Reinforcement Learning Approach for Formula Student Technion Driverless

Project report

Elior Kanfi

305245128

Supervised by:

Mr. Yaron Honen (Technion), Mr. Roman Rabinovitch (Technion)

Semester:

Winter 2021

Date:
10/3/2021

This project was carried out under the supervision of Mr. Yaron Honen and Mr. Roman Rabinovitch from the Faculty of Computer Science at the Technion – Israel Institute of Technology.

This project was contributed by Asher Yartsev.

My thanks go to all that helped and supported this project.

The Code Repository for this project can be found at:
https://github.com/eliork/Reinforcement-Learning-on-Autonomous-Race-Car

Table of Contents

Abstract

"Formula Student" is a competition between universities all over the world, mainly among B.SC. students. Each university forms a team of its students, with the goal a successful construction of a fully functional race car from scratch and competing with their self-built car against other universities in a race against time, within the time span of only one academic year.

The competition is being held in multiple locations around the world in different dates throughout the year.

In recent years the Technion – Israel Institute of Technology team has put its efforts in the competitions in Italy, Czech Republic and Germany, and succeeded multiple times to achieve remarkable results.

With the rise of self-driving cars, following the deep learning revolution, computer vision computation and complex decision problems, it was a matter of time until a competition of self-driving student race cars would be held. 2017 was the first year in which a competition of Formula Student included a self-driving cars competition, in addition to the regular competition.

Shortly after that, the Technion team decided to recruit students for a new team focusing on the new competition, trying to design, build and program the first self-driving race car in Israel, from scratch, solely by students.
The goal was set to complete the project and participate in the competitions by the summer of 2020 in Europe, but unfortunately that was delayed mainly due to the coronavirus crisis in 2020.

This project report elaborates the work of a reinforcement learning algorithm that was designed and implemented in order to contribute to the project, help it and push it to the next level.

Introduction

The objective of this project was to create a Reinforcement Learning algorithm for steering control to the Technion's autonomous Formula Student car, with a simulation of the same settings of the racetrack and the simulation of an actual car that will be used in future competitions.

The competition takes place on a professional Formula racetrack, with two lines of traffic cones, blue on the left and yellow on the right, defining the track, with the start/finish line marked with orange and white color cones.

The track is not known to the teams, and the algorithm designed can't have any assumption on the shape of the track, other than the cones marking the track, so the algorithm's challenge is to be as accurate as it can be on any unknown race track and environment.

In order to train my algorithm, I had to have a proper training environment. Due to safety reasons, and my inability to access a closed road track, and the fear of damaging and crashing the actual car, I had to simulate the car and track environment.
I tried to make it as accurate as possible, to make the transition from the simulation to the real world smooth.

The main goal of my algorithm is to predict the best action for the car – which is the steering angle - at any given moment, in real time, using only a single image as input.

My algorithm is based on a deep learning method, called "Deep Reinforcement Learning".

Training Environment

Training a deep learning algorithm to perform autonomous driving requires a huge amount of data consisted of hours and hours of driving recordings. Since it was not possible to obtain such amounts of data with the real car, I needed a simulation that could demonstrate a realistic formula student environment.

I started by taking an open-source simulation that was created by Microsoft, called “AirSim”. The simulation is based on Unreal Engine 4 (UE4), which is commonly used for creating computer games.

“AirSim” developers programmed various cars into the simulation in order to perform autonomous driving. So, I used an implementation of the Formula Technion vehicle model and environment into the simulation.

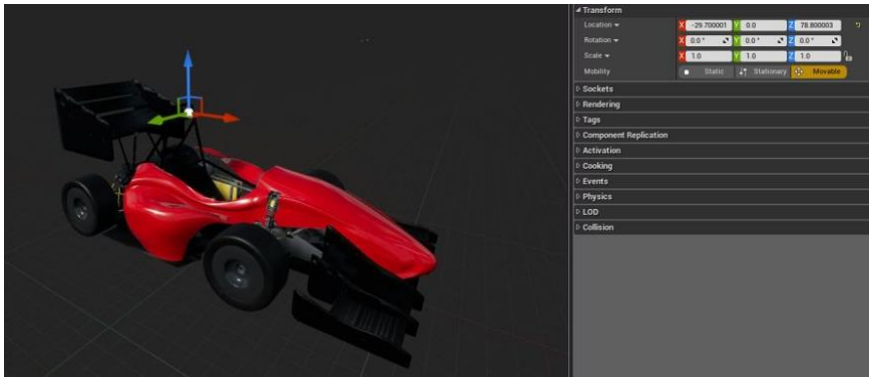
Also, I used the environment that was developed by my colleagues Dean Zadok, Tom Hirshberg and Amir Biran, in their “Formula Technion 2018 – Self-Driving Vehicle Algorithm” project, where they implemented a very realistic version of the actual car and racetrack and trained the car to drive using imitation learning methods.

The actual car and the Simulated car model in UE4

A part of the scene

The traffic cones, simulated

Track creation tool



Position of the Camera on the Simulated car

Training the Model

Introduction to Reinforcement Learning

In reinforcement learning algorithms, we model the problem as an agent acting in an environment. Our agent will be the car, and our environment will be our simulated track, consisted of traffic cones, and the entire landscape.

The agent observes the environment, by getting an image from a camera mounted on the car, and that image is referred as observation, then the agent preforms an action, which will be the steering angle, and receives a reward, good or bad, according to the quality of the action it preformed.

Reward is a number that tells the agent how good or bad the current state of the world is. For example, if the car takes an action that leads to hitting a cone, then the current state of the world is that the car had crashed, and the agent should be penalized for that action. If the car takes an action that keeps the car on track, then the agent should be rewarded for that action.

The agent's goal is to maximize its cumulative reward, called return. By doing so, the agent "learns a policy", meaning learns how to act in the environment, by analyzing the past rewards of his actions, and optimizing the future actions accordingly, to maximize the return. Reinforcement learning methods are ways that the agent can learn behaviors to achieve this goal.

States and Observations

A state s is the complete description of the current state of the world. There is no information about the world that is not included in a state.
An observation o is a partial description of a state, that usually omits some parts of the information in a state. We can say that $o \subseteq s$.

In our example, a state will be the entire state of the racetrack, and of the car, position, speed, etc., and an observation is the image that is captured by the camera mounted on the car. Due to the fact there is only one camera, that sees only part of the environment, we say that the environment is partially observed. If the agent can see the full environment, we say that the environment is fully observed.

In my project, an observation was at first an RGB image, which is a 3-dimensional tensor, and later, in the final model, is a 1-dimensional real valued vector.

Action Spaces

In reinforcement learning, different algorithms apply to different action spaces. An action space is the set of all valid actions in a given environment. For example, in this project the action is the steering of the car. An action space is said to be discrete if it has only a finite number of actions that are available to the agent, and continuous if the actions are real-valued vectors.

I have tried both methods, to test multiple algorithms. First, I used a discrete action space, by dividing the steering value to 11 different possible actions. Later I defined the action space to be continuous, by being the numbers in the range from (-0.5) to (0.5).

Policies

A policy is a set of rules used by an agent to decide which actions to perform. It can be deterministic, though in most applications of reinforcement learning it is stochastic, and usually denoted by π:

$$a_t \sim \pi(\cdot|s_t)$$

Where:
a_t – is the action taken at step t
s_t – is the state of the world at step t
π – the policy

In deep reinforcement learning, we usually work with parameterized policies: policies whose outputs are computable functions that depend on a set of parameters like the weights and biases of a neural network for example, which we can adjust to change the behavior via some optimization algorithm.
We denote those parameters by ϕ or θ thus writing our policy as:

$$a_t \sim \pi_\theta(\cdot|s_t)$$

Trajectories

A trajectory τ is a sequence of states and actions in the world

$$\tau = (s_0, a_0, s_1, a_1, \dots).$$

State transition is what happens to the world when in timestep t the state of the world is s_t and then at timestep $t+1$ the state of the world is s_{t+1} . The transition is only dependent on the most recent action a_t. While the actions come from the agent according to its learned policy, we can say equivalently, the transition is stochastic:

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

Reward and Return

The reward function R is dependent on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1})$$

Sometimes this equation is simplified to just be dependent on the current state-action pair

$$r_t = R(s_t, a_t)$$

The goal of the agent is to maximize the cumulative reward over a trajectory.

The return is infinite-horizon discounted return, which is the sum of all rewards ever obtained by the agent, discounted by how far off in the future they are obtained. Formally this includes a discount factor $\gamma \in (0,1)$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

The discount factor is used because an infinite-horizon sum of rewards may not converge to a finite value, which is hard to deal with in many equations. With a discount factor and under reasonable conditions, the infinite sum converges.

The Reinforcement Learning Problem

As written above, the goal in reinforcement learning is to select a policy which maximizes expected return when the agent acts according to it.

To talk about expected return we first talk about probability distributions over trajectories.

The probability of a T- step trajectory is:

$$P(\tau|\pi) = \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

The expected return, denoted by $J(\pi)$ is:

$$J(\pi) = \int_{\tau} P(\tau|\pi)\, R(\tau) = E_{\tau\sim\pi}[R(t)]$$

And finally, we can say that the central optimization problem in reinforcement learning is expressed by

$$\pi^* = \arg\max_{pi} J(\pi)$$

With π^* being the optimal policy.

Value Functions

In reinforcement learning it is common to compute the value of a state-action pair, meaning the expected return if you start in that state-action pair and then act according to a particular policy forever after. Value functions are used in almost every reinforcement learning problem.

The On-Policy Action-Value Function, $Q^{\pi}(s, a)$ which gives the expected return if you start in state s then take an arbitrary action a and then forever act according to policy π is:

$$Q^{\pi}(s, a) = E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]$$

And accordingly, we define the Optimal Action-Value Function, $Q^*(s,a), as$

$$Q^{\pi}(s,a) = \max_{\pi} \ (E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a])$$

These functions are most often called the Q functions.

The Optimal Action

When the agent is at state s, The optimal policy will select the action that maximizes the expected return from starting in s, as a result, if we have Q^*then we can directly obtain the optimal action $a^*(s)$ by

$$a^*(s) = \arg\max_{a} Q^*(s,a)$$

Bellman Equations

All functions above obey the special self-consistency equations called the Bellman equations, which means that the value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.

The Bellman equation for the on-policy value functions are:

$$Q^{\pi}(s,a) = E_{s'\sim P}[r(s,a) + \gamma E_{a'\sim\pi}[Q^{\pi}(s',a')]]$$

Where $s' \sim P$ is shorthand for $s' \sim P(\cdot\,|s,a)$, $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$ and $a' \sim \pi$is shorthand for $a' \sim \pi(\cdot\,|s')$

The Bellman equation for the optimal value functions are

$$Q^{\pi}(s,a) = E_{s'\sim P}[r(s,a) + \gamma \max_{a'} \ [Q^*(s',a')]]$$

The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

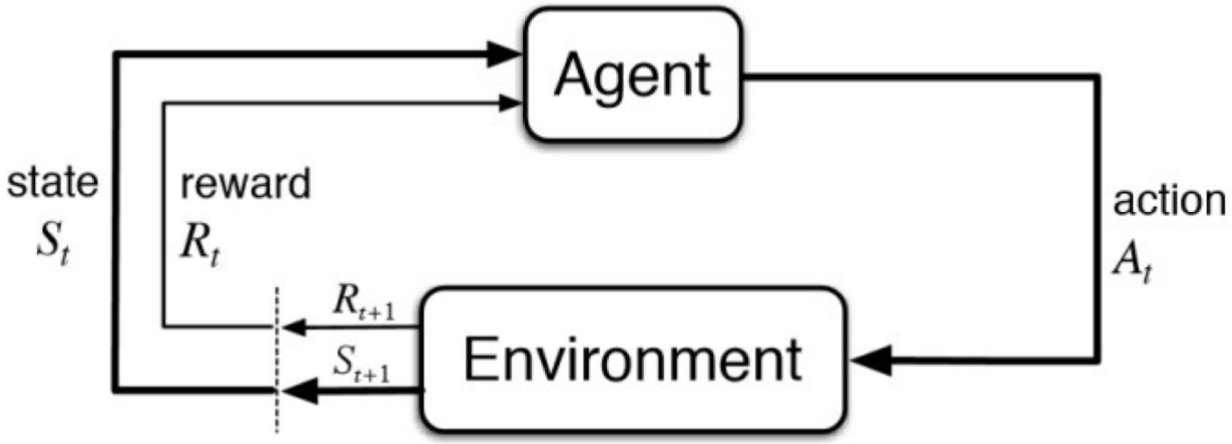
Chosen Model and Algorithm

After examining few different algorithms, I chose to use SAC (Soft Actor Critic), and also to use VAE (Variational Auto Encoder) for this project.
SAC is a state-of-the-art algorithm for reinforcement learning and was used to solve many problems that have a continuous action space, and VAE is an image encoder that reduces remarkably the observation space to a much smaller dimension, which helps the algorithm to learn a good policy both faster and more accurately.

A continuous action space in this context means that the steering angle of the car is transformed to the continuous range of numbers from -0.5 to 0.5, negative numbers represent steering angle to the left, positive numbers represent steering angle to the right, 0 represent straight forward, and the algorithm chooses the action to be taken, meaning the next steering angle of the car, by outputting a number in that range.

In theory, when implemented successfully, a reinforcement learning algorithm can potentially make driving decisions better than a human driver, given the time to train on many environment possibilities, which can take lots of time.

I believe that my algorithm succeeded to learn a good policy, and in this report, I will share the experiments I have made until I have achieved the final model.



Reinforcement Learning framework

Discrete Action Space + Learning from Raw Pixels Approach

At first, I have tried to use a discrete action space, and I have divided the car's steering values to the following:

Action	Steering value
0	-0.5
1	-0.4
2	-0.3
3	-0.2
4	-0.1
5	0
6	0.1
7	0.2
8	0.3
9	0.4
10	0.5

In order to use a discrete action space, I had to use reinforcement algorithms that work on a discrete action space, thinking that approach will lead to faster and more accurate learning.

My first attempt was to use DQN algorithm with CNTK framework inspired from AirSim's original source code.

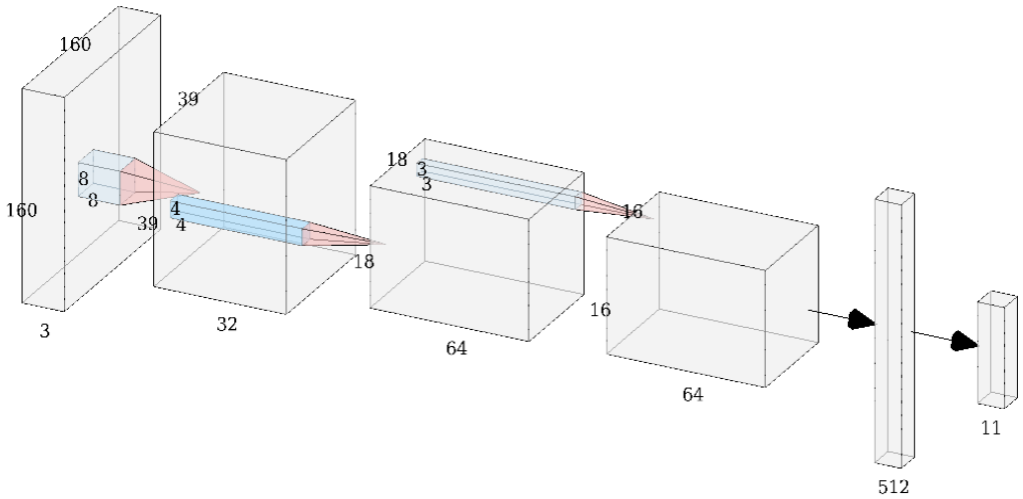
The observation was 160X160X3 image, that was converted to greyscale, and my attempt was to learn from raw pixels.

These experiments didn't perform well, but it was mainly for better understanding of the AirSim API and the interface between AirSim, the DQN algorithm, and the simulation in Unreal Engine.

Later on, I have changed to Keras framework with TensorFlow backend.

The architecture I used in my first attempt was based on three convolutional layers, and two fully-connected layers, with the input being as before an image of 160X160X3 dimension, as follows:

Conv2d 32 8x8
Conv2d 64 4x4
Conv2d 64 3x3
Fc 512
Fc actions number, with action number being 11, as mentioned in the table above.



My attempt was to learn only the steering value, with a fixed throttle value of 0.8 that would get the car to a constant speed of approximately 7.5 m/s. with that speed I thought it is reasonable to assume that the car could steer successfully in corners that are not too sharp and complete a full lap.
I chose a track that will not contain many sharp turns, to ease and fasten the process of learning.

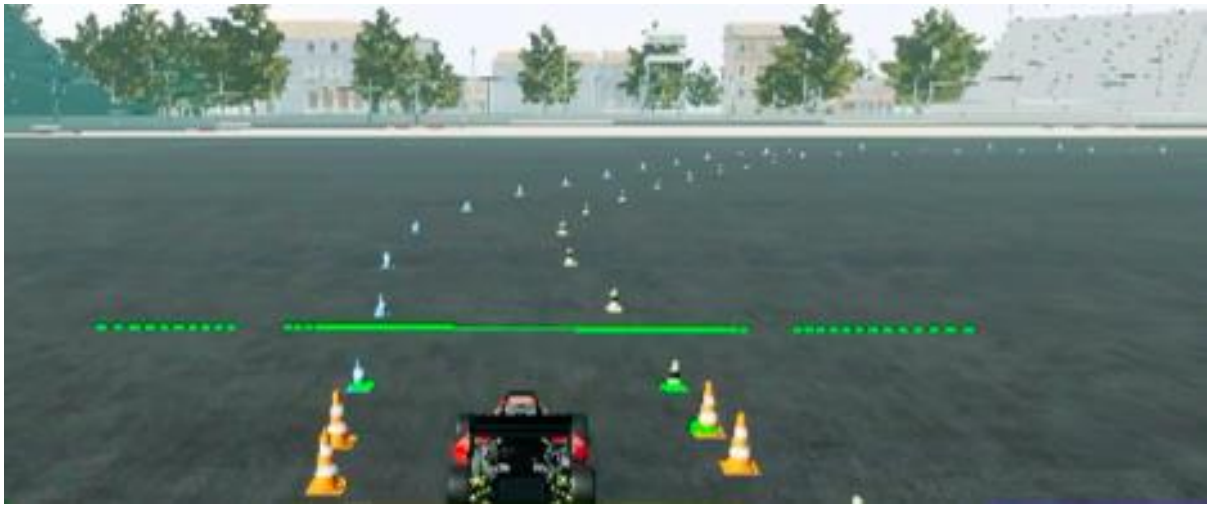
My main work afterwards was to try and test multiple reward functions, and to see the effect of each reward function on the learning process. For each reward function I also supplied a “collision penalty”, where I gave a penalty for hitting cones, and another penalty for exiting the track bounds.

In order to get the collision data, I wanted to use LIDAR technology, since the actual car can also have a LIDAR camera mounted on it, instead of getting the collision information from the simulator.

My thought behind that idea was to get the car to avoid colliding the cones, and also to make the car realize it is about to hit a cone and to end the current training episode faster, rather than getting the collision info that it hit a cone only after the collision, or to make the car try to fix itself if it is about to hit a cone.

LIDAR – Light Detection And Ranging - is a remote sensing method that uses pulsed laser to measure ranges to different obstacles. It is done by illuminating the target with laser light and measuring the reflection with a sensor. Using those measurements, we can tell whether our car is about to collide in a cone and penalize its behavior accordingly.

In the latest version of AirSim, LIDAR is fully integrated allowing users to use it for training agents, and I have done so, mainly to calculate the reward function and to find the best reward function.



AirSim with LIDAR

Reward Functions

Reward by staying in bounds with or without speed bonus

I have extracted the positions of the blue and yellow cones from the track and tried to form a polygon from it. Then, I gave a reward of +1 for every timestep the car stays in the polygon.
In addition, I also tried to add a speed reward function as follows:

$$f(x) = \frac{x - speed_{min}}{speed_{max} - speed_{min}} - 0.5$$

With x representing the current car speed, and $speed_{min}$ is defined as 1 m/s and $speed_{max}$ is defined to 30 m/s. The final reward function was

$$g(x) = 1 + f(x)$$

for every step the car is in the polygon, meaning stays on track, and -10 for exiting the polygon.

This method didn't perform very well, but it was promising.

Reward by staying close to the center of the track with lidar

I have tried a different approach to the reward function. Instead of using prior knowledge, like the polygon that symbolizes the track, I tried to challenge that and to use a reward function that doesn't use any knowledge of the track. I tried to get the points from the lidar camera, and to compute from those points the center of the track, and to give a higher reward as the agent is closer to the center of the track.

I extracted the nearest left and right points from the output of the LIDAR camera, those represented the left cone and the right cone, respectively. Then after having the distance from the car to the left and right cone, I could compute the middle point between these cones, and give the reward according to the distance of the car from the middle point.

The reward function was as follows:

$$g(x) = \frac{1}{e^{\beta \times dist}} - 0.5 + f(x)$$

Where:
β is a hyper parameter
$dist$ is the distance from the car to the middle point of the track.

I set β to 3, and tried also with 2 and 4, and $f(x)$ is the same function as earlier.

I also tried to remove the speed reward function, $f(x)$ that was described above, to this reward function, and to measure the different results, unfortunately, this method did not prove to be successful.

Reward by staying close to the center of the track with lidar with penalties

After forming the function above, I tried to improve it in different ways by penalizing the car for bad actions.
If the car was about to exit the track, I gave it a negative reward of -0.9, but didn't end the episode allowing the car to return to the track and fix its mistakes.

If the car was about to hit the cones, I gave it a negative reward of -5 and reset to the start of the track to begin a new episode.

If the car exited the track, and the LIDAR readings didn't recognize any points, I gave it a negative reward of -10 and reset it to the start.

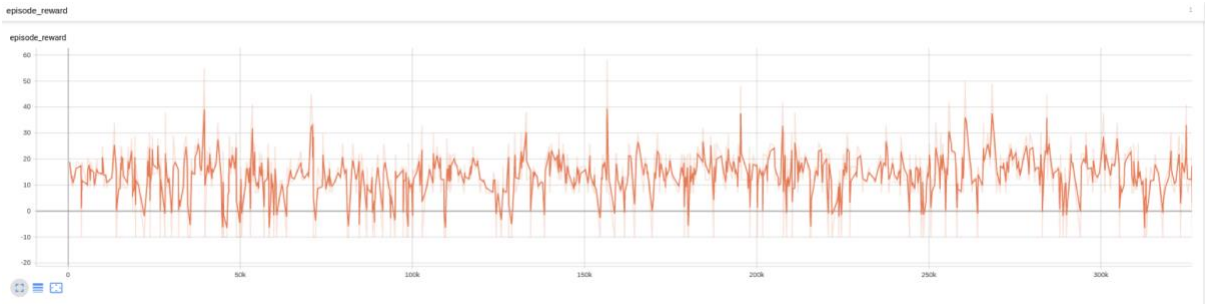
Reward by staying close to the center of the track with two consecutive lidar points

Following unsuccessful runs, I have tried to compute for every LIDAR reading the two consecutive points of both blue and yellow cones, and then tried to compute their center line, and to reward the car for following that line.
When the LIDAR camera recognized the 4 points, I could extract them and divide them to the nearest left, nearest right, next left, next right.
Then I computed the middle point between the nearest left cone and nearest right cone, and the middle point between the next left cone and next right cone, and to form the line connecting these points, and define the reward function to give rewards accordingly.

Since the distance between each pair of consecutive cones is 400 cm, and the distance between a left and right cone is 350 cm, I used simple calculations to get the distance of the car to the center line connecting the two center points of consecutive points.

At first, I set the maximum distance the car can exit the center line to be 500 cm, meaning letting the car to deviate away from the track boundaries for a maximum of 75 cm, in order to make the car learn to “fix” itself if it goes out of bounds, but not hitting the cones. Then I tried to lower that value to 350 cm the same as the track width, so the car will not train on unwanted observations.

After unsuccessful attempts with the reward functions above, I have thought to change framework, and instead of using CNTK, I chose to use stable-baselines framework with TensorFlow backend, and I got back to the planning phase, as it didn't seem like I made any progress with the different reward functions.



Unsuccessful results graph

Changing to Stable Baselines and TensorFlow Framework

Stable-Baselines is a Git repository that has plenty of Reinforcement Learning algorithms implementations, and it is very convenient for anyone who wishes to use the algorithms written and train agents with this methodology.

In order to use their algorithms, I have built a custom OpenAI Gym Environment, defining a class for the environment, and a class for the car. The environment then interfaces with the stable baselines algorithms very easily and I was able to explore many algorithms with just few lines of code.

With the change this framework, I have also decided to change the way to determine the way I compute the center of the track. I added to the simulated track an empty actor in Unreal Engine, which is an invisible point in the center between each pair of blue and yellow cones on the track, and named it 'WayPoint#', with # symbolizing the number of WayPoints I used, and they marked the center positions in the track, so I can calculate if the car deviates from the track or stays inbounds of the track.

I calculated at each step the two closest WayPoints to the car's position, and I calculated the center line, which is the line between those two points that is connecting them, then I calculated the distance of the car from that line, and that gave me the distance from the car to the center line.

I kept using a constant throttle value of 0.8, to keep the car at around 7.5 m/s.

I updated the reward function, and set it to be as follows:

If the distance of the car to the center line that was calculated above is less than 0.1 * track width, then the reward is 1

Else If the distance is less then 0.25 * track width, then the reward is 0.5

Else if the distance is less then 0.5 * track width, then the reward is 0.1

Otherwise – the car is either off track or about to hit a cone – the reward is -10 and reset the session to start a new episode.

With this method, I got quite nice results, and after approximately 24 hours of learning the car could drive quite nicely through mild corners but could not handle sharp turns and did not complete the track.

I wanted to improve those results and also to change the throttle values. I started with a simple change to the action space.

The action space was still consisted of 11 actions, but now if the steering value was between -0.3 and 0.3 then the throttle would be 1, and otherwise, the throttle would be 0.3.

this would simulate acceleration of the car in straight lines and light turns, and a slowing down when the car is doing a sharp turn.

This result did improve slightly the behavior of the car, though the car got to drive too slow on many occasions, and the results were still not satisfying at all.

WayPoints Example

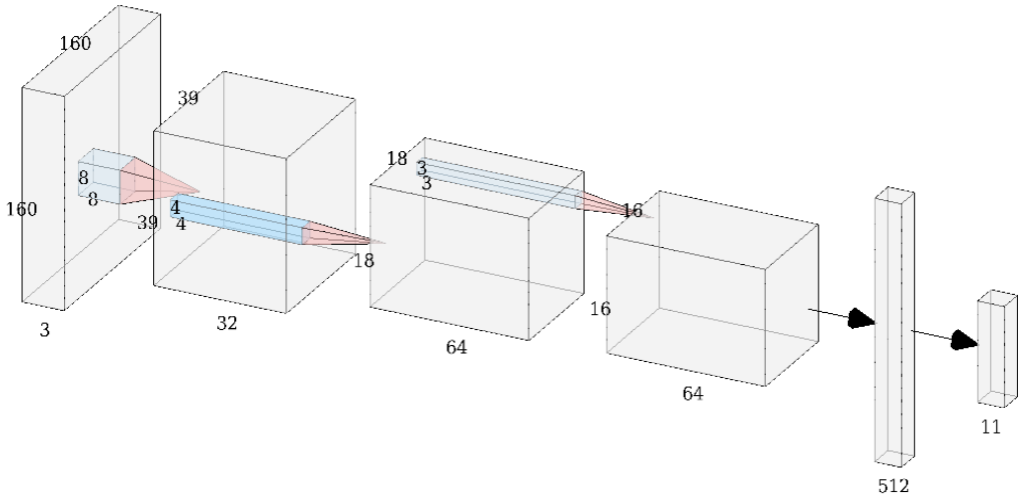
Continuous Action Space

Following the past failures, and after feeling more experienced with reinforcement learning frameworks and algorithms, I decided to change my strategy to train my model with a continuous action space.

At first, I tried to use the PPO algorithm with CNN policy

The CNN Policy was:

Activation function “ReLU”
Conv2d 32 8x8
Conv2d 64 4x4
Conv2d 64 3x3
Flatten
Fc 512
Fc actions number

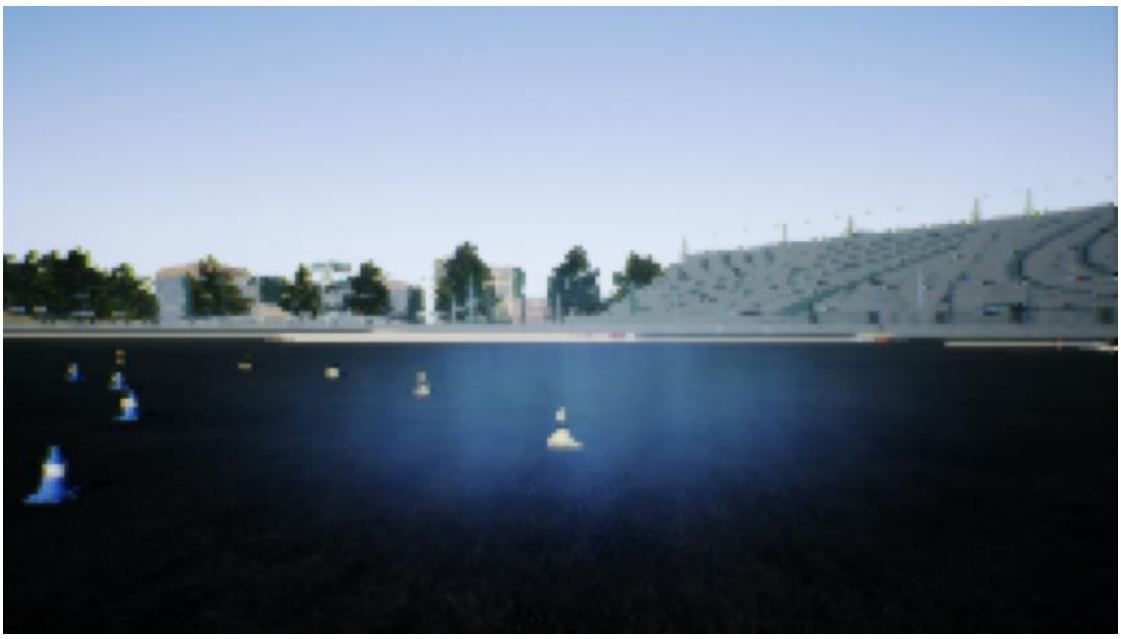


Unfortunately, the PPO algorithm did not perform so well

Afterwards, I tried to use the SAC algorithm, also with the same CNN policy.

