MODULE 5

INSTANCE BASED LEARNING

INTRODUCTION

- Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.
- Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the new query instance
- Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified

Advantages of Instance-based learning

1. Training is very fast
2. Learn complex target function
3. Don’t lose information

Disadvantages of Instance-based learning

- 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.
- In many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

k- NEAREST NEIGHBOR LEARNING

- The most basic instance-based method is the K- Nearest Neighbor Learning. This algorithm assumes all instances correspond to points in the n-dimensional space R^n.
- 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

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

  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,

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

- In nearest-neighbor learning the target function may be either discrete-valued or real-valued.

Let us first consider learning ***discrete-valued target functions*** of the form

$$f : \Re^n \rightarrow V.$$

Where, V is the finite set $\{v_1, \ldots v_s\}$

The k- Nearest Neighbor algorithm for approximation a **discrete-valued target function** is given below:

Training algorithm:

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

Classification algorithm:

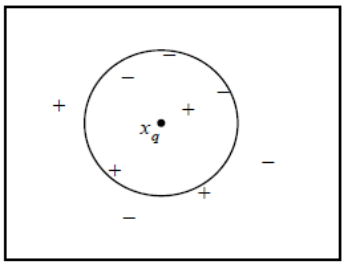
- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

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

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

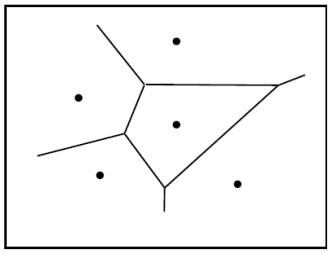
- The value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to x_q.
- If k = 1, then the 1- Nearest Neighbor algorithm assigns to $\hat{f}(x_q)$ the value f(xi). Where x_i is the training instance nearest to x_q.
- For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

- Below figure illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is Boolean valued.



- The positive and negative training examples are shown by "+" and "-" respectively. A query point x_q is shown as well.
- The 1-Nearest Neighbor algorithm classifies x_q as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.

- Below figure shows the shape of this **decision surface** induced by 1- Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples.



- For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the ***Voronoi diagram*** of the set of training example

The K- Nearest Neighbor algorithm for approximation a **real-valued target function** is given below $f : \Re^n \rightarrow \Re$

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

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

Distance-Weighted Nearest Neighbor Algorithm

- The 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.
- For example, in the k-Nearest Neighbor algorithm, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q

Distance-Weighted Nearest Neighbor Algorithm for approximation a discrete-valued target functions

---

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

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

where

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

Distance-Weighted Nearest Neighbor Algorithm for approximation a Real-valued target functions

---

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

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

where

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

Terminology

- **Regression** means approximating a real-valued target function.
- **Residual** is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- **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 $w_i = K(d(x_i, x_q))$

LOCALLY WEIGHTED REGRESSION

- The phrase "**locally weighted regression**" is called **local** because the function is approximated based only on data near the query point, **weighted** because the contribution of each training example is weighted by its distance from the query point, and **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

- Given a new query instance x_q, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding x_q. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance.

Locally Weighted Linear Regression

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

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

Where, $a_i(x)$ denotes the value of the i^{th} attribute of the instance x

- Derived methods are used to choose weights that minimize the squared error summed over the set D of training examples using gradient descent

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

Which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x)$$

Where, η is a constant learning rate

- Need to modify this procedure to derive a local approximation rather than a global one. The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below.

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

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

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 :

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

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \textbf{equ(3)}$$

If we choose criterion three and re-derive the gradient descent rule, we obtain the following training rule

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

The differences between this new rule and the rule given by Equation (3) are that the contribution of instance x to the weight update is now multiplied by the distance penalty **$K(d(x_q, x))$,** and that the error is summed over only the k nearest training examples.

