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

Mikael Grön, Anssi Moisio, Visa Koski, Lassi Knuuttila

November 26, 2017

Contents

1 Features / Scope

We are aiming to make an advanced program. Therefore our program will fill the assignments minimal and optional requirements.

1. Minimal requirements
    - Basic Q-Learning implementation
    - Simple graphics to visualise learning process
    - A system where bots will eventually learn how to perform simple tasks (such as moving forward) by themselves
2. Optional requirements:
    - Application can save learning data to some external file so user doesn't lose progress between sessions (suggested)
    - Ability to fast forward learning steps
    - Bots can share their knowledge with other bots to speed up learning process

To accomplish the optional requirements, our program needs to be able to run at different speeds, so that "fast forward learning steps" is possible. This means that the simulation of the agent can have different speeds.

To have multiple agents that share knowledge, we need a framework in which concurrent computation is possible. **Does this mean threads or processes?**

The Q-learning algorithm never finishes, so it needs to be able to be terminated by the user, therefore threading is needed for a responsive experience.

2 Tools

Plan	Latex
Class Structure	UML
Documentation	Doxygen
Compiler	g++
Compilation manager	cmake and make
Version control	git
Graphics and physics	Box2D
Testing	Google Test

3 Design and Practice

- Regular physical weekly meetings and possibly some web calls, in which every developer lists his progress since the last meeting and states what the next development goals and its blockers are.
- Commit only working code. Meaning that all the tests pass.

- Write code and its tests in tandem, or write test before the actual code.
- Comment properly your code. Especially important because of Doxygen.
- Everything written is in English: comments, names, etc.
- If someone is fast finishing his part, then he can help elsewhere.
- We try to first make a working version which then can be further developed.
- Limit line length to 80 characters.
- Tabs are 4 spaces.

4 Preliminary Schedule

We will have our internal deadlines about 1 to 1/2 weeks before the courses deadlines, so we have time to iron out possible problems that were not seen beforehand.

When bigger obstacles arise, the deadlines are adjusted case by case.

5 Q-learning

The core Q-learning algorithm is quite simple, so its implementation should be quite straightforward. The complexity of the learnable task can raise problems, so developing the algorithm will also mean the defining of the task the Q-learning will learn.

Running the algorithm means having it test different solutions, and finally maximising the reward, when the agent has learned. The learning means that the agent runs multiple times over all the possible states, with all possible actions.

The actor is the thing that is in states from which the actor can do some actions that move the actor into a new state. After every action the system gives the actor a reward. (A punishment is a negative reward.) The optimal behaviour is determined by how the system gives the actor the biggest reward.

5.1 Q-values

Every combination of states and actions has a Q-value. These Q-values are initialized all as 0. The optimal action in a state is the action with the highest Q-value. When the system is learning, an action is performed, and the Q-learning function updates the Q-value which is associated with the current state and performed action:

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

Here 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.

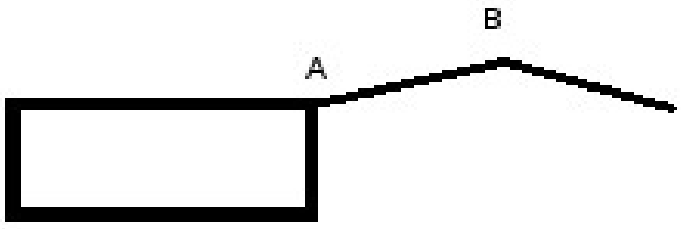
The Q-learning function makes the Q-values more accurate by testing out the combinations of states and actions in the system. The Q-learning is an optimisation method that will find an optimal path which gives the maximum reward. The best path is the global maximum, but there can exist local maxima. The local maxima can be avoided by introducing random actions in the learning process. If the Q-learning has found a local maximum, a random action will hopefully try out something that moves toward the global maximum.

Theoretically the exact Q-values are found by running the learning process for an infinitially long time. This is not practical (duh), so approximate Q-values are enough. The learning needs to be terminated from outside, when a satisfactory accuracy is achieved.

