Car racing with low dimensional input features

Jean-Claude Manoli — jcma@stanford.edu

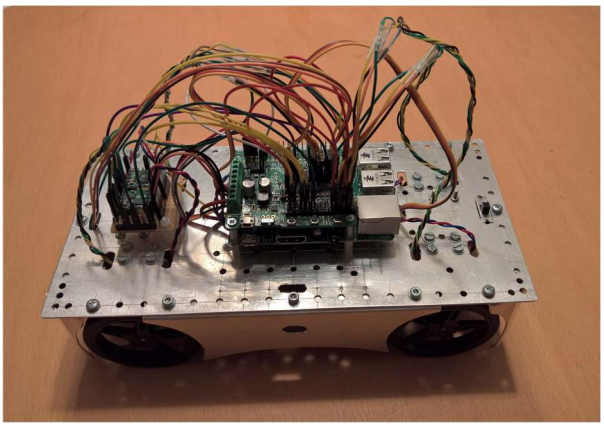
Abstract — The aim of this project is to apply state-of-the-art Deep Reinforcement Learning to pilot a small autonomous race car that has only a few basic sensors. The problem is modeled as a partially observable Markov decision process (POMDP) to handle the state uncertainty, and the policy is trained with a simulated environment that accurately reflects the real world.

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.

The high-tech approach would have been to use high dimensional sensors like video cameras or a lidar. I wanted to see if simpler hardware coupled with deep reinforcement learning would work, so I built a car with only a few basic sensors: wheel encoders provide the car odometer and speed, an orientation sensor provides the heading, and a single sensor measures the track color. The car also has five infrared distance sensors for detecting other cars.

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 and the circuit size is 2×2 m. 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 sensors don't indicate the car's exact position relative to the race track. 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 at high speed.



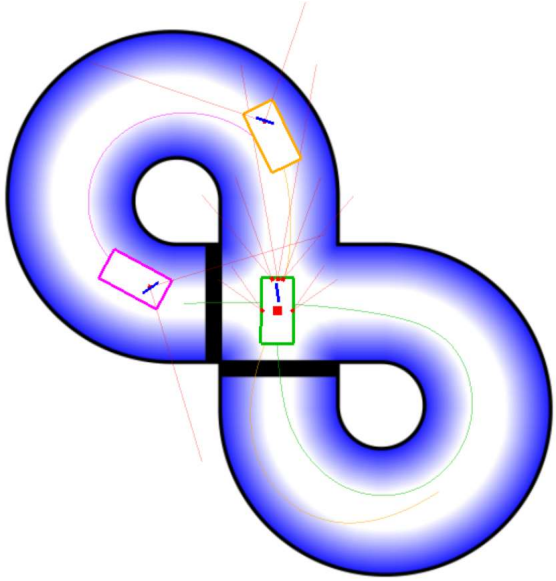


Figure 1: simulated environment rendering of the race track with the AI controlled car (green) and two bots (pink and yellow). The thin red lines represent the IR distance sensor detection cones. The lap and checkpoint thresholds are marked by the thick vertical and horizontal black lines respectively.

In a previous article [1], I showed that a Deep Q-Network can safely and efficiently drive my car around the circuit when no other cars are on the track. The belief state transition model was validated later with tests in the real world.

This time, we will add other cars on the track, modify our model accordingly, and see if we can avoid collisions and overtake slower cars.

2 MODEL

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

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,

- $T: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition model,
- $R: \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function,
- Ω is the observation space,
- $O: \mathcal{S} \to \Omega$ is the observation model,
- $\gamma \in [0,1]$ is the discount factor.

A map of the circuit is provided, with its metrics, so we will use this information in our model.

2.1 State Space

The state space S is the union of the n car states:

$$s = (s^1, \dots, s^n) \in \mathcal{S},$$

$$s^i = \left(x_m^i, y_m^i, \theta_m^i, v^i, \kappa^i, p^i, \psi^i\right)$$

When referring to a single car state, we will usually omit the i car index for readability.

