Self Driving Car Simulation

Kristian Žarn
Faculty of Computer and Information Science
University of Ljubljana

***Abstract*—In this project we present a simple simulation of a self driving car. We first prepared a simple environment in which a car can move, register collisions and detect obstacles. In the first part the car was taught how to drive from the user inputs using neural networks. In the second part the car learned how to drive by itself using genetic algorithm. Both methods turned out to be successful.**

1. Introduction

Neural networks and neuroevolution are powerful algorithms that are used for learning in many real life applications. In this project we wanted to explore these methods by simulating a self driving car. The goal is to make a few laps around the track without crashing.

2. Methods

2.1. Environment

The first step was to prepare a simple environment in which the car can move, register collisions and detect obstacles. For this we used a python programming language and Panda3D framework [1] for 3D rendering. Figure 1 shows a screen-shot of the environment.

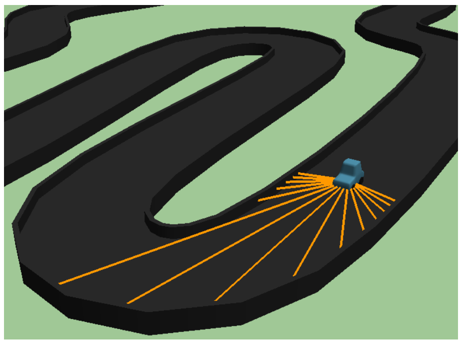
Figure 1. Our setup of the environment. Car has distance sensors in the front and has to make it around the track without crashing.

The car moves forward all the time at a constant speed. We can turn it left or right, but we cannot stop or slow it down. Because of this simple control scheme the state of the car is fully described by its distance sensors and left/right button presses.

2.2. Neural networks

Neural network is used as a decision model of our car. A simple example of a neural network is shown in figure 2. The whole network comprise of individual nodes called perceptrons. Perceptron can have multiple inputs and one output. Each input is assigned a weight and the task of the perceptron is to compute a weighted sum of its inputs and activate its output if the sum if bigger than some threshold.

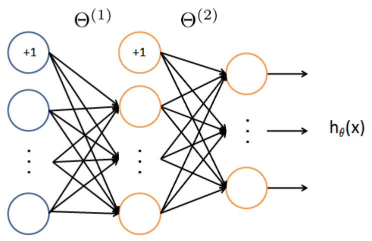

Figure 2. Instance of neural network with one hidden layer.

From this description we can see than the parameters of our model are represented by the weights of neuron connections. Our input layer is as big as the number of sensors and the output layer has two neurons (for left and right). The decision process can be described as follows. At every frame the car has to decide weather to turn left or right. We put the distances from the sensors on the input of the network and we make our decision based on which number of the two output neurons is bigger.

2.3. Back-propagation

Our first approach was to teach the car how to drive from the data. The data was collected by logging the inputs of a user that made a few laps around the track without crashing. The results are discussed in the next section.

Using this approach we have a labeled dataset (sensor distances labeled with left or right). We want to find parameters (weights) that minimize a cost function which is defined based on the difference between our predictions and the labels. Optimization of this function can be done using one of the non-linear optimization procedure (e.g. gradient descent). The main problem is computing the partial derivatives of the cost function that are needed by the optimization procedure. For this task we use an algorithm called back-propagation. The outline is given in algorithm 1.

Algorithm 1 Back-propagation

```
Require: Parameters $\Theta^l$; Training set $X$; Labels $Y$
Ensure: Gradients $\Delta^l$
1: Initialization $\Delta^l = 0$
2: for i = 1 to length($X$) do
3:     Feed-forward input $X(i)$ and obtain activations $a^l$ of neurons at layers $l$
4:     Compute difference on the output layer; $\delta^L := a^L - Y(i)$
5:     Compute differences on other layers, $\delta^{L-1}, \dots, \delta^2$ based on corresponding $a^l$ in $\Theta^l$
6:     Update partial derivatives; $\Delta^l := \Delta^l + \delta^{l+1} a^l$
7: end for
```

2.4. Neuroevolution

Our second approach was to let the car learn how to drive by itself. In this case we do not have the data from the user, so we cannot use back-propagation to find the parameters. Instead we use a procedure called neuroevolution that uses genetic algorithm to find the parameters of the network. For the algorithm to work we need to define a fitness function that tells us how good is the particular set of parameters. In our case we compute it from the time alive and the traveled distance.

