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

Mikael Grön, Anssi Moisio, Visa Koski, Lassi Knuuttila

December 18, 2017

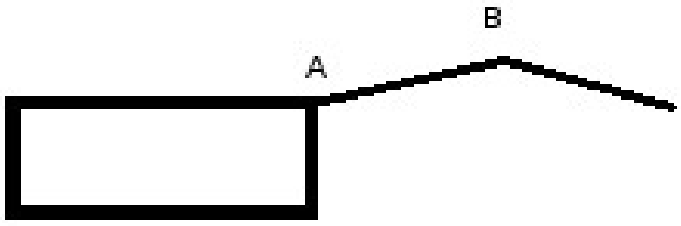
Contents

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.

The crawler looks like this:



We could not get the graphics to work. Problems arose with the GLUT library lacking compatibility with threading.

TODO: visa lassi explain problems some more

A plan-B was to visualize the Qtable with Cairo graphics library which we configured, but we 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.

The original plan was to make the programm support many kinds of learning creatures. This is the case for some of the classes (Qtable, AgentManager) but not all. AgentLearner class has support for crawlers with one or two joints and the Simulation class supports only crawlers with two joints. The finished program is thus compatible with a crawler that has two joints, but compatibility for different kinds of crawlers could be implemented with relatively small changes, which we did not have time for.

1.1 Configuration-file

The support for many kinds of learning Agents is visible also in the configuration-file, where the plan was to have the option to change what Agent you run, without changing source-code. Because of obstacles and not having time, the only

things you can modify are the number of Agents per run, and initializing the Agent(s) QTable from a saved file.

1.2 Evolution

When running more than one Agent evolution is used. This means that the Q-Learning algorithms variables differ some from the first Agents, which has always the same variables. All Agents in a run form the current generation. When one of the Agents reaches to a certain distance the goal, then this Agent is the fittest. All other Agents will copy the fittest Agents QTable into theirs, and every Agent teleports to the starting position. Then the next generation tries to reach the goal. And so on.

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.

TODO: uml picture here

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 actual learning algorithm is in the updateQtable function in the AgentLearner class. The algorithm is:

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha[r_t + \gamma \cdot max_a Q(s_{t+1}, a)]$$

Where s_t is the current state, a_t the current action, α is the learning rate, r_t is the reward from doing the action, γ is the discount factor, $max_a Q(s_{t+1}, a)$ is the next states highest possible Q-value. The reward is calculated from the distance travelled as a consequence of the action.

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 the key so the key cannot be 000000.

2.2 Simulation

TODO: Lassi and Visa: more content

2.3 Thread management

The AgentManager-class controls the treads and their running trough atomic-booleans and mutexes. Every threads task is the AgentTask-function in agent_manager.cpp. AgentManager listens on the users and AgentTasks inputs.

2.4 User Interface

Because there was problems with our graphics library, we do not have a GUI. The program is started from the commandline, where controls listed to control the program when it is running.

The Agent(s) location in the simulation is visible on the commandline as:
"65k: Agent_0 V=127 X=6418.52"
Meaning:

65k	65 thousand actions executed
Agent_0	Line conserns Agent number 0
V=127	Agents velocity in simulation units per 5k actions
X=6418.52	Agents location in simulation units

3 Instructions for using the program

3.0.1 Build

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**
- 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**

3.0.2 Running

The program is run from the built file: build/main. The program is optionally configured from the file: src/configAgent.config. In the config-file you can change how many Agents are running and an optional file for a saved qtable-file.

4 Testing

Testing is done with unittest trough googletest. The test are built as the "qtests"-buildtarget in cmake. The built test are run from file: build/qtests. Valgrind has been used from time to time, but it is not automated.

5 Worklog

5.1 Mikael

My main area I worked on was the planning, building, AgentManager, Interactor and the configuration-file. I helped also partly with nearly all other parts. A detailed log of what I worked on can be found from the GitLab-commit-history, and the time spent was not recorded.

5.2 Others

5.2.1 Week: 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

Lassi: Set up the virtual machine. Studied how Box2D works 5 hours

5.2.2 Week: Nov 13th to 19th

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

Lassi: Continued watching educational videos on Box2D. Created the Simulation part of the UML and project plan with Visa. 10 hours

5.2.3 Week: 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

Lassi: Started implementing Box2D. Busy week from outside the course. 5 hours

5.2.4 Week: 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

Lassi: After a lot of trial and error, Visa and I got the simulation to a point where one could control the crawler with keyboard. We used a testbed from Box2D and decided to keep it so that the Q-learner would control the testbed. 25 hours.

5.2.5 Week: 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

Lassi: Tried to combine the Box2D testbed and the Q-learner with no success 20 hours

5.2.6 Week: 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. 26 hours

Lassi: The week was very eventful and I didn't have that much time for the project. Stripped the testbed off of all OpenGL-components. After that we decided not to use the testbed at all. 3 hours

5.2.7 Week: Dec 18th

Anssi: Wrote the documentation. 6 hours

Lassi: Created the simulation part of the documentation 2 hours: