Self-learned vehicle control using PPO

GROUP11

Oleguer Canal Federico Taschin
27th July 1995 14th June 1996
oleguer@kth.se taschin@kth.se

11th February 2020

Abstract

This work tackles the completion of an obstacle maze by a self-driving vehicle. We solve it combining two main ideas: First, we plan an approximate path running Dijkstra on the environment's Visibility Graph. Second, a fully self-trained agent using PPO (Proximal Policy Optimization) controls the vehicle making it follow the pre-computed path the fastest way possible. Results show a high degree of environment generalization achieved by training on randomized maps of increasing difficulty (Curriculum Learning). Furthermore, our data-driven control approach usually outperforms any of the other heuristic-based methods attempted in both maze completion time and natural driving feel, making us the team with lowest summed time added over all test tracks.

1 Introduction

Unmanned vehicles, such as self-driving cars and drones, are one of the biggest promises of recent developments in Artificial Intelligence. These autonomous agents could operate in open environments (cities, highways), semi-open environments (harbors, mines), and closed environments (factories, warehouses). Self-driving cars and trucks in open environments would decrease transportation costs, improve safety on the roads, provide a comfortable transportation mean to people. Unmanned vehicles in semi-open or closed environments can increase efficiency, reduce personnel costs, and operate in dangerous areas.

While these agents often have some knowledge of the area in which they operate (e.g. a street map, a plant of the factory), a common, desirable feature is flexibility: they need to adapt to changes in the surrounding environment, and operate in unexpected situations.

Motion Planning, the task of producing a sequence of steps from the starting point to destination satisfying the agent's physical constraints, is traditionally considered a central task in this kind of problems. A Motion Plan can have several degrees of detail, from specifying only a sequence of kinematic steps to specifying the full control output. Motion Planning requires a full physical description of agent dynamics as well as a full description of the environment in which the agent operates. These dynamics are often too complex or expensive to solve analytically and approximations are necessary. This inevitably leads to the accumulation of deviations from the planned trajectory, and requires an on-line control to compensate for it. For large environments or complex motion models, a motion plan can be expensive to compute. While a full knowledge of the environment can be obtained in closed or semi-open environments, it cannot be guaranteed in open environments and, in any case, it does not account for changes in that environment. Therefore, for any unexpected change along the trajectory planned by Motion Planning, such as a new obstacle, the agent needs to re-compute the motion plan partially or entirely. However, if the obstacle is unexpected, the agent would probably not know its full spatial configuration and dynamics, and the Motion Planning will be negatively affected.

In this work we completely discard the Motion Planning and approach the problem by using an extremely simple Path Planning combined with Deep Reinforcement Learning to learn control. Reinforcement Learning[19] is a branch of Machine Learning that studies how to maximize the utility of actions taken by agents when interacting with their environment. Deep Reinforcement Learning exploits Deep Neural Networks to perform control in complex action spaces, and has been proven to outperform humans in extremely complex games such as Dota2[16], Go[18]. Deep Reinforcement Learning is also showing promising results in complex robotic control, such as hand dexterity[15] and quadrupedal locomotion[20]. The training of a Reinforcement Learning agent consists in letting it play a large number of episodes and learning how to act in order to maximize the utility. We show that our agents are able to learn from experience and, most importantly, can generalize and perform well for environments they had not seen during the training.

1.1 Contribution

In our work, we show that agents –in this case a car and a drone in simulation– can learn to follow a path that does not take into account their dynamics by exploiting Deep Reinforcement Learning. This goal can be achieved with only a model of the agent dynamics and environment. Note that in robotics tasks such a model is almost always needed even for traditional approaches. Our agents are able to generalize and perform well in previously unseen environments. We eliminate the necessity of Motion Planning for driving tasks, which is often tedious and computationally expensive, as well as that of explicitly implementing hand-written rules for Path Following, therefore eliminating the need of complex control systems, which are often difficult to design and implement. Because we do not integrate agent dynamics when computing the path, the re-planning is extremely fast and can be computed in real time. As result, our agents are able to perform path following in a flexible manner, avoiding collisions and handling unexpected changes in their surroundings.

