Day 15 & 16

30 Days of Machine Learning

Efficient Distance Computations &

Reinforcement Learning and Evaluating Hypotheses

Prepared By

Ahmed Ali

Quaid-e-Awam University of Engineering Science & Technology Nawabshah

Efficient Distance Computations: the KD-Tree

As was mentioned above, computing the distances between all pairs of points is very computationally expensive. Fortunately, as with many problems in computer science, designing an efficient data structure can reduce the computational overhead a lot. For the problem of finding nearest neighbors the data structure of choice is the KD-Tree. It has been around since the late 1970s, when it was devised by Friedman and Bentley, and it reduces the cost of finding a nearest neighbor to O(log N) for O(N) storage. The construction of the tree is $O(N \log^2 N)$, with much of the computational cost being in the computation of the median, which with a naïve algorithm requires a sort and is therefore O(N log N), or can be computed with a randomized algorithm in O(N) time.

The idea behind the KD-tree is very simple. You create a binary tree by choosing one dimension at a time to split into two and placing the line through the median of the point coordinates of that dimension. The points themselves end up as leaves of the tree. Making the tree follows pretty much the same steps as usual for constructing a binary tree: we identify a place to split into two choices, left and right, and then carry on down the tree. This makes it natural to write the algorithm recursively. The choice of what to split and where is what makes the KD-tree special. Just one dimension is split in each step, and the position of the split is found by computing the median of the points that are to be split in that one dimension and putting the line there.

In general, the choice of which dimension to split alternates through the different choices, or it can be made randomly. The algorithm below cycles through the possible dimensions based on the depth of the tree so far, so that in two dimensions it alternates horizontal and vertical splits. The center of the construction method is simply a recursive function that picks the axis to split on, finds the median value on that axis, and separates the points according to that value, which in Python can be written as:

```
# Pick next axis to split on
whichAxis = np.mod(depth,np.shape(points)[1])

# Find the median point
indices = np.argsort(points[:,whichAxis])
points = points[indices,:]
median = np.ceil(float(np.shape(points)[0]-1)/2)

# Separate the remaining points
goLeft = points[:median,:]
goRight = points[median+1:,:]

# Make a new branching node and recurse
newNode = node()
newNode.point = points[median,:]
newNode.left = makeKDtree(goLeft,depth+1)
newNode.right = makeKDtree(goRight,depth+1)
return newNode
```

Suppose that we had seven two-dimensional points to make a tree from: (5, 4), (1, 6), (6, 1), (7, 5), (2, 7), (2, 2), (5, 8) (as plotted in Figure 1).

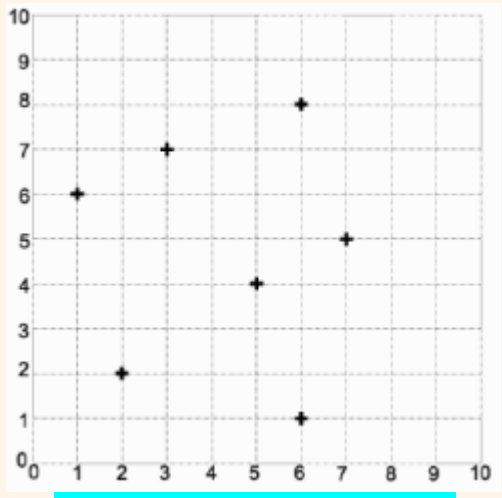


FIGURE 1: The initial set of 2D data.

The algorithm will pick the first coordinate to split on initially, and the median point here is 5, so the split is through x = 5. Of those on the left of the line, the median y coordinate is 6, and for those on the right it is 5. At this point we have separated all the points, and so the algorithm terminates with the split shown in Figure 2 and the tree shown in Figure 3.

