MODULE 4

1. Learning from Examples

- Forms of Learning
- Supervised Learning
- Learning Decision Trees
- Ensemble Learning
- Reinforcement Learning: Introduction
- Passive Reinforcement Learning
- Active Reinforcement Learning
- Generalization in Reinforcement Learning

2. Natural Language Processing

- Language models
- Text Classification
- Information Retrieval

3. Robotics

- Introduction
- Robotics Hardware
- Robotics Perception

LEARNING FROM EXAMPLES

Forms of Learning

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which component is to be improved.
- What prior knowledge the agent already has.
- What representation is used for the data and the component.
- What feedback is available to learn from.

There are three types of feedback that determine the three main types of learning:

- In unsupervised learning the agent learns patterns in the input even though no explicit feedback is supplied.
- In reinforcement learning the agent learns from a series of reinforcements—rewards or punishments.
- In supervised learning the agent observes some example input–output pairs and learns a function that maps from input to output.
- In semi-supervised learning we are given a few labeled examples and must make what we can of a large collection of un labeled examples.

Supervised Learning

The task of supervised learning is this:

Given a **training set** of N example input–output pairs

$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)$,

where each y_j was generated by an unknown function $y = f(x)$,

discover a function h that approximates the true function f.

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

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 output y is one of a finite set of values, the learning problem is called classification, and is called Boolean or binary classification if there are only two values.
When y is a number, the learning problem is called regression.

Learning Decision Trees

- A decision tree represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value.
- The input and output values can be discrete or continuous.
- For now, we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a positive example) or false (a negative example).
- A decision tree reaches its decision by performing a sequence of tests.

As an **example,** we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate **WillWait.**
First we list the attributes that we will consider as part of the input:

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, or >60).

Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with Patrons =Full and WaitEstimate =0–10 will be classified as positive (i.e., yes, we will wait for a table).

An example for a Boolean decision tree consists of an (x, y) pair, where x is a vector of values for the input attributes, and y is a single Boolean output value.

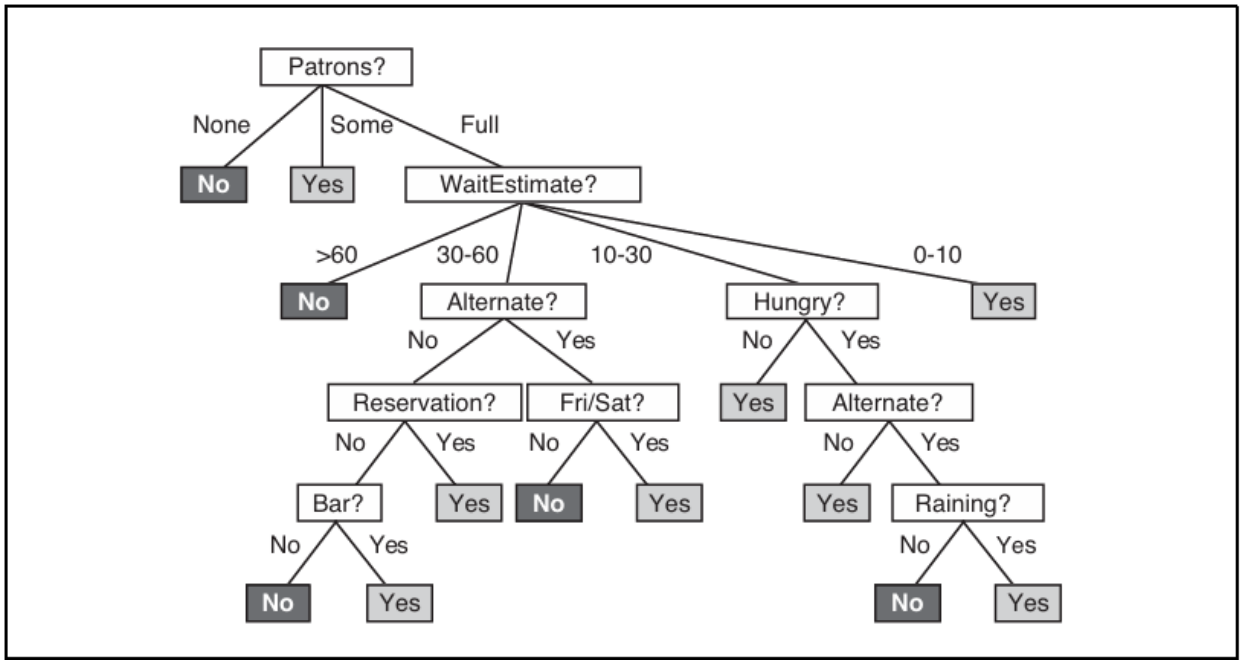

Figure 4: A decision tree for deciding whether to wait for a table.