- x_m the normalized longitudinal car position along the track median line, with values in $[-1,1[$
- y_m the normalized lateral deviation from the track median line, with values in $[-1,1]$ when the car stays within the track boundaries
- θ_m the normalized heading of the car relative to the track median line, with values in $[-1,1]$
- v the longitudinal velocity of the car, with values in $[0, v_{max}]$
- κ the curvature of the car trajectory, with values in $[-\kappa_{max}, \kappa_{max}]$
- p the car's throttle position in $[0,1]$
- ψ the position of the steering wheel in $[-1,1]$

We call (x_m, y_m, θ_m) the *median coordinates*. On the lap threshold, $x_m = 0$, and on the check point threshold, x_m flips to -1.

Because the median coordinates space is not linear, we will also use Cartesian coordinates, with the notation (x, y, θ).

Transforming median coordinates to Cartesian space is straightforward, knowing the dimensions of the circuit. The inverse transform requires knowing if the car has passed the lap threshold to disambiguate between the two possible x_m coordinates at the circuit's intersection.

2.2 Action Space

The action space is

$$a = \left(a_p, a_\psi\right) \in [0,1] \times [-1,1] = \mathcal{A}$$

where a_ψ represents the desired steering position and a_p represents the desired throttle position.

2.3 State Transition Model

The car control logic ensures its speed and trajectory curvature are roughly linear functions of the throttle and steering positions. It also limits the steering and throttle changes to avoid excessive jerkiness and skidding:

$$\begin{pmatrix} p' \\ \psi' \end{pmatrix} = \begin{pmatrix} p \\ \psi \end{pmatrix} + \begin{pmatrix} \min\left(|\Delta p|, v_{p_{max}} \Delta t\right) \operatorname{sign}(\Delta p) \\ \min\left(|\Delta \psi|, v_{\psi_{max}} \Delta t\right) \operatorname{sign}(\Delta \psi) \end{pmatrix}$$

with $\Delta p = p - a_p$ and $\Delta \psi = \psi - a_\psi$.

$v_{p_{max}}, v_{\psi_{max}}$ are the empirically determined maximum throttle and steering velocities.

The car control logic also limits the throttle to the maximum safe value at a given steering position, using the $p_{max}(\psi)$ function.

$$p' = \min\left(p', p_{max}(\psi')\right)$$

The car dynamics can be approximated (in Cartesian space) as:

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ v' \\ \kappa' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \\ v \\ \kappa \end{pmatrix} + \int_0^{\Delta t} \begin{pmatrix} \dot{v} t \cos\theta \\ \dot{v} t \sin\theta \\ v \kappa t \\ \dot{v} t \\ \dot{\kappa} t \end{pmatrix} dt$$

with

$$\begin{pmatrix} \dot{v} \\ \dot{\kappa} \end{pmatrix} = \frac{1}{\Delta t} \begin{pmatrix} v_{max} p' - v + p\mathcal{N}\left(0, \sigma_p^2\right) \\ \kappa_{max} \psi' - \kappa + \mathcal{N}\left(\mu_\psi, \sigma_\psi^2\right) \end{pmatrix}$$

$\sigma_p^2, \sigma_\psi^2, \mu_\psi$ represent the car's mechanical noise and bias.

2.4 Reward

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

$$R_x(s, s') = x_m' - x_m + 2 \cdot \mathbb{1}(x_m' \cdot x_m > 0)$$

Where x_m and x_m' are the median coordinate components corresponding to state s and the new state s' respectively.

When the requested throttle position would make the car skid while turning, a small penalty is calculated based on the excess throttle:

$$R_{skid}(a, s') = \left(a_p - p_{max}(\psi')\right) \Delta t \cdot \mathbb{1}\left(a_p > \mathrm{p_{max}}\right)$$

The penalty for driving outside of the track boundaries is equal to the progress reward R_x corresponding to half a lap:

$$R_{out}(s) = -\mathbb{1}(|y_m| > 1)$$

Finally, the car receives a penalty if it collides with another car:

$$R_{coll}(s') = -\mathbb{1}(\text{Collision}(s'))$$

Unlike the real-world competition, where a collision penalty is given only to the car responsible for a collision (when applicable), the penalty is applied to all the cars involved.

