INFO8010: Project report: Car Racing using deep reinforcement learning

Loïc Champagne[1] and **Philippe Schommers**[2]
[1] *loic.champagne@student.uliege.be (s154314)*
[2] *p.schommers@student.uliege.be (s122617)*

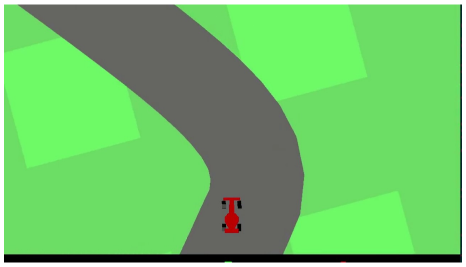


FIG. 1. Car racing game example

I. INTRODUCTION

As we are big believers in autonomous driving, we wanted to start learning the basic principles of training an agent that can drive a car. Of course, our project is nowhere near the realm of requirements a real car has, but we at least want to be able to navigate a simple 2D track. At first, we thought about working on the first Mario Kart game, but providing input and extracting a score proved quite difficult. So, we chose a game specifically made for AI research, for which all interfaces are available to quickly start solving the problem. This game is a variant of the OpenAI gym[1], that adds a few welcome improvements. Our project will consist in designing an agent capable of driving around a randomly generated track as fast as possible.

A. Environment

The game we are using to train our agent is a simple top-down 2D car game with procedurally generated maps. Those maps can be fairly straightforward with just a single route that the car needs to navigate, but can also be made more complex. Indeed, there is a fork available that makes it possible to add obstacles, junctions, intersections, and a bunch of other fun stuff. The training function is also easily changeable, meaning that we could train our model for different objectives. For example, we could give a big penalty if the car exists the road, or we could target fastest completion time, potentially allowing shortcuts.

In our project, we only used the classical map with no intersection, junction or obstacle, although it would have been a nice improvement to train our agent on harder maps, had we had more time.

The game can be configured with a few different input methods, and we decided on using five discrete inputs: turn left, turn right, accelerate, break or do nothing. To be precise, those discrete inputs are actually mapped to the continuous ones provided by the game. The game state can also be configured and we decided to use a stack of the last four gray scale frames of 96x96 pixels.

B. Objectives

The first objective, as suggested by the OpenAI listing, was to have a model that is simply able to drive laps on the simplest track possible (no junctions or intersections, no very steep curves, only one lane, etc). The car should be able to navigate that track easily, ending up with a positive score. Initially, the scoring function is defined as follows:

- -0.1 for every frame
- +1000/N for every track tile visited, N being the total number of tiles in the track

The problem is considered solved when the agent consistently gets at least 900 points, meaning that it needs to finish the game in a timely fashion, as points are lost over time. The game is considered finished when every single tile of the track was visited. This means that, while the agent is allowed to exit the track and to cut corners, it needs to do so while still visiting each tile, otherwise it will have to do another round around the track to find the missing tiles, losing precious points.

This reward function seems to be optimal, as it provides a dense reward environment, which should ensure that our algorithm will converge fast. However, the basic version of CarRacing-v0 doesn't come with early stops, which we added ourselves. Indeed, this basic reward function could cause our agent to take ages before finishing the course, which would be very time consuming to train. After careful thinking, we added early aborts that should not impact scoring in a meaningful way during training. The aborts go as follow:

- When the car gets out of the entire map (not only the track, the whole playing area): at this point, we consider the agent unrecoverably broken, as it cannot see any stretch of road, and thus can't know where to go.
- When there was no reward increase during the past two seconds of play time: this is generally the case

when either the agent is still fairly young and random, when it gets into a state where it doesn't move anymore, or when it missed a few tiles of the track by cutting corners. The latter is the most important issue we tried to fix with the reward function, as otherwise an agent could drive for around 10 000 steps if it failed one tile before timing out, because it would have to redo the whole track. This is not productive in any way, which is why we wanted to avoid it.

- When the total reward is smaller than -200: this is just as an effort to prevent random agents of taking too much time, but most of these aborts are covered by the previous rule.

All three of those early aborts also reduced the reward by 100, making sure that the agent wouldn't optimize too much on bad behaviors.

C. Action space

Although CarRacing-v0 is developed to have a continuous action space, the search and in general optimization is much faster and simpler in an environment with discrete actions. That is the main reason why we chose a custom engine instead of the one provided by OpenAI gym, limiting them to 5 discrete actions (Accelerate, Brake, Left, Right, Do-Nothing). It is important to have an empty action to allow the car to keep driving in a straight line at a constant speed. The environment doesn't have friction, so the car doesn't decelerate when it does nothing.

There are different ways to make the action space discrete. The one used by our custom engine is to directly map continuous actions to discrete ones. An action in OpenAI looks like $[s,\ t,\ b]$, where s stand for steering angle ($s \in [-1,\ 1]$), t for throttle and b for brake ($t,\ b \in [0,\ 1]$). We will use the following mappings:

Discrete action		**Continuous action**
Turn left	$\rightarrow$	$[-1.0,\ 0.0,\ 0.0]$
Turn right	$\rightarrow$	$[+1.0,\ 0.0,\ 0.0]$
Brake	$\rightarrow$	$[0.0,\ 0.0,\ 0.8]$
Accelerate	$\rightarrow$	$[0.0,\ 1.0,\ 0.8]$
Do nothing	$\rightarrow$	$[0.0,\ 0.0,\ 0.0]$

It is also possible to provide combinations of actions, like soft and hard variants, but we did not use them as we wanted to stay in a fairly simple scenario for our first Deep Learning project.

D. Observation state

The state of the game is simply the last four grayscale raw frames of 96x96 pixels. You might have noticed that the game is actually supposed to be colored, but we simply convert those RGB pixels into grayscale ones in order to simplify the neural network. For testing, playing and recording, the game is still colored, and of course of higher resolution. Therefore, the screen that we see is not the same as the one feed to the network. We also hid the information panel, normally displayed on the bottom of the frame, in order to just have the gameplay as observation state. Indeed, we believe that this panel would only act as noise for the network, and it is therefore better to remove it entirely before feeding it to the network.

