TDDC17 Artificial Intelligence

TDDC17-Lab5

Aim

In this lab, you will learn how to control a rocket in a simulated continuous environment with reinforcement learning. We will use the Q-learning algorithm which is simple yet powerful. For pedagogical and computational reasons we will here use classical table-based Q-learning, but coupled with deep learning and large computational resources, this same algorithm is used in many recent advances in AI.

Preparation

Read chapter 17.1 about Markov Decision Processes (MDPs) and 21.1-3 about reinforcement learning in the course book. You only need an very shallow understanding of MDPs but the chapters 21.1-3 are important, especially 21.3 which describes the Q-learning algorithm.

NOTE: Some versions of the book have different numbering, all references herein are to the reinforcement learning chapter.

The Rocket Simulator

The lab environment consists of a simple physics engine which can contain objects that are built from point masses and springs. Different types of actuators can be mounted on these objects such as wheels and rocket engines. The environment can also contain solid objects that can be used for simulating obstacles and/or ground.

The setup for this lab does not contain any obstacles. The rocket that is supposed to be controlled, is shown in figure 1. It has three rocket engines that can be controlled independently. They are either turned on or off, nothing in between is allowed.

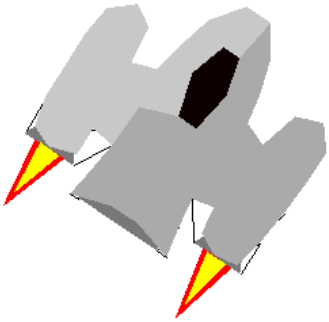

Figure 1: The rocket

The simulated environment is controlled through the following keyboard shortcuts:

- The rocket can be controlled manually by the 'A','S','D' and 'W' buttons the keyboard.
- The simulator is paused on 'R'.
- The simulator is sped up on 'V' and slowed down on 'B'.
- The graphics can be toggled off on 'M' for even faster simulation. This can be used to accelerate learning in parts II and III.
- The learning controller (parts II and III) can be turned off on 'P' and turned on again on 'O'. *Note: This should be done before controlling the rocket manually in later parts, otherwise it will confuse the learning algorithm*
- The learning controller can be told to stop exploring and focus on maximizing known utility by pressing 'E' (toggle). This should be done when it has converged.
- Keys '1' and '2' are unused custom keys which may be used to trigger the corresponding functions in the learning controller if you desire additional functionality.

Part I: Tutorial

The first part of the lab is a tutorial that will introduce you to the environment that is used in this lab and how you can access the sensors and actuators of the simulated rocket to be able to control it.

The rocket has a set of sensors that can be queried. The sensors are all real-valued and their values can be read by calling `getValue()` on the corresponding `DoubleFeature` reference. See the **code skeleton** for part I as an example. The rocket engines can be turned on or off individually by using `engine.setBursting(true)` or false respectively.

Tasks for part I:

1. Prepare the local files e.g.:
   - `cp -r /home/TDDC17/labs/lab5_workspace/ .`
2. Start the Eclipse editor:
   - `eclipse`
3. When the eclipse editor asks for a path to the workspace choose the **lab5_workspace** folder which you copied in step 1.
4. The Eclipse editor will compile the java files automatically. Refresh the project in the workspace by right clicking on it and choosing refresh. You can start the lab by using the dropdown menu on the run button (green circle with a white triangle/play symbol) and selecting **Tutorial**.
5. Open the TutorialController.java file in src subfolder. Implement the `tick()` method so that it receives readings from the sensors "angle", "vx", "vy" and prints them out on the standard output.
6. Try to fly the rocket manually with the "W", "A" and "D" buttons that control the rocket engines and check that your printouts are working. The graphics is unfortunately rather "slow" if you use the thin clients.
7. Try setting the rockets to fire or stop if "vy" or some other sensor falls below/above some threshold.

NOTE: in case you have problems with running the eclipse environment on Linux machines try the following: close eclipse, delete .eclipse directory from your home folder, start ecplise again. If this doesn't help there is a console version of the lab. Following scripts: compile, run_TutorialController, run_QLearningController are located in QLearning subfolder of the lab package.