The total reward is the sum of the reward components:

$$R(s, a, s') = R_x(s, s') + R_{skid}(a, s') + R_{out}(s) + R_{coll}(s')$$

The episode ends if a car exits the track boundaries or if a collision occurs.

2.5 Observation Space

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

$$o = (\theta_z, v_x, c, d) \in \Omega$$

- θ_z the car heading derived by the orientation sensor's embedded fusion algorithm from its gyroscope, accelerometer and magnetometer
- v_x the longitudinal speed of the car derived from the wheel encoders and steering position
- c the floor color sensor RGB measures, (c_r, c_g, c_b)
- d is a tuple $(d_1, d_2, d_3, d_4, d_5)$ with the measures from the left, front-left, front, front-right and right distance sensors

2.6 Observation Model

The observed heading and velocity are based on the car state with some additional bias and gaussian noise:

$$\theta_z = \theta + \mathcal{N}(\mu_\theta, \sigma_\theta^2)$$
$$v_x = \mathcal{N}(v, \sigma_v^2) = \mathcal{N}(v_{max} p + \mathcal{N}(p \cdot \mu_p, \sigma_p^2), \sigma_v^2)$$

The RGB floor color sensor measures also have some gaussian noise:

$$c_i = \mathcal{N}(\text{GetMapColor}(x, y), \sigma_c^2)$$

The distance sensors are modeled after the real infrared sensors specifications, with the following parameters:

- d_{max} the maximum range of the sensor
- d_{min} the minimum range of the sensor
- α_d the half arc of the infrared beam cone
- σ_d^2 the standard deviation of the measures

2.7 Belief Space

The observations do not provide good situational awareness, so the agent needs to maintain a belief of the state over time.

The state belief space is defined as:

$$b_s = (\hat{x}_m, \hat{y}_m, \hat{\theta}_m, \hat{v}, \hat{\kappa}) \in \mathcal{B}$$

Where $\hat{x}_m, \hat{y}_m, \hat{\theta}_m, \hat{v}, \hat{\kappa}$ are estimates of the corresponding car state variables.

2.8 MDP Reformulation

We can now reframe the problem as a belief-state MDP defined by the tuple $(\mathcal{B}, \mathcal{A}, \tau, r, \gamma)$ where:

- $\mathcal{B}$ is the belief space,
- $\mathcal{A}$ the set of actions,
- $\tau: \mathcal{B} \times \mathcal{A} \times \Omega \to \mathcal{B}$ is the belief state transition function,
- $r: \mathcal{B} \times \mathcal{A} \times \mathcal{B} \to \mathbb{R}$ 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 a simulated 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. We will use deep reinforcement learning to train the policy.

The main software components used in this project are depicted in Figure 2.

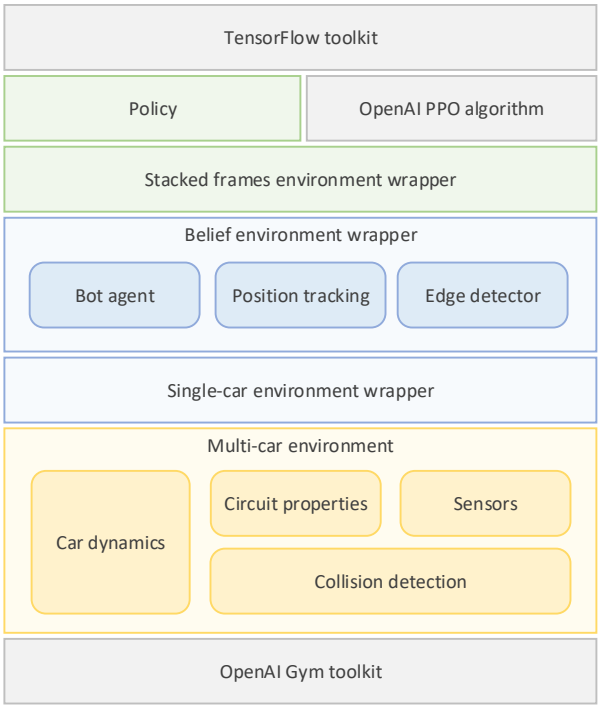


Figure 2: the main software components

3.1 Environment

I have extended the race car simulation environment I built for my previous project to handle multiple cars. The environment, which is based on the OpenAI Gym toolkit [2], models the car dynamics, the circuit characteristics, the infrared sensors and collision detection. It uses a fixed time increment (Δt) set to

achieve 60 steps per second, which corresponds to the lowest sampling rate of the real car sensors.