We could have done both transformations based on the RGB frames received by the game using pre-processing, but did it immediately in the game as an optimization. Indeed, the game is really slow already, making it hard to train, so we didn't want to lose more time for basically nothing. We don't consider this cheating, as we didn't do anything we couldn't have done with pre-processing.

II. BACKGROUND

In this section, we will talk about everything necessary to grasp the content of this report. We will first talk about reinforcement learning and then focus more on the deep learning based approach to RL and finally we will explain PPO, which is the algorithm we used to try to solve this environment.

A. Reinforcement learning

Reinforcement Learning is an area of machine learning, aiming at learning the optimal behaviour in an environment which maximizes the accumulative reward. The decision-making process to control the environment is often considered to be a Markov Decision Process (MDP). More precisely, a MDP consists of a set of states S, which can be finite, infinite or even continuous, a set of executable actions A, which can be discrete or continuous, a transition probability $P(s_{t+1}|a_t, s_t)$ describing the dynamics of an agent interacting with its environment, and a reward function at a given initial, final state after taking a certain action $r(s_t, a_t, s_{t+1}) = \mathbb{E}[R_{t+1}|s_t, a_t, s_{t+1}]$.

Many RL problems follow the setup of a MDP, including the CarRacing task. Reainforcement learning algorithms aim at solving RL problems, which can either be model based, meaning that it leverages the underlying environment model for planning, or model free, meaning that its decision making process does not explicitly consider how the environment changes in response to an action. Model-free algorithms are generally preferred when it is difficult to obtain a sufficiently accurate environment model. The observations of CarRacing are composed of raw pixels, resulting in a complex environment that is difficult to model, hence model-free algorithms, such as actor-critic methods, are better suited.

The training of a RL algorithm can be done on-policy or off-policy. An off policy training is done on top of trajectories generated using a policy different from the current one, whereas on-policy training improves the current policy based on the actions it generates. Here we need to use on policy training, as we cannot generate a lot of experiments and the train the agent with previously generated data. The agent actually needs to interact with the environment directly to be able to extract information.

The direct objectives of RL algorithms generally fall into two categories: value function approximation and policy function approximation.

Value function approximation targets at estimating either the state values, V (or V_t at time t)–i.e. how desirable it is for an agent to be in a state, or the state-action values, Q (or Q_t at time t)–how desirable it is to take a certain action in a state. Their mathematical definitions are

$$Q_t = Q(s_t, a_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}} \left[\sum_{i \geqslant 0} \gamma^i r_{t+1} \right] \quad (1)$$

$$V_t = V(s_t) = \mathbb{E}_{a_t} \left[Q(s_t, a_t) | s_t \right] \quad (2)$$

where γ ($0 \leq \gamma \leq 1$) is the *discount factor* determining the importance of future reward to the current state values. Classic RL methods for value approximation include Monte Carlo methods for state or state-action value approximation, temporal-difference methods such as the off-line Q-learning and the on-line SARSA for state-action value approximation. Once a value approximator is readily trained, planning can be done via e.g. greedily selecting an action that is optimal based on the value approximator.

Policy gradient method, a gradient-based policy learning approach, directly optimizes on top of the action space. Intuitively, the policy parameter updates should encourage actions resulting in positive rewards and vise versa. Specifically, the Policy Gradient Theorem requires the parameters of the policy to be updated using the gradients with respect to the corresponding state-action value. One of the most commonly used policy gradient methods is REINFORCE, where the state-action value is approximated via long-term return. However, a major issue, as one can imagine, for policy gradient is its high variance. To address this, an unbiased state-dependent baseline is usually subtracted from the score.

Another way to reduce policy gradient variance is to simultaneously learn a critic that directly approximates the state-action values, arriving at the actor-critic methods. However, such an approximation can introduce bias. To tackle this, the (approximated) state-value, functioning as a baseline, is often subtracted from the approximated state-action value to form the policy gradient, referred to as the advantage function.

III. DEEP REINFORCEMENT LEARNING

Classic RL algorithms, despite their success with finite MDP and its mathematical rigorousity, experience limitations when facing large-scale problems. For example, many classic value-approximation RL algorithms require to maintain a value look-up table for each state or a state-action pair; however, when the state space grows, it becomes difficult to scale not only in terms of memory but also the excessive amount of samples needed in order to fill up the table relatively accurately–as each state or state-action pair needs to be visited multiple times. A step towards solving such scalability issues is, borrowing an idea from supervised learning to generalize with limited training data, function approximation for value and policy. Intuitively, given a previously unseen state, function approximators can achieve a reasonable estimation by generalizing from experiences on seen states with similar properties. However, due to computational constraints, most classic RL algorithms apply linear function approximation, and mostly on handcrafted features if the observation is of high dimensionality. These practices would clearly limit the capacity of a RL agent, which is especially problematic with a complex environment or task.

Thanks to the recent advances in computational power, tools in Deep Learning (DL) in replacement of classic function approximators have been utilized to overcome this limitation.

An example of a Deep RL (DRL) application is Deep Q-networks (DQN) where an observation is processed using a Convolutional Neural Network (CNN), the subsequent features are fed into a value function represented by a DNN, and the system is trained end-to-end following the theoretical principles of off-policy Q-learning. With the additional help of experience replay to decorrelate replay sampling–i.e. avoid overly similar examples for each training batch, superhuman level can be achieved for many Atari 2600 games. DNNs can also represent policies and be trained with either supervised objectives by transforming policy search or policy gradients such as using REINFORCE. DRL-based actor-critic methods use DNNs to represent both values and policies, where the actor is updated via policy gradients through the feedback from the critic. This family of methods has achieved great success across many RL applications, from mastering the game of GO to Robotics locomotion. As aforementioned, in this thesis, actor-critic methods serve as our algorithmic backbone and we focus on exploring DRL-based advantage actor-critic models.