RADIAL BASIS FUNCTIONS

- One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions
- In this approach, the learned hypothesis is a function of the form

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

- Where, each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance d(xu, x) increases.
- Here k is a user provided constant that specifies the number of kernel functions to be included.
- $\hat{f}$ is a global approximation to f (x), the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u.

Choose each function $K_u(d(x_u, x))$ to be a Gaussian function centred at the point x_u with some variance σ_u^2

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

- The functional form of equ(1) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width σ^2 of each kernel can be separately specified
- The function given by equ(1) can be viewed as describing a two layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values

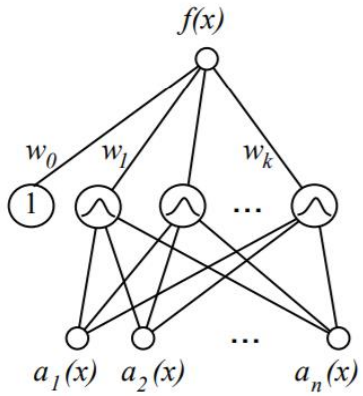
Example: Radial basis function (RBF) network

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.

1. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$
2. Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by

$$E \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2$$

Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently



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

- One approach is to allocate a Gaussian kernel function for each training example $(x_i, f(x_i))$, centring this Gaussian at the point x_i.
  Each of these kernels may be assigned the same width σ^2. Given this approach, the RBF network learns a global approximation to the target function in which each training example $(x_i, f(x_i))$ can influence the value of $\hat{f}$ only in the neighbourhood of x_i.
- A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large.

Summary

- 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 centre and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function.
- One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION.

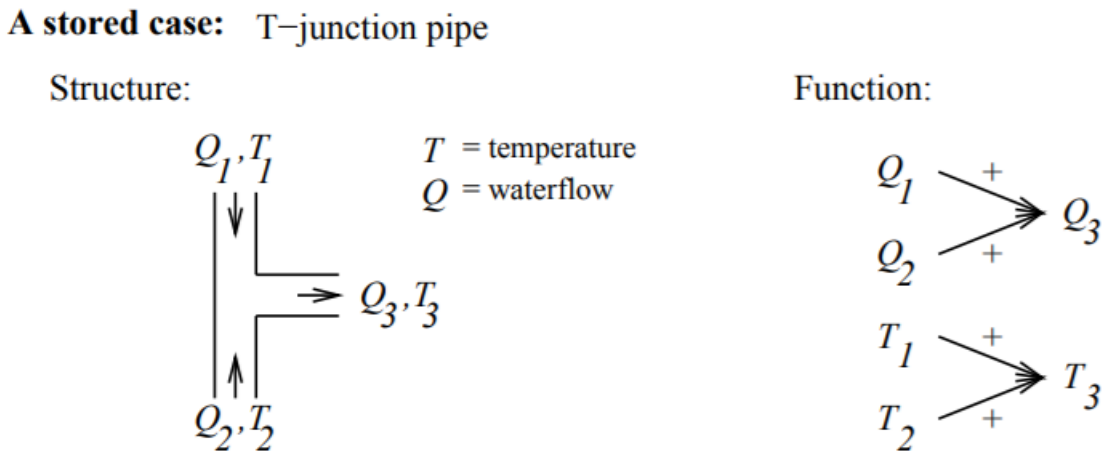
CASE-BASED REASONING

- Case-based reasoning (CBR) is a learning paradigm based on lazy learning methods and they classify new query instances by analysing similar instances while ignoring instances that are very different from the query.
- In CBR represent instances are not represented as real-valued points, but instead, they use a ***rich symbolic*** representation.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs, reasoning about new legal cases based on previous rulings, and solving planning and scheduling problems by reusing and combining portions of previous solutions to similar problems

A prototypical example of a 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.

