RaceCar project – Bečvář

Introduction

Goal of this project was to develop a simple environment allowing us to create a race track and let self learning agents learn their way around the track. The bulk of the work was developed in 2020 as a project for my A-levels. Now most of the code was refactored, cleaned up, improved and optimized in several ways.

Application description

The application uses graphical interface which is done in PyGame. There are two main scripts (parts) in use – track preparation and car class.

Track preparation

Group of scripts taking care about the track data preparation enable the user to create custom race track, place multiple checkpoints including start and select the start rotation.

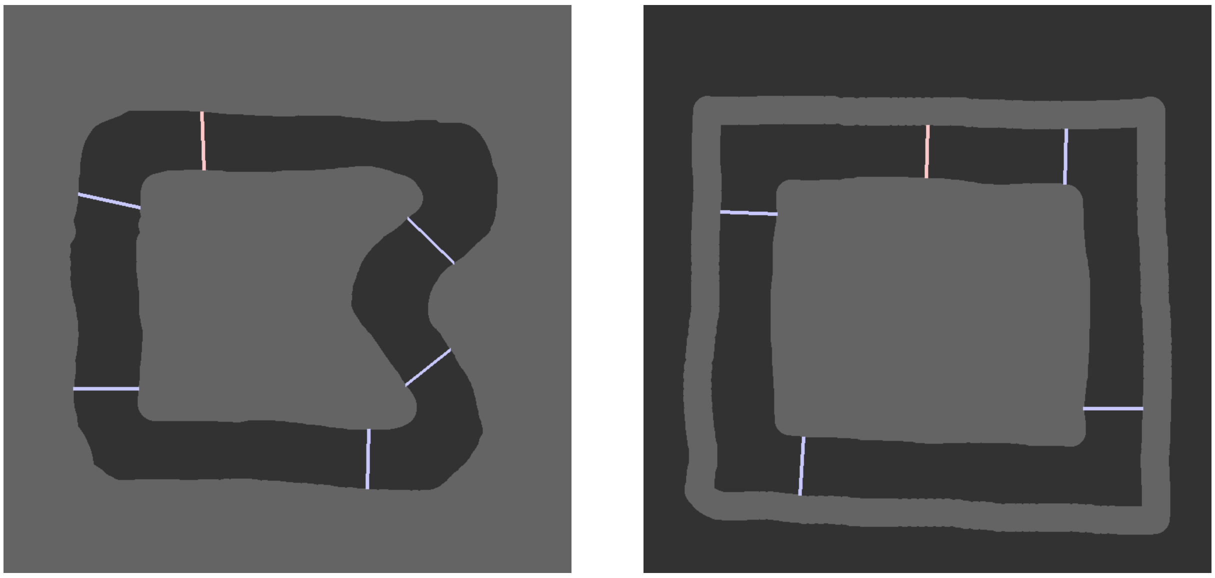

Figure 1: Track 1 (left), Track 2 (right)

Track creation consists of two phases. In first the user draws track borders or walls (in gray color). In this phase user using mouse can paint and erase walls and change the brush size. In the next phase user is tasked with putting down start line (red color) and setting the start rotation and putting down multiple checkpoints (in blue color; not obligatory but advised) by clicking inside the track corridor.

Prepared track can be saved and later loaded for enabling easier experiment setup.

Car script

The second significant group of scripts take care about everything around cars, which are the main actors in future experiments. This group of scripts secure input processing, getting observation data from cars, updating their simplified simulations, collision detection and more.

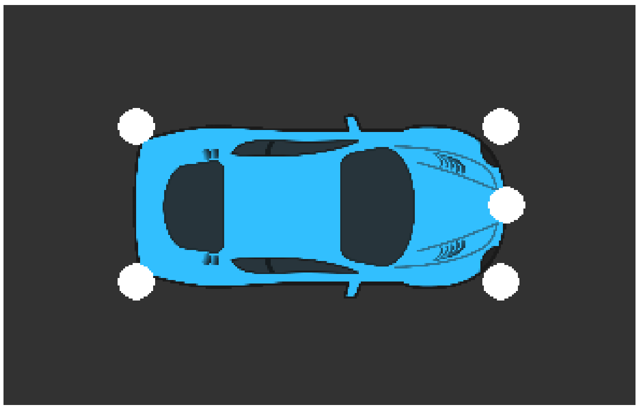

Figure 2: Car with collision points

Car collision are at all times checked against five preset points. If any of these points ever touch a collider, collision is handled for the particular car by itself.