A training set of 12 examples is shown in Figure 4.1. The positive examples are the ones in which the goal WillWait is true (x1, x3,...); the negative examples are the ones in which it is false (x2, x5,...).

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 4.1: Examples for the restaurant domain.

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 sub problems that can then be solved recursively.

Passive Reinforcement Learning

- In passive learning, the agent's policy π is fixed: in state s, it always executes the action π(s).
- Its goal is simply to learn how good the policy is—that is, to learn the utility function Uπ(s).

Figure 4.2 shows a policy for that world and the corresponding utilities. Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm described in Section 17.3. The main difference is that the passive learning agent does not know the **transition model P (s' | s , a),** which specifies the probability of reaching state s' from state s after doing action a; nor does it know the **reward function R(s),** which specifies the reward for each state.

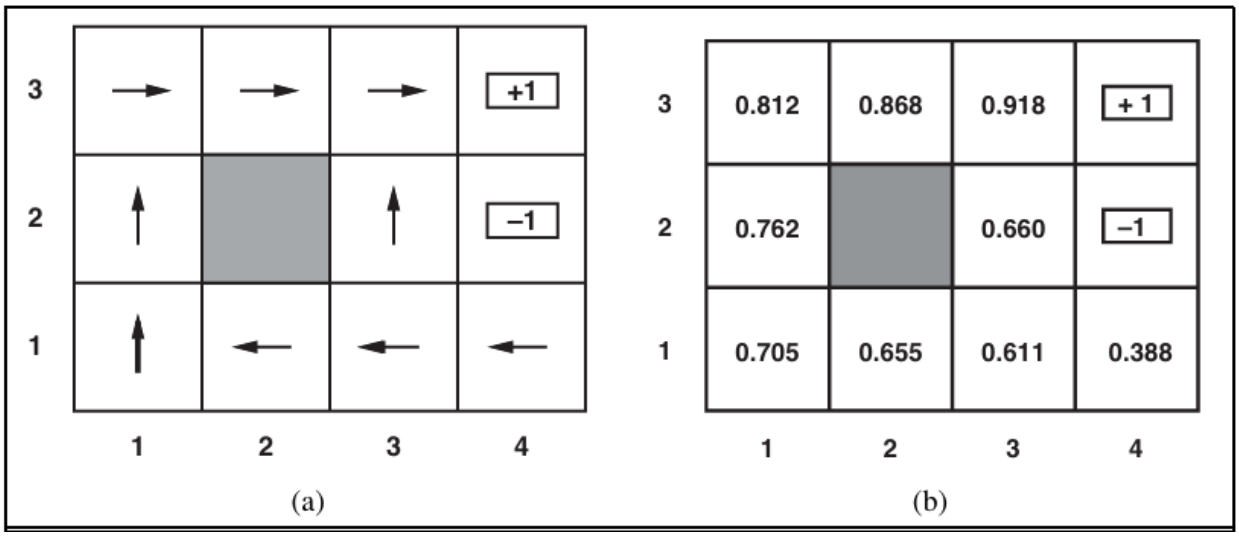

Figure 4.2: (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of R(s)= − 0.04 in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

The agent executes a set of trials in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the

terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state. Typical trials might look like this:

$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (2,1)_{-.04} \rightsquigarrow (3,1)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (4,2)_{-1} .$$

Note that each state percept is subscripted with the reward received. The object is to use the information about rewards to learn the expected utility Uπ(s) associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed.

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

where R(s)is the reward for a state, St (a random variable) is the state reached at time t when executing policy π, and S0 =s. We will include a discount factor γ in all of our equations, but for the 4×3 world we will set γ =1.

Generalization in Reinforcement Learning

- Utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple.
- Function approximation, which simply means using any sort of representation for the Q-function other than a lookup table.
- Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit.
- The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.
- That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states.

For **example**, suppose we represent the utilities for the 4×3 world using a simple linear function.

The features of the squares are just their x and y coordinates, so we have

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y \quad \Longrightarrow \text{eq.1}$$

Thus, if (θ0, θ1, θ2) = (0.5, 0.2, 0.1),

$$\text{then } \hat{U}_\theta(1, 1) = 0.8$$

As with neural network learning, we write an error function and compute its gradient with respect to the parameters.

