UNIT 5

Artificial Intelligence and Machine Learning

Prof. Merin Meleet,
Asst.Professor,
Dept of ISE, RVCE

Textbook

- Machine Learning, Tom M. Mitchell, Publisher: McGraw-Hill Science/Engineering/Math;
- (Unit 4 and 5)

CHAPTER 8
INSTANCE BASED LEARNING

- Systems that learn the training examples by heart and then generalizes to new instances based on some similarity measure.
- It is called instance-based because it builds the hypotheses from the training instances.
- It is also known as memory-based learning or lazy-learning.

- For example, If we were to create a spam filter with an instance-based learning algorithm, instead of just flagging emails that are already marked as spam emails, our spam filter would be programmed to also flag emails that are very similar to them.
- This requires a measure of resemblance between two emails. A similarity measure between two emails could be the same sender or the repetitive use of the same keywords or something else.

- **Advantages:**
- Instead of estimating for the entire instance set, local approximations can be made to the target function.
- This algorithm can adapt to new data easily, one which is collected as we go .
- **Disadvantages:**
- Classification costs are high
- Large amount of memory required to store the data, and each query involves starting the identification of a local model from scratch.

Examples

- K Nearest Neighbor (KNN)
- Self-Organizing Map (SOM)
- Learning Vector Quantization (LVQ)
- Locally Weighted Learning (LWL)

k-NEAREST NEIGHBOUR 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.
- More precisely, let an arbitrary instance ***x*** be described by the feature vector

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

- where ***ar (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

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

KNN Algorithm

https://www.youtube.com/watch?v=4HKqjENq9OU&ab_channel=edureka%21edureka%21Verified
https://www.youtube.com/watch?v=6kZ-OPLNcgE

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.

Apply k-NN to the below given dataset

Sepal Length	Sepal Width	Species
5.3	3.7	Setosa
5.1	3.8	Setosa
7.2	3.0	Virginica
5.4	3.4	Setosa
5.1	3.3	Setosa
5.4	3.9	Setosa
7.4	2.8	Virginica
6.1	2.8	Verscicolor
7.3	2.9	Virginica
6.0	2.7	Verscicolor
5.8	2.8	Virginica
6.3	2.3	Verscicolor
5.1	2.5	Verscicolor
6.3	2.5	Verscicolor
5.5	2.4	Verscicolor

Sepal Length	Sepal Width	Species
5.2	3.1	?

Step 1: Find Distance

$$\text{Distance (Sepal Length, Sepal Width)} = \sqrt{(x-a)^2 + (y-b)^2}$$

$$\text{Distance (Sepal Length, Sepal Width)} = \sqrt{(5.2-5.3)^2 + (3.1-3.7)^2}$$

$$\text{Distance (Sepal Length, Sepal Width)} = 0.608$$

Sepal Length	Sepal Width	Species	Distance
5.3	3.7	Setosa	0.608

Sepal Length	Sepal Width	Species	Distance
5.3	3.7	Setosa	0.608
5.1	3.8	Setosa	0.707
7.2	3.0	Virginica	2.002
5.4	3.4	Setosa	0.36
5.1	3.3	Setosa	0.22
5.4	3.9	Setosa	0.82
7.4	2.8	Virginica	2.22
6.1	2.8	Verscicolor	0.94
7.3	2.9	Virginica	2.1
6.0	2.7	Verscicolor	0.89
5.8	2.8	Virginica	0.67
6.3	2.3	Verscicolor	1.36
5.1	2.5	Verscicolor	0.60
6.3	2.5	Verscicolor	1.25
5.5	2.4	Verscicolor	0.75

Sepal Length	Sepal Width	Species	Distance	Rank
5.3	3.7	Setosa	0.608	3
5.1	3.8	Setosa	0.707	6
7.2	3.0	Virginica	2.002	13
5.4	3.4	Setosa	0.36	2
5.1	3.3	Setosa	0.22	1
5.4	3.9	Setosa	0.82	8
7.4	2.8	Virginica	2.22	15
6.1	2.8	Verscicolor	0.94	10
7.3	2.9	Virginica	2.1	14
6.0	2.7	Verscicolor	0.89	9
5.8	2.8	Virginica	0.67	5
6.3	2.3	Verscicolor	1.36	12
5.1	2.5	Verscicolor	0.60	4
6.3	2.5	Verscicolor	1.25	11
5.5	2.4	Verscicolor	0.75	7