1.2 Outline

In Section 2 we give an overview of Motion Planning and Path Planning techniques, evaluate their strengths and weaknesses and provide a rationale behind our choice of only using Path Planning. Later, we introduce the concept of Reinforcement Learning and its theoretical grounds. In Section 3 we propose a method for expressing the problem of robot navigation in the Reinforcement Learning framework. We describe the learning process and some of the techniques that can be used to speed it up. In Section 4 we describe the evaluation environment of the agents and provide a comparison between Reinforcement Learning control and heuristic or Motion Planning based methods. Moreover, we also contrast our agent times against a human player.

2 Related work

Lot of work has been done in the field of robotics to obtain optimal paths or Motion Plans in the most efficient way. The traditional approaches are often based on search methods in the configuration space, and produce a Motion Plan that partially or completely fulfills the agent constraints.

2.1 Motion Planning

Any phisical agent is constrained by its geometry and the geometry of the environment. It is therefore useful to compute the *Configuration Space Obstacle*, the set of points in the *Configuration Space* that induce a collision between the agent and the environment. This can be purely geometric if the agent only performs translations or can take into account rotations.

Together with its geometry, robot's dynamics contribute to limit its movement. For example, a car is limited by its steering angle, maximum acceleration and velocity, grip, downforce, etc. Motion planning can therefore take some or all of these physical constraints into account.

Rapidly-exploring Random Trees[12] were first proposed by Steven M. Lavalle in 1998. RRTs can find a path from source to destination in a continuous multi-dimensional space, and are *"specifically designed to handle nonholonomic constraints (including dynamics) and high degrees of freedom"*. They explore the Configuration Space of the agent by sampling configurations and incorporating them if they are reachable from those in the tree. Although RRTs find a path -if it exists- with probability 1 in the limit, it has been proven[10] that in a wide range of circumstances it converges almost surely to a non-optimal solution. To overcome the limitations of the RRT algorithm, Karaman and Frazzoli introduced RRT*[11] in 2011. RRT* works in the same way of RRT, but adds optimizations on the tree pushing it towards an optimal solution with probability 1 in the limit. Despite these improvements, RRT and RRT* have limitations: they are much more expensive than path planning methods that do not take into account dynamics, and are not suited for real-time changes of source and destination. Moreover, they need to be fully aware of the environment at planning time, and for any unexpected obstacle in the way re-planning must be triggered. If the geometry of the obstacle is not entirely known, the quality of the plan is negatively affected.

A method that works in dynamic environments is RT-RRT* (Real-Time RRT*)[14]. RT-RRT* is based on RRT* and Informed RRT*[5] and introduces a tree rewiring technique that allows to move the tree root without discarding the sampled path, as well as dynamically change the destination configuration. RT-RTT* limitations are its high memory cost and the fact that it works only for bounded environments and performs poorly for large distances. In our navigation task, however, unbounded environments and long distances must be taken into consideration.

All the methods above do not answer all the requirements of many robotic navigation taks. First, computing a motion plan is often expensive. Second, the environment is often not known a priori in enough detail to produce a motion plan. Third, even when a motion plan is available, a on-line control is needed anyway to compensate for deviations. Moreover, a motion plan needs to be modified on-line whenever there is an unexpected change in the environment. For all these reasons, in our work we completely discard the Motion Planning, and show that comparable results can be obtained from experience with a minimal path planning and agent sensors.

2.2 Path Planning

In this context we define *Path Planning* as the task of producing a minimal set of intermediate points –we call them *checkpoints*– that connect the starting point to the destination. Path Planning does not take into consideration the agent dynamics and the resulting path could be infeasible if the agent is required to pass across the checkpoints exactly. In the first part of our work we require this path to be collision free, but in the last part we suggest that our approach can perform well even when this requirement is relaxed.