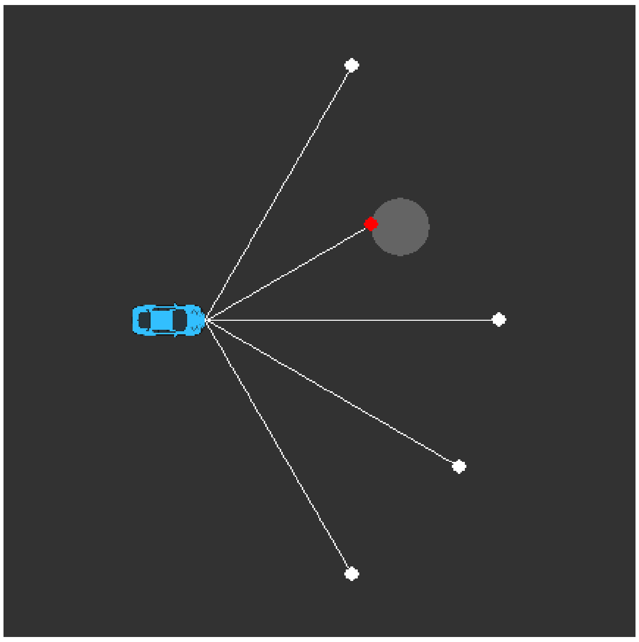

Figure 3: Distance rays

Every car observes its surroundings (**except for other cars – multiple cars do not see each other**) through 5 rays casted from the front of the car out into the world at 30 degree intervals. These rays provide the information about distance to colliders in particular direction. Ray returns the information about a hit distance (if any hit happened).

Distance of these five rays together with $[0; 1]$ normalised tick counter is a 6-tuple representing an immediate observation of each car at single specific time inside the simulation. This observation is used as an input for upcoming learning algorithms.

Cars are handled by two **input controls** (real values between 0 and 1). One controls speeding up and slowing down (cars have hard minimum speed limit) and the other controls turning (and the strength of turning). Both controls have *dead zones* so the input can stay unchanged if the input value is small around zero.

Learning algorithms

I experimented with two different kinds of learning algorithms:

- evolutionary algorithm,
- reinforcement learning algorithm – DDPG

Both of these algorithms use **neural network** (or multiple of them) to represent the main decision making part.

Evolutionary algorithm

Here I used pretty much a basic evolutionary algorithm to evolve a small feed forward neural network with architecture $[6, 8, 4, 2]$. This architecture is a one that proved to be small enough to be fairly time effective and yet be able to learn most of the tracks I was creating for the evolution.

During evolution I used 50 cars running concurrently on the track so evaluation of each generation was fairly quick. Here I think that using EA is a great way for solving this kind of task, because the stochastic exploration of possibilities fits well to the problem of driving through the track without crashing.

I ran experiments with EA on both test tracks with 10 independent repetitions and plotted the results.