At each time step, the environment takes a set of actions, and performs the following tasks:

1. For each car:
   a. Move the car according to the state transition model.
   b. Read the car sensors $o^i = (\theta_z, v_x, c, d)$ according to the observation model.
2. Detect collisions
3. Calculate the rewards for each car
4. End the simulation if a car runs out of the track, if there is a collision, or if the time limit is reached.
5. Return the set of observations.

For each new episode, the cars are set at random positions 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. The cars are distributed on the track making sure there is no imminent collision.

To make it easier to use the environment for training, I created a *single-car environment wrapper* that exposes it as an environment taking a single action and returning a single observation at each step. This environment controls the other cars (bots) with abstract agents.

3.2 Belief Environment

The *belief environment wrapper* transforms the observations into belief states.

The simplified state transition model in section 2.3 would not work well for updating the belief state at the relatively high car speed and a 60 Hz sensors refresh rate. Instead, we use the same car dynamics used by the simulator and some error correction. The steps are listed in Algorithm 1, and described below.

After transforming the car position into Cartesian space, we determine the average acceleration and the vertical angular velocity from the wheel encoders and the orientation sensor. We use those values to compute the new car position. The integral is evaluated using the first order Runge-Kutta algorithm.

We then call the **AdjustPosition** function to correct the x and y coordinates when the car is near the lap and checkpoint thresholds respectively. This uses an edge detector that keeps track of the thresholds using the floor color sensor value. Note how the coordinates are only adjusted up when a threshold is detected and down otherwise. This ensures that the coordinate error is smaller than $v\Delta T/2$ around the track thresholds.

We could use the blue gradients on the track to adjust the position relative to the track median. We could also use the orientation sensor gyroscope and accelerometers to filter the predicted belief. But in practice, it turns out that this method produces results that are accurate enough that we don't need to introduce complex filtering algorithms. Figure 3 shows that the error in the (x, y) estimates remain relatively small during a ten-minute simulation at 2 m/s. The position errors measured in the real world are not much larger.

Algorithm 1: belief update

UpdateBelief(b, a, o)

$(\hat{x}, \hat{y}, \hat{\theta}) = \text{MedianToCartesian}(\hat{x}_m, \hat{y}_m, \hat{\theta}_m)$

$\begin{pmatrix} \dot{v} \\ \dot{\theta} \end{pmatrix} = \frac{1}{\Delta t}\begin{pmatrix} v_x - \hat{v} \\ \theta_z - \hat{\theta} \end{pmatrix}$

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

$(\hat{x}'_m, \hat{y}'_m, \hat{\theta}'_m) = \text{CartesianToMedian}(\hat{x}', \hat{y}', \theta_z)$

$(x, y) =$ **AdjustPosition**$(c, \hat{x}_m, \hat{x}'_m, \hat{x}', \hat{y}')$

if $(\hat{x}', \hat{y}') \neq (x, y)$

$(\hat{x}'_m, \hat{y}'_m, \hat{\theta}'_m) = \text{CartesianToMedian}(x, y, \theta_z)$

return $(\hat{x}'_m, \hat{y}'_m, \theta_z, v_x, \dot{\theta}/v_x)$

AdjustPosition(c, x_m, x'_m, x, y)

If EdgeDetector.**Threshold**(c)

if $x_m < 0$ **and** $x'_m > 0$

$y = -\text{threshold\_offset}$

else if $x_m > 0$ **and** $x'_m < 0$

$x = -\text{threshold\_offset}$

else

if $x_m < 0$ **and** $x'_m > 0$

$x = -\text{threshold\_offset}$

else if $x'_m < 0$

$y = -\text{threshold\_offset}$

return (x, y)

