Reinforcement learning

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.

It can be used to learn tasks such as

- learning to control a mobile robot
- learning to optimize operations in factories
- learning to play board games etc.

Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state.

Eg: when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states.

The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward.

Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

INTRODUCTION

Consider building a learning robot. The robot, or agent, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.

Eg: a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control strategy, or policy, for choosing actions that achieve its goals.

For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

How such agents can learn successful control policies by experimenting in their environment?

Assumption:

The goals of the agent can be defined by a reward function that assigns a numerical value-an immediate payoff-to each distinct action the agent may take from each distinct state.

INTRODUCTION

Eg: The goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.

The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy.

The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

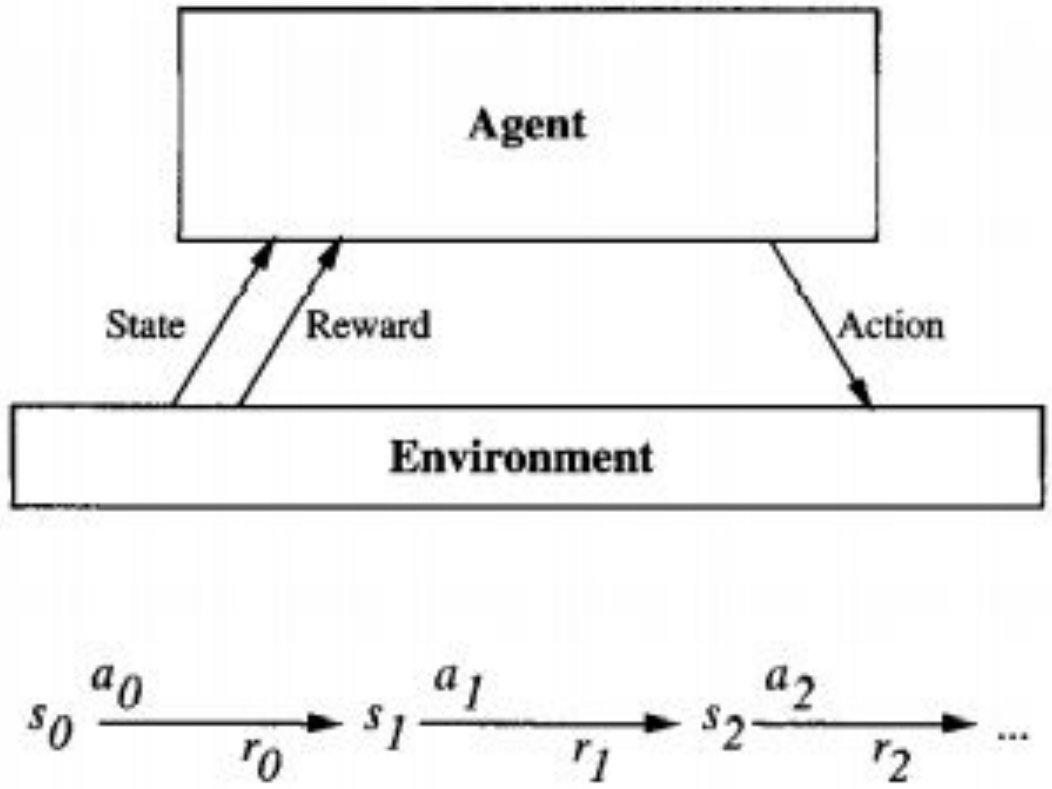


FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

INTRODUCTION

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems .

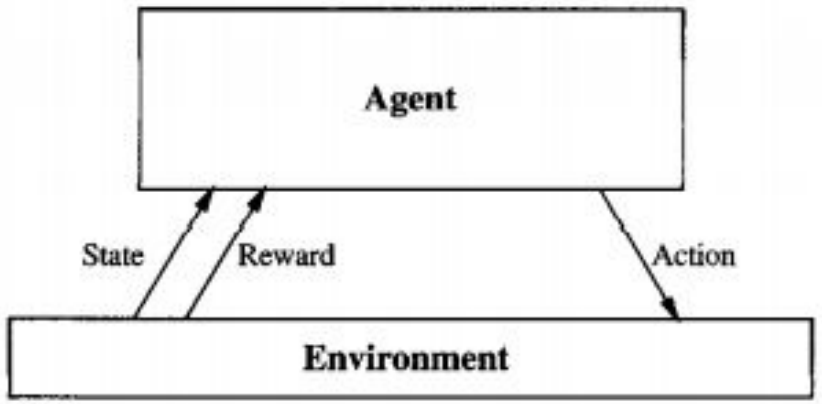
The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$,

that outputs an appropriate action a from the set A, given the current state s from the set S. However, this reinforcement learning problem differs from other function approximation tasks in several important respects.

- Delayed reward
- Exploration
- Partially observable states
- Life-long learning

THE LEARNING TASK

We formulate the problem of learning sequential control strategies.