The problem setting is illustrated in below figure



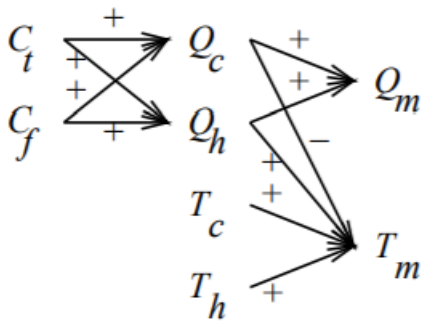
- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label indicates that the variable at the head decreases with the variable at the tail.
- Here Q_c refers to the flow of cold water into the faucet, Q_h to the input flow of hot water, and Q_m to the single mixed flow out of the faucet.
- T_c, T_h, and T_m refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable C_t denotes the control signal for temperature that is input to the faucet, and C_f denotes the control signal for waterflow.
- The controls C_t and C_f are to influence the water flows Q_c and Q_h, thereby indirectly influencing the faucet output flow Q_m and temperature T_m.

A problem specification: Water faucet

Structure:

?

Function:



- 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.

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.

INTRODUCTION

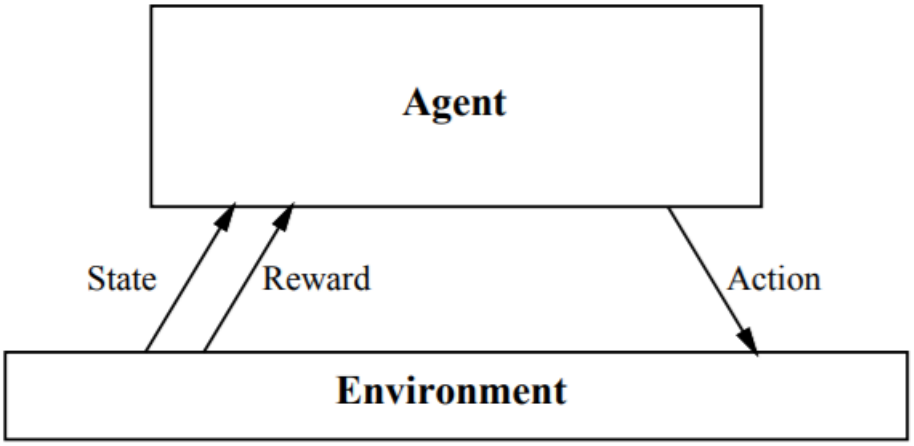
- Consider building a **learning robot**. The robot, or ***agent***, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.
- Its task is to learn a control strategy, or ***policy***, for choosing actions that achieve its goals.
- The goals of the agent can be defined by a ***reward function*** that assigns a numerical value to each distinct action the agent may take from each distinct state.
- 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 is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

Example:

- A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn."
- The robot may have a goal of docking onto its battery charger whenever its battery level is low.
- The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +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.

Reinforcement Learning Problem

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- Agent perform any of a set of possible actions A. Each time it performs an action a, in some state s_t the agent receives a real-valued reward r, 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, $\boldsymbol{\pi: S \rightarrow A}$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.



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

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \text{ , where } 0 \leq \gamma < 1$$

Reinforcement learning problem characteristics

1. **Delayed reward**: The task of the agent is to learn a target function π that maps from the current state s to the optimal action a = π (s). In reinforcement learning, training information is not available in (s, π (s)). Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of ***temporal credit assignment***: determining which of the actions in its sequence are to be credited with producing the eventual rewards.

2. **Exploration:** In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a trade-off in choosing whether to favor exploration of unknown states and actions, or exploitation of states and actions that it has already learned will yield high reward.

3. **Partially observable states:** The agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. In such cases, the agent needs to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

4. **Life-long learning:** Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

THE LEARNING TASK

- Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it.
- The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, $\boldsymbol{\pi: S \rightarrow A}$, for selecting its next action a, based on the current observed state s_t; that is, $\boldsymbol{\pi(s_t) = a_t}$.

How shall we specify precisely which policy π we would like the agent to learn?