In the context of pure Path Planning, RRT and RRT* work in continuous space and can be used by removing all constraints on agent dynamics. When considering a discretized space, popular algorithms to find the shortest path are Dijkstra's algorithm[4] and A*[7]. Seeing grid cells as nodes in a graph connected to their neighbors by edges weighted by the distance from their centers, both Dijkstra and A* find the shortest path along non-occupied cells. The bigger are the grid cells, the more the computed path diverges from the optimal. A more efficient way of computing the Euclidean shortest path from source to destination in the Configuration Space is using the Visibility Graph. If corners of the obstacles are known, a visibility graph can be built by adding edges between *visible* corners, i.e. corners that are connected by a collision-free line. In our work we perform Path Planning by applying Dijkstra's algorithm to the Visibility Graph.

2.3 Reinforcement Learning

Our approach is based on the recent developments in Reinforcement Learning. Reinforcement Learning is a branch of Machine Learning that studies how decision-based agents can maximize the utility of their actions when interacting with the environment. In the Reinforcement Learning framework[19], an agent observes at time t a state $s_t \in S$, takes an action $a \in A(s_t)$, receives a reward R_t from the environment and moves to a new state s_{t+1} with probability $P(s_{t+1}|s_t, a_t)$, where S is the set of possible states and $A(s)$ the set of possible actions that can be taken in a state s. The agent choses actions by following a policy $\pi : S \times A \to [0, 1]$, i.e. $\pi(a, s) = P(a|s)$, the policy π induces a probability distribution over the actions for a given observed state. Problems in Reinforcement Learning are therefore modeled as Markov Decision Processes. The goal of Reinforcement Learning is to find a policy that maximizes the Expected Total Return G

$$G = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \tag{1}$$

where $\gamma \in [0, 1]$ is called *discount factor*. For a given state s and a policy π, it is defined the *Value Function* $V(s)$ as

$$V^{\pi}(s) = \sum_{a \in A(s)} \pi(a, s) Q^{\pi}(s, a) \tag{2}$$

where

$$Q^{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s') \tag{3}$$

$V^{\pi}(s)$ is the Value Function and represents represents the Expected Total Return G by starting in s and following the policy π. $Q^{\pi}(s, a)$ is the State Action Value Function and represents the Expected Total Return by starting in state s, taking action a, and following policy π. The goal of Reinforcement Learning is to optimize the policy π in order to obtain the higher expected reward G. If the a model of the environment is known, it is possible to directly maximize the equations above to obtain an optimal policy. However, when a model of the environment is not known, or the state space is too large, other methods must be used. In our work we use Proximal Policy Optimization[17], a state-of-the-art Policy Gradient Reinforcement Learning algorithm for continuous control tasks. It performs gradient steps to optimize the policy, a Neural Network to predict the value of the updates, and prevents policy updates to fall in low-reward regions by clipping the update.

3 Proposed method

3.1 Environment Setup

Since our path planning is straight forward and does not provide any significant contribution we will focus on the Reinforcement Learning methodologies implemented for vehicle control.
As explained in section 2.3, we frame the environment as a POMDP (Partially Observable Markov Decision Process) in which we defined a state, action and reward system (transition probabilities are given by the simulation). There exist multiple valid ways to build the state-action-reward space, nevertheless after some testing we opted by defining it as follows for both the car and the drone examples:

State: The partial observation of the environment by the agent is through the state vector. This means the state vector is the input which the policy function will map into action probability distribution. In our case the state is composed by the concatenation of the following elements:

- **Next n checkpoints relative position:** For the agent to learn what direction to go it is essential that it knows where the next n checkpoints are located w.r.t. itself. Latest trained models use $n = 4$. In order to efficiently drive through the maze it is critical to not only know the next checkpoint but also be able to anticipate and adapt the trajectory considering future movements. Working in relative coordinates (as opposed to absolute world coordinates) significantly eased the model training and generalization.
- **Relative velocity:** To capture the current dynamic state of the vehicle, the agent needs to know its velocity, both in the forward direction and in the lateral –to understand when its drifting.
- **”Lidar”:** These values represent the distance to closest obstacles for fixed directions in vehicle frame, simulating a lidar sensor without noise. Our tests 3.2 show that an agent aware of its obstacle surroundings outperforms a blind one. Latest trained model for the car uses 12 rays (9 in front and 3 in the back). Similarly, we used 24 rays evenly spaced for the drone.
- **Drone max acceleration:** In the case of the drone we are also appending its maximum acceleration to its state, since it was halved after every crash.

