Utilize reinforcement learning to develop a self-driving car in a simulated environment

Domain background

The detailed photorealistic environments of modern computer games provide cheap and portable development environments to build artificial intelligences (AI) to steer autonomous vehicles (deepdriving, 2017). Computer games have already been used to create breakthrough research in the field of AI in the last years. Google's deep mind team has developed AIs, combining deep neural networks and reinforcement learning, which played competitively in more than 50 Atari 2600 games with the same information available to humans (Minh & al, 2015, Nature). In 2016 different AI's competed against each other in the first-person shooter DOOM, where these AI's developed strategies similar to humans (http://vizdoom.cs.put.edu.pl/competition-cig-2016).

The advantage of using computer games to train these AIs are: Mistakes don't have severe consequences (besides a negative reward); Developing an algorithm is cheaper; Training can be easily parallelized; And the work can be reproduced. Thus, the recent studies, using the realistic vehicle dynamics to train neural networks for self-driving cars (https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html) pave the avenue for the use of reinforcement learning in the field of autonomous vehicle AI for the next years.

Here, a machine learning platform is developed, which allows to train the open source game Torcs (http://torcs.sourceforge.net/). As a basis, gym torcs (https://github.com/ugo-nama-kun/gym_torcs) will be used, which allows to use Torcs from python. But since gym torcs is lacking the possibility to train torcs in parallel. This disables the possibility to use the state of the art machine learning algorithms, such as asynchronous actor critic (https://arxiv.org/abs/1602.01783).

Problem statement

This capstone project will develop a reinforcement learning algorithm using the open source racing game Torcs. Torcs will simulate the driver (race car driver) and the environment (race track) and provide the necessary sensory data to train this agent. An AI algorithm for this agent will be developed using state of the art reinforcement algorithms, i.e. asynchronous actor critic. The goal is to develop an agent which will drive the race car as a fast as possible around the race track.

Datasets and input

The data will be generated using a modified version of Torcs, which sends the data to the agent via a websocket.

ob.angle	[-π,+π]	Angle between the car direction and the direction of the track axis
ob.track	(0, 200)(meters)	Vector of 19 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters
ob.trackPos	(-∞,+∞)	Distance between the car and the track axis. The value is normalized w.r.t. to the track width: it is 0 when the car is on the axis, values greater than 1 or -1 means the car is outside of the track.
ob.speedX	(-∞,+∞)(km/h)	Speed of the car along the longitudinal axis of the car (good velocity)
ob.speedY	(-∞,+∞)(km/h)	Speed of the car along the transverse axis of the car
ob.speedZ	(-∞,+∞)(km/h)	Speed of the car along the Z-axis of the car
ob.wheelSpinVel	(0,+∞)(rad/s)	Vector of 4 sensors representing the rotation speed of wheels
ob.rpm	(0,+∞)(rpm)	Number of rotation per minute of the car engine

Table 1. Inputs which are used for training the agent (taken from https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html)

Proposed solution

To train the agent we start from the already existing implementation of the Deep Deterministig Policy Gradient (DDPG) Algorithm (https://arxiv.org/abs/1509.02971) in python (https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html). As a first step, this algorithm will be analyzed and taken as a benchmark model. Also, the algorithm will be improved by batch normalization, tuning of hyperparameters.

Then as an improved version, an asynchronous actor critic (https://arxiv.org/abs/1602.01783). implementation of the algorithm will be used. This requires to parallelize the execution of gym Torcs. Here, to improve the exploration, the agent will be trained on different tracks simultaneously.

The parallelization is performed by running Torcs in a Docker container on Amazon Web Service.

Evaluation Metrics

Reward at time point t:

R_t = speedX cos (angle) – speedX sin (angle)

Total reward (evaluation metric):

Total_reward = $\sum R_t$

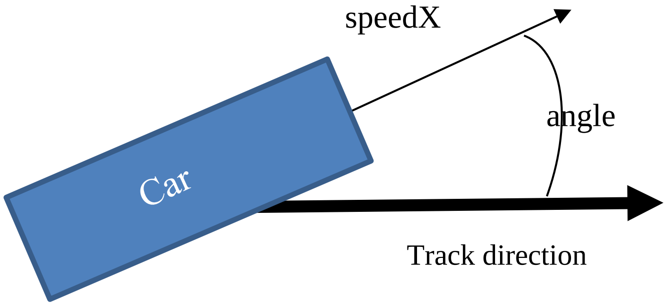

Figure 1. Reward function

As the reward function, we use the accumulated reward of the longitudinal velocity penalized by the transverse velocity, i.e. R_t = ob.speedX cos (ob.angle) – ob.speedX sin (ob.angle), as suggested by (https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html). The evaluation metric will be the accumulated reward of the different agents on a unknown test track.

Benchmark models

which is an implementation of the Deep Deterministig Policy Gradient Algorithm (https://arxiv.org/abs/1509.02971)will serve as a benchmark model.

Project Design

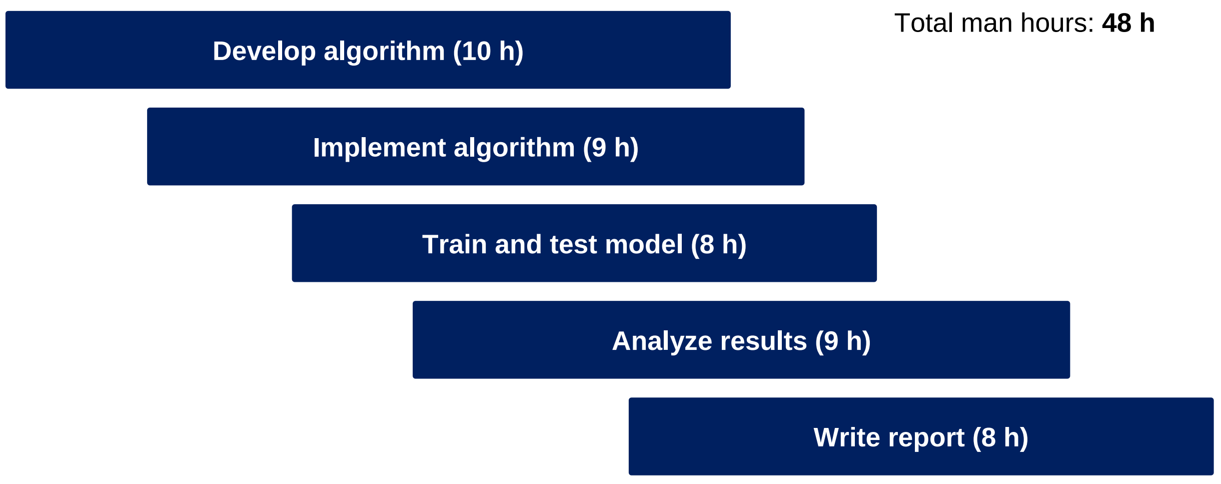
Total man hours: **48 h**
Develop algorithm (10 h)
Implement algorithm (9 h)
Train and test model (8 h)
Analyze results (9 h)
Write report (8 h)

Figure 2. Overview of the project

Development environment

The development machine runs Goggle’s Tensor Flow™ on a Windows 10 machine, thus it requires a Python 3.5 installation. Further to utilize a graphical processing unit for computations, NVIDIA’s CUDA® Deep Neural Network library to utilize a graphical processing unit has been installed The agents run in a containerized Docker environment on AWS.