Point-to-Point Navigation with Deep Reinforcement Learning

Paul-Antoine Le Tolguenec[1,✉]

[1]ENSTA Bretagne, Brest

Nowadays the problem of the autonomous vehicle is one of the most developed subjects in the world of research. Although there are different methods, methods based on deep learning are more and more used in this field. This article illustrates a solution based on Deep Reinforcement Learning.

Deep learning | Reinforcement | Control | Autonomous car

Correspondence: ***paul-antoine.le_tolguenec@ensta-bretagne.org***

Introduction

In recent years, mobile robotics has experienced unprecedented growth. Most of the methods concerning trajectory estimation are based on automatic laws. A very robust method is the linearizing loop control method. But in some cases this method has to be reworked, because it has to be linearised around a working point or other. Another very robust method is that of potential fields, since it also allows to avoid certain obstacles that could appear spontaneously on the way to the robot. But for this type of method it is often necessary to implement other layers that allow to find the path to reach a point in the defined space. Most of the time it is also necessary to develop a finite state machine to manage the high level of the robot. Usual methods are therefore very time-consuming to implement and can lead to mistakes. Moreover, once the control of the robot is set up, optimisation is difficult. With the progress of deep learning, a new control approach has emerged: deep reinforcement learning. The advantage of this method is that once the method is set up it can be generalised to many problems and it is only necessary to change certain parameters. Also this method is permanently optimized. In this article, I expose my work which has allowed me to train a model to make decisions to orientate a car so that it is able to go from point A to point B as quickly as possible by avoiding objects (2) that appear as they go along. To train the model I use the A2C (Advantage Actor Critic) algorithm which uses the principle of reinforcement learning. Part II illustrates the modelling I have carried out and within which I have trained the network. Part III presents the implementation of the method (network architecture, theory ...). Finally, part IV illustrates the experimentation phase.

Modelisation

A. Formalisme. The objective of our work is to automate a car so that it is able to go from point A to point B while avoiding obstacles. The kinematics of the car is modelled by the following equations:

$$\begin{cases} (i) & \dot{x} = v\cos\phi \\ (ii) & \dot{x} = v\sin\phi \\ (iii) & \phi = u_{\pi_\theta} \\ (iv) & v = k \end{cases} \quad \textbf{(1)}$$

At first, I tried to control only the steering angle of the car. Thus u_{π_θ} represents the ϕ output taken via a policy π_θ. The robot is also equipped with three sonars, one frontal and two lateral, which allow to visualize its environment.

B. Simulation. Once the kinematics were defined, I had to simulate the robot and obtain a visual rendering. I used the Kivy library of python. Kivy is a free and open source Python framework for developing mobile apps and other multitouch application software with a natural user interface. The objective was to have a visual rendering of the robot's evolution, but also to be able to interact with the environment via a man-machine interface. Once the simulation is launched, a window opens and the robot evolves in its environment. The operator can interact with the robot's environment by creating walls that he can draw with the mouse. There is also a button to erase all the walls on the map to rebuild others and see if the robot is able to generalise its behaviour in any type of environment.

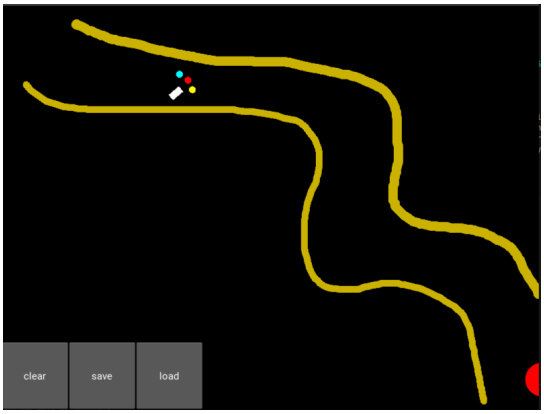

Fig. 1. Simulation of the system

As can be seen in the figure. The car is represented by a white square. The sonars of the car are represented by the three circles at the front of the car.