Actions We did several tests between discrete and continuous actions 3.2, obtaining the best results using a 2-branch continuous action space for both the car and the drone. This can be understood as a classification learning problem (discrete action space) versus a regression one (continuous action space). 2-branch means that 2 set of actions are independent of each other, for example the steering and the throttle.
In the case of the car we picked the following actions:

- **Throttle:** Continuous value between -1 and 1. 1 translates into going forward at full speed and -1 braking or going backwards.
- **Steering:** Continuous value between -1 and 1 where -1 is turning left and 1 turning right.

Similarly, for the done we defined the 2 actions branches as:

- **Move in x axis:** Continuous value between -1 and 1. 1 translates into full acceleration in the positive x direction and -1 in the negative.
- **Move in z axis:** Continuous value between -1 and 1, analogously defined as before with axis z instead.

Rewards It is often very tempting to over-engineer the reward system and introduce some heuristics to ease the training process. Instead, we tried to perform the minimal tuning that guaranteed a successful learning. For this, we introduced the following feedback signals:

- **Reaching checkpoint:** Latest trained models reward the agent with 1 point for passing through one of the checkpoints in its state checkpoint window. If it passed through a checkpoint more advance than the closest one, the reward also adds the sum of checkpoints in between. This is to enhance the agent to find optimal trajectories between checkpoint windows.
- **Time** To guarantee that the agent learns to complete tracks as fast as possible we also add a negative reward for each time-step the agent is running in a given environment.
- **Wall collision:** We add a minimal penalty (-3 for the car, -0.1 for the drone) for crashing against an obstacle. Experiments 3.2 showed that appending this reward made it easier for the agent to learn how to read the "lidar" data. Nevertheless, it is not essential for the policy learning and might even harm in some situations as maybe crashing with the car may result in a lower overall time.

3.2 Learning process

As the main idea of the project is to create an agent capable of generalizing to unseen environments, we implemented a random map generation routine on which to train. Early stages of the project showed that attempting to always train in the same map led into a good performance on that map but a clear overfitting problem. Moreover, experiment results 4 clearly show the benefits of applying a curriculum learning [1] approach. This is: start the training in easy environments and once the agent has mastered it, rise the environment difficulty sequentially. It has been shown [1] to fasten and improve the learning process. In our case, we control the difficulty of the maps using two variables: The proportion of terrain occupied by obstacles and, for the car only, the initial orientation. In early stages of the training, maps have few obstacles and agent starts facing the optimal orientation to complete the track, while final stages are characterized by high number of obstacles and random initial orientations.

Other attempted learning methodologies include Behavior Cloning (BC) [6] and Generative Adversarial Imitation Learning (GAIL) [8]. They both involve human demonstrations on how to behave given a certain state. For this, we recorded over two hours of human demonstrations on randomly generated maps for different state definitions and map difficulties. We did this by manually solving the mazes and recording the sequence of state-actions. Results show a benefit of using both strategies specially when no "lidar" data is included in the state information. Nevertheless, we opted by not using it in the last model as a fully self-trained model already presented a very good performance.