Step 2: Find Rank

Sepal Length	Sepal Width	Species	Distance	Rank
5.3	3.7	Setosa	0.608	3
5.1	3.8	Setosa	0.707	6
7.2	3.0	Virginica	2.002	13
5.4	3.4	Setosa	0.36	2
5.1	3.3	Setosa	0.22	1
5.4	3.9	Setosa	0.82	8
7.4	2.8	Virginica	2.22	15
6.1	2.8	Verscicolor	0.94	10
7.3	2.9	Virginica	2.1	14
6.0	2.7	Verscicolor	0.89	9
5.8	2.8	Virginica	0.67	5
6.3	2.3	Verscicolor	1.36	12
5.1	2.5	Verscicolor	0.60	4
6.3	2.5	Verscicolor	1.25	11
5.5	2.4	Verscicolor	0.75	7

Step 3: Find the Nearest Neighbor

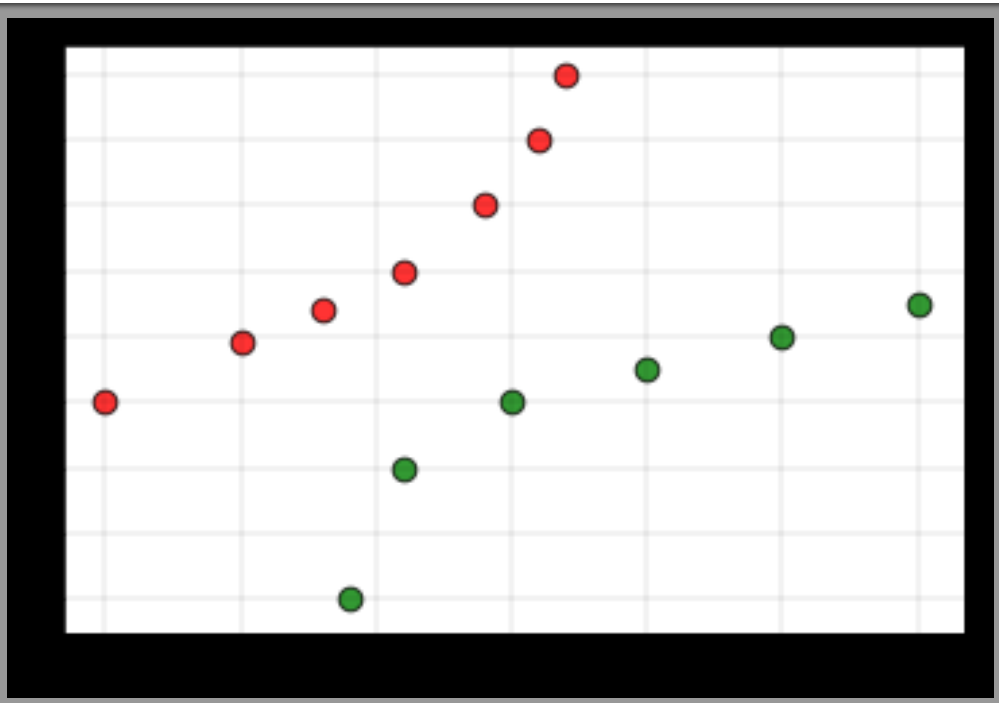
If k = 1 – Setosa

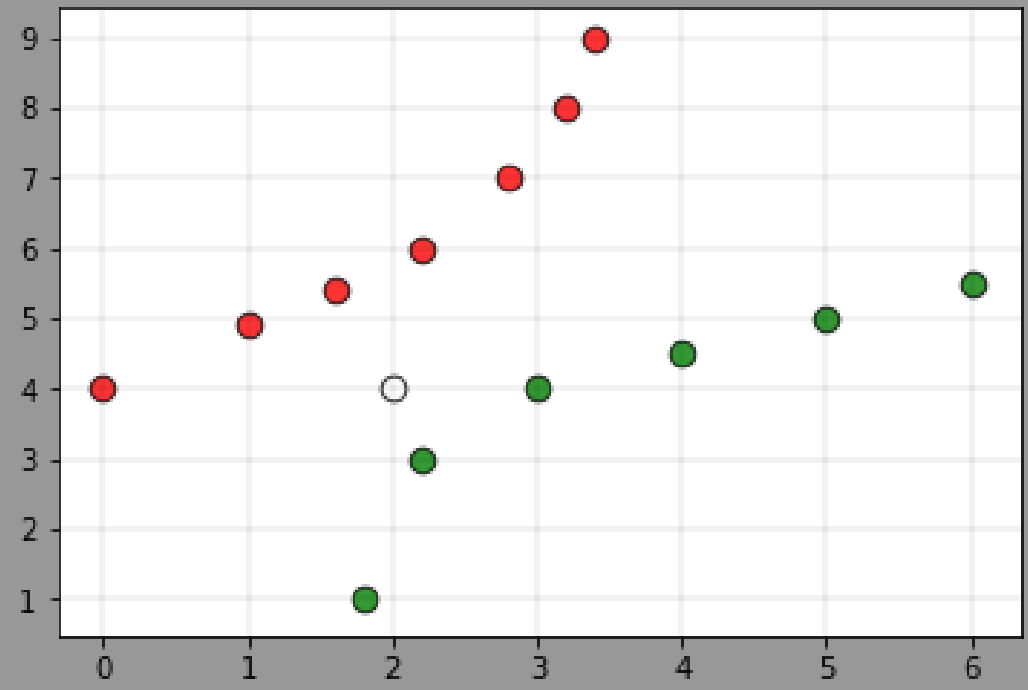
If k = 2 – Setosa

If k = 5 – Setosa

Distance-Weighted k-NEAREST NEIGHBOUR ALGORITHM

- In weighted kNN, the nearest k points are given a weight using a function called as the kernel function.
- The intuition behind weighted kNN, is to give more weight to the points which are nearby and less weight to the points which are farther away.
- Any function can be used as a kernel function for the weighted knn classifier whose value decreases as the distance increases.
- The simple function which is used is the inverse distance function.





Red=Class 0
Green = Class1
kNN Classifies new data point as Class 0

Remarks on k-NN

- It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data.
- The inductive bias corresponds to an assumption that the classification of an instance x_q, will be most similar to the classification of other instances that are nearby in Euclidean distance.
- Sensitive to curse of dimensionality

- kNN is sensitive to :
- Curse of Dimensionality refers to a set of problems that arise when working with high-dimensional data.
- The difficulties related to training machine learning models due to high dimensional data is referred to as ‘Curse of Dimensionality’.
- your data has too many features.