1. One approach is to require the policy that produces the greatest possible ***cumulative reward*** for the robot over time.
   - To state this requirement more precisely, define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \textbf{equ (1)}$$

   - Where, the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions.
   - Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.
   - The quantity $V^{\pi}(s_t)$ is called the ***discounted cumulative reward*** achieved by policy π from initial state ***s***. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later.

2. Other definitions of total reward is ***finite horizon reward,***

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number ***h*** of steps

3. Another approach is ***average reward***

$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

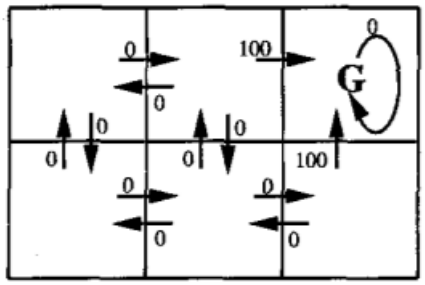
We require that the agent learn a policy π that maximizes $V^{\pi}(s_t)$ for all states s. such a policy is called an ***optimal policy*** and denote it by π^*

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \qquad \textbf{equ (2)}$$

Refer the value function $V^{\pi*}(s)$ an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state ***s***.

Example:

A simple grid-world environment is depicted in the diagram

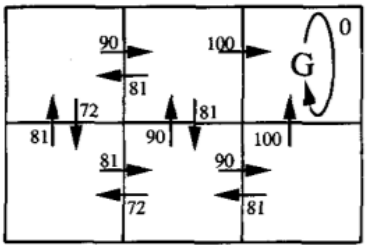


$r(s, a)$ (immediate reward) values

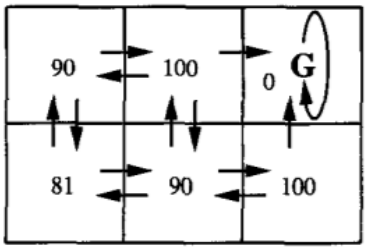
- The six grid squares in this diagram represent six possible states, or locations, for the agent.
- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
- The number associated with each arrow represents the immediate reward ***r(s, a)*** the agent receives if it executes the corresponding state-action transition
- The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labelled G. The state G as the goal state, and the agent can receive reward by entering this state.

Once the states, actions, and immediate rewards are defined, choose a value for the discount factor γ, determine the optimal policy π^* and its value function $V^*(s)$.

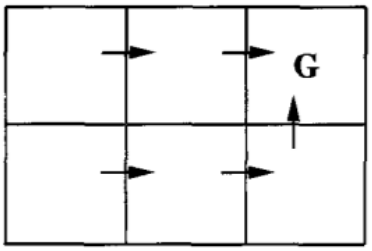
Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom centre state is

$$0+\gamma\ 100+\gamma^2\ 0+\gamma^3\ 0+\ldots = 90$$

Q LEARNING

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

The training information available to the learner is the sequence of immediate rewards $r(s_i,a_i)$ 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.

What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1

The optimal action in state ***s*** is the action ***a*** that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))] \qquad \textbf{equ (3)}$$

The *Q* Function

The value of Evaluation function ***Q(s, a)*** is the reward received immediately upon executing action a from state ***s***, plus the value (discounted by γ) of following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \qquad \textbf{equ (4)}$$

Rewrite Equation (3) in terms of ***Q(s, a)*** as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \qquad \textbf{equ (5)}$$

Equation (5) makes clear, it need only consider each available action ***a*** in its current state ***s*** and choose the action that maximizes ***Q(s, a)***.

