December 2016

Driving a car with low dimensional input features

Jean-Claude Manoli — jcma@stanford.edu

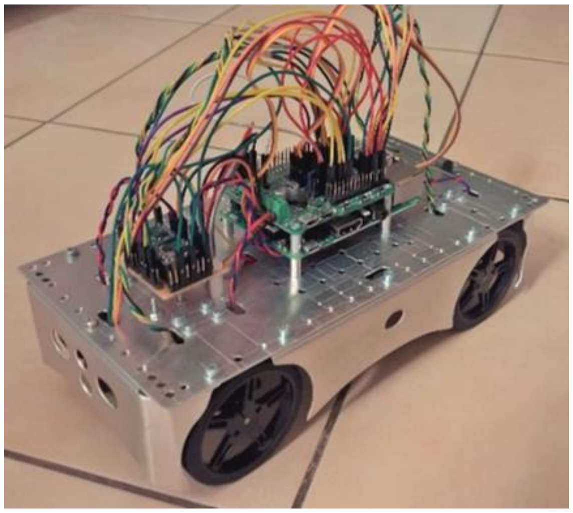
Abstract — The aim of this project is to train a Deep Q-Network to drive a car on a race track using only a few basic sensors. This problem is similar to a blind person learning to ride a bicycle from one place to another with only a white cane and a good sense of balance.

1 INTRODUCTION

The motivation for this project comes from a robotics competition organized by the Milliwatt Group in Lausanne, Switzerland. The 2017 edition is about building and programming scaled-down autonomous cars that will race against each other around a small circuit illustrated in Figure 1. Three cars will be racing simultaneously on the same circuit, which promises to be exciting given that the track has an intersection.

In this article, we focus on controlling the throttle and steering a single car around the circuit while keeping it within the track boundaries. Adding more cars on the circuit will be the object of a future project.

The car I have built has only a few basic sensors: wheel encoders provide the car odometer and speed, an orientation sensor provides the heading, and a sensor measures the track color. The car also has five infrared distance sensors for detecting other cars, which won't be used for now.



Optimally driving this car on the race track is challenging, even for a human that can see the circuit. The car's top speed is 2.5 m/s, the circuit size is 2 by 2 m and the track width is 45 cm. At maximum steering, the car will start skidding at about half the top speed. This problem is also challenging for a software agent, because the car cannot know its exact position on the circuit. The blue gradients on each side of the track only cover one third of its width, which delays lateral deviation detection. Without a good estimate of the car's lateral position, the track color sensor reading cannot be used for deciding which direction the car needs to be steered towards. Finally, steering and velocity changes take time, and require anticipating the turns.

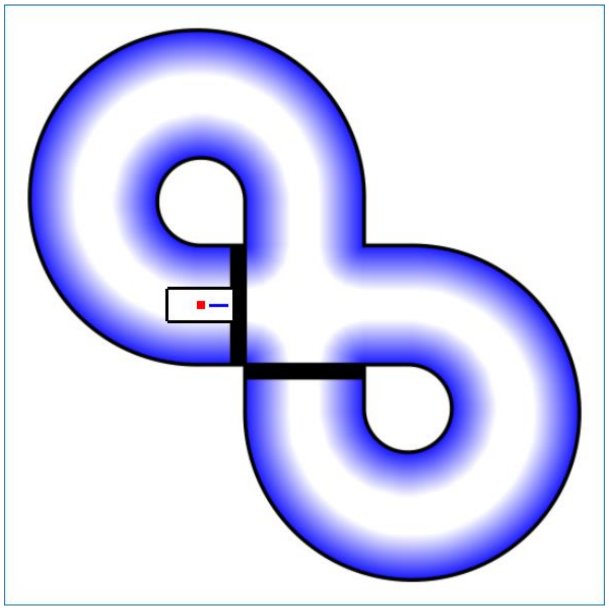


Figure 1: simulated environment rendering of the race track with the car at the lap threshold. The red dot shows the location of the floor color sensor, and the blue line represents the steering wheel position.

While there is plenty of research around autonomous driving, most this work assumes that the vehicle has high dimensional input features like video cameras or other sophisticated sensors.

With an original problem and no related work to refer to, we need to take some time to explain how we modeled it.

2 MODEL

This problem can be posed as a discrete-time partially observable Markov decision process (POMDP) with continuous state and discrete action spaces, characterized by the tuple $(S, A, T, R, \Omega, O, \gamma)$, where

- S is the state space,
- A the set of actions,
- $T: S \times A \to S$ is the deterministic state transition function,
- $R: S \times S \to \mathbb{R}$ is the reward function,
- Ω is the observation space,
- $O: S \times A \times S \to \Omega$ is a function that determines the observation from state transitions,
- $\gamma \in [0,1]$ is the discount factor.

2.1 State Space

The state space S is defined as:

$$s = \{x, y, \theta, \psi, p, v, c_b, x_m\} \in S$$

x, y	the Cartesian coordinates of the car with values in $\mathcal{C} = [-1,1] \times [-1,1]$.
θ	the car heading, with values in $[-\pi, \pi]$.
ψ	the position of the steering wheel in $[-1,1]$. It is proportional to the car's trajectory curvature $\kappa = \kappa_{max}\, \psi$, with $\kappa_{max} = 4.5$.
p	the car's throttle position in $[0,1]$.
v	the car's longitudinal speed in $[0, v_{max}]$ with $v_{max} = 2.5\ m\ s^{-1}$.
c	the color sensor RGB values.

The position of the car can also be expressed relative to the track median line with $\{x_m, y_m\} \in \mathcal{M}$, which we call the "median coordinates".

x_m	the normalized longitudinal car position along the track median line, with values in $[-1,1]$. On the lap threshold, $x_m = 0$.
y_m	the normalized lateral deviation from the track median line, with values in $[-1,1]$.

It would be tempting to just use median coordinates to track the car position in our state, but the non-linearity of this space does not make it practical for integrating the car's position over time. Median coordinates are more natural for controlling the car, so we will use them later.

Transforming median coordinates to Cartesian space is straightforward. The inverse transform however requires knowledge of where the car comes from when it is located on the circuit's intersection.

2.2 Observation Space

The observation space Ω contains values that can be derived from the car sensors:

$$o = \{d, c_b, \hat{\theta}, \psi, \hat{v}\} \in \Omega$$

d	represents the normalized distance driven relative to the lap and checkpoint lengths. Similar to the x_m median coordinate, d is *zero* on the lap threshold line, and increases proportionally to the distance driven as the car moves toward the checkpoint. On the checkpoint threshold line, d flips to -1, and increases proportionally to the distance driven as the car moves toward the lap threshold. Note that the values of d can be larger than 1.
c_b	the normalized difference between the blue and red channels of the RGB color sensor. Because the track colors are white, black and shades of blue, its range is $[0,1]$.
$\hat{\theta}$	the normalized car's heading in $[-1,1]$, proportional to θ plus some Gaussian noise.
ψ	the position of the steering wheel.
$\hat{v}$	the normalized longitudinal speed in $[0,1]$, proportional to v plus some Gaussian noise.

2.3 Action Space

The action space is

$$a = \{a_\psi, a_p\} \in \{-1,0,1\} \times \{-1,0,1\} = A$$

where a_ψ represents a ψ_{step} steering position increment and a_p represents a p_{step} throttle increment.

2.4 Reward

In the competition, the cars are awarded points for each completed lap. Such a sparse reward system would make reinforcement learning difficult, so instead, we reward the car at each time step based on its progress along the track:

$$R(s, s') = x'_m - x_m$$

When the car drives outside the track boundaries, the episode ends and the reward is set to:

$$R(s, s_{out}) = -\frac{v_{max}\Delta t}{1 - \gamma}$$

This constant value corresponds to the maximum discounted reward that can be accumulated by following the track median at maximum speed. Using this value is a good way to ensure that the penalty is not set too low nor too high, which would hurt the learning algorithm performance.

2.5 Belief Space

Deriving the correct action from only the current observation would not lead to good results. To make the problem tractable, the agent needs to maintain a belief of the car's position over time. The belief space uses median coordinates, and is defined as:

$$b = \{\hat{x}_m, \hat{y}_m, \hat{\theta}, \psi, \hat{v}\} \in B$$

Updating the prior belief b after taking action a and observing o is given by the transition function $\tau: B \times A \times \Omega \rightarrow B$. Because the car sensors are deemed accurate enough, $\hat{\theta}, \psi, \hat{v}$ can be taken directly from the observation, and the transition function only needs to derive the $\hat{x}_m, \hat{y}_m$ coordinates.

2.6 MDP Reformulation

With a reasonably good position estimate, we can now reformulate the problem as a regular MDP which should be much easier to solve. It is defined by the tuple (B, A, τ, r, γ) where

- B is the belief space,
- A the set of actions,
- $\tau: B \times A \times \Omega \rightarrow B$ is the belief state transition function,
- $r: B \times B \rightarrow \mathbb{R}$ is the reward function,
- $\gamma \in [0,1]$ is the original discount factor.

3 METHOD

To solve this problem, we will build an agent that can interact with the environment. The agent implements a policy that takes the current observation, outputs the next action to perform and then receives the resulting observation and a reward from the environment.

A simulated environment was built for this project, using the OpenAI Gym toolkit [1]. The high-level components involved are depicted in Figure 2, and described below.

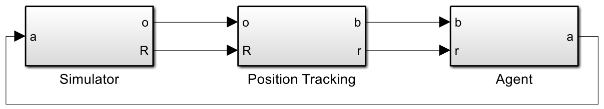


Figure 2: control loop components

3.1 Environment

The environment contains a simulator that models the car dynamics and the circuit characteristics. It uses a fixed time increment (Δt) set to achieve 60 steps per second, which corresponds to the lowest sampling rate of the car sensors.

At each time step, the environment takes an action from the agent and performs the following tasks:

1. If the action $\{a_\psi, a_p\}$ would make either ψ or p exceed their limits, the corresponding action component is set to 0.
2. Skidding was not modeled in the simulator, so the environment may override the throttle action a_p to ensure the car velocity stays below the safe speed based on the desired steering value $\psi + \psi_{step}\, a_\psi$.
3. The following variables are integrated to determine the new position:

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ \psi' \\ p' \\ v' \\ d' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \\ \psi \\ p \\ v \\ d \end{pmatrix} + \int_{t=0}^{\Delta t} \begin{pmatrix} v \cos \theta \\ v \sin \theta \\ v\, \kappa_{max}\, \psi \\ \psi_{step}\, a_\psi \\ p_{step}\, a_p \\ v_{max}\, p \\ v \end{pmatrix} dt$$

   We assume that the car velocity is a linear function of the throttle control.

4. c_b and x_m are updated based on the position $\{x, y\}$.
5. The observation $o = \{d, c_b, \hat{\theta}, \psi, \hat{v}\}$ is returned after adding some Gaussian noise to all the variables, except for the steering position ψ.

An episode runs until the car exits the track boundaries or until a fixed time limit is reached (10 seconds). To avoid wasting training time, we also added a rule to stop the simulation if the car makes a turn at the track crossing.

For each new episode, the car is set at a random position along the track median, with the orientation and steering matching the median line characteristics plus some small random offset. This improves the learning by providing a wide range of states at the beginning of the training.

3.2 Position Tracking

The belief state is maintained by tracking the estimated car position $\{\hat{x}, \hat{y}\}$ in Cartesian space. This is done by integrating the position from the previous belief:

$$\begin{pmatrix} \hat{x}' \\ \hat{y}' \\ \hat{\theta}' \\ \psi' \\ \hat{v}' \end{pmatrix} = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{\theta} \\ \psi \\ \hat{v} \end{pmatrix} + \int_{t=0}^{\Delta t} \begin{pmatrix} \hat{v} \cos \hat{\theta} \\ \hat{v} \sin \hat{\theta} \\ \hat{v}\, \kappa_{max}\, \psi \\ \Delta\psi \Delta t \\ \Delta v \Delta t \end{pmatrix} dt$$

with $\Delta\psi = \psi' - \psi$ and $\Delta v = \hat{v}' - \hat{v}$.

The belief state position $\{\hat{x}_m, \hat{y}_m\}$ is then determined by transforming the Cartesian coordinates into median space, using d from the observation to resolve the ambiguity at the crossing.

The position estimate will drift over time, but it gets corrected using the observation:

- When the floor sensor reads a blue gradient ($c_b <> 0$), we can use its value to correct the car's lateral position relative to the track:

  $$\hat{y}'_m = \operatorname{sign} \hat{y}_m \left(\frac{1}{2} w_a + c_b w_b\right)$$

  with w_a being the width of the white area on the track and assuming c_b is linear with respect to the blue gradient width w_b.
- Each time the car crosses the lap threshold, we can reset the $\hat{x}_m$ value to 0.
- Each time the car crosses the checkpoint threshold, we can reset the $\hat{x}_m$ value to -1.

With the simulated environment, the reward value is taken directly from the simulator. Otherwise it can be determined from the position estimates.

3.3 Baseline Agent

A simple reflex agent is used as a baseline for this project. It uses a PD controller that tries to minimizes $|\hat{y}_m|$ to keep the car within the track boundaries:

$$f(b) = \kappa + k_p y_m + k_d \dot{y}_m$$

$$a_\psi = \begin{cases} 1 & \text{if } f(b) < -\epsilon \\ -1 & \text{if } f(b) > \epsilon \\ 0 & \text{otherwise} \end{cases}$$

With $k_p, k_d \in \mathbb{R}$ and $\epsilon \in [0,1]$.

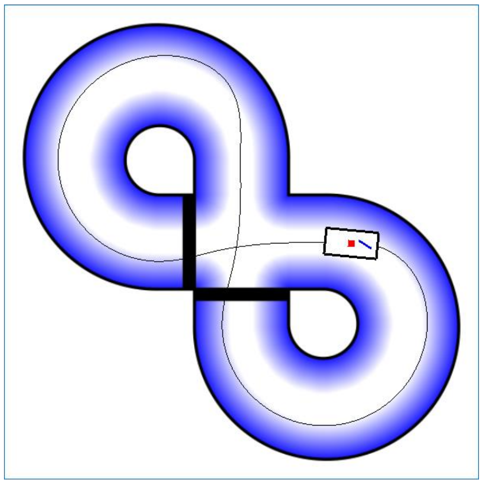


Figure 3: trajectory from the baseline agent

This agent reliably drives the car with maximum throttle set at 75%. As we can see in Figure 3, its trajectory is not optimal. Above 75% throttle, the PD controller is unable to react quickly enough to keep the car within the track boundaries.

3.4 DQN Agent

Our agent seeks to maximize the expected discounted reward. Following an optimal policy, the value of a state-action pair is given by the Bellman equation of the Q-function:

$$Q^*(b,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a' \in A} Q^*(b',a') \,\middle|\, s,a\right]$$

With this problem, current actions have little influence beyond one or two seconds in the future. At 60 steps per second, this is achieved by setting the discount factor γ at 0.98.

We implement a Deep Q-Network (DQN) [2] with a number of optimizations to approximate the Q-function $Q(b,a;W)$ with parameters W. The network is trained with the loss function:

$$L_i(W_i) = \mathbb{E}_{b,a,r,b'}\left[\left(y^{(i)} - Q(b,a;W_i)\right)^2\right]$$

Double Q-learning DDQN [3] is used to obtain the targets:

$$y^{(i)} = r + \gamma Q\left(b', \arg\max_{a' \in A} Q(b',a';W_i)\,; W^-\right)$$

where W^- are the parameters of a separate target network, which are frozen for a fixed number of iterations (10,000 steps).

We accumulate the experience from an entire episode before training the network with a minibatch of samples. Rather than sampling from the memory uniformly, we use a SumTree memory to implement prioritized experience replay [4].

At the end of each episode, we calculate the effective discounted reward $q_{min}^{(i)} = r^{(i)} + \gamma q_{min}^{(i+1)}$ of all the steps and store it in the memory as the tuple (b, a, r, b', q_{min}). Then we use this value when evaluating the targets for a minibatch:

$$y^{(i)} = \max\left(q_{min}^{(i)}, r + \gamma Q\left(b', \arg\max_{a' \in A} Q(b',a';W_i)\,; W^-\right)\right)$$

With the q_{min} values, the network gets trained with the actual discounted reward as the lower bound of each sample state instead of starting off with just its immediate reward. Intuitively, this should reduce the training time.

This q_{min} technique not only reduced the number of episodes required for training, but also improved the stability of our network and allowed us to reduce the size of the minibatches to 8 samples for each new experience step.

4 RESULTS

The following table summarizes the performance of the baseline and DQN agent, averaged over 100 episodes with a time limit set to 10 seconds.

Test	Score	Crashes
Baseline, 75% throttle	**16.0**	**0**
Baseline, 100% throttle	**12.9**	**35**
DQN	**22.1**	**0**

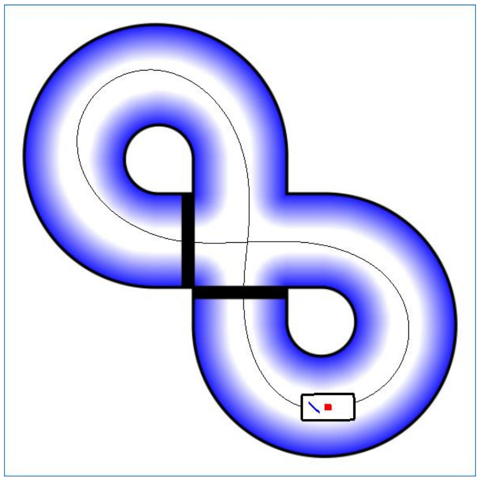


Figure 4: trajectory from the DQN agent

As we can see, our DQN agent outperforms the baseline and manages to keep the car within the track boundaries at high speed. Its trajectory, shown in Figure 4, appears to be a good compromise between driving safely and minimizing lap times.

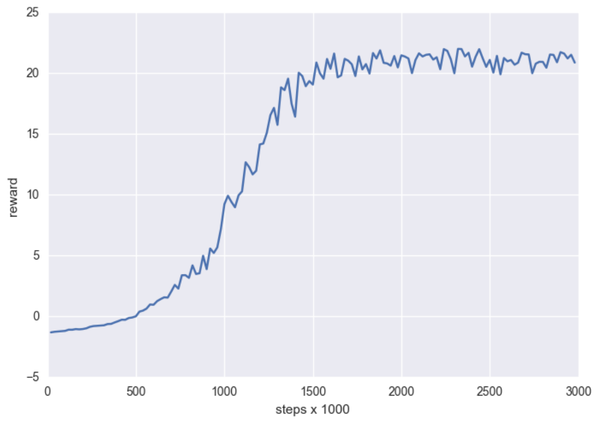


Figure 5: average reward vs. training steps

5 CONCLUSION

We have achieved our goal of using a DQN to efficiently drive our car on the circuit.

Properly modeling the problem was key to getting the DQN algorithm to work. It took several iterations of the model before the DQN agent was able to drive the car past the first checkpoint.

Getting the network to converge required a lot of patience and tenacity. Small changes to the algorithm or the simulated environment can have dramatic effects on the network stability, and it can take several hours of tuning the hyperparameters to find out if the changes were any good.

6 FUTURE WORK

There is still a lot to do before being ready for the competition that motivated this project. The next steps are:

- Try the Dueling Network Architecture [5] and other optimizations to see if the results can be improved further.
- Test the DQN agent in the real world and, probably, tune the simulator to better mimic the real environment.
- See if we can use machine learning to provide estimates of the car's true position based on the observations.
- Add other cars in the simulation and see if we can learn to avoid collisions and overtake slower cars using the distance sensors.
- Repeat the real-world testing and simulator tuning with other cars on the track.

In a more general context, it would also be interesting to find out if the q_{min} optimization generalizes to other types of problem.

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba — *OpenAI Gym*, arXiv:1606.01540, 2016

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattle, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis — *Human-level control through deep reinforcement learning*, Nature 518, 2015

[3] Hado van Hasselt, Arthur Guez, David Silver — *Deep Reinforcement Learning with Double Q-learning*, arXiv:1509.06461

[4] T. Schaul, J. Quan, I. Antonoglou, D. Silver — *Prioritized Experience Replay*, arXiv:1511.05952, 2015

[5] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas — *Dueling Network Architectures for Deep Reinforcement Learning*, arXiv:1511.06581, 2015