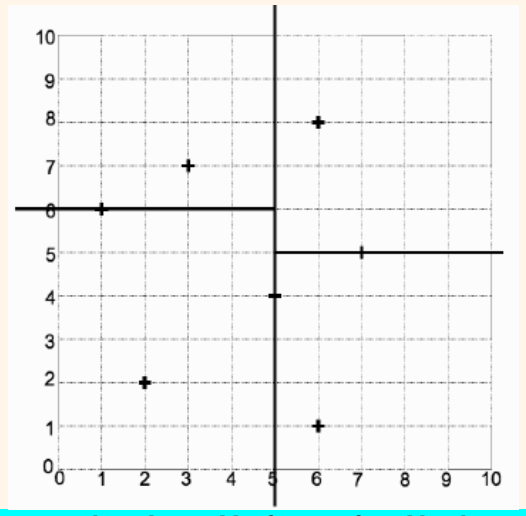


FIGURE 2: The splits and leaf points found by the KD-tree

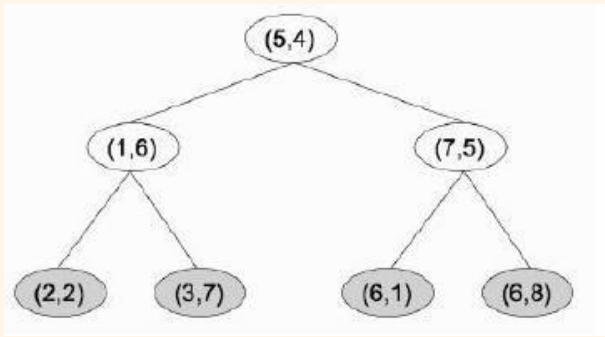


FIGURE 3: The KD-tree that made the splits.

Searching the tree is the same as any other binary tree; we are more interested in finding the nearest neighbors of a test point. This is easy: starting at the root of the tree you recurse down through the tree comparing just one dimension at a time until you find a leaf node that is in the region containing the test point. Using the tree shown in Figure 3 we introduce the test point (3, 5), which finds (2, 2) as the leaf for the box that (3, 5) is in. However, looking at Figure 10 we see that this is not the closest point at all, so we need to do some more work.

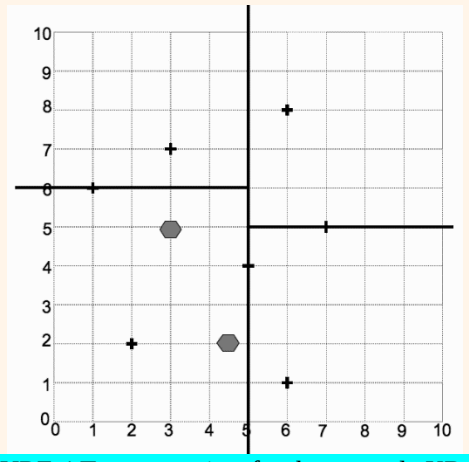


FIGURE 4 Two test points for the example KD-tree

Distance Measures:

We have computed the distance between points as the Euclidean distance, which is something that you learnt about in high school. However, it is not the only option, nor is it necessarily the most useful. In this section we will look at the underlying idea behind distance calculations and possible alternatives. If I were to ask you to find the distance between my house and the nearest shop, then your first guess might involve taking a map of my town, locating my house and the shop, and using a ruler to measure the distance between them. By careful application of the map scale you can now tell me how far it is. However, when I set out to buy some milk I'm liable to find that I have to walk rather further than you've told me, since the direct line that you measured would involve walking through (or over) several houses, and some serious fence-scaling. Your 'as the crow flies' distance is the shortest possible path, and it is the straight-line, or Euclidean, distance. You can measure it on the map by just using a ruler, but it essentially consists of measuring the distance in one direction (we'll call it north-south) and then the distance in another direction that is perpendicular to the first (let's call it east-west) and then squaring them, adding them together, and then taking the square root of that. Writing that out, the Euclidean distance that we are all used to is:

$$d_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

where (x1, y1) is the location of my house in some coordinate system (say by using a GPS tracker) and (x2, y2) is the location of the shop.

If I told you that my town was laid out on a grid block system, as is common in towns that were built in the interval between the invention of the motor car and the invention of innovative town planners, then you would probably use a different measure. You would measure the distance between my house and the shop in the 'north-south' direction and the distance in the 'east-west' direction, and then add the two distances together. This would correspond to the distance I had to walk. It is often known as the city-block or Manhattan distance and looks like:

$$d_C = |x_1 - x_2| + |y_1 - y_2|.$$

The point of this discussion is to show that there is more than one way to measure a distance, and that they can provide radically different answers. These two different distances can be seen in Figure 5. Mathematically, these distance measures are known as metrics. A metric function or norm takes two inputs and gives a scalar (the distance) back, which is positive, and 0 if and only if the two points are the same, symmetric (so that the distance to the shop is the same as the distance back), and obeys the triangle inequality, which says that the distance from a to b plus the distance from b to c should not be less than the direct distance from a to c.