Those efforts also were not very successful.

After further reading and advisement from my supervisors, I came to the conclusion that the problem might be in the observation space, as learning a policy from observation space of 160 X 160 X 3 is very difficult and very slow, and a reduction to the size of the observation space is required in order to improve performence.



Observation for example

Training a Variational Auto Encoder

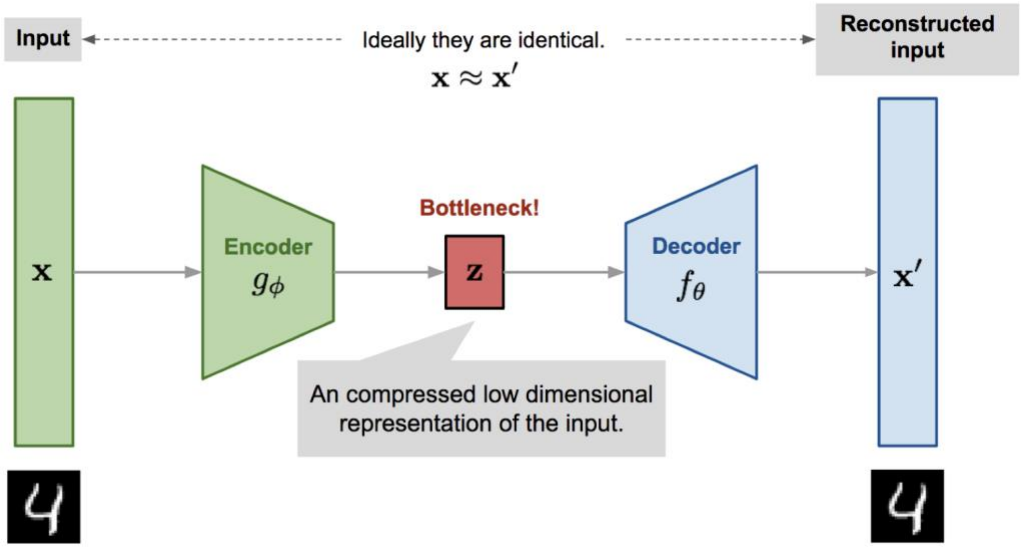
After many attempts to learn from raw pixels, I still couldn't achieve any performance that was satisfying, I decided to try a different approach.

First, I cropped the original image to get a cropped version of the observation of shape 80X160X3

Original Image *Cropped Image*

I decided to train a VAE (Variational Auto Encoder), to reduce the input images from 80X160X3 dimension to a single dimension vector of 512.

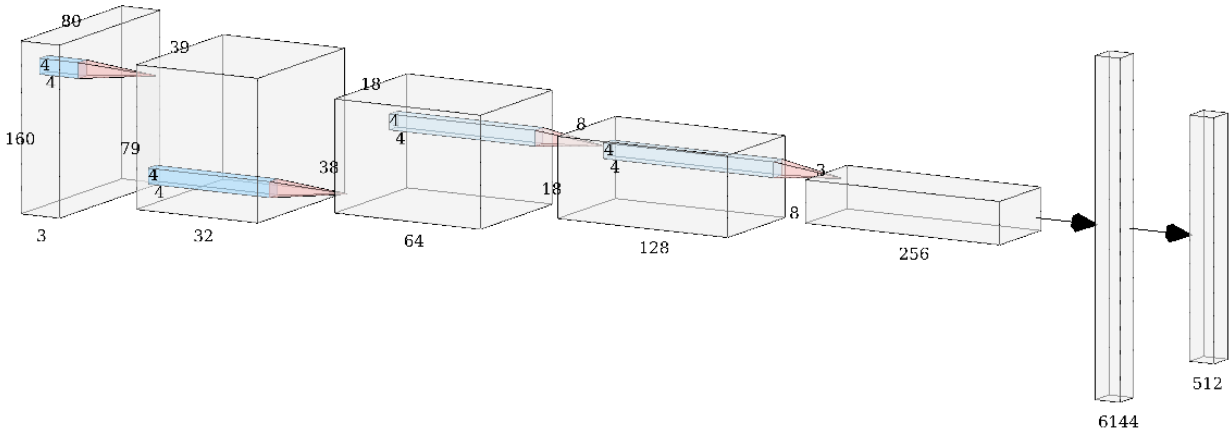
In order to do that, I defined a buffer that will store all the images seen by the car in a single episode, then, I trained the VAE on that set of images. The network of the encoder part of the VAE model was consisted of 4 Conv2d layers



Auto Encoder framework

Input was an 80X160X3 image
Conv2d 32 4x4
Conv2d 64 4x4
Conv2d 128 3x3
Conv2d 256 3x3
Flatten layer
FC 512

All conv2d layer had a strides number of 2, and an activation function “ReLU”.



The output of the network was a single vector of 512. Then, I concatenated to that vector the last 20 actions of the car, in order to improve stability of driving, and to avoid shaky driving when driving in a straight line.

In order to track the output images of the VAE I constructed a decoder part of the VAE model that had the same architecture, only backwards.

The images that came from the VAE were close enough to the original ones, which was very satisfactory.

That changed the observation space from 80X160X3 to a single 1-dimension vector of 532.

Then, I passed that observation to SAC algorithm, and trained the network with the SAC algorithm, and I was able to achieve quite good results, and almost succeeded to complete a full lap of the track.