It is also worth mentioning that many DRL models can seamlessly incorporate recurrent modules such as a long short-term memory unit (LSTM), to take into account longer temporal dependencies. That property is especially important when the environment is partially observable or if a long horizon planning is essential for an agent to excel. Although, the hidden units with compressed history leads to a non Markovian policy, more resembling the belief states of a partially observable MDP (POMDP). Alternatively, without using recurrent modules, a stack of consecutive frames can be used as input to engage more temporal information but on a much shorter term.

IV. PPO2

We decided to use Proximal Policy Optimisation 2 [2] (PPO2) for solving this problem, as this algorithm is very stable and has already proven to give good result in this kind of problem. As a matter of fact, PPO is really good in discrete environment which is exactly the case here. PPO is designed to be really easy to tune, which is great for our first hand-on experience with deep reinforcement learning. Furthermore, this algorithm doesn't need data of previous gameplay as it is a on-policy algorithm. This will help us a lot as we don't need to lose time in gathering data. Another positive point for PPO2 is that it can be adapted to be asynchronous, which gives us the possibility of multiple parallel environments to improve convergence by reducing the correlation between samples, which is a problem of DRL.

A. PPO2: details

PPO is a policy gradient method and a Q-learning algorithm, which means that the algorithm tries to approximate the Q function as shown above (see Equation 1), while policy gradient method means that PPO learns online[3] and does not use a replay buffer to store past experiences.

The Loss function used by PPO is a variant of the Policy gradient loss shown below:

$$L^{PG(\theta)} = \hat{\mathbb{E}}_t \left[log\pi_\theta(a_t|s_t)\hat{A}_t \right] \tag{3}$$

Where π_θ is the policy action, which is a neural network that takes the state as input and suggests an action as output, and $\hat{A}_t$ is an estimate of the advantage function which estimate the relative reward of the selected action. Intuitively, the Advantage is an estimate for how good an action is compared to the average action for a specific state. For more detail, $\hat{A}_t = Discounted\ rewards\ -\ Baseline\ estimate$ where the discounted rewards is:

$$Discounted\ rewards = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{4}$$

where γ is an hyperparameter that makes it possible to favor reward that the agent is able to get quickly to reward it can gain later, depending on its value and R_t is the reward at time t. While the Baseline Estimate is actually an estimation of the Value function (see Equation 2) which is obtained from earlier experience. The intuition behind this Loss is that if the advantage is positive, the gradient is also positive and thus the probability of such an action in that state increases.

But the PPO researcher did not use this loss, as getting good results via policy gradient methods is challenging because they are sensitive to the choice of step size — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking many millions of time steps to learn simple tasks.

In order to solve this problem, they created a new loss function inspired by TRPO[4], where the idea is to constrain the step size to avoid big steps that may cause catastrophic drops in performance. With their new approach for the loss function, they also solve the computational problem of TRPO, which is that the latter involves the calculation of the second-order derivative and its inverse, which is a very expensive operation. Hence to solve that they came with a new objective which is called Clipped Surrogate Objective. This objective is the following:

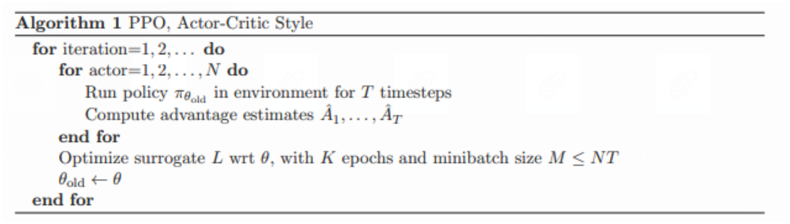
$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{5}$$

Now the objective function becomes a minimum between two terms, where the first term is $r_t(\theta)$ multiplied by the estimate of the advantage function. The goal of that is to avoid big changes in policy, making sure that the agent doesn't "forget" good actions because it explored too much. With

$$r_t(\theta) = \pi_\theta(a_t|s_t)\ /\ \pi_{\theta_{old}}(a_t|s_t) \tag{6}$$

this is the ratio between the current policy with the old policy. This permits to see by how much the new policy has changed from the old one, and it can be proven that $r_t(\theta)$ is equivalent to $log\pi_\theta(a_t|s_t)$ which comes from the Policy gradient loss (see Equation 3). PPO thus has to store two different policy networks. The second term of the minimum is a clip operation that prevents big step if the current policy network change too much from the old one as shown in `Figure` 5.

Furthermore, Most techniques for computing variance-reduced advantage-function estimators make use of a learned state-value function $V(s)$ and as they are using a neural network architecture that shares parameters between the policy and value function, they must use a loss function that combines the policy surrogate (see Equation 5) and a value function error term. This objective can further be augmented by adding an entropy bonus to ensure sufficient exploration, as suggested in past work. Combining these terms, they obtain the following objective, which is (approximately) maximized each iteration:



```
Algorithm 1 PPO, Actor-Critic Style
  for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
      Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
      Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \le NT$
    $\theta_{\text{old}} \leftarrow \theta$
  end for
```

FIG. 2. PPO algorithm

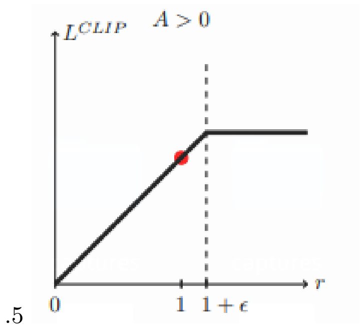


FIG. 3. In this case the advantage is positive and we want to augment the likelihood of this action but not too much to avoid taking a too big step that may lead to a bad result

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF} + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{7}$$

where c_1, c_2 are hyperparameters, S denotes an entropy bonus, and L_t^{VF} is a squared error loss $(V_\theta(s_t) - V_t^{targ})^2$. Finally, the algorithm of PPO is shown on `Figure` 2. We can see that this algorithm is divided into two steps, where the first one generates episodes by interacting with the environment and calculates the Advantage function using the fitted baseline estimate. Then, the second part of this algorithm is to collect this learned experience to perform the gradient ascent. This separation between actor and critic allows PPO to run asynchronously. Thus PPO can use multiple environment at a time to gather experience in parallel that will then be use in the second step.

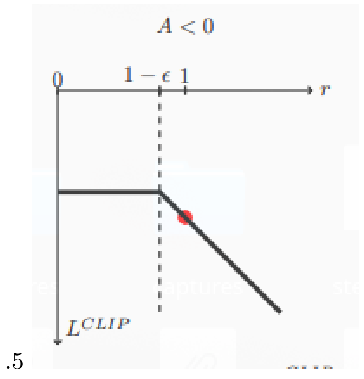


FIG. 4. In this case the advantage is negative and we want to diminish the likelihood of this action but we don't want the likelihood to fall to a probability of 0

FIG. 5.

V. GETTING STARTED

A. Setting up

First things first, we had to get the game running, and be able to run a simple Proximal Policy Optimization algorithm on it. This proved quite difficult, as OpenAI doesn't work correctly on Windows and has its quirks on MacOS. As we needed a fairly powerful system with a GPU, we decided to go ahead and install the latests Ubuntu version (20.04) on our dedicated system, as Ubuntu is generally recommended for Machine Learning tasks. In particular, TensorFlow only offers instructions for Ubuntu. However, neither NVidia nor TensorFlow have updated their software repositories for Ubuntu 20.04 yet, so we would have been better off with Ubuntu 18.04.

Then, we used Anaconda and created a dedicated environment. The game, OpenAI and stable baselines all have very particular required versions of various software, which we had to find out the hard way one by one. We tried to always use the latest supported versions of every software, in order to limit problems in a few months, when the whole ecosystem will have new breaking changes all over the place. The most important requirements are:

- Python 3.7.4 (newer versions don't work, we didn't bother checking older ones)
- TensorFlow 1.8 to 1.14 (we chose 1.14)
- Numpy 1.14.5

Here is a (hopefully) complete script to get the code to work, assuming that CUDA and all GPU related dependencies were installed by following TensorFlow's tutorial[5]:

```
$ apt install swig cmake libopenmpi-dev
↪  zlib1g-dev ffmpeg
$ conda create dl
$ conda activate dl
$ conda install python=3.6.4
$ conda install tensorflow-gpu=1.14
$ git clone
↪  https://github.com/filoozom/dl-gym.git
$ cd gym
$ pip install -e ."[Box2D]"
$ pip install stable-baselines
$ pip install tensorflow-gpu
$ pip install pillow OpenCV-python
$ pip install pyglet -U
$ pip install numpy==1.14.5
$ pip install optuna
$ pip install mpi4py
$ pip uninstall gym
```

As our code uses a local version of Gym, but imports always went to the version installed by pip, we removed it as a security measure. Basically, it either uses our version or throws an error because it doesn't find the package.

Despite carefully selecting every dependency to make sure that they would be compatible with the software we needed, we still got tons upon tons of warnings related to deprecations and incompatibilities. Ignoring them seemed to mostly work for us.

B. Forking the game

Before starting to work on the actual deep learning, we decided to fork the latest version of OpenAI's Gym and to merge CarRacing-v1 into it[6]. This allowed us to be sure that the code and its dependencies are up to date, and gave us a simple commit[7] we could look at to check all differences with the original CarRacing game. This was quite useful for debugging.

Then, we tweaked the code a little bit without changing the gameplay at all. For example, we hid the game windows during training, as they are a pain to deal with when they constantly close, open and steal focus.

The code also doesn't work correctly in grayscale, at least not when we only want one frame per state. The issue is that it entirely removes the 4th element (number of channels) of the NHWC data format, instead of setting it to one. The NHWC format used by the game is simply a shape that defines respectively: the number of images in the batch, the height of the image, its width and the number of channels of the image (so 3 for RGB, 1 for grayscale, X when we want X grayscale frames per state). This meant that our convolutional layers didn't work, as they expect the input shape to have 4 elements. The author did that to allow us to get multiple frames per state when using grayscale, but apparently forgot to add an exception when we want both grayscale and a single frame per state, which was going to be our basic test case. Basically, we had:

```
[None, 200, 300]
```

instead of:

```
[None, 200, 300, 1]
```

In the end, we decided not to work with CarRacing-v1, and rather ported a few features over to our custom fork of CarRacing-v0. There were a lot of issues with NotAnyMike's fork, some of which we already talked about, and we wanted to stay closer to OpenAI's intended environment. We actually lost a lot of time training our agent on CarRacing-v1 with undocumented functions, and then wondering why we couldn't get a score over 150. For example, the red and white stripes were removed from steep curves, the game stopped once half the tiles were visited, and the reward function was completely changed. We thus ported the following features to our own fork, based on OpenAI's CarRacing-v0:

- Grayscale observation space
- Multiple frames per state
- Discrete action space with the 5 aforementioned actions
- Early stops in order to speed up training, detailed in I B Objectives

C. Related work

There are a few papers out there that try to solve CarRacing-v0, but not a lot that do so using PPO. Still, there are always things to learn from related work.

A lot of our tweaks come from Mike Woodcock's blog[8]. He's the person who improved CarRacing-v0 and published CarRacing-v1 on GitHub, which we used at first before creating our own fork. As mentionned before, this newer version adds a lot of features to the game to make it harder and easier to work with. Sadly, it also changes the reward function as well as a bunch of other things we didn't want to deal with. Our goal was to keep it simple. Still, his work was very inspiring. For example, he created a discrete action space by binding it to the continuous one from the original game, which makes the environment easier to solve, as concluded by *Discretizing Continuous Action Space for On-Policy Optimization*[9]. We also used his idea of feeding not only the latest frame to the neural network, but also a certain amount of the previous ones.