- If we have more features than observations than we run the risk of massively overfitting our model — this would generally result in terrible out of sample performance.

- One additional practical issue in applying k-NNs 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

Locally weighted Regression

- The nearest-neighbor approaches described in the previous section can be thought of as approximating the target function f (x) at the single query point $x = x_q$
- Locally weighted regression is a generalization of this approach.
- It 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.

- For example, we might approximate the target function in the neighborhood surrounding ***x,*** using a linear function, a quadratic function, a multilayer neural network, or some other functional form.
- 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.

General Approach

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

- Let us consider the case of locally weighted regression in which the target function f is approximated near ***x***, 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 ith attribute of the instance ***x***.
- we derived methods to choose weights that minimize the squared error summed over the set D of training examples

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

- This will lead us to the gradient descent training rule

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

where $\acute{\eta}$ is a constant learning rate

- Final Learning Rule derived

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

Kernel function

It is not possible to find a hyperplane or a linear decision boundary for some classification problems.
If we project the data in to a higher dimension from the original space, we may get a hyperplane in the projected dimension that helps to classify the data.

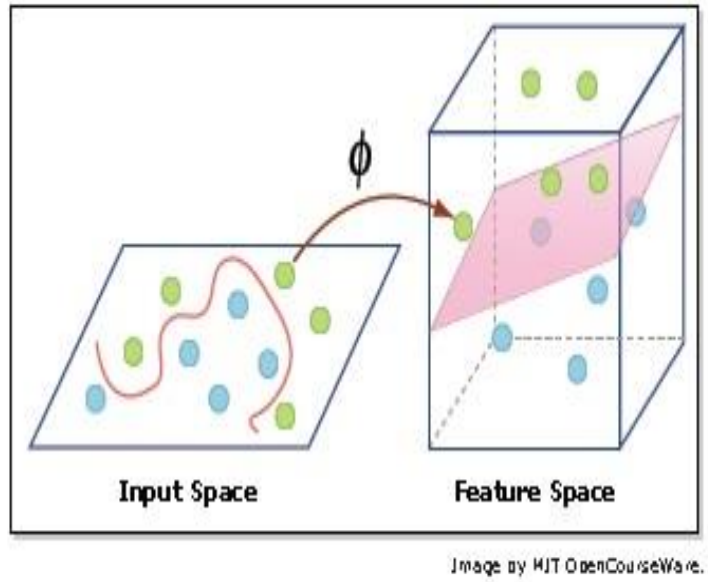


Image by MIT OpenCourseWare.

As we shown in the figure, it is impossible to find a single line to separate the two classes (green and blue) in the input space. But, after projecting the data in to a higher dimension (i.e. feature space in the figure), we could able to find the hyperplane which classifies the data.
Kernel helps to find a hyperplane in the higher dimensional space without increasing the computational cost much. Usually, the computational cost will increase, if the dimension of the data increases.

RADIAL BASIS FUNCTIONS

- A Radial basis function is a function whose value depends only on the distance from the origin.
- In effect, the function must contain only real values.
- Alternative forms of radial basis functions are defined as the distance from another point denoted C, called a center.
- Part of Neural Networks

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

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad \text{Equation (8.8)}$$

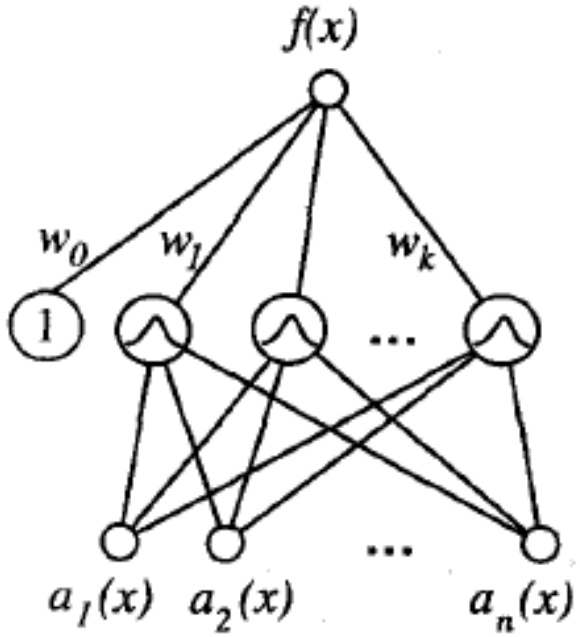
- where each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. Here k is a user-provided constant that specifies the number of kernel functions to be included.

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

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

Radial Basis Function Networks

The function given by Equation (8.8) 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.



A radial basis function network.
Each hidden unit produces an activation determined by a Gaussian function centered at some instance 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.

CASE-BASED REASONING

Instance-based methods such as k-NN and locally weighted regression share three key properties.

- First, they are lazy learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed.
- Second, they classify new query instances by analyzing similar instances while ignoring instances that are very different from the query.
- Third, they represent instances **as** real-valued points in an n-dimensional Euclidean space.

- Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third.
- In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs

Eg: The CADET system

- The 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.

https://www.youtube.com/watch?v=8p7dzVSr4XU&ab_channel=AIandGames

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET.
- Multiple retrieved cases may be combined to form the solution to the new problem.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving

Chapter 13

Reinforcement Learning
UNIT 5 –Part 2

Introduction to Reinforcement Learning

https://www.youtube.com/watch?v=YUbFQlMXShY&ab_channel=SimplilearnSimplilearn

- 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.
- Reinforcement learning is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones.
- In general, a reinforcement learning agent is able to perceive and interpret its environment, take actions and learn through trial and error

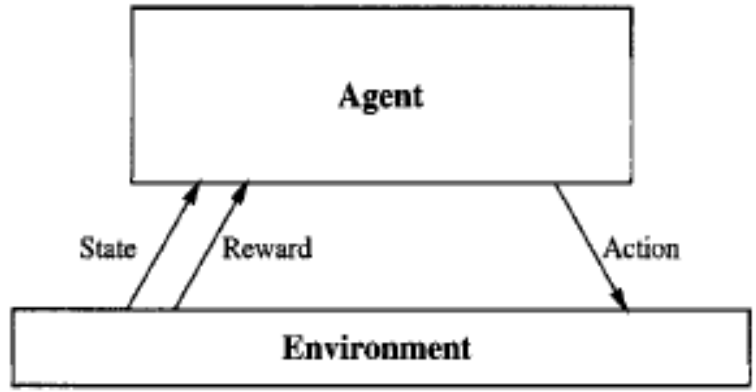
- Example: learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games.
- Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state.
- For example, when 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

Terminology

- 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.
- For example, 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

- 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.
- For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +loo) 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.