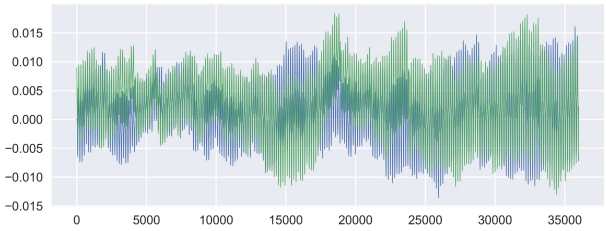


Figure 3: (x, y) position error during a ten-minute simulation

3.3 Belief Augmentation

The belief wrapper also augments the belief state with the following values:

$$b_{augm} = (d_l, d_a, \kappa_t, \kappa_{ta})$$

d_l and d_a are obtained by passing the distance sensor values through a low-pass filter and a moving average filter respectively. The low-pass filter removes the noise from the raw values, and the moving average provides a simple short-term

memory in the case the other car moves out of the visibility cones of the sensors.

κ_t is the track curvature at the car's current position, and κ_a is the weighted mean of the track curvature sampled ahead of the car, with weights decreasing as the distance increases. κ_a is especially useful for anticipating turns.

3.4 Stacked Frames

Like with Atari classic games [3], our learning algorithm will perform better if we stack the last four belief frames. The *stack frames environment wrapper* performs this operation on top of the belief wrapper.

3.5 Bot Agent

A simple reflex agent is used for the bots and as a baseline for this project. To reduce the complexity of the simulation, the bots use a single wide-angle distance sensor.

A PD controller steers the car using some of the belief state values:

$$f(b) = k_y y_m + k_{\dot{y}} \dot{y}_m + k_\theta \theta_m + k_{\dot{\theta}} \dot{\theta}_m + k_\kappa(\kappa_a - \kappa) + k_{\dot{\kappa}}(\dot{\kappa}_a - \kappa)$$

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

With $k_y, k_{\dot{y}}, k_\theta, k_{\dot{\theta}}, k_\kappa, k_{\dot{\kappa}} \in \mathbb{R}$, $\epsilon \in [0,1]$ and $\psi_{step} = 0.1$.

The throttle is maintained below the safe speed based on the desired steering value.

The bot will slow down when the distance sensor value and its derivative reach certain thresholds. The thresholds are stricter when the car approaches the intersection from the left.

All the parameters are carefully tuned to ensure that the bots do not cause collisions during training.

The trajectory of the bots can be adjusted to maintain a certain position relative to the track median in the turns. That and a variable maximum speed can provide a variety of opponent behaviors for training. We'll use two different settings: the first one, called *Reflex*, stays on the outside of a curve, and the second, called *Sharp* stays on the inside.

3.6 Training

We'll use an implementation of a policy gradient method developed by OpenAI called *Proximal Policy Optimization* (PPO) [4]. We chose this algorithm because it performs comparably or better than state-of-the-art approaches, like the ACER [5] algorithm, while being much simpler to implement and tune.

In policy gradient methods, we use a neural network to represent a policy $\pi_\theta(a_t|s_t)$. The parameters θ are optimized to maximize the expectation over future rewards.

PPO uses the following cost function:

$$L_t(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

where c_1, c_2 are coefficients, L_t^{VF} is a squared-error loss, and S denotes an entropy bonus.

L_t^{CLIP} is based on the "surrogate" objective used in the TRPO [6] algorithm. It prevents large policy updates by clipping the probability ratio:

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

where

- θ is the policy parameter
- $\hat{\mathbb{E}}_t$ is the empirical expectation over a batch of samples
- $r_t(\theta)$ is the ratio of the probability under the new and old policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

- $\hat{A}_t$ is the estimated advantage at timestep t for a length-T trajectory:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$
where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

- ϵ is a hyperparameter, usually 0.1 or 0.2

The OpenAI implementation of the PPO algorithm runs the policy for T timesteps in N parallel environments and uses the collected timestamps for an update.

The policy and value functions are approximated by two independent feedforward neural networks. Both networks have two hidden layers of 512 neurons with the hyperbolic tangent activation function.

4 RESULTS

The following table summarizes the performance of the baseline and the PPO agent, with a time limit set to 180 seconds.