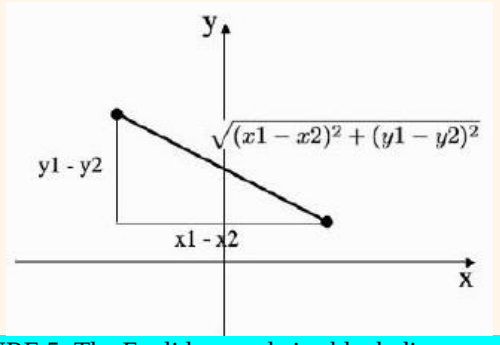


FIGURE 5: The Euclidean and city-block distances between two points

Most of the data that we are going to have to analyze lives in rather more than two dimensions. Fortunately, the Euclidean distance that we know about generalizes very well to higher dimensions (and so does the city-block metric). In fact, these two measures are both instances of a class of metrics that work in any number of dimensions. The general measure is the Minkowski metric, and it is written as:

$$L_k(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^k\right)^{\frac{1}{k}}.$$

If we put k = 1 then we get the city-block distance (Equation (7.12)), and k = 2 gives the Euclidean distance (Equation (7.11)). Thus, you might possibly see the Euclidean metric written as the L2 norm and the city-block distance as the L1 norm. These norms have another interesting feature. Remember that we can define different averages of a set of numbers. If we define the average as the point that minimizes the sum of the distance to every datapoint, then it turns out that the mean minimizes the Euclidean distance (the sum-of-squares distance), and the median minimizes the L1 metric.

There are plenty of other possible metrics to choose, depending upon the dataspace. We generally assume that the space is flat (if it isn't, then none of these techniques work, and we don't want to worry about that). However, it can still be beneficial to look at other metrics. Suppose that we want our classifier to be able to recognize images, for example of faces. We take a set of digital photos of faces and use the pixel values as features. Then we use the nearest neighbor algorithm that we've just seen to identify each face. Even if we ensure that all the photos are taken fully face-on, there are still a few things that will get in the way of this method. One is that slight variations in the angle of the head (or the camera) could make a difference; another is that different distances between the face and the camera (scaling) will change the results; and another is that different lighting conditions will make a difference. We can try to fix all these things in preprocessing, but there is also another alternative: use a different metric that is invariant to these changes, i.e., it does not vary as they do. The idea of invariant metrics is to find measures that ignore changes that you don't want. So, if you want to be able to rotate shapes around and still recognize them, you need a metric that is invariant to rotation.

A common invariant metric in use for images is the tangent distance, which approximates the Taylor expansion in first derivatives and works very well for small rotations and scaling's.

for example, it was used to halve the final error rate on nearest neighbor classification of a set of handwritten letters. Invariant metrics are an interesting topic for further study, and there is a reference for them in the Further Reading section if you are interested.

Reinforcement Learning and Evaluating Hypotheses

. 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:

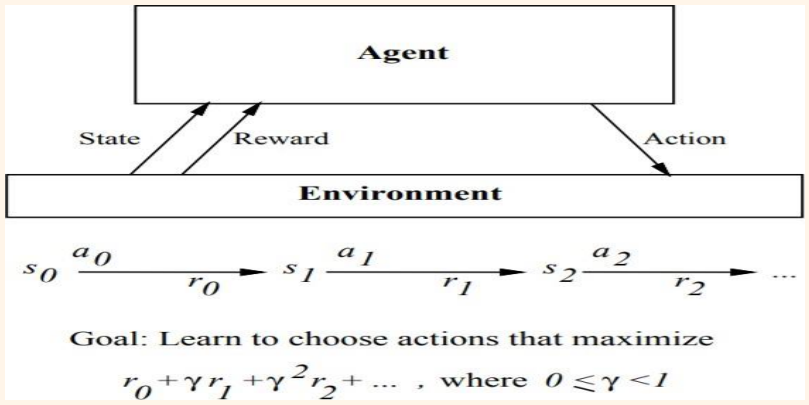
- A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn."
- The robot may have a goal of docking onto its battery charger whenever its battery level is low.
- The goal of docking to the battery charger can be captured by assigning a positive reward (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 performs any of a set of possible actions A. Each time it performs an action a, in some state st the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states si, actions ai, and immediate rewards ri as shown in the figure.

• The agent's task is to learn a control policy, π: S → A, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.



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 situation's 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.

4.1. 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 st, chooses a current action at, and performs it.

• The environment responds by giving the agent a reward rt = r(st, at) and by producing the succeeding state st+l = δ(st, at). Here the functions δ(st, at) and r(st, at) 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, π: S → A, for selecting its next action a, based on the current observed state st; that is, (st) = at.

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 Vπ (st) achieved by following an arbitrary policy π from an arbitrary initial state **st** 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 rt+i is generated by beginning at state st 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 Vπ (st) 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**

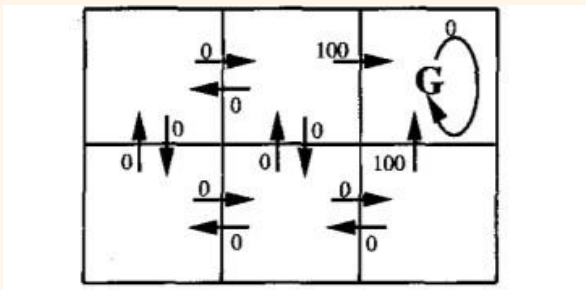
$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

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 Vπ (st) 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 Vπ*(s) an optimal policy as V*(s). V*(s) gives the maximum discounted cumulative reward that the agent can obtain starting from state s.