Algorithm 2 Genetic algorithm

```
Require: Initial generation; size of generation N
Ensure: Generation of improved parameters
1: while True do
2:     Evaluate the fitness function for cars in generation
3:     Initialize new empty generation
4:     for i = 1 to N do
5:         Selection: pick two parents according to fitness
6:         Crossover: create child by combining parents
7:         Mutation: change some of the child parameters
8:         Add child to new generation
9:     end for
10:    Replace old generation with the new one
11: end while
```

The outline of such procedure is given in algorithm 2. We initialize our genetic algorithm with first generation of N cars with random parameters. Then we run the simulation and obtain their fitness values. Now we want to generate new generation of N cars from the old generation. To make a single child we sample two parents p_1 and p_2 (with fitness values f_1 and f_2) from the old generation (the cars with higher fitness are more likely to be picked). Crossover of parents gives us a child. Its parameters are obtained by taking some parameters from the first parent and some from the second (in the ratio of $f_1 : f_2$). To avoid getting stuck in local maximum we also mutate the child's parameters by randomly replacing, scaling and adding adding values to them. This is done to a very small percentage of parameters. At the end of this iteration we replace our old generation with the new one and repeat the procedure until we are satisfied with the performance.

3. Results

3.1. Back-propagation

In our first approach the car had 15 sensors and we used a neural network with two hidden layers with 15 and 10 neurons respectively. We learned from 1908 logged states. The testing was done on the second track where the data was not collected.

The performance of the car was surprisingly good. While collecting the data we tried to drive as close to the middle of the road as possible to avoid crashing. From this data the car has learned that it can get much closer to the edge as shown in figure 3. The car can finish both of our tracks without crashing.

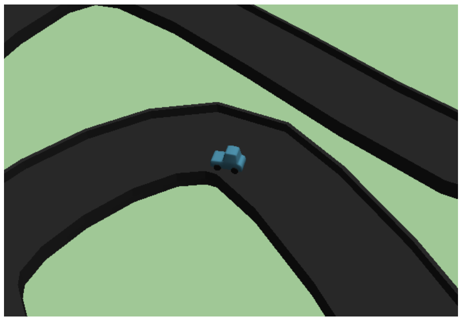

Figure 3. Self driving car trained with user input data using backpropagation. The car has learned to handle the corners very close the the edge.

3.2. Neuroevolution

In our second approach the car had 9 sensors and the neural network had two hidden layers both with 5 neurons. Generation size was set to 15. The learning was again successful. The car has learned how to get around the track in about 40 generations.

We tried to make the problem more difficult by introducing the enemy cars. We added 5 red cars controlled by our previously trained neural network that our car had to avoid as seen on figure 4. After about 85 generations our car has learned to do 3 laps around the track without crashing.

4. Discussion

Both of our approaches turn out to be successful in training a self driving car for our simple environment. The next step would be to make the state of the car and the environment more complex. For example, by introducing more accurate car physics and more complex traffic rules. Comparing the performance to other learning algorithms (e.g. reinforcement learning) would also be interesting.

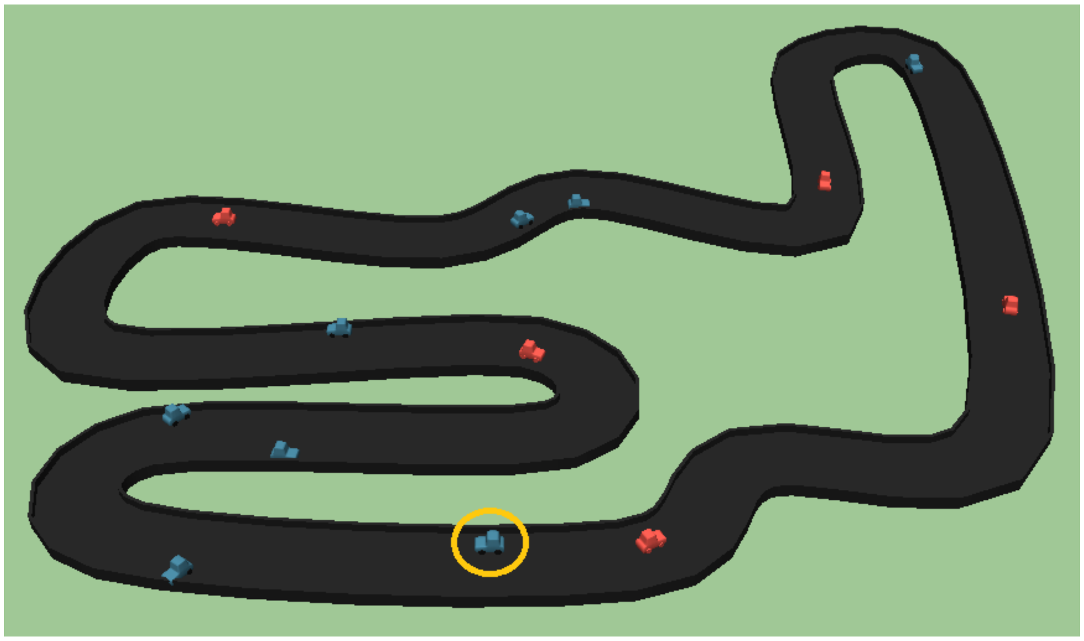

Figure 4. Self driving car trained with genetic algorithm. The car has to avoid crashing into red cars. The car marked with orange circle is the one that is still alive.

References

[1] M. Goslin and M. R. Mine, "The panda3d graphics engine." *IEEE Computer*, vol. 37, no. 10, pp. 112–114, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/computer/computer37.html#GoslinM04