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \cdots$$

Goal: Learn to choose actions that maximize

$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$, where $0 \leq \gamma < 1$

FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

Markov Decision Processes

Assume

- finite set of states S
- set of actions A
- at each discrete time agent observes state $s_t \in S$ and chooses action $a_t \in A$
- then receives immediate reward r_t
- and state changes to s_{t+1}
- Markov assumption: $s_{t+1} = \delta(s_t, a_t)$ and $r_t = r(s_t, a_t)$
  - i.e., r_t and s_{t+1} depend only on *current* state and action
  - functions δ and r may be nondeterministic
  - functions δ and r not necessarily known to agent

Agent's Learning Task

Execute actions in environment, observe results, and

- learn action policy $\pi : S \rightarrow A$ that maximizes

$$E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$$

  from any starting state in S

- here $0 \leq \gamma < 1$ is the discount factor for future rewards

Note something new:

- Target function is $\pi : S \rightarrow A$
- but we have no training examples of form $\langle s, a \rangle$
- training examples are of form $\langle \langle s, a \rangle, r \rangle$

Value Function

To begin, consider deterministic worlds...

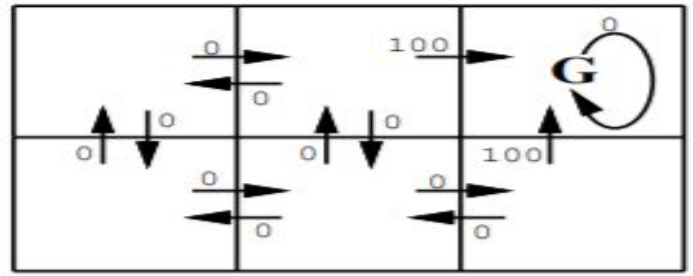
For each possible policy π the agent might adopt, we can define an evaluation function over states

$$\begin{aligned} V^{\pi}(s) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

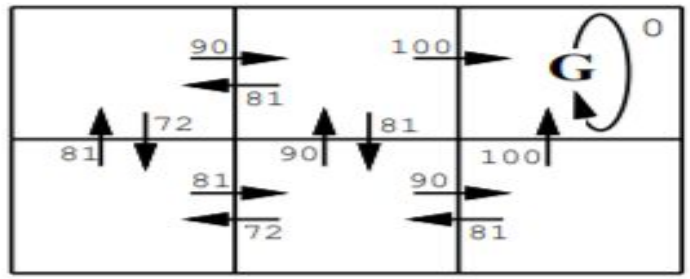
where $r_t, r_{t+1}, \ldots$ are generated by following policy π starting at state s

Restated, the task is to learn the optimal policy π^*

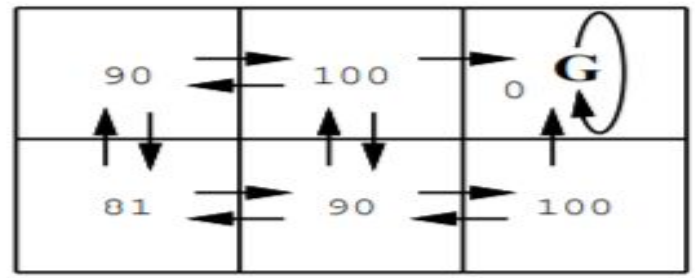
$$\pi^* \equiv \operatorname*{argmax}_{\pi} V^{\pi}(s), (\forall s)$$



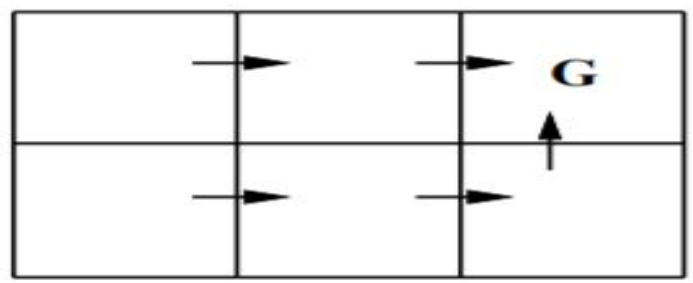
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

Q LEARNING

How can an agent learn an optimal policy for an arbitrary environment?

- It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does not provide training examples of the form (s, a).
- Instead, the only training information available to the learner is the sequence of immediate rewards r(si, ai) for i = 0, 1,2, ...
- Given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

Q LEARNING

We might try to have agent learn the evaluation function V^{π^*} (which we write as V^*)

It could then do a lookahead search to choose best action from any state s because

$$\pi^*(s) = \operatorname{argmax}_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

A problem:

- This works well if agent knows $\delta : S \times A \rightarrow S$, and $r : S \times A \rightarrow \Re$
- But when it doesn't, it can't choose actions this way

Q Function

Define new function very similar to V^*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

If agent learns Q, it can choose optimal action even without knowing δ!

$$\pi^*(s) = \operatorname{argmax}_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \operatorname{argmax}_a Q(s,a)$$

Q is the evaluation function the agent will learn

Training Rule to Learn Q

Note Q and V^* closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write Q recursively as

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Nice! Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s