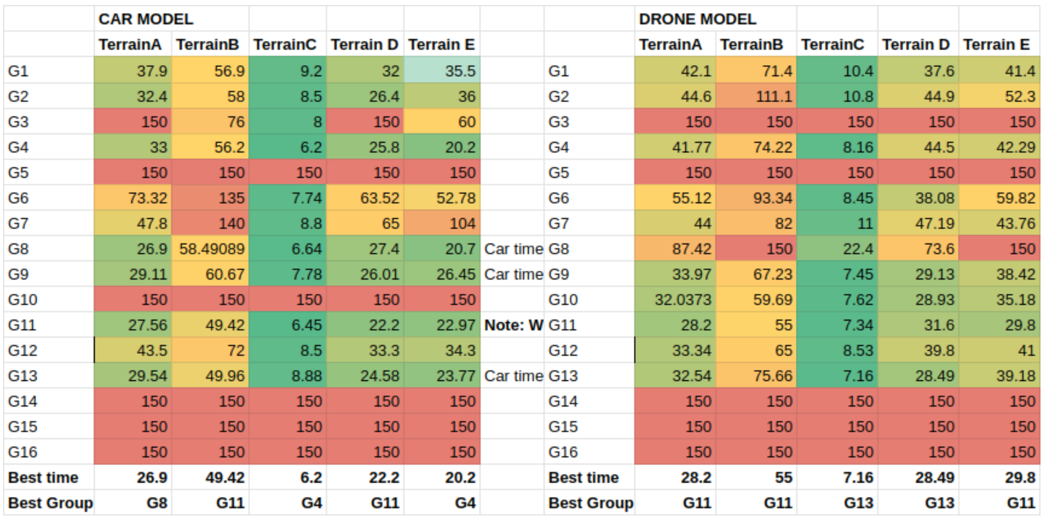
It is important to notice that the assessment of the impact of different state definition, learning techniques and learning meta-parameters is a high time consuming job. Consider that not only a single training on a regular machine takes around eight hours to give decent results, but also the effect of each parameter is tuned blindly. To cope with that kind of constrains we had to continuously run tests changing state definition and parameters in an efficient way. For this, we made use of the fractional factorial experiment design [13] running multiple training instances in different machines to later assess the effect of each variable in our case. In particular, aside from our two laptops we had two servers training models without interruption during the least two weeks.

3.3 Implementation

The implementation was done in Unity in combination with the open-source toolkit ML-Agents [9]. This framework allows to convert the world coded into Unity as a GYM [2] environment. This enables the use of already implemented Reinforcement Learning algorithms with Python such as the ones by OpenAI baselines [3]. In our case, we had to adapt the vehicle AI controller code to work as an ML-Agents agent. After that, we could build the Unity project and use that executable as an environment from where to take samples to train the algorithm.

4 Experimental results

We tested our system in a competition against other teams on three known and two unknown maps at the time of the development. It is important to remark that our agents had never seen any of the evaluation maps during training period as we intended to show its generalization ability.
Results show a better performance than most of the other teams as seen in table 1.



	CAR MODEL							DRONE MODEL				
	TerrainA	TerrainB	TerrainC	Terrain D	Terrain E			TerrainA	TerrainB	TerrainC	Terrain D	Terrain E
G1	37.9	56.9	9.2	32	35.5		G1	42.1	71.4	10.4	37.6	41.4
G2	32.4	58	8.5	26.4	36		G2	44.6	111.1	10.8	44.9	52.3
G3	150	76	8	150	60		G3	150	150	150	150	150
G4	33	56.2	6.2	25.8	20.2		G4	41.77	74.22	8.16	44.5	42.29
G5	150	150	150	150	150		G5	150	150	150	150	150
G6	73.32	135	7.74	63.52	52.78		G6	55.12	93.34	8.45	38.08	59.82
G7	47.8	140	8.8	65	104		G7	44	82	11	47.19	43.76
G8	26.9	58.49089	6.64	27.4	20.7	Car time	G8	87.42	150	22.4	73.6	150
G9	29.11	60.67	7.78	26.01	26.45	Car time	G9	33.97	67.23	7.45	29.13	38.42
G10	150	150	150	150	150		G10	32.0373	59.69	7.62	28.93	35.18
G11	27.56	49.42	6.45	22.2	22.97	**Note: W**	G11	28.2	55	7.34	31.6	29.8
G12	43.5	72	8.5	33.3	34.3		G12	33.34	65	8.53	39.8	41
G13	29.54	49.96	8.88	24.58	23.77	Car time	G13	32.54	75.66	7.16	28.49	39.18
G14	150	150	150	150	150		G14	150	150	150	150	150
G15	150	150	150	150	150		G15	150	150	150	150	150
G16	150	150	150	150	150		G16	150	150	150	150	150
Best time	**26.9**	**49.42**	**6.2**	**22.2**	**20.2**		**Best time**	**28.2**	**55**	**7.16**	**28.49**	**29.8**
Best Group	**G8**	**G11**	**G4**	**G11**	**G4**		**Best Group**	**G11**	**G11**	**G13**	**G13**	**G11**