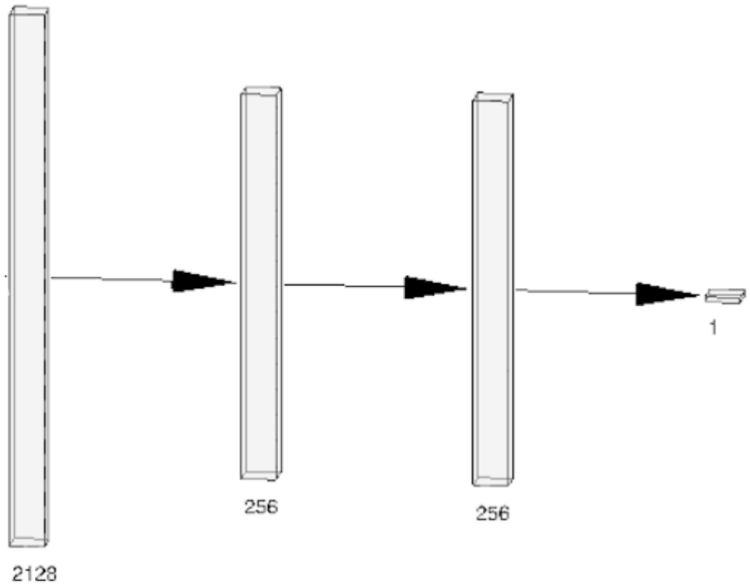
VAE output and input example

Final Model

Finally, after having successful attempts with the VAE approach, I decided to stack 4 observations, together, creating a 2-D observation space of size 4 X 532, that was the input to the SAC algorithm, with a network of two 256 layers.

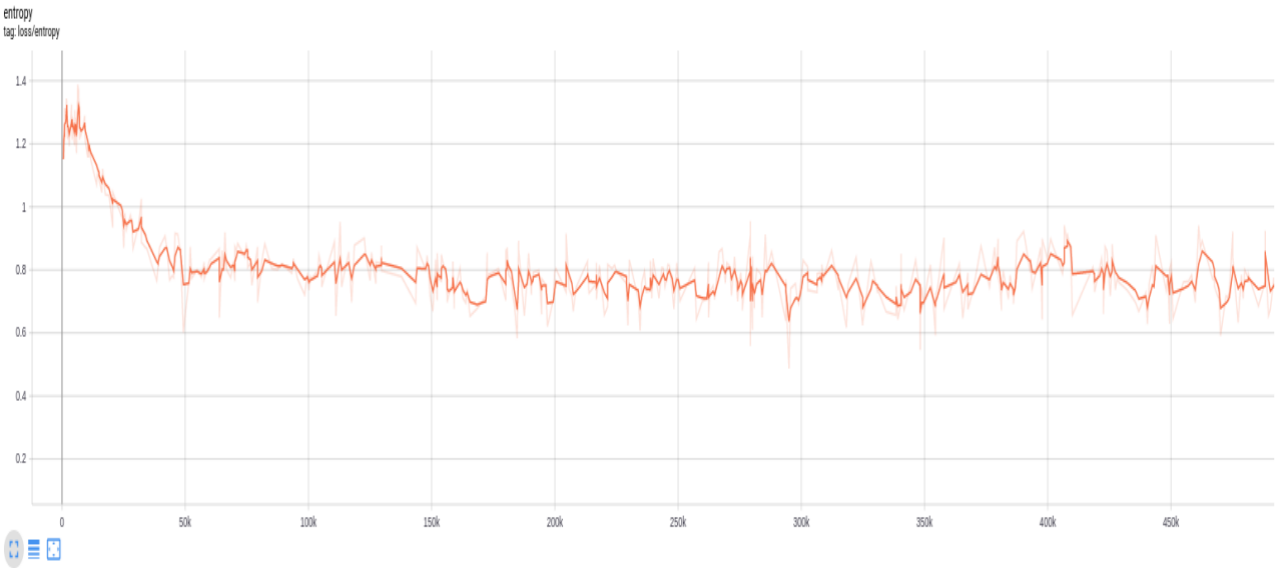
That attempt had the best results, as the model was able to learn very fast how to steer successfully through turns and curves, and completed a full lap in about 40k timesteps (about 1 hour), which was the best performance I achieved yet.

Then, I trained the model on more tracks, and after 3 or 4 episodes on completely new and unseen tracks, the model learned was able to complete a full lap on the new unseen track which was very impressive

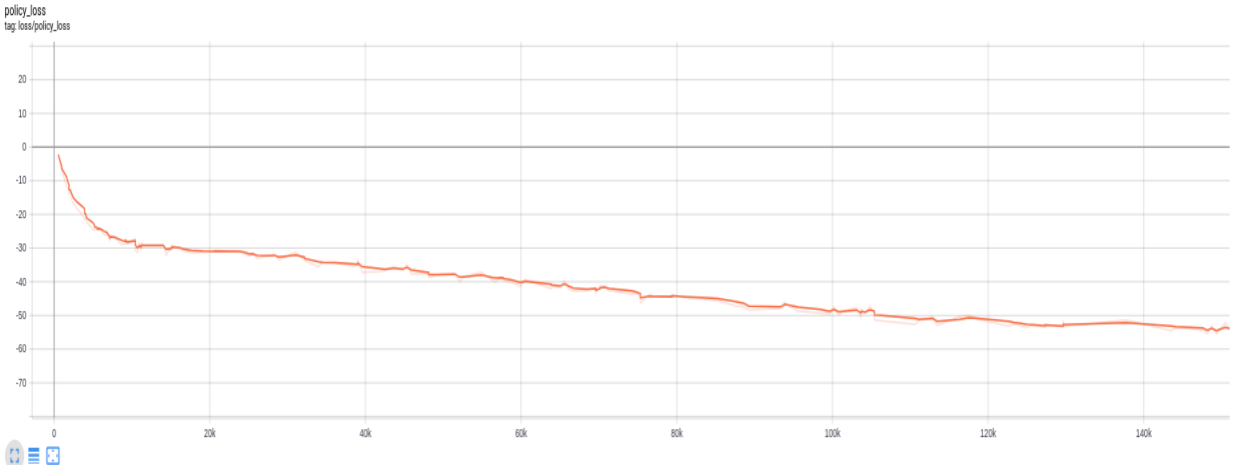


Final Model Architecture

Car succeeds to complete lap repeatedly and reward converges



Entropy changes throughout training



Policy loss throughout training

Conclusions

I started by taking the virtual environment that was built for the autonomous formula student competition, and I wanted to train an algorithm in Reinforcement Learning to self-steer the car through the cones on the racetrack.

In theory, Reinforcement Learning have the potential to perform better than human drivers.

In my work, I have tested both continuous action space models and discrete action space models. In my initial thought, I was certain that discrete action space models will outperform the continuous ones.