Q-learning Algorithm

For each s, a initialize table entry $\hat{Q}(s, a) \leftarrow 0$

Observe current state s

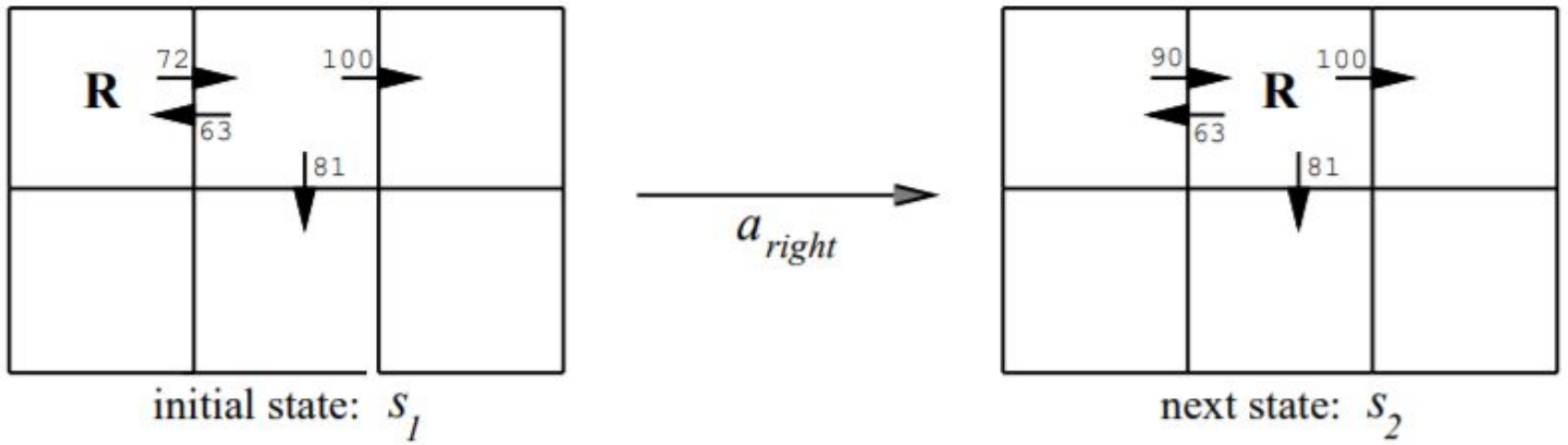
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

Updating $\hat{Q}$



$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

Properties of Q learning

The two general properties of this Q learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all $\hat{Q}$ values to zero.

1. the $\hat{Q}$ values never decrease during training.

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

2. Throughout the training process every $\hat{Q}$ value will remain in the interval between zero and its true Q value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

INSTANCE BASED LEARNING

Introduction

- Instance-based learning methods simply store the training examples.
- Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance.
- Instance-based methods are sometimes referred to as "lazy" learning methods because they delay processing until a new instance must be classified.
- A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

Introduction

- Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified.
- Many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space.
- This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.
- Instance-based methods can also use more complex, symbolic representations for instances.

Introduction

Disadvantages of instance-based approaches

- The cost of classifying new instances can be high.
  -This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered.
  -Techniques for efficiently indexing training examples are required.
- They typically consider all attributes of the instances when attempting to retrieve similar training examples from memory.

Introduction

- Instance based learning includes
  - nearest neighbor
  - locally weighted regression methods
  - case-based reasoning methods

k-NEAREST NEIGHBOR LEARNING

- K-nearest neighbors (KNN) algorithm is a type of supervised ML algorithm which can be used for both classification as well as regression predictive problems.

The following two properties would define KNN well –

- **Lazy learning algorithm** – KNN is a lazy learning algorithm because it does not have a specialized training phase and uses all the data for training while classification.
- **Non-parametric learning algorithm** – KNN is also a non-parametric learning algorithm because it doesn't assume anything about the underlying data.

K-nearest neighbors (KNN) algorithm uses 'feature similarity' to predict the values of new datapoints which further means that the new data point will be assigned a value based on how closely it matches the points in the training set.

k-NEAREST NEIGHBOR LEARNING

We can understand its working with the help of following steps −

Step 1 − For implementing any algorithm, we need dataset. So during the first step of KNN, we must load the training as well as test data.

Step 2 − Next, we need to choose the value of K i.e. the nearest data points. K can be any integer.

Step 3 − For each point in the test data do the following −

3.1 − Calculate the distance between test data and each row of training data with the help of any of the method namely: Euclidean, Manhattan or Hamming distance. The most commonly used method to calculate distance is Euclidean.

3.2 − Now, based on the distance value, sort them in ascending order.

3.3 − Next, it will choose the top K rows from the sorted array.

3.4 − Now, it will assign a class to the test point based on most frequent class of these rows.

Step 4 − End

k-NEAREST NEIGHBOR LEARNING

- This algorithm assumes all instances correspond to points in the n-dimensional space.
- The nearest neighbors of an instance are defined in terms of the standard Euclidean distance.
- Let an arbitrary instance x be described by the feature vector