Figure 1: Final times for maps A, B, C, D, E up to date on 11th Feb. 2020, 22:20. Our group is G11

4.1 Experimental setup

The car was tested on the five maps of Figure 2. Maps A, B, and C were known from the beginning and have been useful for the firsts evaluations of the agents (validation set). Maps D and E were unknown to the agent for the whole training (test set). These maps contained a wide variety of features such as straight lines, narrow corridors, tight curves, and complex shapes.

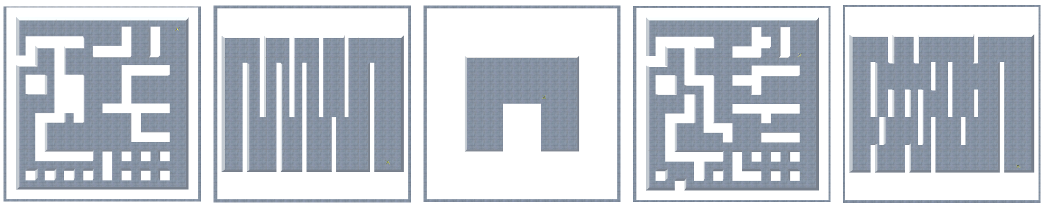


Figure 2: From left to right, maps A, B, C, D, E

4.2 Analysis of Outcome

By visually comparing the vehicle driving style against the other proposed heuristic techniques we can see a more "natural" feel to it. Figure 3 shows an example of driving path performed by the car and the drone. It is interesting to observe how the agents do not pass over all the checkpoints, but learned to skip them when not necessary. We regard this behavior as an important example of how Motion Planning can be learned on top of the Path Planning, without the need of explicitly program it. While other approaches present a string feel in its steering style, our learned method resembles that one of a human, only steering when taking a turn or intending to gain control. Furthermore, our agent is capable of dealing with unexpected situations in real time as is able to perceive its surroundings using the "lidar" information. Nevertheless, it still struggles when crashing or having to start or recover from bad orientations. We believe that with further training and parameter tuning this could be easily overcame.

Some other interesting behaviours observed involve: break before taking a turn or break when going too fast to react in case of obstacle. However, both car and drone still present important limitations. In the case of the car, in an attempt to complete the maze too fast it sometimes "jumps" key checkpoints so that it becomes trapped in a situation from which it does not know how to go back to main track. It is clear that it understands its environment but it still hasn't trained enough to know how to explore to discover an exit. Another possible improvement for the trap problem could be given by exploiting a Long-Short Term Memory (LSTM) network for the agent policy.