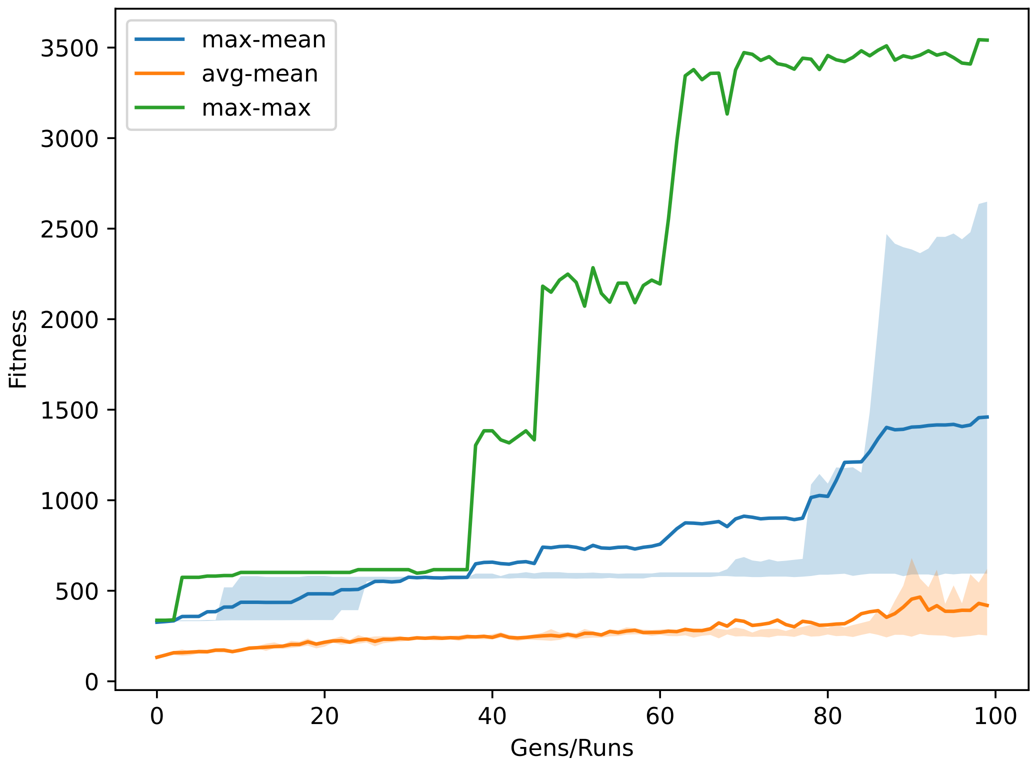

Figure 4: Track 1 results

On track 1 we can clearly see that the first chicane proved to be very difficult with most of the first half spent on learning just how to navigate this section. After that, the runs that were able to pass this section, were reasonably quick in finding their ways around the whole circuit.

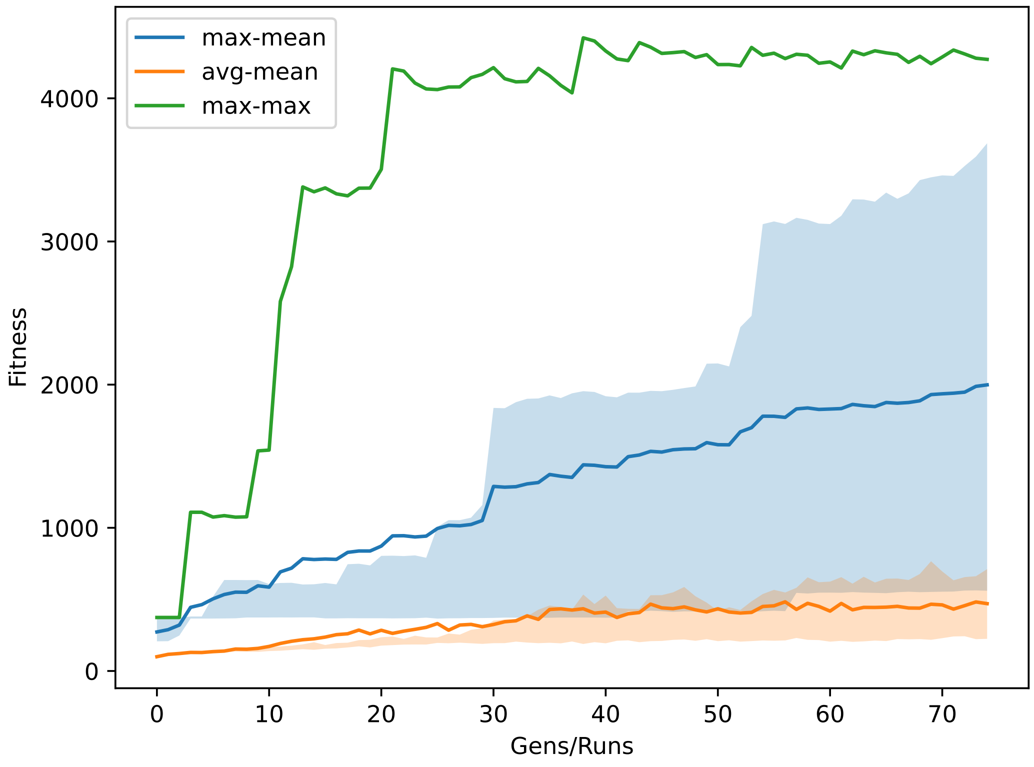

Figure 5: Track 2 results

The shape of track 2 is much simpler. Results prove that once individuals were able to pass first and second corner, turning into other corners was mostly figured out and the only thing left to optimize was speed. We see constant growth of best fitnesses, which would most likely continue to grow till the hard limit on maximum simulation steps.

DDPG – Deep deterministic policy gradient

By searching online for reinforcement learning method I stumbled onto a problem. Most of the methods like to used discrete actions (methods like Q-learning, DQL, Actor-Critic). So I settled on using a bit more advanced DDPG method, which combines the benefits of Q-learning method through its use of target network and replay buffer and Actor-Critic method with two distinct sets of actor and critic networks.

The benefit here is that this method works with continuous actions by default which is exactly what we need.

During training we use only one car. The method uses replay buffer so we are storing set amount of memories to be sampled during training. Those memories consist of state, used action, reward, new state and done flag 5-tuple. Method uses 4 neural networks at a time (2 actor critic pairs – actual and target networks – similar to Deep Q-learning). When training, similar to DQN, we are using the target networks to calculate the quality of next state through critic networks. Then similar to Actor-Critic method we train both critic and actor network and then we do a soft-update of target networks (specific to DDPG). The method is online so we train the networks after every simulation step (sampling memory buffer and updating networks) and is fairly computationally expensive.

There is usually a **exploration-exploitation** problem while handling continuous actions. DDPG solves this by using *Ornstein-Uhlenbeck process* to add random noise to selected actions which drives agent to try and explore possibly new states. During training I do a decay of this randomness so the agent has a learning period firstly with more exploration and then more specialization and improvement.

I tried multiple network architectures for this task (actor-critic methods usually ask for quite a big networks). Here the architecture with 2 hidden layers with 64 nodes each proved to be the best for me. I also tried a network with 128 and 64 nodes in hidden layers respectively and it had potential but was problematic when getting stuck in hard positions and was sometimes unable to escape those local minima.

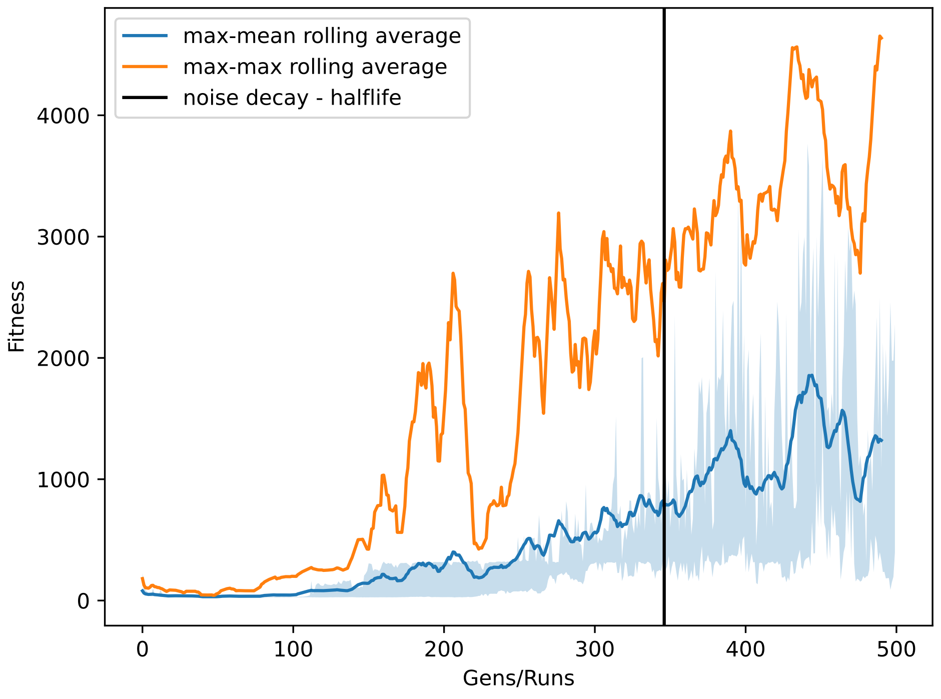

Figure 6: Track 1 results – DDPG [64,64]

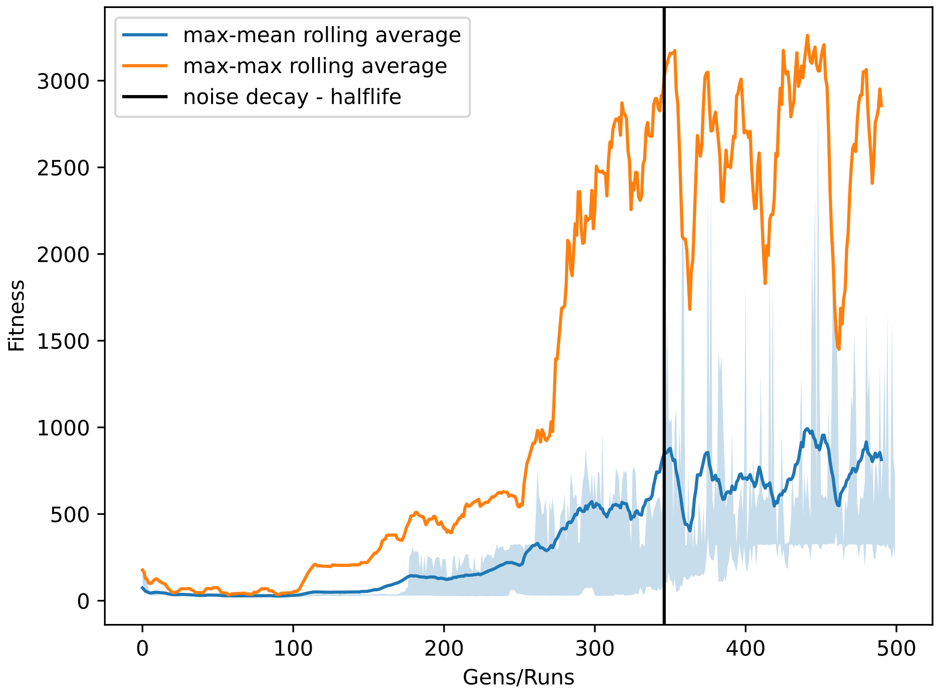

Figure 7: Track 1 results – DDPG [128,64]

We can see comparison of both approaches. The first (smaller architecture) approach seemed to be quite stable at improving its performance. On the other hand the bigger architecture had to be a bit lucky to find its way through the first tricky part, but still struggled afterwards.