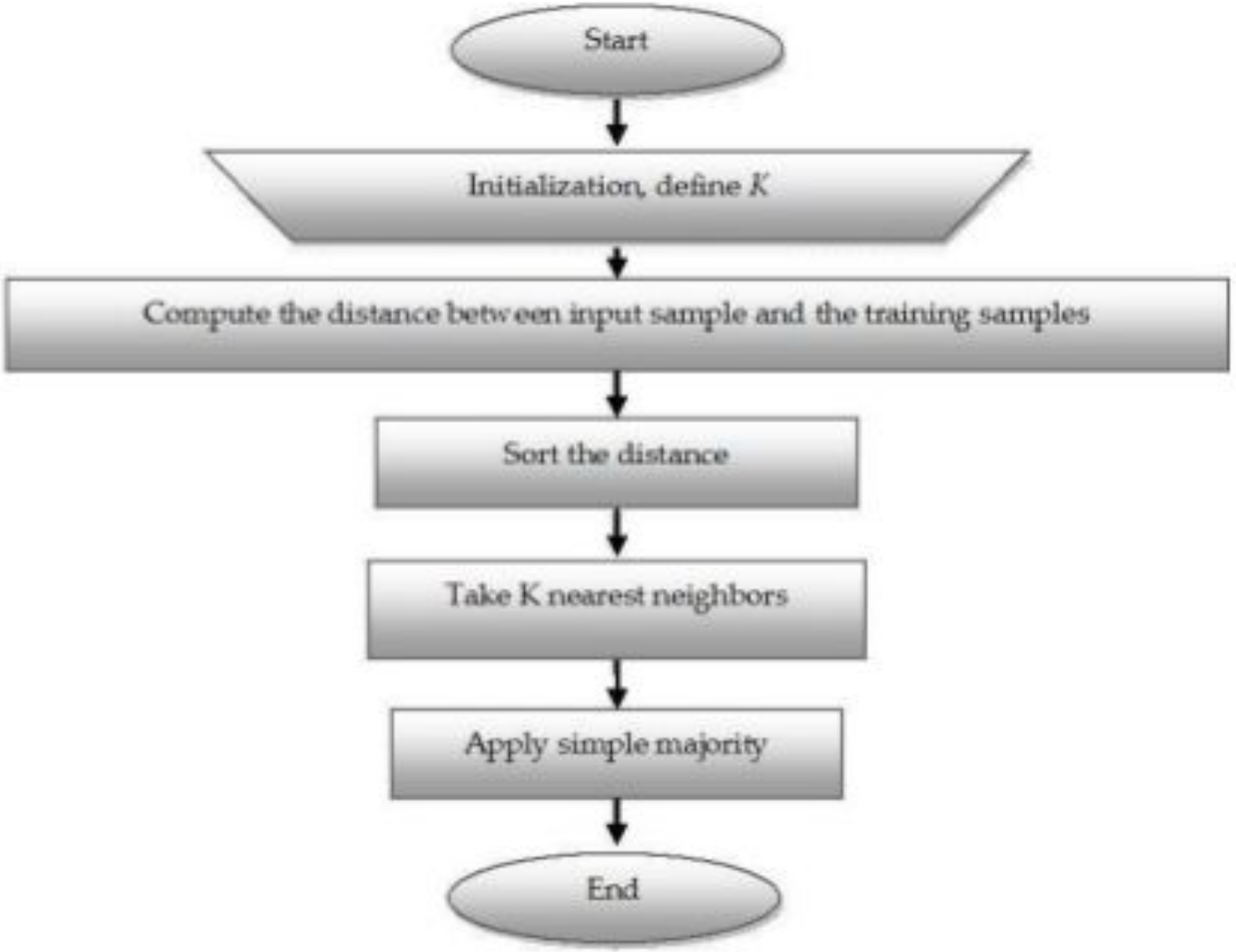
$$< a_1(x), a_2(x), \ldots a_n(x) >$$

where $a_r(x)$ denotes the value of the r^{th} attribute of instance x.

Then the distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

KNN Classifier Algorithm



KNN Classifier Algorithm Example

Name	Age	Gender	Sport
Ajay	32	M	Football
Mark	40	M	Neither
Sara	16	F	cricket
Zaira	34	F	cricket
Sachin	55	M	Neither
Rahul	40	M	cricket
Pooja	20	F	Neither
Smith	15	M	cricket
Laxmi	55	F	Football
Michael	15	M	Football

Classify the new instance Angelina : Age : 5 Gender: F

K=3

KNN Classifier Algorithm Example

Calculate the distance between test data and each row of training data

Name	Age	Gender	Sport
Ajay	32	M =0	Football
Mark	40	M =0	Neither
Sara	16	F =1	cricket
Zaira	34	F = 1	cricket
Sachin	55	M = 0	Neither
Rahul	40	M = 0	cricket
Pooja	20	F = 1	Neither
Smith	15	M = 0	cricket
Laxmi	55	F = 1	Football
Michael	15	M = 0	Football

KNN Classifier Algorithm Example

Euclidian Distance

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

Euclidian Distance between 1st training example and new instance is

Name	Age	Gender	Sport
Ajay	32	M =0	Football
Angelina	5	Gender: F = 1	

Euclidian Distance = sqrt((5-32)^2 + (1-0)^2) = 27.0185

KNN Classifier Algorithm Example

Euclidian Distance between 2^{nd} training example and new instance is

Euclidian Distance = $\text{sqrt}((5-40)^2 + (1-0)^2) = 35.014$

KNN Classifier Algorithm Example

Name	Age	Gender	Euclidian Distance
Ajay	32	M =0	27.0185
Mark	40	M =0	35.014
Sara	16	F =1	11
Zaira	34	F = 1	29
Sachin	55	M = 0	50.01
Rahul	40	M = 0	35
Pooja	20	F = 1	15
Smith	15	M = 0	10
Laxmi	55	F = 1	50
Michael	15	M = 0	10.05
Angelina	5	F = 1	cricket

