Chapter 1

Lab 3 — Reinforcement Learning

1.1 INTRODUCTION

Here you will use TD-learning to get an agent to move around in a little environment. The environment is represented as a matrix (so a grid-world) where 0's represent free space, 1 represents the goal location, and -1's represent walls and obstacles. You task is to get the agent to find the goal(s) and avoid the obstacles. The agent gets a reinforcement of 1 at the goal location, -1 when it tries to bump into a wall or leave the environment, and otherwise. Thus, the agent is rewarded for finding path to the goals and for avoiding obstacles. In the first part, the states of the system are represented as states in the environment. In the second part, you will use a neural network to generalize over states, so that similar states are assigned similar Q values.

1.2 Part I: Reinforcement Learning of the States and Actions.

In the first part, we will learn the appropriate action for every state of the system. The states will be the location in the environment (represented as a grid), and the goal is to learn $Q(s,a)$.

1.2.1 Tasks

1. Train the system using Q-learning (or Sarsa) on a very small grid. Examine Q and convince yourself that it is learning appropriately. What do you expect Q at the goal state to be? What do you expect Q to be after one learning sequence? After many learning sequences?

2. Train the system on a larger environment. Record how the time to reach the goal

state and the average reinforcement changes as the system goes through more and more learning sequences. (The average reinforcement reveals how well the agent is at avoiding obstacles.) Does the agent learn a short path to the goal?

3. Repeat the above for TD(λ) learning. Compare the two methods for speed of learning and learned performance.

4. The agent is not rewarded for finding a short path to the goal. In order to encourage it to find a short path, you could change the reinforcement it receives to the following: -2 for running into walls or leaving the environment, 1 for finding the goal, and -1 otherwise. In order to maximize the reinforcement received, it needs to find a short path. The reinforcement is given by the function `response`. Copy response.m to your directory and modify it to give the desired reinforcement. Does this lead to shorter paths?

1.3 RESOURCES

I have written some simple TD-learning functions

Environments: I have made five: TinyWorld, Line, BigEmpty, MazeWorld, and Stadium. To load them, type

```
load TinyWorld
```

where TinyWorld can be replaced by the names of the other environments. These are simply matrices, so you can easily make others. You can view them using

```
imagesc(TinyWorld); colormap(gray);
```

(The colormap is optional.)

Qlearn: A simple Q-learning function. The form is

```
[newQ,statetrace,rewardtrace]=Qlearn(Q,Environ,start,parameters)
```

where

Q is the Q matrix;

Environ is the environment matrix, e.g. TinyWorld;

start is the starting state in the environment,

parameters is a list of parameters: ε used by ε-greedy algorithm, γ, learning rate, a flag which determines whether the algorithm implements Q-learning (0) or Sarsa (1), and the maximum number of states the agent goes through before the learning ends. E.g. parameters could be `[0.1 0.9 0.1 0 1000 ]`. Missing values or NaN are replaced by defaults.

This learns the complete matrix of all states and actions. Q is a matrix of the form $Q(i, j, a)$ where (i, j) is the location in the environment, and a takes values: 1 (down), 2 (right), 3 (up), and 4 (left).

This algorithm stops when the goal has been reached or the maximum number of states has been visited. It outputs

newQ The updated Q matrix,

statetrace a list of the states visited,

rewardtrace a list of the rewards. So, for example, `mean(rewardtrace)` gives the average reward in the learning sequence, and `length(rewardtrace)` gives the number of states visited in the learning sequence.

TDLambda: similar to Qlearn, only implements a TD(λ) version of Sarsa. The variables are the same except λ is an additional parameter. It comes after the learning rate and before the Sarsa flag.

`initialize_Q`: This function initializes the starting state to a random, allowed state and Q to zero. It is used as follows,

```
[startstate,Q]=initializeQ(Environ);
```

show_run: shows the states visited in a learning sequence. Use is

```
show_run(Environ,statetrace);
```

where Environ is the name of the environment, and statetrace is output from the learning function.

show_Q: This shows the direction of best move according to the Q matrix. If there is more than one best direction, it displays a dot. It is used as follows,

```
show_Q(Environ,Q);
```

imageQ: This is another visualization tool. It shows the Q matrix for each of the four directions. It is called like this,

```
imageQ(Q);
```

In this part of the lab the states are simply the coordinates of the environment.

1.4 Part II: Thinking about Generalization With a Neural Network

In a large environment, there will be too many states to the approach in which every possible combination of coordinates is a different state. Firstly, the agent may not have

enough memory to remember so many state – action pairs. Secondly, it may take an extremely long time to explore enough of the states so that good routes are known from most of the states.

To alleviate this, we will use a neural network to approximate the Q-function and, hopefully, generalize an appropriate Q-value for unexplored states. We will do this using the simplest learning rule, namely Q-learning. To see how this works, first let us remember the Q-learning update rule. If s is the current state, and a is the current action, and under that action $s \to s'$, the update equation is

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma \max_{a^*} Q(s', a^*) - Q(s,a) \right]. \tag{1.1}$$

Consider a simple neural network with the states as inputs and with one output per state, where the outputs should be the Q values. Denote the network output as $f_a(s|w)$. This is abstract, and only serves to remind us that it depends on some set of weights and biases denoted w. Instead of updating Q, we will train the network with the targets,

$$f_a(s|w) \to r + \gamma \max_{a^*} f_{a^*}(s'|w). \tag{1.2}$$

We are going to implement this. Consequently, we would like to find the simplest neural network which can do the task.

This part is done with paper and pencil.

1.4.1 Tasks

1. Convince yourself that equation (1.2) is the correct update rule.
2. Consider empty 2-d space with a goal somewhere. With a pencil and paper, sketch what the Q function would look like. If that gives you trouble, consider a simple 1-d scenario. Sketch the shape of the Q-function. (If that gives you trouble, ask for help.)
3. What network architecture is required to do this task? Could a linear associator do this. How about a sigmoidal perceptron? Decide what type of architecture is appropriate.
4. Consider what the input – output pairs will be for our situation in which there are four actions, and two inputs. Write down the targets in detail. (You are going to implement it in the next section.)

1.5 Part III — Implementing the Neural Network

We are now going to apply the neural network to this task. In order to do this, you will need to modify the Qlearn function. I have made a function called nnetQlearn which is called like this,

```
[newnet,statetrace,rewardtrace]=nnetQlearn(net,Environ,start,parameters)
```

where `net` is the name of the neural network. Cruelly, however, I have not implemented the computation of the target and the network update. I have also made a function called `net2Q` which computes a Q table from the network. It is called like this,

```
Q=net2Q(Environ,net);
```

This allows you to use previous functions like `show_Q` to visualize what is going on.

1. Copy this function to your working directory
2. Open it in an editor. Matlab has an build in editor, called edit.
3. There is a comment where you need to edit. Follow the directions. Compute the target, and then train the network on that target. Since you want to train the network incrementally, use `adapt` rather than train.
4. To use nnetQlearn, initialize a network to use 'traingd', which is a learning algorithm which can be used incrementally. Then repeatedly run nnetQlearn and observe Q using net2Q and see how it evolves.
5. You may find that the initialize network produces Q values which make exploration difficult (we cannot initialize all Q's to zero using this approach). Therefore, you might want to start with a large value of ε in the ε-greedy algorithm, and decrease it as the system learns. For example,

   ```
   [newnet,statetrace,rewardtrace]=nnetQlearn(net,Environ,start,[0.5])
   ```

   would search randomly half the time.