corresponds to a reduction in the uncertainty of a random variable while larger entropy corresponds to an increase in the unpredictability of a random variable.

1. **Low entropy** => low level of uncertainty about the outcome of a random variable
2. **High entropy** => high level of uncertainty about the outcome of a random variable

Similarly, we can define entropy in terms of the amount of information acquired from a random variable.

1. **Low entropy** => low amount of information acquired from the outcome of a random variable as there is a low level of uncertainty about the outcome
2. **High entropy** => high amount of information acquired from the outcome of a random variable as there is a high level of unpredictability about the outcome

We have introduced the concept of entropy because we have said before that we use a greedy approach to select at each call of the decision tree procedure the best attribute that goes as far as possible towards a categorization of the examples by using the IMPORTANCE function.

It turns out that we can implement the IMPORTANCE function using the concept of entropy.

Text, letter

Description automatically generatedFirstly, we are going to see how to actually compute the entropy of a random variable:

Now, we are going to explain how to use entropy to choose the best attribute at each call of the decision tree procedure.

Each call of the decision tree will have n examples from the dataset. We can compute the entropy of these examples in the following way (assuming a BOOLEAN classification):

* Let p be the positive examples
* Let n be the negative examples

Then the entropy of the set of n examples is . The previous value tells us about the level of unpredictability of the n examples and so given that the value is low then there is a low level of unpredictability in the examples which means that there is a larger amount of either positive examples or negative examples, on the contrary if the value is high then there is roughly a balanced amount of positive and negative examples.

Once we compute the entropy of the set of n examples, we need to find the attribute A that goes as far as possible towards the classification of the examples and so an attribute A that reduces the entropy of the examples as much as possible.

It is possible to compute the expected entropy of an attribute A on a set of n examples in the following way:

* Given that attribute A has k values then it will split the examples in k partitions
* We can compute the entropy of each of these partitions weighted by the probability that each partition occurs in relation to the set of n examples. Given that each partition k has pk positive examples and nk positive examples then the probability of each partition to occur in relation to the set of n examples is where p is the positive examples and n the negative examples in the set of n examples and the entropy of the partition is .
* In the end we obtain the expected entropy for the attribute A also called Remainder(A) and we need to compute . Remainder(A) is computed in the following way:

A close up of a clock

Description automatically generated

assuming that the Attribute A has d possible values

The Remainder(A) tells us that given that we arrive at an hypothetic node that describes A in the decision tree then it is **EXPECTED** to arrive at another node B from A that has an entropy equal to Remainder(A).

* We need to do the above procedure for all possible attributes left and we need to select the attribute A that reduces as much as possible the entropy of the set by using the following formula

Indeed the greater the value of is, the more reduction in entropy occurs and so more gain we obtain towards a classification of the examples.

However, always remember that such a procedure is just a greedy procedure and does not ensure to find the smallest possible decision tree.

Text, letter

Description automatically generated

The procedure of the Importance function as outlined above using entropy stresses once again that the resulting decision tree is heavily influenced by the data set that we choose as we compute Remainder(A) by computing the probability of each of the partitions that are produced times their entropies which depends on the data set. Thus, in order to obtain a decision tree that really reflects the true function we need to select the data set appropriately.

## Overfitting

On some problems, the DECISION-TREE-LEARNING algorithm will generate a large tree when there is actually no pattern to be found. Consider the problem of trying to predict whether the roll of a die will come up as 6 or not. Suppose that experiments are carried out with various dice and that the attributes describing each training example include the colour of the die, its weight, the time when the roll was done, and whether the experimenters had their fingers crossed. If the dice are fair, the right thing to learn is a tree with a single node that says “no,” But the DECISION-TREE-LEARNING algorithm will seize on any pattern it can find in the input. If it turns out that there are 2 rolls of a 7-gram blue die with fingers crossed and they both come out 6, then the algorithm may construct a path that predicts 6 in that case. This problem is called **OVERFITTING.**

In other words, overfitting occurs when a supervised learning agent learns an hypothesis function h from a training set which is consistent with the training set but actually it does not generalise well with previously unseen data because it has captured patterns in the training set which represent noise or are not actually relevant to the domain of the problem in consideration.

When is Overfitting more likely to occur:

1. When the hypothesis space is large. Indeed, when the hypothesis space is large it is likely to select a complex function that is consistent with the training set but may generalise bad with previously unseen data
2. When the input attributes is large. Indeed, when the input attributes are many it is likely that some of these variables are not actually relevant to the domain of the problem and so represent unnecessary noise

How to combat Overfitting:

1. Select large training set

A close up of a person

Description automatically generated

One important property of decision trees is that it is possible for a human to understand the reason for the output of the learning algorithm. (Indeed, this is a legal requirement for financial decisions that are subject to anti-discrimination laws.) This is a property not shared by some other representations, such as neural networks.

A picture containing text, newspaper

Description automatically generated­

## Equivalence between Remainder(A) and H(Y|A)

I have proved in a paint file in this file’s folder that the definition of Remainder(A) and H(Y|A) of the lecture are actually equivalent. Note that Y is the random variable we want to know its entropy of.

Thus, now let’s see what the formula for H(Y|A) is:

A picture containing diagram

Description automatically generatedWe know how to compute Gain(A), in the prof lecture it is called Gain(Y,A).

We know that Gain(A) = H(Y) – Remainder(A).

In the prof lecture it is equivalent to Gain(Y,A) = H(Y) – H(Y|A) which literally means the following:

The gain in the entropy of Y given that we choose attribute A is equal to the entropy of Y – the entropy of Y given that we choose attribute A.