The *Reinforcement Learning for a Simple Racing Game*[10] paper details a solution that uses transfer learning, which we thought about at first too, but didn't seem necessary in this particular case. While interesting and exploring a lot of different methods, this paper seems to only reach an average score of a bit more than 50 points, which is too low for our requirements.

VI. METHODS

In order to train an agent able to drive in this environment, we tried several different policy networks like the CnnPolicy of the baseline library, CnnLnLstmPolicy and finally our own implementation based on CnnPolicy, which is in turn based on NatureCNN. The Nature CNN architecture is shown on `Figure` 6. This architecture was used as policy network for PPO2 (see `Section` IV for more details on PPO).

The final parameters that we used for training are:

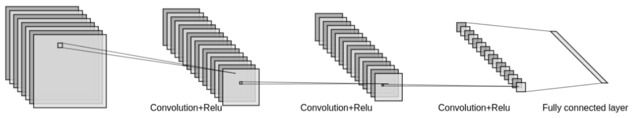


FIG. 6. Architecture of the Nature CNN

hyperparameter name	value
gamma	0.99
n_steps	32
ent_coef	0.01
learning_rate	0.00005
vf_coef	0.5
max_grad_norm	100
lam	0.95
nminibatches	4
noptepochs	4

Where gamma is the γ in equation 4 and its impact is explain bellow the equation, ent_coeff and vf_coeff are respectively c_2 and c_1 in equation 7 and n_steps is the N value in the PPO algorithm shown on `Figure` 2. Also, ent_coeff is the hyperparameter responsible to deal with the tradeoff between exploitation and exploration, where a high value favors exploration. Finally, n_steps is the number of steps to execute before updating the policy with the new knowledge.

As a matter of fact, almost all of these parameters are the default ones except two which are the learning rate and n_steps. We changed the learning rate because we tried several values for 100k steps and plotted the average score for each of these values and the value of 0.00005 had the best average score, although we are fairly sure that the change is only anecdotal. We also decided to diminish n_steps to better use our GPU and thus fasten the training, but it didn't really make a big difference.

We also tried to optimize those parameters with a grid search performed with Optuna, but the results with those parameters were again deemed anecdotal, and PPO doesn't rely too much on hyperparameters anyways. This is why we decided to mostly stick to the default ones.

VII. RESULTS

A. Qualitative

We trained our agent for 1.5 or 2.5 million steps at a time, and took some notes about behaviors we saw when testing the resulting model. This helped us identify issues, and made it possible to react by changing parameters or improving early aborts. At the end of our 18 million training steps, we saw that our agent was able to finish the track quite a few times, but still struggled in some specific cases. Videos of some successful runs can be found on our GitHub[11]. Alternatively, it should not be too difficult to create one using our CLI and weights. Simply run:

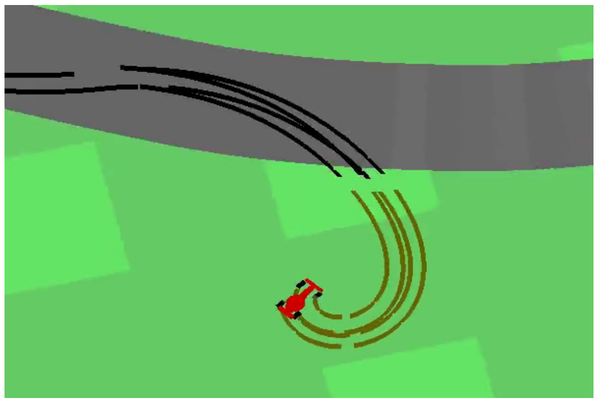
```
$ python cli.py video -c 1
```

and have a look at the "videos-tmp" folder in case the run was successful.

1. *Following the track*

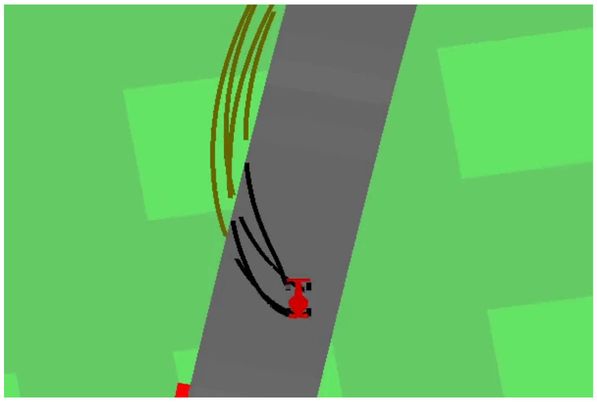
The agent is quickly able to follow the track. Indeed, it generally takes less than 100k steps – or around 10 minutes – for the agent to be able to navigate the track, until it inevitably slips due to a steep curve.

2. *Braking*



The agent has immense trouble to learn to break, and generally starts off driving at full speed for its whole lifespan, as it's the easiest way to gain points early on. This results in failures when taking any type of curves, resulting in going out of the track and slipping, generally in an unrecoverable way. One idea would have been to clip the reward gained by visiting a tile. Right now, the reward function gives a lot of points very fast for visiting more and more tiles, giving the agent the urge of driving really fast. This would however have forced us to alter the reward function in a way that would lose context with the initial reward function, so we didn't pursue that solution. We could have trained the agent on this particular reward function and then tested it on the original one, but feared that this would cause more troubles down the line, and deemed it not worth it. At some point after 5 - 6 million steps, the agent started to learn to brake in tight curves, which we attributed to both the red and white lines on the ground and to the 4 frames we use in our observation space.