To get accurate Q-values, the different combinations of states and actions need to be tested multiple times during the learning process. The number of Q-values are the number of states times the number of actions in every state. Having a smaller amount of Q-values leads to a shorter learning time. More Q-values can lead to more complex behaviour for the learned actor. This depends on the system.

5.2 Modelling of Agents

We model agents as a combination of sensors and actors. The sensors determine which state the agent is in, and the actors can perform actions. We limit our systems to be two dimensional (2D), meaning the world and agents are 2D. We will implement our program to support other 2D systems, but we will start by studying a simple case of the agent in the picture:



We will model this agent that moves trough a flat 2D world. The agent gets a positive reward when it moves to the right, and a negative reward when it moves to the left.

The agent has a body and an arm. The arm consists of two beams and two joints (A and B). The angles of the joints determine the state of the actor, and actions are performed by rotating the beams, changing the joints' angles. Both joints can move 360° continuously, but we need to quantisize the angles, so we can have a limited amount of states. We can also be wise gods and limit the angle the joints can move. We can see that in our agent's case that the arm will

not affect the state if it flails in the air, not touching the ground. Also, we say that the beam between the joints cannot occupy the same space as the body. Therefore we can limit the permitted angles of the joints.

We limit the A joint to move 135°, and the B joint to move 270°. We quantisize the A joint's angle into 2^4 steps, and the B joint's angle into 2^5 steps. This gives the actor $2^4 \cdot 2^5 =$ **512 states**. The smallest angle registered at the joint, due to the quantization, is $135°/2^4 = 270°/2^5 \approx 8.4°$. The angle can be more precise when simulating the agent, but the Q-learning algorithm will just "see" the quantizised angle.

A joint can be moved by performing an action. We limit the movement to be just one speed, which is one quantization step. A joint can only move clockwise, counter-clockwise or not move at all. Multiple joints raises the possible actions to be 3 to the power of the number of joints. Our agent has two joints, therefore there is $3^2 =$ **9 actions** to perform in a state.

The number of Q-values is states times actions. Therefore there is $512 \cdot 9 =$ **4608 Q-values**. This should be a manageable amount to use the Q-learning algorithm on. If a Q-value is a float, which is 4 bytes, then the Q-values will occupy 18kB of memory. Thus the memory size would be tolerable, even when having multiple agents learning at the same time.

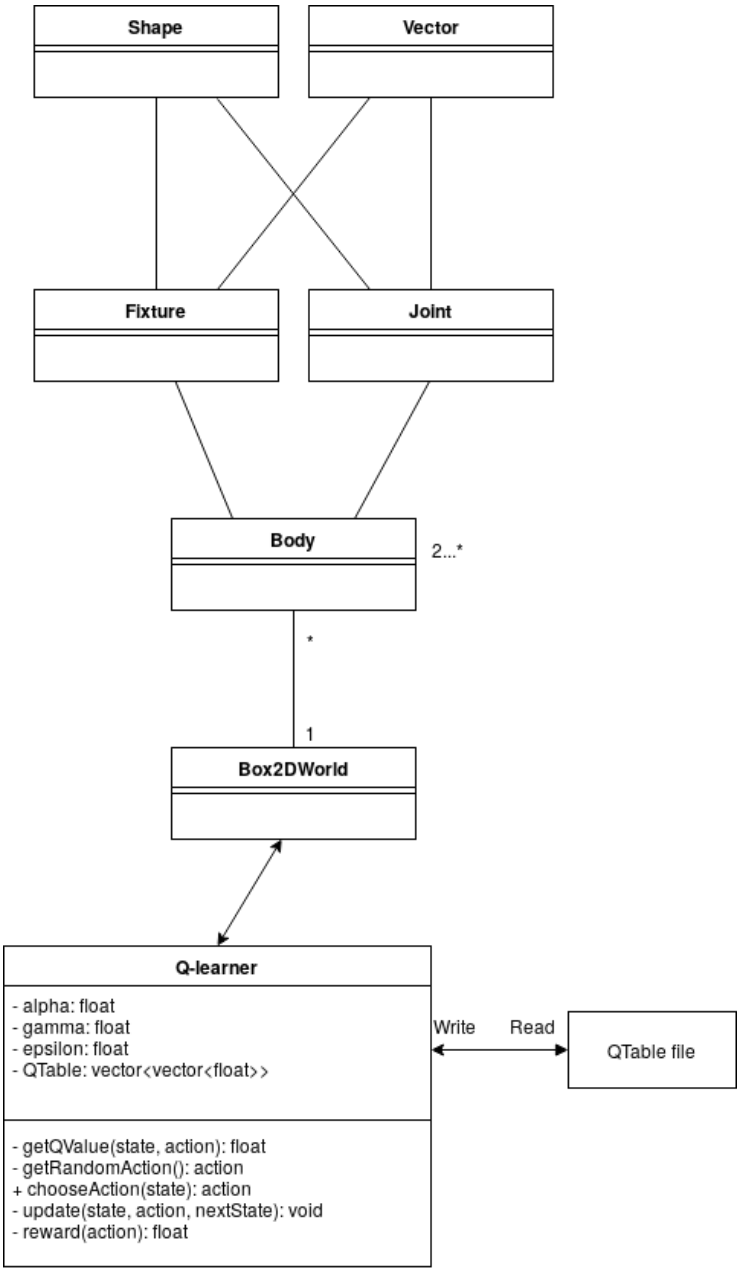
The analysis of the number of Q-values could be too high or low, but that can be adjusted when trying to run the algorithm.