New instance: Height=159 & weight=68 , k=3

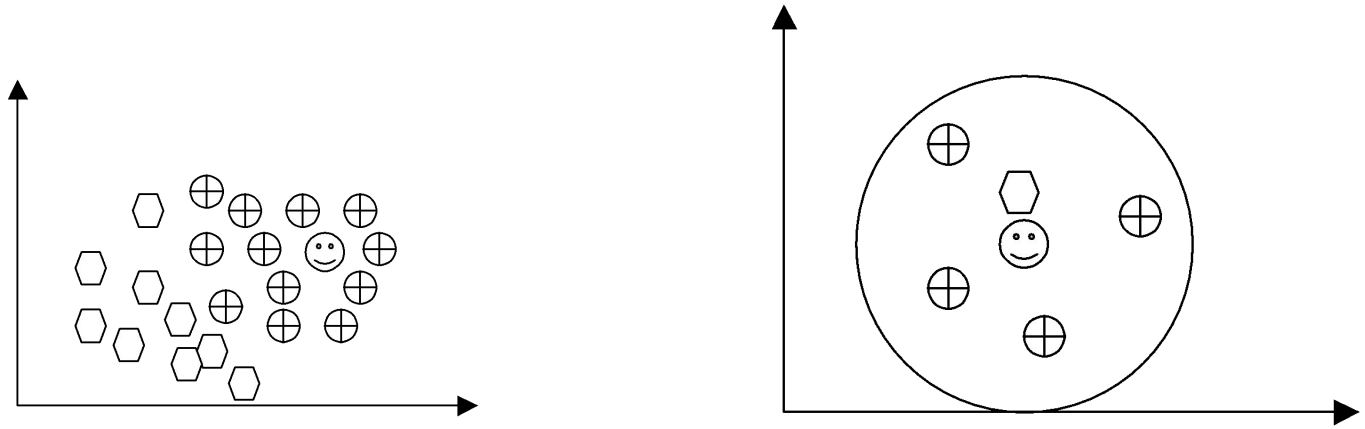
Height	Weight	T-shirt Size
158	63	M
160	59	M
160	60	M
163	60	M
163	51	M
160	64	L
163	64	L

Name	Acid Durability	Strength	Class
Type-1	7	7	Bad
Type-2	7	4	Bad
Type-3	3	4	Good
Type-4	1	4	Good

Test-Data → acid durability=3, and strength=7, class=?

Example

Simple 2-D case, each instance described only by two values (x, y co-ordinates). The class is either ⊕ or ⬡



Need to consider :

1) Similarity (how to calculate distance)
2) Number (and weight) of similar (near) instances

k-NEAREST NEIGHBOR LEARNING

- Consider, the case of learning a discrete-valued target function of the form $f : \Re^n \rightarrow V$, where V is the finite set $\{v_1, \ldots v_s\}$

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

---

k-NEAREST NEIGHBOR LEARNING

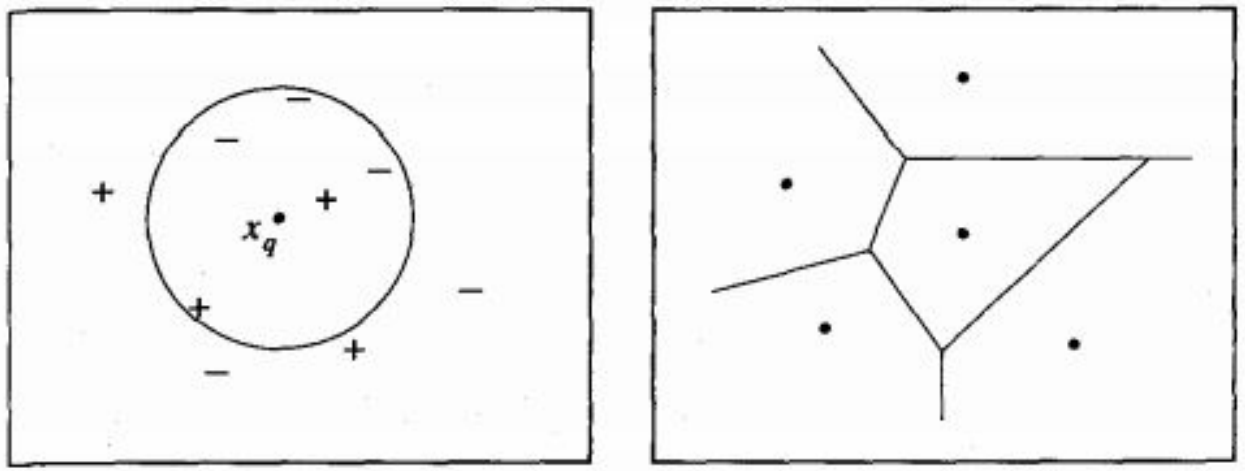


FIGURE 8.1
k-NEAREST NEIGHBOR. A set of positive and negative training examples is shown on the left, along with a query instance x_q to be classified. The 1-NEAREST NEIGHBOR algorithm classifies x_q positive, whereas 5-NEAREST NEIGHBOR classifies it as negative. On the right is the decision surface induced by the 1-NEAREST NEIGHBOR algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance space closest to that point (i.e., the instances for which the 1-NEAREST NEIGHBOR algorithm will assign the classification belonging to that training example).

k-NEAREST NEIGHBOR LEARNING

- The k-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions.
- To accomplish this, we have the algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value.
- More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$

  replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

k-NEAREST NEIGHBOR LEARNING

Distance-Weighted NEAREST NEIGHBOR Algorithm:

One refinement to the k-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors.

This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

where $$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

k-NEAREST NEIGHBOR LEARNING

Distance-Weighted NEAREST NEIGHBOR Algorithm:

- We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

Remarks on k-NEAREST NEIGHBOR Algorithm

- The distance-weighted k-NEAREST NEIGHBOR algorithm is a highly effective inductive inference method for many practical problems.
- It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data.
- By taking the weighted average of the k neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples.

Inductive bias of k-NEAREST NEIGHBOUR:

- The classification of an instance x_q, will be most similar to the classification of other instances that are nearby in Euclidean distance.

Remarks on k-NEAREST NEIGHBOR Algorithm

Issues with KNN:

The distance between instances is calculated based on all attributes of the instance.

The distance between neighbors will be dominated by the large number of irrelevant attributes. This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the **curse of dimensionality.**

Solution:

- Weight each attribute differently when calculating the distance between two instances.
- The true error can be estimated using cross-validation.
  - select a random subset of the available data to use as training examples, then determine the values of zl . . . z, that lead to the minimum error in classifying the remaining examples. By repeating this process multiple times the estimate for these weighting factors can be made more accurate.
- Completely eliminate the least relevant attributes from the instance space

Remarks on k-NEAREST NEIGHBOR Algorithm

- One additional practical issue in applying k-NEAREST NEIGHBOR is efficient memory indexing. Because this algorithm delays all processing until a new query is received, significant computation can be required to process each new query. Various methods have been developed for indexing the stored training examples so that the nearest neighbors can be identified more efficiently at some additional cost in memory.

LOCALLY WEIGHTED REGRESSION

- Locally weighted regression constructs an explicit approximation to f over a local region surrounding x_q.
- Locally weighted regression uses nearby or distance-weighted training examples to form this local approximation to f.
- The phrase "locally weighted regression" is called

  local - the function is approximated based a only on data near the query point

  weighted - the contribution of each training example is weighted by its distance from the query point

  regression - approximating real-valued functions.

LOCALLY WEIGHTED REGRESSION

- Consider the case of locally weighted regression in which the target function f is approximated near x_q, using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

- Redefine the error criterion $E(x_q)$ as below:

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

LOCALLY WEIGHTED REGRESSION

Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in \text{ k nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

If we choose this criterion and rederive the gradient descent rule, we obtain the following training rule

$$\Delta w_j = \eta \sum_{x \in \text{ k nearest nbrs of } x_q} K(d(x_q, x)) \; (f(x) - \hat{f}(x)) \; a_j(x)$$

More complex functional forms are not often found

(1) the cost of fitting more complex functions for each query instance is prohibitively high

(2) these simple approximations model the target function quite well over a sufficiently small subregion of the instance space.

RADIAL BASIS FUNCTIONS

In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$$

where

x_u is an instance from X

$K_u((d(x_u, x))$ is the kernel function defined so that it decreases as the distance $d(x_u, x)$ increases.

k is the number of kernel functions to be included

Kernel function is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function K such that $w_i = K(d(x_i, x_q))$.

RADIAL BASIS FUNCTIONS

Usually the function $K_u((d(x_u, x))$ is a Gaussian function centered at the point x_u with some variance σ_u^2.

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

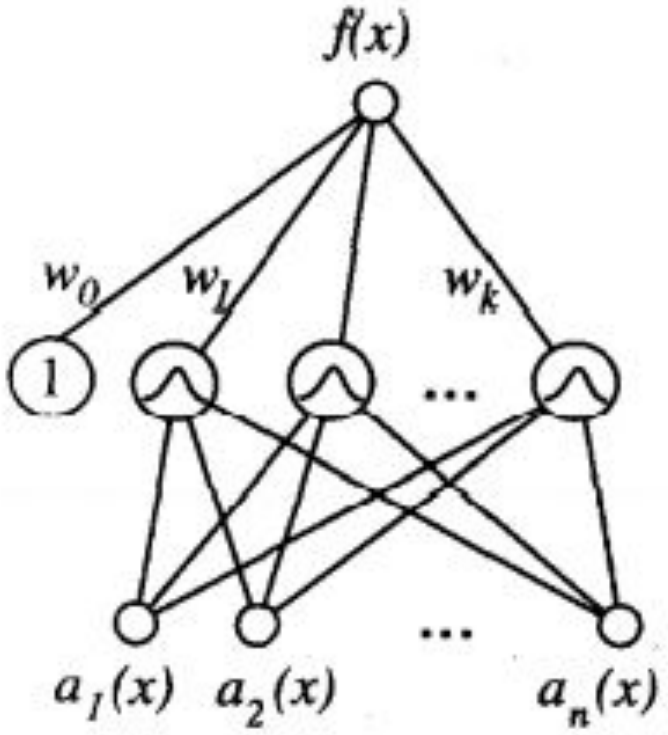


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

RADIAL BASIS FUNCTIONS

Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions.