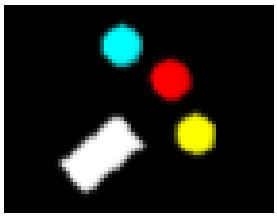

Fig. 2. Robot sonars

The walls are represented by the yellow lines. And the objective of the robot is represented by the red dot. At first, the robot can cross walls but when it does so its speed decreases and it gets a negative reward because it has crossed the wall and because it will arrive less quickly at its target.

Proposed method

The objective of our work is to train an agent to be able to choose the orientation angle of the car according to his perception of the environment.

C. network structure. The structure of the network is quite simple. The network takes as input: the distances output by the sonars, and the angle formed by the direction of the robot and the line passing through the centre of the robot and target. To simplify the teaching process, we have discretised the robot's exit space. The robot can make three decisions: do nothing, increment its angle by +20 degrees or -20 degrees. Of course, making the exit area discreet means poorer results in the long term. But it allows for a quicker convergence towards a more stable policy (1).

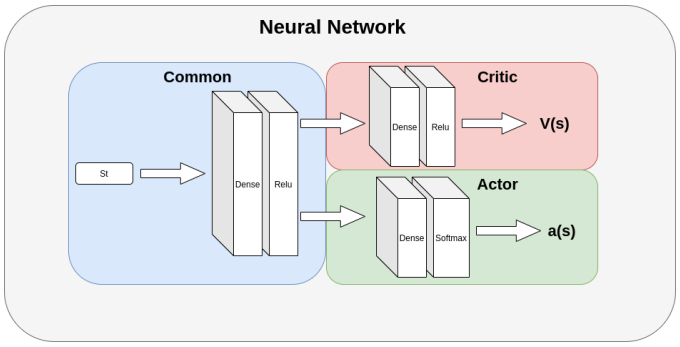

Fig. 3. Neural Network Structure

As you can see the network is divided into two parts(6):

- Critic

  The Critic part is used to determine the value of the state in which the agent is in. This part corresponds to the value-based part of the learning. It allows to stabilize the learning and to converge towards a global maximum of the policy.

- Actor

  The actor part corresponds to the part of the agent that takes action. It is the policy-based part of learning. The actor part pulls out a probability distribution $\pi_\theta(s)$ which makes it possible to determine what probability is for the agent to take this or that action given the state.

The advantage of such a method is that a continuous space of action can be used.(8) Unlike conventional Deep Q-Learning methods.(9) The problem is that algorithms such as the DDPG (Deep deterministic policy gradient) which provide a continuous policy space do not guarantee to find the optimal policy for a given network and MDP, which is the case for a stochastic critical actor. Moreover, they introduce many problems that make them difficult to converge.(4) That's why we discretize the space in this problem. Here he purpose of learning is to change the probability distribution in the direction that gives the maximum reward.

The value part allows to compare the quality of an action carried out in relation to the value of a state already estimated to know if this action is more optimised than what has been done before.

D. Advantage Actor Critic algorithm. To update the network parameters we used reinforcement learning. We used one of the most powerful reinforcement learning algorithms before the A3C. A2C is an algorithm halfway between value based learning and policy based learning.

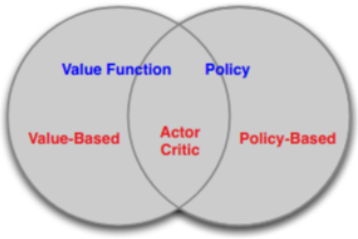

Fig. 4. Actor Critic

The Advantage Actor Critic algorithm takes the following form :

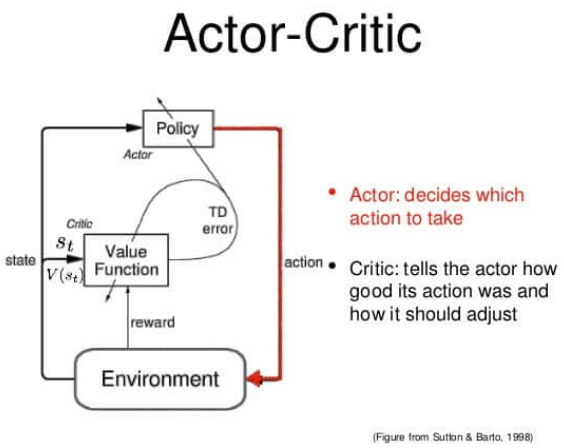

Fig. 5. Actor Critic

As can be seen in figure 4, the principle of the critical actor is

Algorithm 1 N-step Advantage Actor-Critic

procedure N-STEP ADVANTAGE ACTOR-CRITIC
  Start with policy model π_θ *and value model* V_ω
  repeat:
    Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$
    for t *from* $T-1$ *to* 0:
      $V_{end} = 0$ if $(t+N \geq T)$ *else* $V_\omega(s_{t+N})$
      $R_t = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k (r_{t+k}$ *if* $(t+k<T)$ *else* $0)$
    $L(\theta) = \frac{1}{T}\sum_{i=0}^{T-1}(R_t - V_\omega(S_t)) \log \pi_\theta(A_t|S_t)$
    $L(\omega) = \frac{1}{T}\sum_{i=0}^{T-1}(R_t - V_\omega(S_t))^2$
    Optimize π_θ *using* $\nabla L(\theta)$
    Optimize V_ω *using* $\nabla L(\omega)$
end procedure

to take an action thanks to the actor and to criticise the action achieved thanks to the critic. But to criticise this action you need something to compare this action with the critic. This is the Advantage part. There are several ways of expressing Advantage. But the most common method is to take as the advantage : $Rt - V_{\pi_\theta(s,a)}$

with $Rt = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k (r_{t+k})$.

We have used the Monte-Carlo evaluation. The Monte-Carlo method involves letting an agent learn from the environment by interacting with it and collecting samples. This is equivalent to sampling from the probability distribution P(s, a, s') and R(s, a).

However, Monte-Carlo (MC) estimation is only for trial-based learning. In other words, an MDP without the P tuple can learn by trial-and-error, through many repetitions.

In this learning process, each "try" is called an episode, and all episodes must terminate. That is, the final state of the MDP should be reached. Values for each state are updated only based on final reward Rt, not on estimations of neighbor states — as occurs in the Bellman Optimality Equation.

MC learns from complete episodes and is therefore only suitable for what we call episodic MDP.

Here it's not the case, but we are going to use the same method to take into account more information.

As can be seen in figure 5, the advantage part of the algorithm can also be called the TD error.

In the literature one initially finds $Rt = \gamma V_{t+1} + r_t$ but recent work on replay experience has shown that taking into account more rewards (like the monte-carlo evaluation) makes it easier to converge towards a global minimum of the cost function estimating the MDP (Markovian decision process). On the other hand, using this method implies a certain volatility of the estimator since a stock affects the value of a state further in time, and it often takes longer to converge. It is therefore a question of finding the best hyper-parameters, which are often found empirically.

E. Reward process. In reinforcement learning methods the reward process is very important as it allows the robot to perceive its environment. More importantly, it corresponds to the MRP (Markovian reward process) that we try to estimate thanks to the neural network (the critical part). So if the reward process does not correspond exactly to the environment, the agent will not correctly estimate the behaviour it has to adopt.

$$\begin{cases} R_t = -2, \text{ if } None(\text{life penalty}) \\ R_t = +1, \text{ if } distance_t < distance_{t-1} \\ R_t = -5, \text{ if } x, y \in \Omega_{sand} \end{cases} \quad \textbf{(2)}$$

The life penalty, allows the agent to get out of certain situations such as when he comes into contact with a wall and gets stuck. If a negative life penalty is not applied, the robot may get stuck in its situation.

Experimental discussion and results

The neural network used is quite simple so the learning process is not long but the final results can be improved. To evaluate the quality of a reinforcement learning project, it is not enough to look at the final quality of the model. It is necessary to evaluate the training time, the response time of the system, and to determine on parameters such as the learning curve if the model can still learn or if it is not the case.

Learning. It is difficult to visualize neural network learning for reinforcement learning problems. Especially when using a critical actor. In fact, if the error on the critic decreases it does not necessarily mean that the actor learns correctly. But it is really the actor who interests us. The error on the actor can be interpreted but it does not allow us to visualize the quality of the policy used by the agent at a given moment t. To visualize the evolution of an agent in his environment, we must take into account the number of rewards he acquires over time.

