C++ Course Project Documentation

qlearning2

Team Skele aihe

Esa Koskinen, Simo Muraja, Olli Kauppinen, Jussi Hirvonen

2016-12-21

Skele aihe Documentation

Table of Contents:

1. Overview

Skele aihe is a program that aims to teach cars to drive around a track as fast as possible without crashing into a wall by using neural networks and q-learning. In its current version, very little learning actually happens. Though the neural network has been implemented and functions without exceptions, the network's weights have not been adjusted in a way that would allow the cars to stay on track or accelerate properly. Users can, however, enjoy realistic top-down 2D car physics and drive around the track themselves if they so wish.

This document details the program's structure and logic, and gives insight into how the team has worked during the project. As is the case for many projects, there have been both high and low points, but we have pulled together and managed to reach many of our goals. Unfortunately, at this point in time, some goals remain yet to be reached, but we will no doubt be able to look back on this project as a valuable learning experience.

2. Architecture, software logic, and class relationships

We followed our initial plan on the design and architecture with only a couple of changes. When the program is launched, the Launcher starts a Controller which acts as the bridge between all the different modules of the software. The Controller is initialized with a large amount of constants defined in constants.txt, which are parsed with FileReader. Having all the constants in a separate file makes adjusting the training parameters and other relevant things much easier. When initialized, the Controller starts the simulation by creating a Car object, a Track object and a NeuralNetwork object.

After initialization, the Controller starts the GUI, which doubles as the main logic loop of the program. At fixed time intervals, the GUI makes a call to the Controller's stepForward() function, which advances everything in the program by one step. During each step, new sight line vectors and other data is gathered to be fed into the Neural Network as the input state, the network's values are calculated, and the output is used as a new Q-value. The Q-values are used to form a probability distribution and an action for the car is chosen using that distribution. The network is then adjusted through Q-learning, which is handled by the Learning class. Finally, the action is passed to the Car object, physics are applied and a new frame is drawn.

The NeuralNetwork class is fairly self-contained and can take care of for example building and connecting itself and calculating its output values based on current inputs. Actions for individual Nodes are practically all handled through the NeuralNetwork. Friend class declarations are used to make the nodes' weight adjustment easier when training the network. Controller only needs to create a new network and to call appropriate functions to get output values for Car's actions and have the network learn.

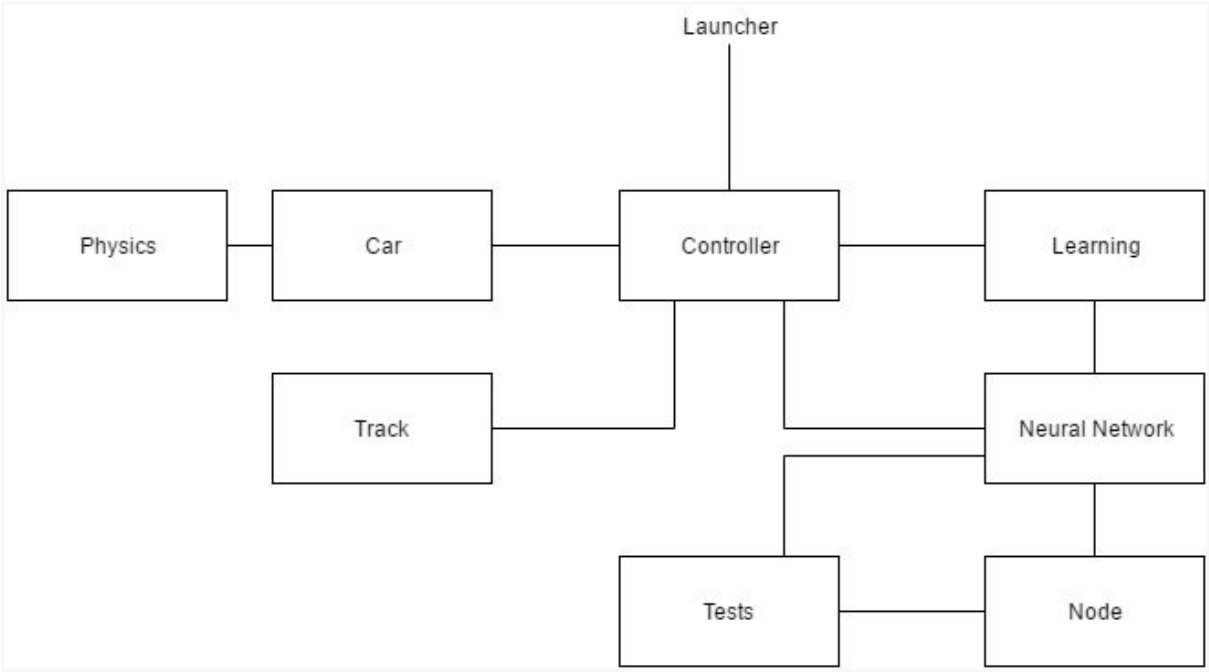
The relevant methods in the NeuralNetwork and Learning classes with regard to running the simulation and training the network are NeuralNetwork's getAction() and Learning's adjustNetwork(). When called, they fire off a sequence of function calls internally to determine the desired outcomes for the car and the network respectively. The getAction function approximates the rewards for various actions using the neural network. It then uses a Boltzmann probability distribution to weigh exploration of actions over exploitation of learnt patterns early on, gradually shifting towards the latter as the simulations run on. For the most part, the implementation is similar to the one described in Steve Dini and Mark Serrano's Q-learning experiment.[1]