3. *Going backwards*

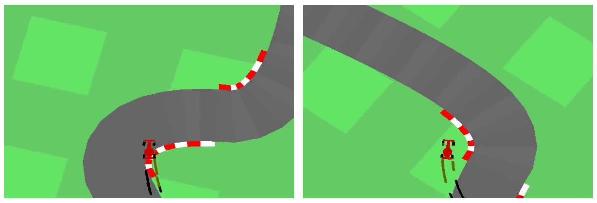


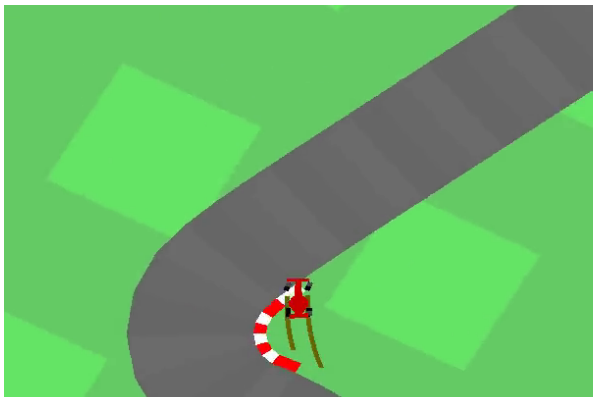
When the car misses a curve, it is sometimes able to recover after having slipped for a bit. Sadly, this often results in the car driving in the wrong direction afterwards, effectively tracing back its own path. When there are two steep curves on the track, this can happen twice, resulting in an endless loop of doing the same part of the track in both directions again and again. We added the 2 seconds without reward increase early abort to fight this issue, in order not to lose too much time training a bad state.

4. *Splipping twice*

The car is very prone to slipping when driving fast and turning at the same time. Again, the car didn't learn to brake for steep curves before half a dozen million steps, and thus mostly always ended up out of the track and slipping. Often, this ends up in the car going backwards, but sometimes it slips twice, effectively recovering and being able to resume its adventure. As the car is generally slower after recovering than before, it is more likely that it will be able to take the steep curve on the second try after a double slip.

5. *Cutting corners*





Once the reward function was tweaked a bit by adding early abort, the model started training faster, and the agent started to cut corners. This is perfectly safe and authorized, as long as it doesn't skip any tiles, and gives a speed advantage if executed correctly. The issue is that the agent generally does skip tiles, and is thus never able to finish the track, ending up with a sub-optimal score after the two second early abort. It did however learn to brake and take corners correctly towards the end. At that point, cutting still occurred, but generally only in complicated curves like the double curve represented earlier. Both pictures are actually just a few frames apart, and we can see that the second curve (the one being cut) immediately followed another steep curve.

6. *Safe driving*

This characteristic was only observed when training the agent in an environment where it wasn't allowed to exit the track. Basically, the agent would never really speed up, and try to stay as safe as possible, cruising the track at low speed. While this achieved the goal of finishing the track, it didn't do it quickly, and thus wouldn't win a race and doesn't satisfy the solve condition of reaching more than 900 points.

7. *Afraid of moving*

Similarly to the safe driving characteristic, this one was only seen when the agent wasn't allowed to exit the track. This caused the agent to basically freeze in steep curves and staying in place, instead of trying to take the turn. It just sits there, waiting to be killed by an early abort condition.

8. *Slow start*

This is the same as the agent being afraid of moving, except it is like that since the beginning when starting in any type of curve. When that happens, the agent takes ages to actually start up, losing precious seconds of time. At worst, it actually just tries to turn without accelerating, just staying in place until it times out. This is probably due to the fact that the agent learned not to accelerate in steep curves, as it causes slipping. Sadly, the agent does not make the difference between sitting still and having a certain speed in curves. This is the reason why we moved from using a single grayscale frame to using the latest four, which should allow to agent to have a sense of speed. Indeed, if the four last frames are identical, it means that the car is standing still. When the differences between the first and last images are big, the car is presumably driving at faster speeds.

9. *Final model*

In the final model, the most common issue was simply to miss a part of the track. This happens mostly because of cutting corners, but is also sometimes the result of a recovery. Indeed, it's great news when the car is able to recover, but sadly it almost always misses then tiles of the road when it was outside of it. It's thus never able to visit all tiles.

The agent is also still sometimes prone to slipping, but it's fairly rare and only happens on complex tracks.

B. Quantitative

1. *Benchmark*

Before getting into technical details, let us get a quantitative benchmark of our agent. We wrote a small script that evaluates the agent an arbitrary number of times, and calculates the percentages of successful runs and the average score. This script is included in our CLI, and can be ran like this:

```
# Calculates an average on 50 runs
$ python cli.py benchmark --count 50
```

Of course, the lower values aren't entirely accurate because of our early aborts, meaning that in most cases the agent should be able to finish the run with extra time, and in some it would get an infinitely low score if it froze. Of course, we capped the negative score at -200 in order to avoid an infinite loop.

Out of the 50 agents we benchmarked, 20 managed to finish the track with 900 points or more, giving a success rate of 40%. The mean score was 848 points. These results are very encouraging. As basically all runs between 800 and 900 points can be attributed to corner cutting, it means that basically no big errors were executed by our agent. To be precise, 35 agents finished with a score of over 800, and 48 over 750. The 3 worst scores were 631, 652 and 758, and the 3 best ones were 946, 943 and 942. The standard deviation was 80, meaning that the agent is fairly consistent. We also didn't have any outliers, meaning that no agent failed miserably.

2. *Training*

We trained our agent for 18 million steps total over a few days. This was done in two batches of 1.5 million steps, followed by 6 batches of 2.5 million steps. We will analyze the first two batches independently, as the first part of training is generally more interesting, and it allows us to have the same axes for each graph. Each graph contains all episode rewards in the background, and the darker line in the smoothed average. Stacking the lines instead of keeping a continuous line also makes them easier to compare. For each line except the very first one, the average is incorrectly displayed for the first 100k steps or so. This is because of the way TensorBoard displays the data when resuming the training of a model.

As should be fairly obvious on each first graph, the orange line is the very first training done on this model, and the blue line represents the second half of the first 3 million steps. On the second, the lines are in the right order too, so respectively red, light blue, green, grey, orange and dark blue.