6 Class 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.

Most of the programs classes are dynamically created, aka. template classes.



6.1 Simulation

The simulation is divided into the graphics and the physics. Box2d provides only physics simulation, so we have to write our own visual representation of world and agents with a graphics engine like SFML. Box2d uses float numbers in

simulation, and that may cause minor inaccuracies in the world. Main features of Box2d that are going to be used in project are: bodies, joints, fixtures, gravity and friction.

Body consists of shapes that are connected to each other with joints and/or fixtures.

Joints are rotatable and provide core for agent to move in world. If a joint is forced to turn where it can not turn, it will not turn.

Fixtures are used to make a fixed connections between the shapes.

Gravity and friction are used to enable the agent's moving abilities.

6.2 Agent Simulation Initialization

Loads agent parameters from file and sets the learning and simulation parts.

6.3 Q-learning

The Q-learner class inherits the Agent class (or otherwise gets the information what kinds of states and actions are applicable and what the sensors can detect) and initializes itself accordingly. It will initialize its Q-table from a file or as zeros. The agents may have different actors and sensors, therefore all data structures and classes need to support dynamic creation.

The Qlearner class will have attributes alpha, which is the learning rate, and gamma, the discount factor. The discount factor will determine the trade-off between rewards of distant and near future and the learning rate determines how much of an effect each reward has on the corresponding q-value, i.e. what is the trade-off between newly learned values and values learned in the past.

The basic functionalities are choosing an action based on the current state and the Q-table, getting a reward based on action, and updating the Q-table based on the reward. These functionalities will each consist of multiple functions and smaller classes.

To choose an action will be a major part of the Q-learning implementation. An example of the involved functions could be:

- getQValue(state, action) function that returns the Q-value of a state-action pair from the Q-table.
- maxValue(state) returns the maximum Q-value of the actions of this state
- bestAction(state) compares actions (calls maxValue which in turn calls getQValue) and returns the best for a given state
- getRandomAction() returns a random action
- chooseAction(state) will decide which action is chosen. It will call bestAction(state) and/or getRandomAction() depending on epsilon. It will communicate the action to the simulation and get the information back about what happened in the simulation when the action was performed.

Choosing the action may be partly random. The value of epsilon will determine to what degree the action is random; if epsilon is zero, the chosen action is always the action with the maximum Q-value. Simple way to use epsilon would be to set it to be some positive constant, e.g. 0.1. A more advanced use of epsilon is to have it approach zero as the learning process converges.