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.



Goal: Learn to choose actions that maximize

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

Reinforcement learning problem differs from other function approximation tasks in several important respects

- ***Delayed reward.***
- ***Exploration :*** The learner faces a tradeoff in choosing whether to favor ***exploration*** of unknown states and actions (to gather new information), or ***exploitation*** of states and actions that it has already learned will yield high reward (to maximize its cumulative reward)
- ***Partially observable states:*** it may be necessary for the agent 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.
- ***Life-long learning:*** Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors.

THE LEARNING TASK

- Markov Decision Process
- In a Markov decision process (MDP) 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 δ and r are part of the environment and are not necessarily known to the agent

- In an MDP, 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*, $\pi : S \rightarrow A$, for selecting its next action a_t based on the current observed state s_t; that is, $\pi(s_t) = a_t$. How shall we specify precisely which policy π we would like the agent to learn? One obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time. To state this requirement more precisely, we define the cumulative value $V^\pi(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^\pi(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

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 as described above (i.e., $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$, etc.). Here $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards

- The quantity $V^{\pi}(s_t)$ defined by the previous Equation (13.1) is often called the discounted cumulative reward achieved by policy n from initial state s.

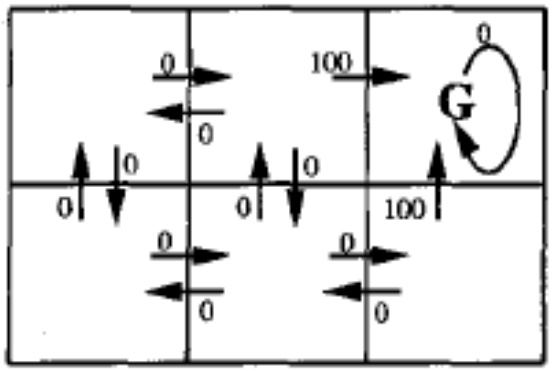
We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy π that maximizes $V^{\pi}(s)$ for all states s. We will call such a policy an *optimal policy* and denote it by π^*.

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \tag{13.2}$$

To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state s; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s.

Sample Grid –World Environment

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.
Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into the state labeled G.

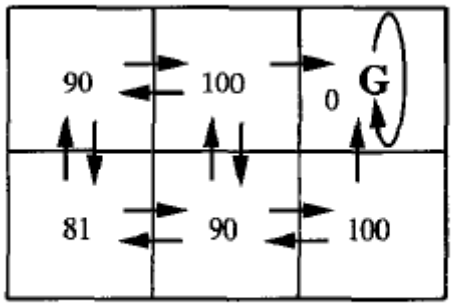


$r(s, a)$ (immediate reward) values

It is convenient to think of the state G as the goal state, because the only way the agent can receive reward, in this case, is by entering this state.
Note in this particular environment, the only action available to the agent once it enters the state G is to remain in this state. For this reason, we call G an absorbing state

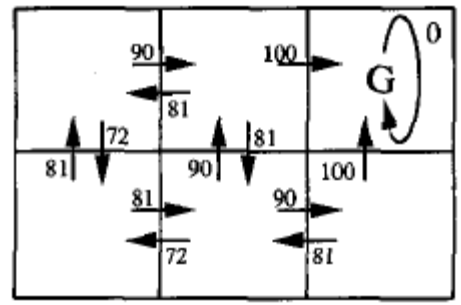
- NOTE: discount factor
- The discount factor essentially determines how much the reinforcement learning agents cares about rewards in the distant future relative to those in the immediate future.
- If $\gamma=0$, the agent will be completely myopic and only learn about actions that produce an immediate reward.
- If $\gamma=1$, the agent will evaluate each of its actions based on the sum total of all of its future rewards.

- Once the states, actions, and immediate rewards are defined, and once we choose a value for the discount factor γ, we can determine the optimal policy π^* and its value function $V^*(s)$. In this case, let us choose $\gamma = 0.9$.
- Like any policy, this policy specifies exactly one action that the agent will select in any given state.
- the optimal policy directs the agent along the shortest path toward the state G.



$V^*(s)$ values

Each grid square represents a distinct state, each arrow a distinct action.
The immediate reward function, ***r(s,*** a) gives reward 100 for actions entering the goal state G, and zero otherwise.
Values of ***V*(s)*** and ***Q(s, a)*** follow from ***r(s,*** a), and the discount factor $y = 0.9$.