An Algorithm for Learning *Q*

- Learning the Q function corresponds to learning the **optimal policy**.
- The key problem is finding a reliable way to estimate training values for ***Q***, given only a sequence of immediate rewards ***r*** spread out over time. This can be accomplished through ***iterative approximation***

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

Rewriting Equation

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

- **Q learning algorithm:**

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the 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:

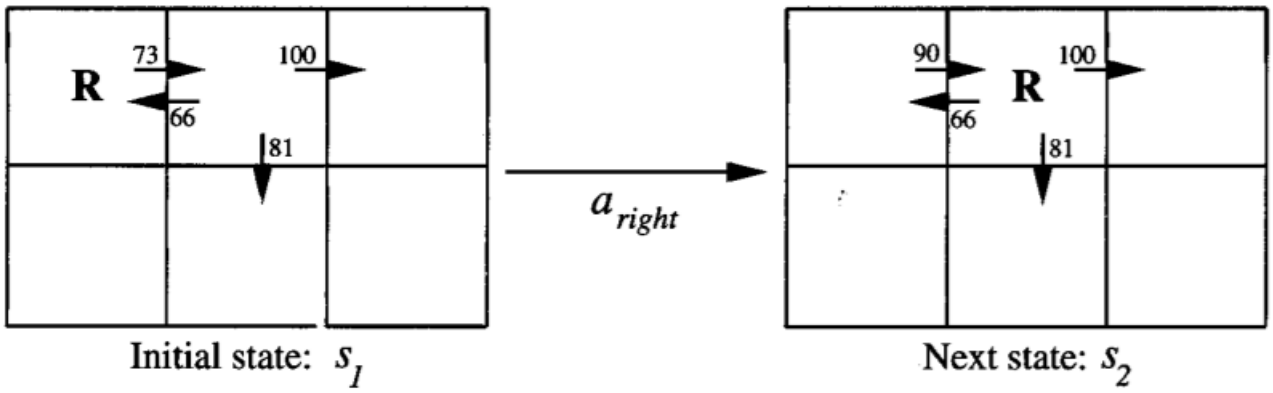
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

- Q learning algorithm assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$
- $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function

An Illustrative Example

- To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to $\hat{Q}$ shown in below figure



- The agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition.
- Apply the training rule of Equation

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

to refine its estimate Q for the state-action transition it just executed.

- According to the training rule, the new $\hat{Q}$ estimate for this transition is the sum of the received reward (zero) and the highest $\hat{Q}$ value associated with the resulting state (100), discounted by γ (.9).

$$\begin{aligned} \hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90 \end{aligned}$$

Convergence

Will the Q Learning Algorithm converge toward a Q equal to the true Q function?

Yes, under certain conditions.

1. Assume the system is a deterministic MDP.
2. Assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, $|\mathbf{r(s, a)}| < \mathbf{c}$
3. Assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

Theorem Convergence of Q learning for deterministic Markov decision processes.

Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$.

The Q learning agent uses the training rule of Equation $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$ initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a.

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \end{aligned}$$

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $Q_{n+1}(s, a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all s, a. Now after the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma \Delta_0$. After k such intervals, the error will be at most $\gamma^k \Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \rightarrow 0$ as $n \rightarrow \infty$. This proves the theorem.

Experimentation Strategies

The Q learning algorithm does not specify how actions are chosen by the agent.

- One obvious strategy would be for the agent in state **s** to select the action ***a*** that maximizes $\hat{Q}$(s, a), thereby exploiting its current approximation $\hat{Q}$.
- However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to explore other actions that have even higher values.
- For this reason, Q learning uses a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability.
- One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Where, $\mathbf{P(a_i|s)}$ is the probability of selecting action $\mathbf{a_i}$, given that the agent is in state **s**, and $\mathbf{k > 0}$ is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ values

MODULE 5

EVALUATING HYPOTHESES

MOTIVATION

It is important to evaluate the performance of learned hypotheses as precisely as possible.

- One reason is simply to understand whether to use the hypothesis.
- A second reason is that evaluating hypotheses is an integral component of many learning methods.

Two key difficulties arise while learning a hypothesis and estimating its future accuracy given only a limited set of data:

1. **Bias in the estimate**. The observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.

2. **Variance in the estimate.** Even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

ESTIMATING HYPOTHESIS ACCURACY

Sample Error –
The sample error of a hypothesis with respect to some sample S of instances drawn from X is the fraction of S that it misclassifies.