To my surprise, SAC, which is a continuous action space algorithm, had the best performance in all my experiments, and was able to complete a full lap in about 40k timesteps (about an hour) of an unseen track, and training from scratch. After training a model for 2,000,000 timesteps, the model performed very good on other unseen tracks, with 3 or 4 episodes only needed to complete a full lap on the new track.

I was able to save a model after finishing learning period of 2,000,000 timesteps (around 24 hours) on a certain track, then load it on another track, and continue the learning for another 2,000,000 timesteps.

I have repeated these steps on multiple tracks, and eventually tested the car on an unlearned track, and the car performed very well, succeeding many times to complete a full lap, only few times getting out of bounds or hitting a cone.

This would not have been possible unless the reduction of dimensions with VAE. With VAE, the results were tremendously better, both in learning time, and in the quality of driving.

The car was able to complete a lap in about few hours of training, while when learning from raw pixels, after 24 hours of training the car only learned a few turns, but could not complete a lap, which was very disappointing.

During my experiments, I have also tried to train a VAE of different latent sizes. I have found out that VAE with latent sizes lower than 512 were not able to see the next 3 pairs of cones, which was necessary to improve training time and driving quality.

When I used 512 as the VAE latent size, the encoded image could show most of the time the 3 consecutive pairs of cones, and that helped the car to plan steering for future turns and curves.

I have also come to the conclusion that the lighting of the environment highly affects the quality of the encoding in the VAE. I have trained the VAE on very specific lighting and weather scenarios.
Every change in the scenarios, or even reflection of the sun on other objects, reduces the quality of the encoding, and the car is not able to learn a good driving policy that way.
It is most needed to either train the VAE on multiple lighting and weather scenarios, or to maintain the same lighting and weather scenarios always in order to reproduce results across different tracks.

Concatenation of the last 20 actions proved to be very helpful in avoiding “shaky” driving, meaning when the car is driving at a straight line of cones, we expect it to drive straight, and not to steer endlessly between right and left. Adding the last 20 actions helped to lower the “shakiness” of the car and was able to achieve much smoother drive.

Throughout my model testing I have tried many network architectures, including multiple layers, and different layer sizes. Out of all the tests I have done, I could clearly see that deeper networks lengthened the optimization time that was done at the end of every episode and made the training process much longer in terms of wall time, however, it hardly impacted the quality of driving.

The car didn’t learn to drive properly faster, and it had no significant difference than a network with just 2 layers, which was a surprise to me.

The usage of frame stacking proved to be excellent. The idea is taken from Google DeepMind, where their engineers found out that it is highly difficult to let the network learn anything with movement only by supplying a single image, and by supplying 4 consecutive images, we can “teach” the network about movement in the environment and make the network learn much better about the world and the actions needed to be taken

Lastly, it was very clear to notice the “exploration-exploitation trade-off” problem in this project. It is very noticeable that the car sometimes acts completely random, by taking a hard right turn when it should keep straight for example. That is done in order for the car to explore more of the environment, searching from time-to-time different paths to go to, hoping to improve the reward and the quality of driving.

SAC algorithm handles the trade-off by the addition of entropy regularization parameter. Entropy is a measure of randomness of the car, how much is it likely that the car will act randomly in the next episode. This parameter is being changed constantly, allowing both the exploration of the environment from time to time, and also to exploit the learned policy and the car exploits its knowledge on how to drive and to steer in the current state and observation it knows.

Future Work

- Throttle control: The next big challenge will be to learn also the throttle control alongside with the steering control. For that there some changes need to be made to the action space, and of course to the reward function. However, if successful, this could lead to driving superior to human abilities, which will be fascinating to develop and test.

- Reward functions: I am certain that many reward functions can be used, and they might perform better, or worse than the one I have used. That could be an interesting experiment to see the difference in performance of my final model with different reward functions, both in driving quality and training time, as in how fast the car succeeds to complete a full lap.

- Frame stacking: I have tried to stack 4 images together. I am sure that different stacking options can be applied and there might be better options, which could be a good experiment to see how different stacking affect the results.

- Action concatenation: I have seen that concatenating the last 20 actions to the observation helps in smoothing the driving and improving the driving quality. However, I haven't tried other forms of concatenating last actions, and it's interesting to see if adding more or less actions will change the smoothness of driving.

- Improving VAE model: As I have mentioned earlier, I have trained the VAE on a specific lighting and weather conditions, which hardly resembles the everyday life weather and lighting. It will be an improvement to the entire model in my opinion, if the VAE will be improved and trained on different lighting scenarios and on different weathers, which might result in better encoding of each image and better driving.

- VAE size: Another Interesting quality that might be improved in the VAE model is the latent space size. Maybe a size larger than 512 might be more effective and might encode more detail that will improve the model, or it might also harden on the model to learn as the observation space will be larger. Also, maybe using a different methodology might be able to reduce the latent space to lower than 512, while still keeping a good representation of the original image, that could also improve the model tremendously.

- Network Architecture: The final model is just 2 fully connected layers of 256. I think that an optimization on the layer size or number of layers can be made to improve performance. Also, hyperparameter tuning might also help and improve the training process and might lead to a better driving quality and performance of the car.

Bibliography

1. AirSim – RL
   https://github.com/hoangtranngoc/AirSim-RL

2. All You Need to Know About Variational Autoendocer
   https://blog.bayeslabs.co/2019/06/04/All-you-need-to-know-about-Vae.html

3. Zadok, D.; Hirshberg, T.; Biran, A.; Radinsky, K. and Kapoor, A. (2019). Explorations and Lessons Learned in Building an Autonomous Formula SAE Car from Simulations. In Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH, ISBN 978-989-758-381-0, pages 414-421. DOI: 10.5220/0008120604140421
   https://arxiv.org/abs/1905.05940

4. From Autoencoder to Beta-VAE
   https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

5. Learning to Drive in A Day
   https://github.com/r7vme/learning-to-drive-in-a-day

6. Learning to Drive Smoothly in Minutes
   https://github.com/araffin/learning-to-drive-in-5-minutes

7. Microsoft AirSim
   https://github.com/microsoft/AirSim

8. OpenAI GYM
   https://gym.openai.com/

9. Stable-Baselines
   https://github.com/hill-a/stable-baselines