$Q(s, a)$ values

The value of V* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100.
Thereafter, the agent will remain in the absorbing state and receive no further rewards.
Similarly, the value of V* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100).
Thus, the discounted future reward from the bottom center state is
$o+ \gamma 100+ \gamma^2 0+ \gamma^3 0+...=90$

Q LEARNING

- How can an agent learn an optimal policy π* for an arbitrary environment?
- the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2,$
- The agent should prefer state *s1* over state *s2* whenever $V^*(sl) > V^*(s2)$, because the cumulative future reward will be greater from *s1.* .

- 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.

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

Eqn 1

- Thus, the agent can acquire the optimal policy by learning *V**, provided it has perfect knowledge of the immediate reward function ***r*** and the state transition function δ.
- Learning *V** is a useful way to learn the optimal policy only when the agent has perfect knowledge of δ and *r.*

The Q function

- Let us define the evaluation function *Q(s, a)* so that its value is the maximum discounted cumulative reward that can be achieved starting from state *s* and applying action *a* as the first action.
- In other words, the value of *Q* is the reward received immediately upon executing action *a* from state *s*, plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \qquad \text{Eqn 2}$$

- Rewriting eqn 2 using the Eqn 1

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

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:

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

- $s \leftarrow s'$

- we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function.
- In this algorithm the learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair.
- The table entry for the pair (s, a) stores the value for Q^(s a)- learner's current hypothesis about the actual but unknown value Q(s, a).
- The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero).

- The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward r = r(s, a) and the new state s' = δ(s, a).
- It then updates the table entry for Q^(s,***a)*** following each such transition, according to the rule:

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

Convergence

- Will the algorithm converge toward a $\hat{Q}$ equal to the true Q function?
- The answer is yes, under certain conditions.
- First, we must assume the system is a deterministic MDP.
- Second, we must assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, $|r(s, a)| < c$.
- Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often.
- By this third condition we mean that if action a is a legal action from state s, then overtime the agent must execute action a from state s repeatedly and with nonzero frequency as the length of its action sequence approaches infinity.

Updating Sequence

- it can learn the $\boldsymbol{Q}$ function (and hence the optimal policy) while training from actions chosen completely at random at each step, as long as the resulting training sequence visits every state-action transition infinitely often.
- A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically.

NONDETERMINISTIC REWARDS AND ACTIONS

- Here we consider the nondeterministic case, in which the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes.
- In robot problems with noisy sensors and effectors it is often appropriate to model actions and rewards as nondeterministic.
- In such cases, the functions $\delta(s, a)$ and $r(s, a)$ can be viewed as first producing a probability distribution over outcomes based on s and a, and then drawing an outcome at random according to this distribution

- In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic.
- The obvious generalization is to redefine the value V^{π} of a policy π to be the *expected value* (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^{\pi}(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

- where, as before, the sequence of rewards r_{t+i} is generated by following policy $\boldsymbol{\pi}$ beginning at state *s.*

As before, we define the optimal policy π^* to be the policy π that maximizes $V^\pi(s)$ for all states s. Next we generalize our earlier definition of Q by taking its expected value.

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \tag{13.8}$$

where $P(s'|s,a)$ is the probability that taking action a in state s will produce the next state s'. Note we have used $P(s'|s,a)$ here to rewrite the expected value of $V^*(\delta(s,a))$ in terms of the probabilities associated with the possible outcomes of the probabilistic δ.

To summarize, we have simply redefined $Q(s,a)$ in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

New Training rule

Consider, for example, a nondeterministic reward function $r(s, a)$ that produces different rewards each time the transition $\langle s, a \rangle$ is repeated. In this case, the training rule will repeatedly alter the values of $\hat{Q}(s, a)$, even if we initialize the $\hat{Q}$ table values to the correct Q function. In brief, this training rule does not converge. This difficulty can be overcome by modifying the training rule so that it takes a decaying weighted average of the current $\hat{Q}$ value and the revised estimate. Writing $\hat{Q}_n$ to denote the agent's estimate on the nth iteration of the algorithm, the following revised training rule is sufficient to assure convergence of $\hat{Q}$ to Q:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \quad (13.10)$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s, a)} \quad (13.11)$$

where s and a here are the state and action updated during the nth iteration, and where $visits_n(s, a)$ is the total number of times this state-action pair has been visited up to and including the nth iteration.

END OF UNIT 5