Running it on your personal computer

As the lab is self-contained you can just follow the directions above and copy over the directory to your personal computer. To make things easier we also provide a **.zip file** with the workspace folder from above. Just extract that and point Eclipse to it during start-up. If you do not have Eclipse installed it is recommended that you download the Juno release from **here** as the workspaces in the latest version is incompatible with the Eclipse version used in the labs. Note that on some Windows and Eclipse combinations you may need to switch the path symbols from '/' to '\' in the project Properties->Run/Debug Settings->Arguments dialog box.

Part II: Q-Learning Angle Controller

In this part of the lab you will construct a Q-learning controller that will turn to rocket to face up all the time. In the next part this will be extended to full hovering behavior. This requires you to implement three parts:

1. A state space representation and a reward function function based on the sensors (inputs)
2. An action representation (possible outputs)
3. The actual Q-learning algorithm.

To do this you will be editing two files, `QLearningController.java` and `StateAndReward.java`. This will later be extended for the final hover controller in part III below.

As the only objective is to face up, the state space and reward function for this part only needs to rely on the angle of the rocket. The states need to be discrete in this simple table version of Q-learning. The sensor variables can be discretized in many ways and it is up to you how this should be done in your implementation. A function for uniform discretization within certain bounds is provided in the **code skeleton**. You do not have to write your own discretization methods for this lab. Note that the state is represented by a string in this implementation, which you can construct in any way you wish from the chosen variables as long as it uniquely identifies each state. It is strongly recommended that you test it by flying around to make sure most states the rocket can/will be in are represented. Remember that the agent only sees the states you define, so if you give all downward facing angles one state the agent will not know to which side it is facing. More is generally better but may take longer time to learn. For the angle controller this is rarely a problem.

The action representation should be implemented in `performAction(int action)` so that the given integer maps to a rocket activation pattern. The simplest pattern is to use four actions for {None,Left,Right,Middle}, but in our experience it can be useful to at least have actions for firing both the left/right rocket and the middle at the same time.

The simple table-based Q-learning algorithm that we will use in this lab is described in chapter 21.3 (Figure 21.8) in the course book. It is **strongly** recommended that you have an understanding of Q-learning before proceeding with this lab. You may implement the Q-function (or anything in this lab) in any way you want, but we have given you the beginnings of an implementation in `QLearningController.java`. This implementation already contains logic for doing most of the basic internal book keeping required, in addition to some suitable default parameters for the missing Q-learning update that you need to implement. Look around in the source file to identify useful constants (top) and helper function (below) that you can use. The table of Q-values are stored in a Map of strings to double, `HashMap<String, Double> map = new HashMap<String, Double>();`, where the index string is the string representation of the state appended to the action number. Use `map.put(qString, value)` to set a Q-value and `map.get(qString)` to read it. More information about the HashMap can be found **here**. **Make sure that you understand what is going on in the program.** It can be a good idea to start off by answering question 6 below to gain a better understanding of what you are doing.

A simple *epsilon-greedy* exploration function is used in the action selection unless exploration is de-activated in the simulator ('E'). This simply selects a random action with epsilon probability (0.5 in this case) and the highest utility action with probability (1 - epsilon). This should be turned off when the agent has converged to evaluate its performance.

The implementation also contains a domain specific trick to speed up learning by repeating each action a number of times until the agent reaches a new state. This is to lessen the impact of using a simple table-based learner on a problem that is inherently continuous, as most of the time no state change would occur due to the discretization not having a fine enough grid.

Q-Learning Advice

Remember that you can speed up learning in the simulator by pressing 'V' and turn off the graphics on 'M'. Check from time to time if it still seems to be making progress. This can be observed both visually and by looking at reward and Q-values in the console print-outs. With the given discount factor gamma (0.95) a good rule of thumb is that when converged in this domain the Q-values should be roughly 20 times the reward it is on average generating, and if it behaves well this average should in turn be close to the maximum possible reward you defined. Do not forget to turn off exploration while evaluating its performance!

Some common problems

- Do not forget that if you turn on a rocket engine it will be persistently on until you turn it off. You need to explicitly switch off all engines you do not want to be on for an action.
- Be careful with where you use the old state and the new state in the Q-update that you implement.
- Be careful that the state string you concatenate together is unique for each variable combination, including the actions that are later added! It is good practice to separate the variables by some character so they cannot overlap.