The adjustNetwork function on the other hand uses a modified version of backpropagation and a simplified gradient descent algorithm to adjust the network. Rather than using typical backpropagation, this seemed simpler to implement while still providing a good amount of practicality. It turns out that unfortunately this method Essentially, the node weights aim towards providing such values that the long-term reward gained from different sets of inputs increases over time. This group of algorithms was explored in general-purpose use-cases by Cary Huang with moderate success.

[1] Steve Dini and Mark Serrano, Combining Q-Learning with Artificial Neural Networks in an Adaptive Light Seeking Robot. (2012)

Physics class holds the main functionality for car's behaviour in relation of the world. The Car class has actions for updating the car's position and rotation and asks the Physics class about its vision, friction and collision, while also providing getters for the Controller to use.

The Track class builds a track for the simulation by computing the position of track boundaries in the world and storing a means to visualize them into the object. The class also creates textures for the finish line and kerbs to be used in the render() function, which the Controller calls to draw the track in the GUI.



Dia 1: An updated class diagram

3. Using the software

Instructions for building and running the software

We have included a makefile in the root folder of the git repository. Running “make” in this folder builds a project file called “qlearn”. Running this file will start the project. We use the libraries Box2D and SFML in our project. Box2D is wholly included within the repository and the SFML headers are expected to be found in their standard path. This means that the project should compile fine on the Aalto Linux machines.

How to use the software: a basic user guide

The learning process is automatic and requires no user input after the program has started. There are two different modes that can be activated by the user. Pressing the D key toggles the simulation between learning and manual driving. When the manual driving mode is active, the user can control the car using the arrow keys and drive around the track freely. The F key controls the fast-forward mode. Fast-forward mode makes the simulation run 20x faster until disabled. The intended purpose for this is to speed up the learning; controlling the car manually in fast-forward mode is nigh impossible.

In usability, the current version of the software still falls short of what we envisioned in our project plan. We were planning on adding more features that were supposed to be accessible through the GUI. These would have included, for example, the ability to dynamically change the learning parameters. This would have allowed the user to affect the main simulation. Other shortcomings of the software are discussed in more detail below, in the Known Issues section.

4. Testing

Initially we planned to focus on integration tests since their errors should typically give us enough information to fix possible flaws quickly while covering large areas of the program, but for the most part we placed higher priority on implementing new features. Testing these features was mainly done by actually running the program and seeing if the new functions work as intended. We did add a few unit tests, which are discussed in more detail below.

Neural Network

As the Neural Network by nature is very compatible with unit tests, a small number of tests were written to ensure the Node and Network classes are working as intended. These tests include connecting nodes, setting and getting values, calculating values based on inputs with small networks and full scale network tests.

Running these tests actually revealed a couple of bugs and assisted in implementing additional functionality. For instance, initially the Network inadvertently created clones of Nodes instead of modifying the old ones. This caused the output of the Network to always be zero. To fix this, we switched to using pointers instead.

Learning

Learning was mainly tested by tweaking related constants by hand and running the program to see what happens, printing as much information to the console as possible.

In hindsight, had the learning been implemented earlier, it would've been wise to unit test commonly used functions and methods to make sure that they not only work, but are error and crash free.