Definition: The sample error (***$error_S(h)$***) of hypothesis h with respect to target function f and data sample S is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

True Error –
The true error of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution $\mathcal{D}$.

Definition: The true error ($error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Confidence Intervals for Discrete-Valued Hypotheses

Suppose we wish to estimate the true error for some discrete valued hypothesis h, based on its observed sample error over a sample S, where

- The sample S contains n examples drawn independent of one another, and independent of h, according to the probability distribution $\mathcal{D}$
- $n \geq 30$
- Hypothesis h commits r errors over these n examples (i.e., $error_s(h) = r/n$).

Under these conditions, statistical theory allows to make the following assertions:

1. Given no other information, the most probable value of $error_{\mathcal{D}}(h)$ is $error_s(h)$
2. With approximately **95% probability**, the true error $error_{\mathcal{D}}(h)$ lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

Example:

Suppose the data sample S contains n = 40 examples and that hypothesis h commits r = 12 errors over this data.

- The ***sample error*** is $error_s(h) = r/n = 12/40 = 0.30$
- Given no other information, ***true error*** is $error_{\mathcal{D}}(h) = error_s(h)$, i.e., $error_{\mathcal{D}}(h) = 0.30$
- With the 95% confidence interval estimate for $error_{\mathcal{D}}(h)$.

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

$= 0.30 \pm (1.96 * 0.07)$ $= 0.30 \pm 0.14$

3. A different constant, ***ZN***, is used to calculate the **N% confidence interval**. The general expression for approximate N% confidence intervals for $error_{\mathcal{D}}$ (h) is

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

Where,

$N\%$:	50%	68%	80%	90%	95%	98%	99%
z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

The above equation describes how to calculate the confidence intervals, or error bars, for estimates of $error_{\mathcal{D}}$ (h) that are based on $error_s$(h)

Example:

Suppose the data sample S contains n = 40 examples and that hypothesis h commits r = 12 errors over this data.

- The ***sample error*** is $error_s$(h) = r/n = 12/40 = 0.30
- With the 68% confidence interval estimate for $error_{\mathcal{D}}$ (h).

$$error_S(h) \pm 1.00 \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

= 0.30 ± (1.00 * 0.07)

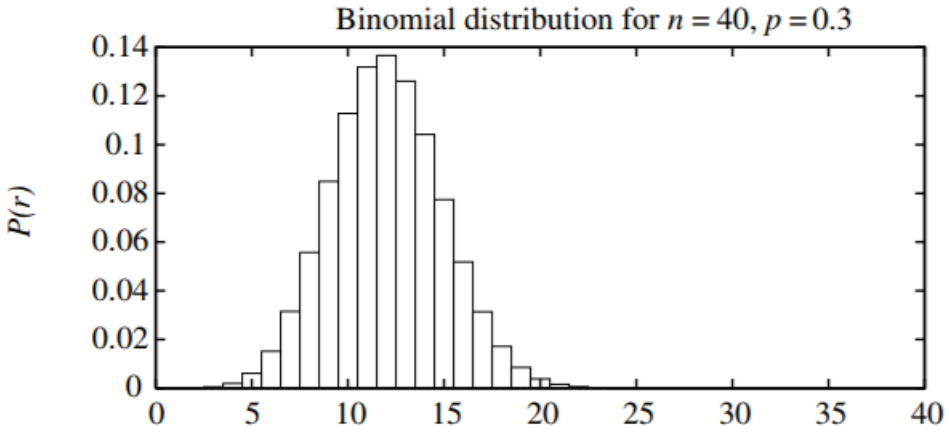
= 0.30 ± 0.07

BASICS OF SAMPLING THEORY

Error Estimation and Estimating Binomial Proportions

- Collect a random sample S of *n* independently drawn instances from the distribution *D*, and then measure the sample error $error_s$(h). Repeat this experiment many times, each time drawing a different random sample S_i of size *n*, we would expect to observe different values for the various $error_{si}$(h), depending on random differences in the makeup of the various S_i. We say that $error_{si}$(h), the outcome of the i^{th} such experiment, is a ***random variable***.