Tasks for part II:

1. Select the QLearning project in the Eclipse workspace and open the `QLearningController.java` and `StateAndReward.java` file in the src subfolder.
2. You can start the lab by using the dropdown menu on the run button (green circle with a white triangle/play symbol) and selecting **QLearning**.
3. Implement the state and reward functions for the **angle** controller in `StateAndReward.java`. Test it by flying around manually and watch print-outs of the state to make sure all the important states are covered.
4. Implement the action representation in the method `performAction(int action)` in `QLearningController.java`. Test it by temporarily setting it to perform some action in `tick()`, the main decision loop.
5. Implement the Q-learning algorithm in `QLearningController.java` according to the instructions above. Run it for a sufficient number of iterations until it stabilizes, <100k should be enough for most representations with the angle controller. Remember to turn off the exploration while evaluating it (see instructions for the simulator above). If it calms down and faces upwards it has converged to a good policy, otherwise let it continue for a while more or revise your choices above.
6. Experiment with turning off the controller on 'P' and manually maneuvering it into bad situations before turning the controller on again on 'O'. Make sure that it seems to recover eventually.
7. Question 1 (theory): In the report, a) describe your choices of state and reward functions, and b) describe **in your own words** the purpose of the different components in the Q-learning update that you implemented. In particular, what are the Q-values?
8. Question 2: Try turning off exploration from the start before learning. What tends to happen? Explain why this happens in your report.

Part III: The Full Q-Learning Hover Controller

Now that you have implemented Q-Learning and a simple state and reward function for the angle controller above, you can extend this to full hover control by simply implementing state and reward functions that correspond to the task of hovering. The state and reward function calls in the `tick()` function need to be changed to point to these new hover versions.

Since the rocket is supposed to learn how to hover, it is important that both the state space and reward functions are designed appropriately. Complicating matters is that the underlying control problem is continuous and we are discretizing it to work with the simple table based version of Q-learning. The size of the state-space and therefore the Q-table therefore increases exponentially with the number of variables we include, and the granularity of the discretization has a large impact. Calculate the size of the state space times the number of possible actions (which gives the size of the Q-function) and check that it is not too large. If it is more than a few thousand states the Q-learning algorithm can take a long time to converge on the lab computers (>10min).

There are certainly tradeoffs involved in the state space design but some methods have previously been used sucessfully. For example, the angle seems to be the most important sensor variable and as many discrete values as possible should be assigned to it during the discretization. The horizontal velocity require fewer values and the vertical velocity will only need a small number of discrete values.

Try also to make the reward function as simple as possible. A good idea can be to simply come up with separate rewards for each state variable and add them together. It is also strongly recommended that you make sure rewards are all positive to make debugging easier. Many students have tried to write complicated reward functions and their own discretization functions and most of them had to redo everything from scratch because of the debugging mess that followed when something did not work properly. Keep it as simple as possible and you will be alright.

A sufficiently good reward function can be written in one or a few lines of code and a state space that is completely defined by the uniform discretization methods is also sufficent for the purpose of this lab. You do not need rock solid hover behavior to pass this lab, a bit of drift is perfectly fine. It is possible to get very nice hover behavior with simple state and reward functions, but you may need more states and therefore more iterations (upwards a million) of learning to reach this. A good idea is to start small and then when the controller is doing okay (at least outperforming the simple angle controller) you can attempt to add more values to the discretization. Also remember the Q-Learning tips from the previous section. If you are feeling adventurous the learning constants or other aspects of the implementation may be tuned to better suit your solution, but this can be difficult and time consuming.