Controller

The Controller object that holds the main logic of the program steadily grew in size as the project moved ahead. Testing it was mainly done through printing debug logs when running the program to pinpoint the location of the error.

Car, Physics and GUI

The car, its physics and the GUI underwent intensive testing that was carried out by constantly driving around the track in manual driving mode. We would always attempt to break any new functionality in all possible ways before approving them.

One thing that we discovered while testing was that the car could get stuck to the inner edge of curves if approached while hugging the edge of the track. This was fixed by adding kerbs to the curves that would allow the car to bypass the track limits slightly in places where it would previously not be able to move. We also found out that the car was able to get out of bounds by driving through the wall on the finish line. The boundaries had been added to every other part of the track except the finish line, which was then fixed.

5. Known issues

The Skele aihe software is far from perfect. The state of the program was, admittedly, pretty lackluster until just before the final demonstration. The shape of the program is fleshed out better now, but the very titular functionality of Q-learning was left unfinished due to time constraints. The primary reason for this might be that we took a difficult project and aimed for the stars without really knowing what we were undertaking.

Much of our time during the project was spent on fixing random and frequent crashes. We managed to completely fix these in the end (the program should be able to run indefinitely). Effectively a fourth of the project was used fixing the few tricky bugs causing these crashes, and they might have not been avoidable with our prior level of C++ understanding. Fixing them deeply cut into time that we would have wanted to use into building new features and making the learning effective.

As for the features, perhaps the first and foremost issue is in the usability. The program runs fine, but there is little actual interaction between the user and the software. Moreover, the existing interactive parts are really only related to debugging. The manual driving mode works wonderfully, and could have been easily expanded into its own game, but that was not the theme of the project.

If we had more time, we would have wanted to find a working set of parameters for the learning algorithm to use. As it stands, the cars are operated by a neural network which adjusts itself based on each run. From debugging it is mostly confirmed that the network and learning work as intended, and it would seem that the cars try to avoid certain actions in some situations after a few runs. However, it cannot be said that the cars actually learn to avoid the walls; any actions that correct course are often made too late.

In the current version, the learning always has to start from scratch. Another worthwhile feature would have been the ability to save the runs that have been made into a file, which lists the car's actions at each step. It should be easily possible to replay the runs afterwards from the file, since the physics are deterministic due to a fixed time step. This would have allowed us to utilize previous learning to start the simulation from an already developed stage.

There are some problems of more general nature as well, such as a small-ish amount of tests, questionable changes in working practices, et cetera, but none of these actually became an issue due to good communication. There are also some TODOs and issues remaining in the program that we simply did not have the time to go over. However, we constantly kept our list of priorities flexible and always worked on what we felt were the most pressing problems when considering the success of the project. It could be said that any downsides apart from the aforementioned are merely an indication of the issue being low in priority when compared to the near-catastrophes we've had along the project.

6. Developer responsibilities

As we largely stuck to our original plan of having a central Controller object act as an intermediary between all other parts of the code, it was easy to split the responsibilities of different branches between team members. Esa and Simo worked together on the pivotal part of the project: neural networks and learning. Olli delved into the Box2D physics and built a car that utilizes them. Jussi took up the SFML library and the track. Our interests seemed to align well with the initially assigned areas and we carried on like this for the whole project.

Since Simo was doing a lot of the early coding, he ended up implementing the majority of the NeuralNetwork and Node classes and thoroughly testing them. Esa, joining in later, took care of the components responsible for learning, and wrote the core logic for figuring out what the car should do each step. Of course, a lot of discussion was had between the two of them throughout the project. Simo later added a FileReader to simplify the initialization process.

Olli implemented Car and Physics classes and few physics related classes that basically overloaded some of the Box2D functionality. These were used for car's vision, counting checkpoints and checking for collisions. He also greatly assisted Jussi in creating the track by implementing the code that adds track parts to the physics engine.

Jussi came up with the solution for the track that would make it easy to gauge how far the car has travelled. He built the GUI and later the track utilizing the physics code written by Olli. Jussi created the graphical representation of the track and also added the problem-solving kerbs. He took on the task of creating a makefile for the project since he happened to be the first one in the team to try and compile the project using Aalto Linux machines. The final major addition that Jussi made to the project was the complete overhaul of Physics to make the car behave much more realistically.