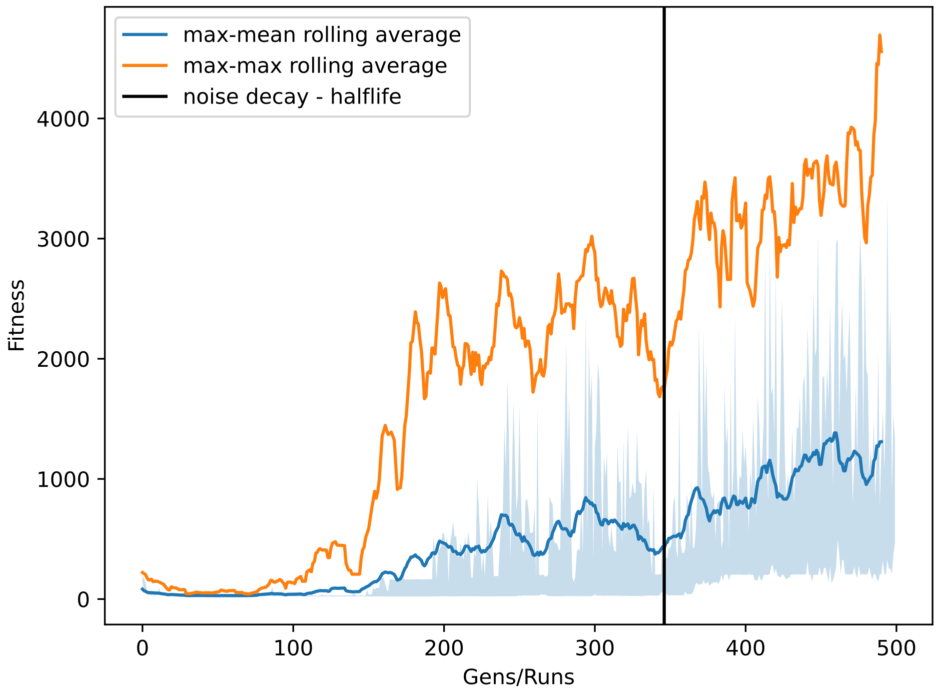

Figure 8: Track 2 results – DDPG [64,64]

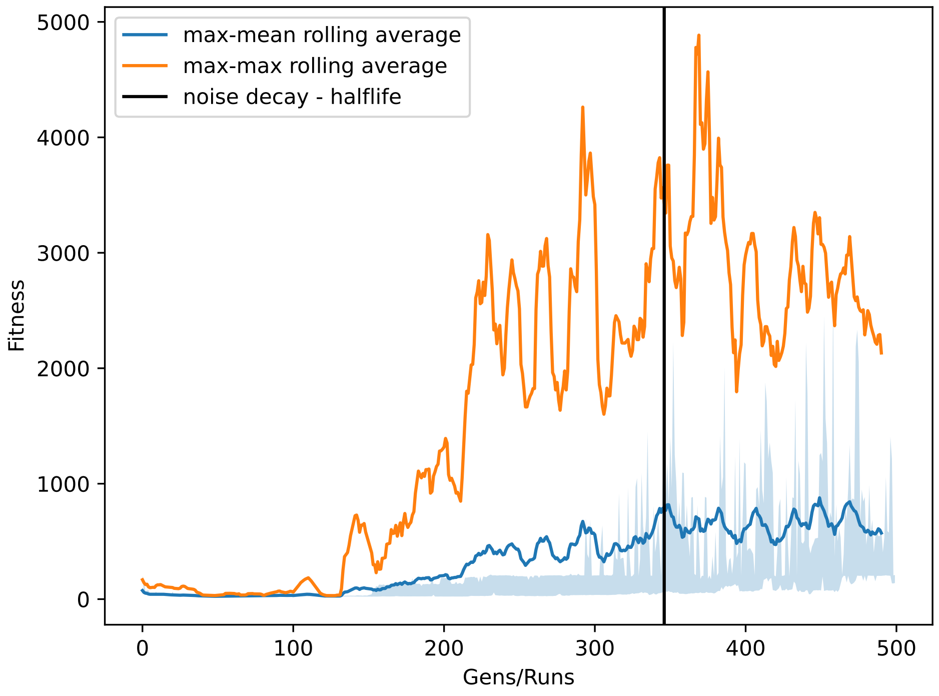

Figure 9: Track 2 results – DDPG [128,64]

On the second track we can see similar outcome but because of the simplicity of the track, both of approaches did well. However, the first one shows signs of constant improvement, which would make it the architecture of choice for further experiments.