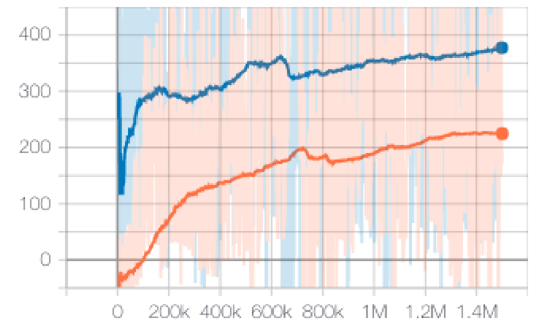
As a side note, training for 2.5m steps started to take 25% more time after the first 5.5m steps, and stayed relatively constant afterwards. This can be attributed to the fact that the agents started to perform better and that early aborts started to get triggered less, lengthening the time of completion.

3. *Episode reward*

See `Figure` 9. As we can see on the orange line, the agent is very quickly able to score positively. In fact, less than 100k steps were required before attaining an average of 0, and the agent was already able to average over 200 points after the first 1.5m steps. The model improved quite a lot during the next 4m steps too, more than doubling its average score. After around 5m steps, the improvements take way longer to occur, which makes sense, as optimizations to the model are more subtle and specific.

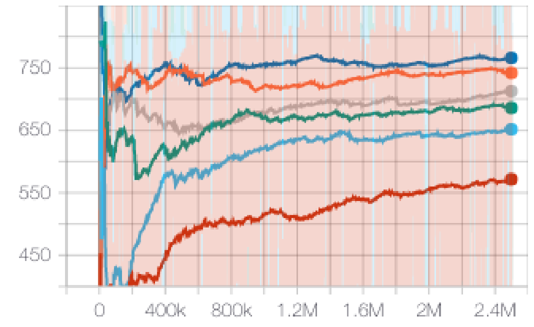
4. *Advantage*

See `Figure` 12. We can clearly see on this graph that the further we train our algorithm, the further the advantage tends to be 0. This means that that the baseline model becomes better at estimating the return of an action, which is indeed what we want.



(a)

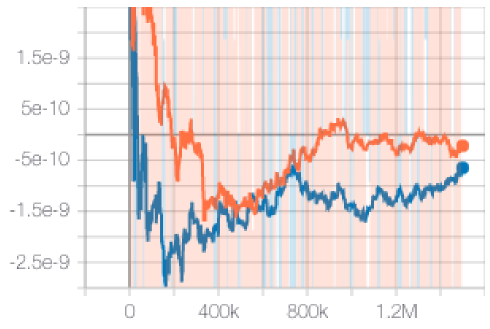
FIG. 7. Episode reward evolution of the first two batches



(a)

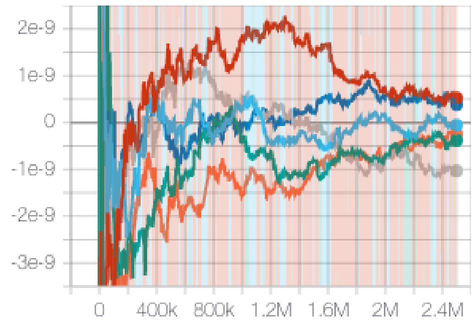
FIG. 8. Episode reward evolution of the other batches

FIG. 9.



(a)

FIG. 10. Advantage evolution of the first two batches



(a)

FIG. 11. Advantage evolution of the other batches

FIG. 12.

5. *Discounted reward*

See `Figure` 15. This graph confirms furthermore the conclusion made for the advantage. This seems obvious as the Advantage is calculated based on the discounted reward. We can see that the curve always goes up as the training goes on, meaning that the further the algorithm is trained, the better its action choice are as they bring back more points.

6. *Loss*

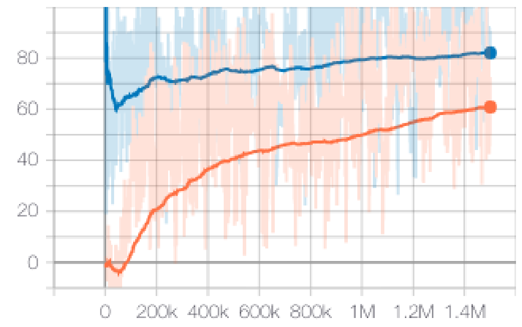
See `Figure` 18. It may seem strange at first glance to see a loss so high but you have to remember that we are in the case of reinforcement learning and thus the problem becomes a maximization problem and thus the higher, the better. Therefore we can see the improvements of our agent the further it is trained.

VIII. DISCUSSION

A. Performance

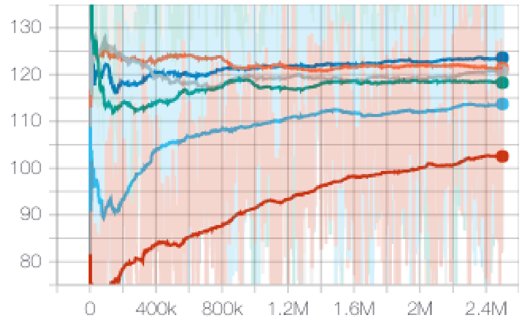
As stated previously, the goal of this environment is to achieve a score of 900 in average. This is only possible by finishing the track at a fast pace, as the most points to be won is 1000 when all tiles are visited, and every frame decreases the score by 0.1. If the agent doesn't complete the track, it loses 100 points, making it impossible to reach the passing score. Our agent reached a mean score of 765 points. This means that in most cases it can navigate the track really well, and it barely ever fails miserably anymore. Sadly, for a significant amount of tracks, the agent also skips corners, giving an early boost in score, but also making it impossible to finish the track with more than 900 points. In those cases, we kill the agent and decrease its score by 100 points.

