Toni Pellinen, Kuura Parkkola, Miisa Martio, Nuutti Sten

Q learning - Project plan

Introduction

Our objective is to make a simple program that teaches a car how to drive using an artificial neural network taught by a q-learning algorithm. The first milestone is to reach the minimum requirements for the project. If we have excess time and energy, we will also implement at least some optional parts. Threading the workload could for example be a welcome addition. The idea behind the learning process is that every generation learns something from the previous ones. These may include something like how to adjust speed and when and how hard to turn to avoid collisions.

Distribution of roles in the group

Toni will be focusing on the graphics since he has some experience using QT-library before. He will also help with other parts where the help is needed.

Kuura takes responsibility for the simulation physics. As working with this part may be difficult before some work on the graphics is done, will he also help the other group members in the beginning.

Miisa will implement the car and focus on how it observes its surroundings. If the implementation of the car doesn’t cause any major problems, she will also participate in implementing the surroundings.

Nuutti will be working on the actual q-learning algorithm and the artificial neural network it needs to function.

Schedule

1st week:
Getting familiar with needed libraries and learning more about Q learning and neural networks. Writing some code but nothing needs to work at this point.

2nd week:
Midterm evaluation. Something should work at this point or at least we need to have good progress with the project. We’ll hopefully also have basic graphical environment for testing the car and the physics.

3rd week:
If we're diligent and lucky, our car should drive pretty well at this point. At least with manual control. In the third week all the components excluding the neural network should be functional. The neural network is probably the hardest component and its also difficult to test and debug if the system around it is not functioning properly.

4th week:
Hooray, project is ready! Few sleepless nights before the deadline. Car should be driving along the track without any problems. If there's any time left, it's possible to add additional features, for example performance improvements, for the project at this point.

Q and ANN

The steering algorithm of the simulated car will consist of a very simple artificial neural network and a Q -function that "teaches" it. As we have no particular experience of either of the two, we attempt to keep the inputs and layer structures as simple as possible. These could also be implemented with a genetic algorithm later but for now the main objective is to create a functioning Q-ANN combination that is also reasonably light to run.
There are a few steps along the way, and they might be taken in following order:

1. Study the basic theory of both algorithms
2. Design a preliminary structure for both and an interface for the inputs and control
3. First attempt to code it.
4. Try to come up with a way to test the code before everything else is quite ready. Might be difficult but at least it can be checked for major problems like memory leaks at this stage.
5. Try to fit everything with the car and physics etc. quite probably start all over again.
6. Test run the car until it works.

Simulation physics

For the simulated car to function somewhat realistically in its environment, a simple physics engine needs to be created. This engine needs to take care of stopping the car in case it hits an obstacle as well as model different forces that are relevant in a 2d environment. In the scope of this project structure wise, the physics engine is most likely going to be one of the main classes hosting the world and the car.

Some things the engine needs to do:

- Make the car slide in case traction is lost due to too high velocity in corners
- Make the car skid in case too much breaking is applied
- Make the car stop slowly in case the ‘gas pedal’ is released
- Handle crash physics when collisions with walls and possible other vehicles occur

The physics behind these tasks are not too complicated and can be calculated with some very basic formulas. However, implementing this to a code is most likely a task that will take a fair bit of time. In general these physics are light to run and will not need much optimizing, nor will they cause lag in the simulation.

The steps for creating the physics will be something in the lines of following:

1. Doing some research on the basic concepts of physics programming. Most likely based on some simple games as this simulation has a close resemblance to one.
2. Playing around, testing and learning to work with newly acquired information to create an intuition to using it.
3. Designing the functions for the different physics phenomena taken into account in the simulation.
4. Figuring out how time is modeled in the simulation. Basically this means determining the interval and structure of the runloop.
5. Actually programming a prototype of the physics engine and testing it in a simplified graphics environment that hopefully is already done by this time.
6. Tweaking and improving the engine as needed later on until it meets it’s requirements.

The physics class should hold at least the following functions:

- Physics.ApplyDrag
    - Function that multiplies the car velocity values by some multiplier (less than one) that might depend on the velocity
- Physics.CheckCollisions
    - Checks whether any corner of the car’s hitbox overlaps with the edges of the track
- Physics.CheckSkid
    - Returns true if deceleration is too hard
- Physics.CheckSlip
    - Returns true if velocity and steering joined exceed some given value
- Physics.SetVelocityVector
    - In case traction is lost, the direction of the car should not change. Only resistive forces apply. This function should take of that.

The car and its surroundings

The car is the key component in this simulation as it is the object the q-learning algorithm is used to control. The car will be implemented as a class inherited from the physics engine. The car class should include information at least of the following variables:

- The velocity vector parameters of the car
- The amount of acceleration, deceleration and steering (control values)
- Depending on implementation, possibly the location of the car
- Whether the car is at skid or not
- The hitbox of the car

The surroundings of the car will be defined as a matrix where the walls are marked as ones and everything else as zeroes. The size of this matrix will be defined later.

The car should also be able to observe its surroundings. This will be achieved by implementing a class which will be inherited from the car class. This class should include three or five “sensors” that will keep track of the surroundings in different directions. In other words, a “sensor” should track the nearest wall in its direction and the distance between this wall and the car.

The steps needed for implementing these classes are:

1. Learning about the basics
2. Planning how to implement the classes
3. Trying to implement the classes
4. Fixing the code when needed
5. Making changes and trying to improve the code

Graphics

The Simulation will have a simple graphical user interface. The car and the track are displayed using simple geometric shapes. GUI will also have a few buttons to give easier control over the simulation. The graphical user interface will be implemented using the QT graphics library. Being fairly simple to use, the QT library is an optimal tool for creating the visual setup in a fairly short time. This shouldn’t be one of the hardest modules to make in this simulation.