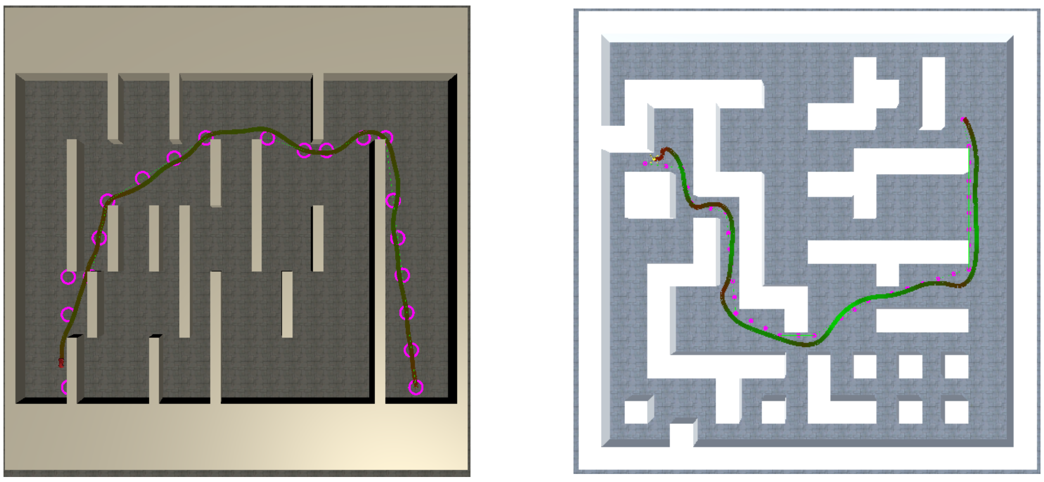


Figure 3: Paths performed by car and drone respectively. Green represents acceleration, red braking. Pink circles are checkpoints computed by Path Planning.

This would allow the agent to "remember" the obstacle structures and make it easier to find a way out. We believe that the use of a LSTM network would generally improve the performances of the agents or allow a reduction of the number of "lidar" points. However, it would probably make the learning slower. In the case of the drone, the main drawback we are facing is the fact that after a crash its maximum acceleration gets decreased. During the training process, crashing does not happen often and the agent is not used to the control change, making it perform worse after a crash. Moreover, this increases the probability of another crash which worsens the situation. We believe that further training would help solvent the situation.

In order to understand how our agents perform with respect to a human player, we played both the drone and the car in all the maps and keep our best time. Results can be seen in Table 1. It has to be noted that a human player has a great advantage: while the agents can see only the next 4 checkpoints and the lidar measurements, a human player can see the whole video frame and the minimap. Despite this advantage, our drone agent performance is comparable to a human. The car agent is, however, slower than a human in all maps.

5 Conclusions and further improvements

In this work we prove the effectiveness of Reinforcement Learning in continuous control tasks. We propose a training method that allows a dynamically

Map	Human Drone	Agent Drone	Human Car	Agent Car
A	31.4	**28.2**	**17.7**	27.56
B	DNF*	**55**	**47.5**	49.42
C	**6.6**	7.34	**5.67**	6.45
D	**25.1**	31.6	**17.3**	22.2
E	**29.3**	29.8	**16.8**	22.97

Table 1: Respectively: drone times for human player and agent, car times for human player and agent. *In map B the human player with the drone Did Not Finish the track in the maximum limit of 60 seconds.

constrained agent to perform effective control without needing a motion plan. Compared to other solutions, summed up in figure 1, our approach performs better overall. Moreover, despite not having been trained for this specific purpose, our agents are capable of avoiding obstacles that were unknown when the path plan was computed. The control is however not always perfect leading to some crash episodes in the drone, whose dynamics are more complex. Both agents suffer from the "trap" issue: when an unexpected obstacle is too large they remain stuck in indecision. This suggests the use of a LSTM network instead of the current Multi Layer Perceptron, in order to give the agent a "memory" that could allow it to overcome this situations. A crucial issue in the proposed method is the difficulty of tuning, since trainer and environment hyperparameters are high in number, and trainings are slow.

References

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, pages 41–48, 2009.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[5] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. Informed rrt*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic. *CoRR*, abs/1404.2334, 2014.

[6] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. Integrating behavior cloning and reinforcement learning for improved performance in sparse reward environments, 2019.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[8] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning, 2016.

[9] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. sep 2018.

[10] S. Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. 06 2010.

[11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[12] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[13] Douglas C Montgomery and John Wiley. *D esign and Analysis of Experiments Eighth Edition*. 2013.

[14] Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. Rt-rrt*: a real-time path planning algorithm based on rrt*. 11 2015.

[15] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

[16] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 2019.

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

[18] Schrittwieser J. Simonyan K. et al. Silver, D. Mastering the game of go without human knowledge. *Nature*, 2017.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[20] Vassilios Tsounis, Mitja Alge, Joonho Lee, Farbod Farshidian, and Marco Hutter. Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning, 2019.