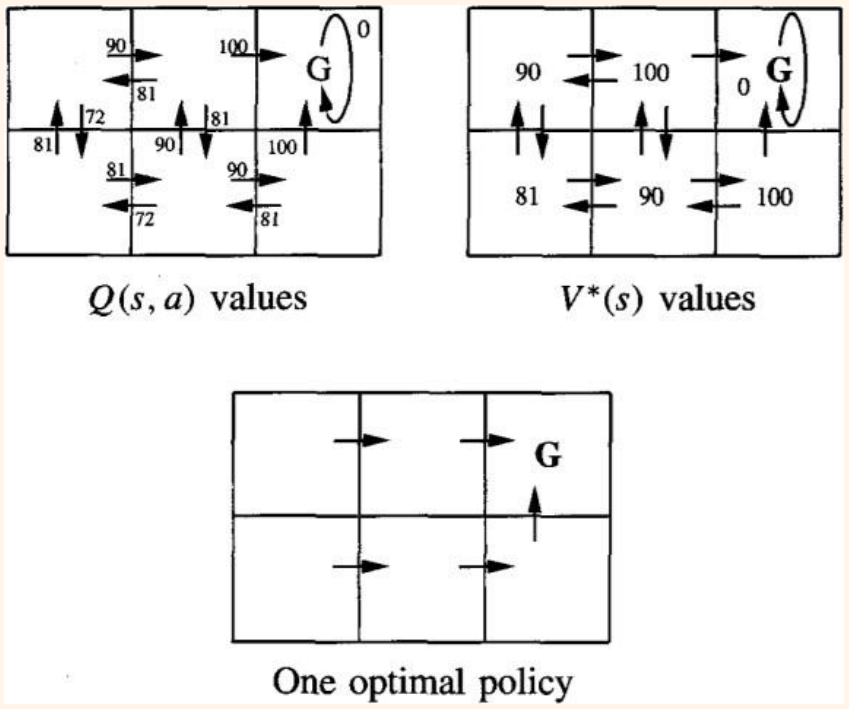
Example: A simple grid-world environment is depicted in the diagram



$r(s, a)$ (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.



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+ γ 100+ γ2 0+ γ3 0+... = 90

4.2. 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(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.

What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state sl over state s2 whenever V*(sl) > V*(s2), because the cumulative future reward will be greater from sl 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.

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

4.3. 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)**.

4.4. 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

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

4.5. 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:

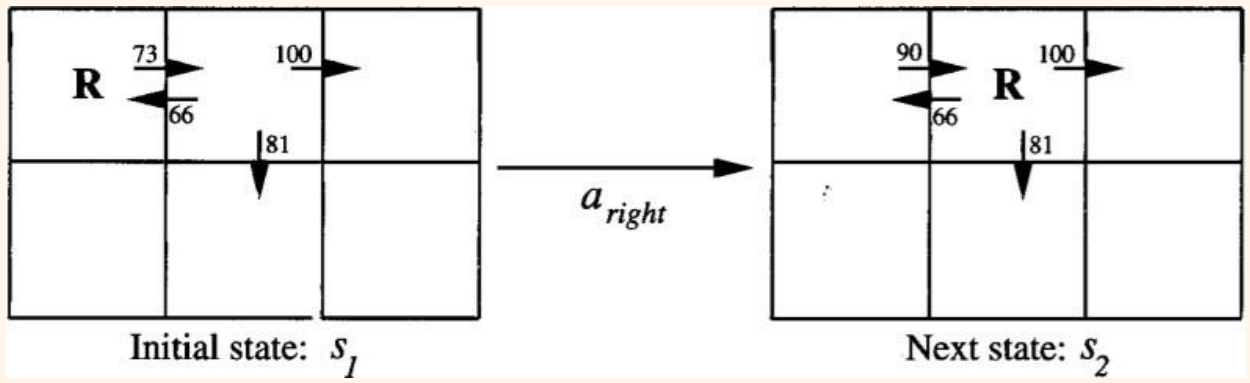
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

- 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

5.6. 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).

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

4.7. 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, $|r(s, a)| < c$

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

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

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 $\hat{Q}_{n+1}(s, a)$ 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.

Up next:

Experimentation Strategies