Q-learning Project Documentation - C++
ELEC-A7150

Mikael Grön, Anssi Moisio, Visa Koski, Lassi Knuuttila

December 18, 2017

Contents

1 Overview

Main functionalities of this Q-learning implementation include:

- Support for a crawler creature that has an arm with two joints. The crawler's purpose is to crawl to the right in a 2D-world using its arm.
- Many crawlers learning at the same time, in different threads.
- A command line user interface from which user can pause learning and save the Q-table.
- Crawlers can share their knowledge with other crawlers to speed up the learning process.

We could not get the graphics to work. Problems arose with the GLUT library lacking compatibility with threading. ¡visa lassi explain problems some more¿

A plan-B was to visualize the Qtable with Cairo graphics library which we configured, but didn't have time to implement the visualization.

As a substitute for graphics, the command line interface prints out the crawlers' locations and speeds at regular intervals. It can be observed from the prints that a crawler learns a maneuver by which it acquires some top speed. After acquiring a top speed, the movement is stable (velocity is constant) if the exploration factor is set to a low value, so that the crawler does not make random actions.

¡mikael explain the evolution and config from file¿

2 Software structure

The main divide of the program is: learning and simulation. The simulation and the learning parts communicate with each other mainly by the learning part telling the simulation to do an action. The simulation will then simulate the action, returning the new simulated change of the agent to the learning part. The learning part evaluates the simulated change, determining what the reward for the performed action was, updating the Q-value.

In other words, the simulation part contains the world and the body of the agent. The learning part contains the "brain" of the agent.

2.1 Q-learning

The learning process takes place in the AgentLearner class. An object of the AgentLearner class uses a Qtable object to store and access the Q-table. A Qtable object has a two-dimensional map structure for storing the Q-values of each state-action pair. A map structure is used instead of, for example, a vector structure which was the original plan, so the size of the Q-table is possible to scale up without the access time increasing.

The Qtable class is supports dynamic creation and an object is initialized from a vector of actors and a vector of sensors. Initializer needs the number each actor's possible actions and the number of each state-detecting sensor's possible states. Qtable is a map of state-maps that contain actions for that state. The size of the Qtable will be states * actions. If an agent has, for example, three actors with 3, 2 and 2 possible actions, for each state there is 3 * 2 * 2 = 12 possible actions. If three state-detecting sensors can detect each 10 different states, the total number of states is 10*10*10 = 1000. This would make a Qtable of a size 1000 * 12.

The Qtable class has functions for finding the best action and finding the optimal Q-value for a state of the crawler. These are used in the learning process to update the Q-values.

The AgentLearner needs to convert the state of the learning crawler to a key for the Qtable object. The states are quantisized from the continuous states that the crawler can have in the simulation. StateKeys are in the format int stateKey = 141315, where each sensor's state is represented by two digits (14, 13, 15). Sensors' states are in the range 1...quantizationSteps. ActionKeys are in the format int actionKey = 30302 where each sensor's state is represented by two digits (3, 3, 2). Actors' moves are enumerations: 0 still, 1 counterclockwise and 2 clockwise and 1 is added to them so the key cannot be 000000.

3 Instructions for building and using the program

Cmake creates the Makefiles, and make uses the Makefiles to build the project.

- Install required packages if they're not installed already **sudo apt install cmake libcairo2-dev**
- Move to the build folder in the q-learning-9 folder. **cd q-learning-9/build/**
- Create Makefile. **cmake ..**

Build targets can be listed, built together, built separately and removed:

- List targets that can be built. **make help**
- Compile everything. **make** or **make all**
- Compile main. **make main**
- Compile tests. **make qtests**
- Remove compiled targets. **make clean**
- The executable compiled targets will be in the build folder as: **main** and **qtests**

4 Testing

Source files for testing are in the 'test' folder. Unit tests are implemented for the classes that pertain to the learning algorithm. more content

5 Worklog

Nov 6th to 12th

Anssi: Configured the programming environment, i.e., linux virtual machine. Planned the learning classes; what functions they will need and what member data they will use. Planned the distribution of roles between Mikael and myself, as we are the group members that implement the learning algorithm. 15 hours

Nov 13th to 19th

Anssi: Continued planning the project, made the UML picture for the plan. 3 hours

Nov 20th to 26th

Anssi: Planned the Q-learning implementation with Mikael and started coding. My responsibilities are the AgentLearning and Qtable classes. 10 hours

Nov 27th to Dec 3rd

Anssi: Implemented functions in the Qtable class and in the Agent class and re-named the Agent class as AgentLearner. Made tests for these functions. Changed Q-table data structure from vectors to maps. 30 hours

Dec 4th to 10th

Anssi: Implemented functions in the Qtable class and in the Agent class, for converting actions and states to keys for the Q-table map and choosing the Action. Also for saving and loading Q-table from a file. 25 hours

Dec 11th to 17th

Anssi: Finalized AgentLearner class and Qtable class and implemented the Simulation class with Visa. Tried to visualize the Q-table with Cairo graphics library, but didn't have time to finish. Debugged and optimized the learning process. 30 hours

Dec 18th

Anssi: Wrote the documentation. x hours