Learning consists, in the abstract, of getting rewards and updating the Q-table according to rewards. Its basic functions should include:

- getReward(action) will evaluate an action based on what happened in the simulation when the action was performed. The evaluation will be based on predetermined values; what the agent should do in the world, what are her highest goals and what is the meaning of her existence, e.g. to move right.

- updateQValue(state, action, nextState) will call getReward and update the Q-value of the given action using the parameters α and γ, or learning rate and discount factor, respectively.

Some of these functions might be better divided to multiple functions.

There can be also some smaller classes in addition to the Q-learner class, if needed. The Q-learner class can be an abstract class that has children for different kinds of agents.

There should also be functions to manage the Q-table data in an external file, so that the Q-table is saved when the program is closed: saveQTable, loadQTable, and to show the Q-table: printQTable.

The data structure for the Q-table that contains all state-action pairs and the Q-values has a few possible alternative implementations. It could be a two-dimensional matrix, for example a vector of state-vectors that contain q-values for each action. An other possibility is a vector of state-action pair objects that contain the Q-value as an attribute.

7 Distribution of Roles

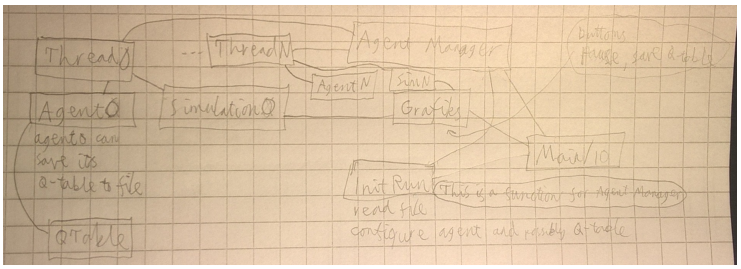
Mikael and Anssi will implement the learning part. Visa and Lassi will implement the simulation part.

The Q-learning implementation could be divided (between Mikael and Anssi) for example to (A) functions that communicate with the simulation (getReward), update the Q-table (updateQValue, updateQTable) and manage the Q-table data (saveQTable, loadQTable, printQTable etc.) and (B) functions that choose the action based on the Q-table.

The simulation can be divided to three main parts: The Agent, physics and graphics. Lassi and Visa will either do these parts together or separately.

8 Modification / Addition to plan

Things Anssi and Mikael came up with 25.11.2017.



The map above is pseudo UML. It shows the structure of the learning part and the interface to the simulation and graphics.

Main controls IO and gives control of learning and simulation to AgentManager. AgentManager initializes the run from configure files, that tell what the Agent "looks" like in the learning and simulation. Here a saved Q-table can be used.

The AgentManager creates a thread per agent. In every thread there is a agent and its simulation. The agent has a Q-table. The simulation and main/IO has controll of the graphics.

8.1 Communication Between Learning and Simulation

In a thread the learnig process looks like this: The simulation waits, while the agent chooses an action. The choosen action is sent to the simulation, and the agent waits. The simulation resives the action and the simulation simulates it. The simulation will have a response from the action. The simulation sends the response to the agent, where after the simulation waits. The agent resives the response and from it calculates the revard and the new state, and updates the Q-table according to the Q-learning algorithm. And so on.

The communication between the learning and the simulation is this:

- Action is a vector of pairs: (actor-ID, actor-action)
- Response is a vector of pairs: (sensor-ID, sensor-input)

8.2 Multiple Agents Teaching Eachother

When running multiple agents, they will teach eachother by the fittest agent copying its Q-table to the other agents. The fittest agent is determined by having some goal in the simulation, like a finishline, that ends the current generation. The AgentManager controls the evolution of the agents, when copying the Q-table. When moving to the next generation the Q-learning functions parameters (learning-rate, etc.), can also be changed. The different agents will have somewhat differing parameters. E.g. different amounts of random-actions will result agents that will explore different things.

8.3 Contents of Configure Files

- Describes the agent and world (shape, actors, sensors) The actors and sensors quantization and limitations are needed.
- Q-learning-function-parameters (learning-rate, ...)
- Saved Q-table (optional)