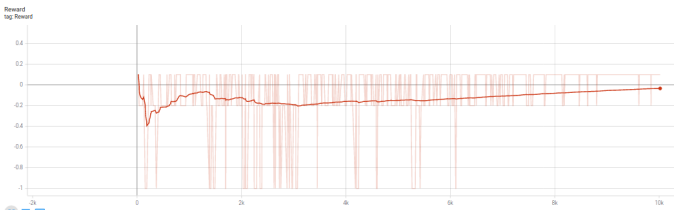

Fig. 6. Rewards curve

As can be seen on the reward curve the model converges to an acceptable policy after 10K iterations, which is about 5 minutes. We can consider that the model learned very quickly. The environment is quite simple since at first we don't use walls.

Obstacle avoidance. Our initial goal was to create an agent capable of avoiding all the obstacles he could find. With the chosen network and the established configuration our agent is able to reach his target avoiding most of the obstacles.

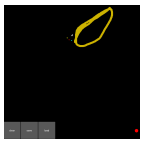

Fig. 7. Avoiding one Obstacle

Even following a path.

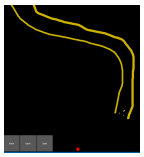

Fig. 8. Avoiding one Obstacle

However in some configurations the robot is sometimes forced to cross the wall to reach the target.

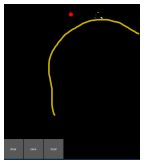

Fig. 9. The U problem.

So if the model presents very good results, it is still perfectible. To check the results go to : Actor-Critic.

Prospects for improvement. As the results obtained are the result of a detailed search for the best hyper-parameters of the model (learning rate, gamma, etc.), there are three ways of improving the agent's performance:

- **Building a deeper network** One of the possible reasons is that the network that has been built is not sufficient to find the most optimal policy in this Markovian decision process problem. One of the solutions is therefore to create a deeper network and thus to add layers of neurons.
- **Add an lstm layer** LSTM (7) are recurrent neural networks. They make it possible to store information in memory. So in case the robot is stuck in a corner, it would be able to know where it comes from. For the moment it only goes with what it sees. Keeping information in memory could probably optimise the robot's performance. However, using this type of network leads to problems that can be difficult to solve, such as the vanishing gradient.(3)
- **Implementing another algorithm** As we have seen, there are several ways to train a neural network to find the CDM. But one of the most optimal methods is the A3C (5) algorithm. (asynchronous advantage actor-critic) This method maximizes the exploration of several agents communicating with each other to find the best policy.

Conclusion

Our method makes it possible to approximate the CDM and find an acceptable policy. However, the model can still learn. For the moment a more conventional method such as a vector field would work without any doubt. But the advantage of reinforcement learning methods is that it can be applied to a totally different problem without changing the agent structure and therefore the method. This is not the case with conventional methods where each problem introduces a new method.

1. Bhatnagar, S., R. S. Sutton, M. Ghavamzadeh, and M. Lee (2009) "Natural actor–critic algorithms," *Automatica* 45(11), 2471–2482, ISSN 0005-1098
2. Evans, B., H. W. Jordaan, and H. A. Engelbrecht (2021) "Autonomous obstacle avoidance by learning policies for reference modification,"
3. Hu, Y., A. E. G. Huber, J. Anumula, and S. Liu (2018) "Overcoming the vanishing gradient problem in plain recurrent networks," *CoRR* abs/1801.06105
4. Matheron, G., N. Perrin, and O. Sigaud (2019) "The problem with DDPG: understanding failures in deterministic environments with sparse rewards," *CoRR* abs/1911.11679
5. Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016) "Asynchronous methods for deep reinforcement learning,"
6. SC., W. (2003) *Artificial Neural Network*, The Springer International Series in Engineering and Computer Science,
7. Staudemeyer, R. C., and E. R. Morris (2019) "Understanding LSTM - a tutorial into long short-term memory recurrent neural networks," *CoRR* abs/1909.09586
8. van Hasselt, H., and M. A. Wiering (2007) "Reinforcement learning in continuous action spaces," in *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pp. 272–279
9. Yang, Z., Y. Xie, and Z. Wang (2019) "A theoretical analysis of deep q-learning," *CoRR* abs/1901.00137