1. Allocate a Gaussian kernel function for each training example (x_i, f (x_i)), centering this Gaussian at the point x_i. Each of these kernels may be assigned the same width.

   -One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly

2. Choose a set of kernel functions that is smaller than the number of training examples.

RADIAL BASIS FUNCTIONS

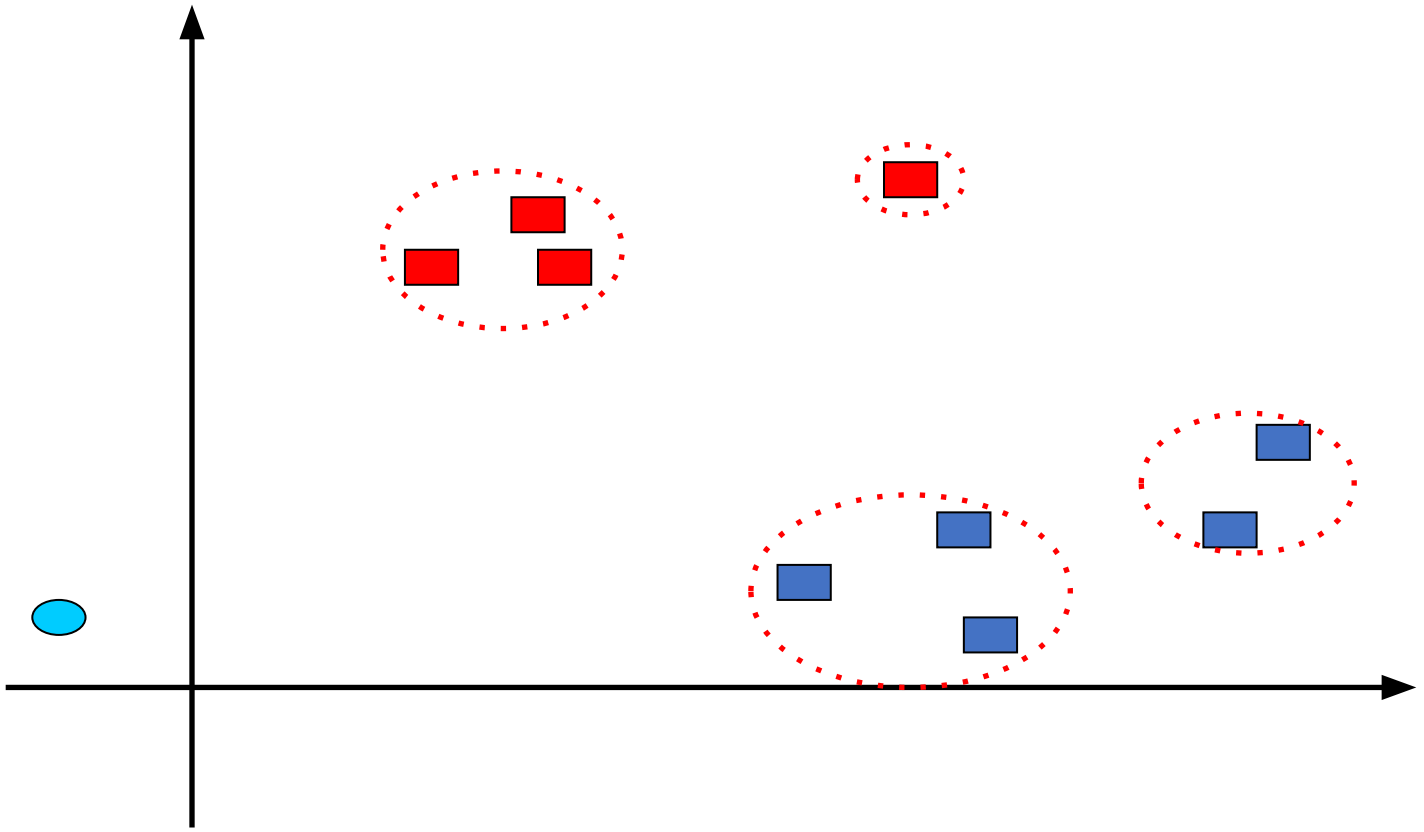
The set of kernel functions may be distributed with centers spaced uniformly throughout the instance space X.

Alternatively, we may wish to distribute the centers nonuniformly, especially if the instances themselves are found to be distributed nonuniformly over X.

-Pick kernel function centers by randomly selecting a subset of the training instances, thereby sampling the underlying distribution of instances.

-Identify prototypical clusters of instances, then add a kernel function centered at each cluster. The placement of the kernel functions in this fashion can be accomplished using unsupervised clustering algorithms that fit the training instances to a mixture of Gaussians.

RADIAL BASIS FUNCTIONS



Summary of RADIAL BASIS FUNCTIONS

Radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions.

The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular center and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function.

Advantage:

They can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION.

The input layer and the output layer of an RBF are trained separately.

CASE-BASED REASONING

- Case-Based reasoning (CBR), broadly construed, is the process of solving new problems based on the solutions of similar past problems.

- CBR is reasoning by remembering:
  It is a starting point for new reasoning

- Case-Based Reasoning is a well established research field that involves the investigation of theoretical foundations, system development and practical application building of experience-based problem solving.

CASE-BASED REASONING

Everyday examples of CBR :

- An auto mechanic who fixes an engine by recalling another car that exhibited similar symptoms

- A lawyer who advocates a particular outcome in a trial based on legal precedents or a judge who creates case law.

- An engineer copying working elements of nature (practicing biomimicry), is treating nature as a database of solutions to problems.

- Case-based reasoning is a prominent kind of analogy making.

CASE-BASED REASONING

Case – previously made and stored experience item

Case-Base – core of every case – based problem solver

- **collection of cases**

One of the core assumptions behind CBR is that
similar problems have similar solutions.

CASE-BASED REASONING

- A case-based problem solver solves new problems primarily by reuse of solutions from the cases in the case-base.
- For this purpose, one or several relevant cases are selected.
- Once similar cases are selected, the solution(s) from the case(s) are adapted to become a solution of the current problem.
- When a new (successful) solution to the new problem is found, *a new experience is made*, which can be stored in the case-base to increase its competence, thus implementing a learning behavior.

CASE-BASED REASONING

There are three main types of CBR that differ significantly from one another concerning case representation and reasoning:

1. Structural : a common structured vocabulary, i.e. an ontology
2. Textual : cases are represented as free text, i.e. strings
3. Conversational : a case is represented through a list of questions that varies from one case to another ; knowledge is contained in customer / agent conversations.

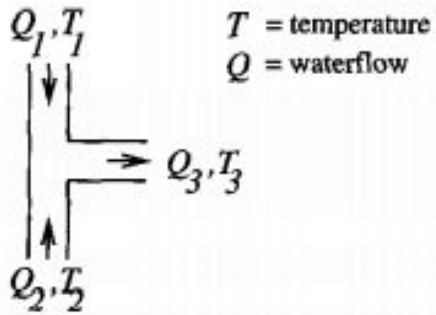
Example of CASE-BASED REASONING

- The CADET system employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.
- It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems.
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

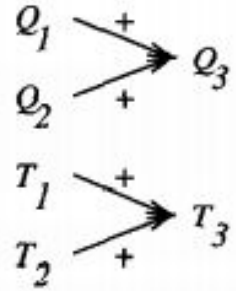
Example of CASE-BASED REASONING

A stored case: T–junction pipe

Structure:



Function:

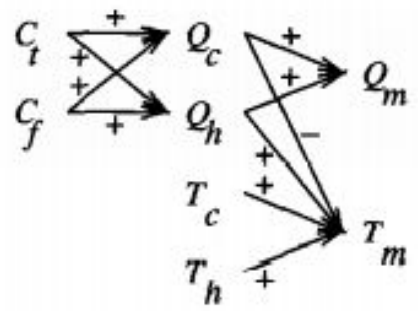


A problem specification: Water faucet

Structure:

?

Function:



Example of CASE-BASED REASONING

- Given this functional specification for the new design problem, CADET searches its library for stored cases whose functional descriptions match the design problem.
- If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem.
- If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification.

Example of CASE-BASED REASONING

- The system may elaborate the original function specification graph in order to create functionally equivalent graphs that may match still more cases.
- It uses general knowledge about physical influences to create these elaborated function graphs. For example, it uses a rewrite rule that allows it to rewrite the influence

$$A \xrightarrow{+} B$$

as

$$A \xrightarrow{+} x \xrightarrow{+} B$$

Example of CASE-BASED REASONING

By retrieving multiple cases that match different subgraphs, the entire design can sometimes be pieced together.

In general, the process of producing a final solution from multiple retrieved cases can be very complex.

It may require designing portions of the system from first principles, in addition to merging retrieved portions from stored cases.

It may also require backtracking on earlier choices of design subgoals and, therefore, rejecting cases that were previously retrieved.

CADET has very limited capabilities for combining and adapting multiple retrieved cases to form the final design and relies heavily on the user for this adaptation stage of the process.

Example of CASE-BASED REASONING

- In CADET each stored training example describes a function graph along with the structure that implements it. New queries correspond to new function graphs.
- Thus, we can map the CADET problem into our standard notation by defining the space of instances X to be the space of all function graphs.
- The target function f maps function graphs to the structures that implement them.
- Each stored training example (x, f (x)) is a pair that describes some function graph x and the structure f (x) that implements x.
- The system must learn from the training example cases to output the structure f (x_q) that successfully implements the input function graph query x_q.

Generic properties of case-based reasoning systems

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET. This may require a similarity metric different from Euclidean distance, such as the size of the largest shared subgraph between two function graphs.
- Multiple retrieved cases may be combined to form the solution to the new problem. This is similar to the k-NEAREST NEIGHBOR approach, in that multiple similar cases are used to construct a response for the new query. However, the process for combining these multiple retrieved cases can be very different, relying on knowledge-based reasoning rather than statistical methods.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving.

One simple example of this is found in CADET, which uses generic knowledge about influences to rewrite function graphs during its attempt to find matching cases. Other systems have been developed that more fully integrate case-based reasoning into general search based problem-solving systems.

Eg: ANAPRON (Golding and Rosenbloom 1991) and PRODIGY/ANALOGY (Veloso 1992).