This means that, while our agent didn't quite meet the environment's goal, it still learned to drive really well. It is actually still improving during training, but time and processing constraints didn't allow us to pursue training any further. Indeed, the agent only improves by a few points in average every hour, making the whole process quite long. We're also not entirely sure if the current configuration will make it possible to ever reach a perfect agent. We might need to add a penalty when the agent



(a)

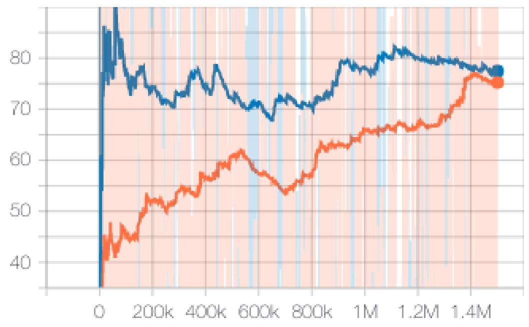
FIG. 13. Discounted reward evolution of the first two batches



(a)

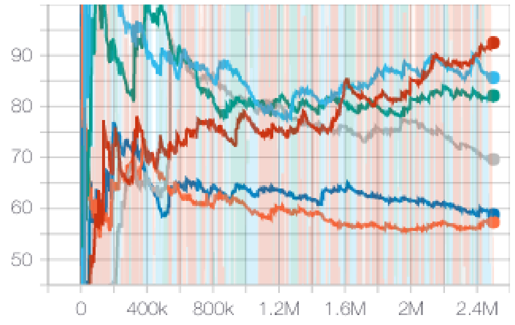
FIG. 14. Discounted reward evolution of the other batches

FIG. 15.



(a)

FIG. 16. Loss evolution of the first two batches



(a)

FIG. 17. Loss evolution of the other batches

FIG. 18.

skips parts of the track, but we're not necessarily big fans of that, as it would be too big of a hint, and might be considered cheating.

B. What we would have liked to explore further

As mentionned previously, we didn't quite manage to solve the environment in the strict sense of the term, as we didn't reach an average score of 900 or higher. Still, we think that we have a good basis and are already proud of our agent. We would have liked to explore a few options further, but as always with university projects, there's a balance to strike between time investment and depth of the work. Sadly, we weren't able to allocate more time and resources to this project, but still found a few leads for further improvements.

First, we want to try out the tweak to the reward function detailed in the previous section. Adding an early abort with a negative score when the agent misses a tile could be very interesting, and might improve the agent. However, this might not be optimal, as this would constrain the agent to specific actions we believe to be better, without letting it find the best ways by itself, which would also generalize better. This is a reward shaping issue, and we chose not to modify the original function except for adding early aborts. Given more time, we would definitely have tried this out, at least to compare performance to our current agent. This could also arguably be characterised as the same type of early abort we already implemented, and might be interesting for benchmarking either way.

Then, we would have liked to train the agent longer. At the time of writing, we trained the agent for 18 million timesteps, resulting in an average of 795 points. The average episode reward is still growing, but very slowly. We would have liked to train it for 30 or even 50 million timesteps, just to see the difference. It is probably not worth it, but the only way to find out is to try.

Finally, we would have liked to try out different values of the gamma value, which is the discount factor in the PPO algorithm. We trained our agent with a gamma value of 0.99, indicating that we prefer the reward we get immediately to reward we get later. This might be one of the reasons that cause our agent to skip corners, and thus miss tiles. Playing with hyperparameters in general could be beneficial, although PPO doesn't rely a lot on them in theory. As stated previously, we tried a few runs with Optuna, but the parameters didn't really end up making significant changes, at least not in the same order as our early aborts.

C. Possible improvements

Once the basic environment has been solved, it would be nice to play a bit with more complex environments. We could add obstacles and intersections, to further mimic actual car driving. Of course, it is going to be quite difficult to train the agent to take the shortest path, but it should at least be able to avoid obstacles and choose one path on junctions. Then, we could also start playing with the angle of curves, or place obstacles near curves to make the game harder. We could also try to simulate different weather conditions, making the car slip faster when it's wet for example. Basically, there's a ton of small tweaks we could add to make the game harder and test our model further. Of course, we would be able to use our current model as a base on which to build upon.

D. Tips for future work

It is essential to make sure that you understand the problem you are trying to solve before starting to work on it. We knew that we were working on a racing game with a given objective, but there are a lot of things we didn't know that slowed us down significantly. For example, we only understood the concept of missing tiles when the agent was actually able to drive a very decent track, but wasn't able to finish the game. The missing tiles are actually very slightly highlighted in the game, but it's hard to catch when not paying attention.

An other issue, as explained previously, is that we didn't make sure that the fork we were planning on using still respected the hypothesis of the original environment. We basically lost a week of work because the fork changed the reward function and added a strange early abort without us knowing and without it being documented.

In regards to actual Deep Learning, it is important to understand which algorithms suit your particular problem, and doing a few quick benchmarks is generally a good idea too.

IX. CONCLUSION

We used Proximal Policy Optimization in order to try to solve OpenAI's CarRacing-v0 environment.

Although we were not able to strictly solve the environment by scoring and average of over 900 points, we still got very close to it and learned a lot of things. This was a great first project in the world of Deep Learning. After 18 million training steps, our agent was able to solve 40% of tracks, and scored 848 points in average, which is not that far of from the 900 points goal.

[1] OpenAI Gym. `https://gym.openai.com/envs/CarRacing-v0/`.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[3] learns from what its agent encounters.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[5] `https://www.tensorflow.org/install/gpu`.

[6] `https://github.com/filoozom/dl-gym/tree/carracing-v1`.

[7] `https://github.com/filoozom/dl-gym/commit/925265b866346308bb4759a1c5bce694fae2d369`.

[8] `https://notanymike.github.io/Solving-CarRacing/`.

[9] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. *CoRR*, abs/1901.10500, 2019.

[10] Pablo Aldape and Samuel Sowell. Reinforcement learning for a simple racing game. `https://web.stanford.edu/class/aa228/reports/2018/final150.pdf`.

[11] `https://github.com/filoozom/dl-gym/tree/master/videos`.