This simple table based Q-learning converges rather slow compared to (for example) methods that either uses function approximation (supervised learning) on the Q-function or learns an internal model of the environment. If you want to see how fast your implementation converges over time you can use the class `TestPairs` to keep track of the cumulative reward. The following code example shows how the `TestPairs` class can be used:

```
// Import IO
import java.io.*;

// Member variables in QLearningController
TestPairs pairs = new TestPairs();
double sumReward = 0.0;
int nrTicks = 0;
int nrWrites = 0;


public void writeToFile(String filename, String content) {
  try {
    FileOutputStream fos = new FileOutputStream(filename);
    fos.write(content.getBytes());
  } catch (Exception e) {
    e.printStackTrace();
  }
}

...

// Inside the tick() method
int nrTicksBeforeStat = 10000; // An example
if (nrTicks >= nrTicksBeforeStat) {
  TestPair p = new TestPair(nrTicksBeforeStat * nrWrites, (sumReward / nrTicksBeforeStat));
  pairs.addPair(p);
  try {
    writeToFile("output.m", pairs.getMatlabString("steps", "result"));
  } catch (Exception e) {
    e.printStackTrace();
  }
  sumReward = currentReward;
  nrTicks = 0;
  nrWrites++;
} else {
  nrTicks++;
  sumReward += currentReward;
}
```

Every 10000th tick, the cumulative rewards are written as a Matlab file "output.m". The file can be inspected in its raw form or can be plotted in Matlab with the commands:

```
output;
plot(steps, result);
```

Ask your lab assistant if you have any questions about how to start and use Matlab.

If you run into problems (and you probably will) and wonder whether the bug is in the reward function, state space definition or implementation of the Q-Learning algorithm, switch back to the state and reward functions for the angle controller which should make the Q-learning algorithm converge quickly (use the Matlab file printouts to check the improvement).

Tasks for part III:

1. Implement the state and reward functions for the hover controller in `StateAndReward.java` and point the corresponding function calls in the `tick()` method in `QLearningController.java` to your new functions. If you cannot get behavior that is even close to hovering after 500k iterations you need to either revise your state and reward choices according to the instructions above, or there may be a bug in your Q-learning implementation. A good idea can be to turn off the controller on 'P' and maneuver it off-target and then turn it on again to see how it recovers like you did with the angle controller in part II. Do not forget to turn off exploration on 'E' when evaluating it!
2. Demonstrate your solution to the lab assistant As it can be difficult and time consuming to get perfect hover behavior we are fairly lenient as long as it can be seen that it is at least trying.
3. Hand in a report describing your solutions to part II and III, including Questions 1-2 from Part II as well as 3 and 4 below.

4. Question 3 (theory): The true state variables in this problem, s = (angle, x,y,vx,vy), are actually real numbers. For simplicity, we used table-based Q-learning in this lab, where you manually discretized the true state variables into a discrete state represented by a string s, used to learn the Q-table Q[s,a]. As you may have found, it can be difficult to find a discretization that gives good results.
   As mentioned in the RL lecture, *Fitted* Q-iteration is a technique where instead of using a table, supervised learning (SL) is used to approximate a continuous Q(s,a) function inside the Q-updates. Remember, supervised learning approximates any unknown function f(x) = y given examples of x and y. Here x could be (s,a) and y the new Q-values from the Q-learning update. This way you can use the true (real number) state variables directly, and do not need to manually find a discrete representation. *(Clarified on 2018-10-16)*

   a) If you were to apply such Fitted Q-iteration to the rocket angle control problem above, would a linear model be sufficiently expressive to approximate the Q-table of the angle controller? For example, even ignoring action and just using the real number angle *a* between -pi..+pi, could *a**w1 + w0, for some values of the linear parameters w0 and w1, be a good fit for the Q-values of the angle controller? (Hint: You could pick some example angles and plot them against your reward function, on this particular problem you can assume that higher-reward states also have higher Q-values).

   b) Otherwise, suggest some sufficiently powerful SL model/representation that could be used to learn a Q-function for this angle controller. (Hint: The ML lecture on supervised learning might be useful to refresh your memory on different types of models).

5. Question 4 (theory): For this question, assume you had plenty of computational power rather than a weak lab computer. Like above you want to approximate Q(s,a), but instead of the rocket state s containing a known real number angle, let it contain the image (about 2 million raw pixel values) from the rocket simulator graphics that are shown on the screen. We clearly cannot make a Q-table for all possible combinations of such pixel values, and even using a supervised learning approximation like above, many standard models struggle with such large inputs. What specialized model or architecture could we use to still efficiently learn from image inputs? (Hint: The ML lecture on deep learning might be useful). *(Clarified on 2018-10-16)*

Acknowledgements

This lab was developed by Per Nyblom and later modified by Olov Andersson.

Page responsible: **Patrick Doherty**
Last updated: 2018-10-16