If uj (s) is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total:

$$E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2.$$

The rate of change of the error with respect to each parameter θi is ∂Ej / ∂θi, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} .$$

This is called the Widrow–Hoff rule, or the delta rule, for online least-squares. For the linear function approximator ˆ Uθ(s) in Equation 1, we get three simple update rules:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) , \\
\theta_1 &\leftarrow \theta_1 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) x , \\
\theta_2 &\leftarrow \theta_2 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) y .
\end{aligned}$$

NATURAL LANGUAGE PROCESSING

Language Models

- A language can be defined as a set of strings; "print (2 + 2)" is a legal program in the language Python, whereas "2) + (2 print" is not.
- Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a grammar.
- Formal languages also have rules that define the meaning or semantics of a program.

1) N-gram character models

- A written text is composed of characters—letters, digits, punctuation, and spaces in English.
- Thus, one of the simplest language models is a probability distribution over sequences of characters.
- we write P(c1:N) for the probability of a sequence of N characters, c1 through cN.
- A sequence of written symbols of length n is called an n-gram, with special case "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. A model of the probability distribution of n-letter sequences is thus called an **n-gram model.**
- An n-gram model is defined as a Markov chain of order n −1.

2) Smoothing n-gram models

- The process of adjusting the probability of low-frequency counts is called smoothing.
- A better approach is a backoff model, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n −1)-grams.
- Linear interpolation smoothing is a backoff model that combines trigram, bigram, and unigram models by linear interpolation.
- It defines the probability estimate as

$$\widehat{P}(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i)$$

where $\lambda 3 + \lambda 2 + \lambda 1 = 1$.

- The parameter values λi can be fixed, or they can be trained with an expectation–maximization algorithm.

Information Retrieval

Information retrieval is the task of finding documents that are relevant to a user's need for information.

The best-known examples of information retrieval systems are search engines on the World Wide Web.

An information retrieval (henceforth IR) system can be characterized by

1. **A corpus of documents:** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. **Queries posed in a query language**: A query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AINEAR book] or [AI book site:www.aaai.org].
3. **A result set:** This is the subset of documents that the IR system judges to be relevant to the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.
4. **A presentation of the result set**: This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space, rendered as a two-dimensional display.

IR system evaluation

- Consider an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments.
- Traditionally, there have been two measures used in the scoring: **recall and precision.**
- We explain them with the help of an example.
- Imagine that an IR system has returned a result set for a single query, for which we know

which documents are and are not relevant, out of a corpus of 100 documents.

- The document counts in each category are given in the following table:

	In result set	Not in result set
Relevant	30	20
Not relevant	10	40

- **Precision** measures the proportion of documents in the result set that are actually relevant.
  In our example, the precision is 30 / (30 + 10) = 0.75. The false positive rate is 1 − 0.75 = 0.25.
- **Recall** measures the proportion of all the relevant documents in the collection that are in the result set.
  In our example, recall is 30 / (30 + 20) = 0.60. The false negative rate is 1 − 0.60 = 0.40.

ROBOTICS

Introduction

- Robots are physical agents that perform tasks by manipulating the physical world.
- To do so, they are equipped with effectors such as legs, wheels, joints, and grippers.
- Effectors have a single purpose: to assert physical forces on the environment.
- Robots are also equipped with sensors, which allow them to perceive their environment.
- Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.

Robotics Hardware

Sensors

- Sensors are the perceptual interface between robot and environment.
- Passive sensors, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
- Active sensors, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor.
- Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time.
- Whether active or passive, sensors can be divided into **three types,** depending on whether they sense the environment, the robot's location, or the robot's internal configuration.
  - **Range finders** are sensors that measure the distance to nearby objects.
    - In the early days of robotics, robots were commonly equipped with **sonar sensors.**
    - **Sonar sensors** emit directional sound waves, which are reflected by objects, with some of the sound making it back into the sensor.
    - **Stereo vision** relies on multiple cameras to image the environment from slightly different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects.

- For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.

➢ **Tactile sensors** measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.

- Most **location sensors** use range sensing as a primary component to determine location.
- Outdoors, the **Global Positioning System (GPS)** is the most common solution to the localization problem.
- GPS measures the distance to satellites that emit pulsed signals.

➢ **Proprioceptive sensors,** which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with shaft decoders that count the revolution of motors in small increments.

Effectors

- Effectors are the means by which robots move and change the shape of their bodies.
- To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a degree of freedom (DOF).
- We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move.

For example,

- a rigid mobile robot such as an AUV has six degrees of freedom, three for its (x, y, z) location in space and three for its angular orientation, known as yaw, roll, and pitch.
- These six degrees define the kinematic state or pose of the robot.
- The dynamic state of a robot includes these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.