7. Work log

On week one, we made it our goal to hold weekly meetings for the duration of the project. Ultimately, there was only a single week when we didn't see each other at all. Most of our team members were typically present whenever we met and the meetings were of varying length, depending on our personal schedules. At times we would chat and brainstorm for a maximum of two hours and every now and then we would spend an entire evening programming together.

On the first week we sat down two times to agree upon general working practices and software architecture and class structure. After deciding on what needs doing we divided the tasks in a distinct, easy-to-understand way (see Developer responsibilites) and for the following weeks all team members were mostly focused on developing their own branches.

It wasn't until the last week of the project when our team really came together again and kicked things into overdrive. Welding the different parts of our project together proved to be more than a day's worth of collective work. Looking back, we definitely underestimated how much time it would take to find suitable learning parameters. Our project could have been amazing if we only had succeeded in polishing the Q-learning component of the program.

What follows are individual comments for the different phases of the project and estimates for the weekly working hours.

Simo

Hours not including group meetings and planning.

Week 1 - 15 hours:

Sketching out and writing skeletons for Node and Neural Network classes, planning things with Esa. Studying machine learning theory.

Week 2 - 14 hours:

Writing most of Node and Neural Network classes. Studying more machine learning theory.

Week 3 - 10 hours:

Studying even more machine learning theory, adding functionality to Node, NN, Controller, wherever I was able to do work in.

Week 4 - 25 hours:

Studying machine learning theory, pulling hair. Writing unit tests, testing and fixing Node and NN, adding some functionality and helping a bit with learning. Writing documentation.

After the final demonstration:

Adding the FileReader, moving all constants to one location.

Olli

Week 1 - 11 hours:

Studying Box2D documentation and tutorials. Started to try things out in Box2D's testbed and began implementing basic functionality.

Week 2 - 13 hours:

Basic car functionality (eg. steering and acceleration) written. Physics regarding friction were very crude. Started working on vision. Basic code for track part physics ready.

Week 3 - 14 hours:

Implementing car's vision using ray casting and collision checking. Finalized code for track part physics.

Week 4 - 17 hours:

Fixing compilation errors and segmentation faults. Completely rewrote the code for collision checking. A lot of tweaking car's parameters to behave more realistically. Fixed physics-related bugs. Writing documentation.

After the final demonstration:

Adding FileReader to makefile.

Esa

Hours include group meetings and planning.

Week 1 - 18 hours:

During the first week most of my time was used on writing the plan, coming up with ways to implement Q-learning, studying background material related to the tasks I took and setting up some development environments for the project. I also spent some time managing some general things related to the project.

Week 2 - 12 hours:

My second week was mainly used implementing some backbone structure for the program and trying to get it to compile successfully. Turns out that I was unable to use the libraries both on my home computer and in Niksula… Research of the machine learning material continued, and we had a few discussions with the team.

Week 3 - 16 hours:

Mostly reading up on the theory and figuring out how the codebase got so messy all of a sudden. Started implementing the actual mechanisms using the neural networks and the way they learn.

Week 4 - 33 hours:

During the last week I've had several near-sleepless nights while enjoying putting things together so that they hopefully somewhat work. There were a few very tricky bugs that affected the whole action search logic and caused numerous crashes. Turns out that they were, for the most part, related to the inaccuracy and limited size of floats. I also made a lot of learning-related parameters easier to adjust through constants that were all in one place. Additionally, there were some preparations to be done related to the project's end and the demo.

Jussi

Week 1 - 2 hours:

Drew a draft of how the graphical user interface could look like.

Week 2 - 20 hours:

Added SFML and built a basic foundation for a graphical user interface. I also constructed a makefile for us, which was challenging but also a really good learning experience.

Week 3 - 19 hours:

Built the track which the car races on, applied a lot of trigonometry to figure out the graphical placement of track parts and to make curves seamless.

Week 4 - 10 hours:

A lot of errors were still present in the code at this point, due to some team members programming in an environment where they were unable to compile the project. Spent a lot of time fixing these problems and clarifying the code. Worked on car behaviour alongside Olli and added kerbs to prevent the car from getting stuck at certain parts of the track.

After the final demonstration:

Overhauled physics entirely, refactored GUI into the Controller (was previously mistakenly part of the Track class), fixed time steps, added toggles for manual driving and fast-forward modes.