Test	Reward	Ep length	Ratio
Baseline 2 cars	199	180	1.1
Reflex, fixed speed	303	136	2.2
Reflex, variable speed	234	108	2.2
Reflex+Sharp, variable speed	172	84	2.0
Baseline 1 car	347	180	1.9
PPO, no bot	426	180	2.4

All the training was performed with one bot. The training requires 30 million timesteps, which takes more than six hours on a four CPU workstation. Early trials with two bots per race were taking twice as much time to train and led to no performance improvement.

The bots had a fixed or average speed of 1 m/s while our agent car was set with a top speed of 2 m/s.

Since the reward is mostly based on the progress along the track, the ratio column provides good comparison on how fast the car completes each lap.

The *baseline 2 cars* is a race between a Sharp agent and a Reflex bot. Since the Sharp agent is unable to overtake the bot, it is almost the same performance as the bot itself.

The *baseline 1 car* is a Sharp agent with no bot, and the *PPO no bot* shows how much the PPO trained agent was able to optimize its trajectory (regardless of which bot it trained with).

The three other results show that the PPO trained agent gets higher rewards than the corresponding baseline, going about twice as fast as the bot. In other words, the agent has learned how to overtake the slower bot. However, their average episode length is much lower than the time limit, which indicates that most episodes end in a collision.

We can see in the top left plot of Figure 4 that the average episode length spikes at almost 10,000 timesteps (almost the three-minute limit) early during training, and the reward is just under 150. At the end of the training, the reward doubles while the episode length goes down 40%. This pattern is present in almost all the trainings. It seems that, like humans, the agent is more prudent as it is learning how to drive, and later it starts taking risks.

The best result is obtained by training against the Reflex bot with a constant speed. It goes down when the bot speed is varied by up to 30%, and gets even lower when training alternates between a Reflex and a Sharp bot. This indicates that the training fails to generalize to different car behaviors.

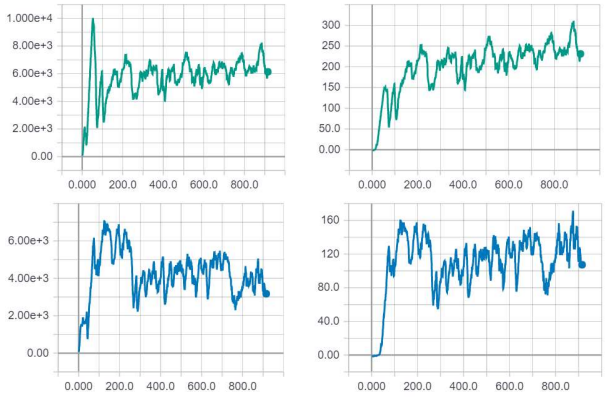


Figure 4: average episode length (left) and rewards (right) for the *Reflex, fixed speed* (green) and the *Reflex+Sharp, variable speed* (blue)

5 CONCLUSION

I have partially achieved my goals for this project: the car can overtake slower cars while optimizing its trajectory to complete the laps as fast as possible. However, this comes at the cost of a higher collision rate which would not be acceptable in the real world. When the learning is tuned to reduce the collision rate, the car is unable to overtake slower cars.

Future work:

- Implement a Hybrid Reward Architecture (HRA) [7], which might work well with our reward structure.
- Compare those results with a model that uses high dimensional input features, like a LIDAR or video camera.

The source code for this project is available at https://github.com/jcsharp/DriveIt/tree/aa228

REFERENCES

[1] Jean-Claude Manoli — *Driving a car with low dimensional input features*, Stanford University, CS229, Autumn 2016, http://stanford.io/2igKSO6

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

[3] 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. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis — *Human-level control through deep reinforcement learning*, Nature 518.7540, pp. 529-533, 2015

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov — *Proximal Policy Optimization Algorithms*, arXiv:1707.06347, 2017

[5] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, Nando de Freitas — *Sample Efficient Actor-Critic with Experience Replay*, arXiv:1611.01224, 2017

[6] S. Kakade and J. Langford — *Approximately optimal approximate reinforcement learning,* ICML. Vol. 2. 2002, pp. 267–274

[7] Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, Jeffrey Tsang — *Hybrid Reward Architecture for Reinforcement Learning*, arXiv:1706.04208, 2017