- Imagine that we were to run k random experiments, measuring the random variables $error_{s1}(h)$, $error_{s2}(h)$ . . . $errors_{sk}(h)$ and plotted a histogram displaying the frequency with which each possible error value is observed.
- As ***k*** grows, the histogram would approach a particular probability distribution called the ***Binomial distribution*** which is shown in below figure.



A Binomial distribution is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable ***X*** follows a Binomial distribution, then:

- The probability ***Pr(X = r)*** that ***X*** will take on the value ***r*** is given by ***P(r)***

- Expected, or mean value of X, $E[X]$, is

$$E[X] \equiv \sum_{i=0}^{n} iP(i) = np$$

- Variance of X is

$$Var(X) \equiv E[(X - E[X])^2] = np(1-p)$$

- Standard deviation of X, σ_X, is

$$\sigma_X \equiv \sqrt{E[(X - E[X])^2]} = \sqrt{np(1-p)}$$

The Binomial Distribution

Consider the following problem for better understanding of Binomial Distribution

- Given a worn and bent coin and estimate the probability that the coin will turn up heads when tossed.
- Unknown probability of heads ***p***. Toss the coin ***n*** times and record the number of times ***r*** that it turns up heads.

  Estimate of $p = r / n$
- If the experiment were *rerun*, generating a new set of ***n*** coin tosses, we might expect the number of heads ***r*** to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for ***p***.
- The Binomial distribution describes for each possible value of ***r*** (i.e., from 0 to n), the probability of observing exactly ***r*** heads given a sample of ***n*** independent tosses of a coin whose true probability of heads is ***p***.

<u>The general setting to which the Binomial distribution applies is:</u>

1. There is a base experiment (e.g., toss of the coin) whose outcome can be described by a random variable 'Y'. The random variable Y can take on two possible values (e.g., Y = 1 if heads, Y = 0 if tails).
2. The probability that Y = 1 on any single trial of the base experiment is given by some constant p, independent of the outcome of any other experiment. The probability that Y = 0 is therefore (1 - p). Typically, p is not known in advance, and the problem is to estimate it.
3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_n$. Let R denote the number of trials for which $Y_i = 1$ in this series of n experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable R will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \qquad \textbf{equ (1)}$$

Mean, Variance and Standard Deviation

The Mean (expected value) is the average of the values taken on by repeatedly sampling the random variable

Definition: Consider a random variable Y that takes on the possible values $y_1, \ldots y_n$. The expected value (Mean) of Y, E[Y], is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i)$$

The Variance captures how far the random variable is expected to vary from its mean value.

Definition: The variance of a random variable Y, Var[Y], is

$$Var[Y] \equiv E[(Y - E[Y])^2]$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean E[Y].
The square root of the variance is called the standard deviation of Y, denoted σ_y

Definition: The standard deviation of a random variable Y, σ_y, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]}$$

In case the ***random variable Y is governed by a Binomial distribution***, then the Mean, Variance and standard deviation are given by

$$E[Y] = np$$
$$Var[Y] = np(1-p)$$
$$\sigma_Y = \sqrt{np(1-p)}$$

Estimators, Bias, and Variance

Let us describe $error_s(h)$ and $error_D(h)$ using the terms in Equation (1) defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$

$$error_D(h) = p$$

Where,

- n is the number of instances in the sample S,
- r is the number of instances from S misclassified by h
- p is the probability of misclassifying a single instance drawn from *D*

- Estimator:
  $error_s(h)$ an ***estimator*** for the true error $error_D(h)$: An estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn

- Estimation bias: is the difference between the expected value of the estimator and the true value of the parameter.

  Definition: The estimation bias of an estimator Y for an arbitrary parameter p is

$$E[Y] - p$$