CITY UNIVERSITY OF HONG KONG

COURSE PROJECT REPORT

CityU Car Survives Through Hong Kong

Author:
Bonan LIU
Yuzhan LIU
Ziyi PU
Zonghang TIAN
Dong XIE

Supervisor:
Dr. Clint Chin Pang HO

Master of Science in Data Science

School of Data Science

April 12, 2021

CITY UNIVERSITY OF HONG KONG

Abstract

School of Data Science

Master of Science in Data Science

CityU Car Survives Through Hong Kong

by Bonan LIU
Yuzhan LIU
Ziyi PU
Zonghang TIAN
Dong XIE

This project first simulated the complex traffic situation in Hong Kong as the environment: narrow and sinuous roads layout, obstacles alongside, pedestrians and animals come from all the directions accidentally, besides, limited time for the commute. A 3D back-engine supercar(named UFO) was then modeled in that environment, whose task is to drive to the target terminal point safe and as quickly as possible.

We control UFO through Unity ML-Agent API, deploy the Reinforcement Learning algorithm to help UFO learning to survive from random routes. TODO: some conclusion

Contents

Chapter 1

Introduction

1.1 Introduction of Deep Reinforcement Learning

1.2 Motivation and Target

We notice that Hong Kong faced a very complex traffic situation than many other cities, even supercities like Shenzhen or Washington. Roads here are much narrow and sinuous, and some segments are located between mountain and sea. The very high density of buildings makes it possible that pedestrians suddenly appear from any horizontal direction, not to say wild pigs and other animals.

We try to simulate the traffic situation of HK, and 'build' a self-driving car, deploy RL algorithms to help it driving to the target terminal point safe and as quickly as possible.

1.3 Architecture of This Report

- Chapter 1: Introduction of this project
- Chapter 2: Supported methods and materials
- Chapter 3: Enviroment building and training
- Chapter 4: Numerical results and discussion
- Chapter 5: Conclusion and future directions

Chapter 2

Supported Methods and Materials

2.1 Reinforcement Learning Theory and Methods

Deep Reinforcement Learning is a subfield of machine learning that combines reinforcement learning (RL) and deep learning. Deep RL incorporates deep learning into the solution, allowing agents to make decisions from unstructured input data without manual engineering of the state space.Deep RL algorithms are able to take in very large inputs (e.g. every pixel rendered to the screen in a video game) and decide what actions to perform to optimize an objective (eg. maximizing the game score). Deep reinforcement learning has been used for a diverse set of applications including but not limited to robotics, video games, natural language processing, computer vision, education, transportation, finance and healthcare.
Various techniques exist to train policies to solve tasks with deep reinforcement learning algorithms, each having their own benefits. At the highest level, there is a distinction between model-based and model-free reinforcement learning, which refers to whether the algorithm attempts to learn a forward model of the environment dynamics.

2.1.1 Optimization Methods

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy algorithm which can be used for environments with either discrete or continuous action spaces. There are two primary variants of PPO: PPO-Penalty and PPO-Clip.Here, well focus only on PPO-Clip.
PPO-Clip doesnt have a KL-divergence term in the objective and doesnt have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Key Equations
PPO-clip updates policies via

$$\theta_{k+1} = arg \max_{\theta} \ \mathbf{E}_{s,a\sim\pi_{\theta_k}} \left[L\left(s, a, \theta_k, \theta\right) \right],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective. Here L is given by

$$L\left(s, a, \theta_k, \theta\right) = min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s|a), g(\epsilon, A^{\pi_{\theta_k}}(s|a)) \Big),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & A \geq 0 \\ (1-\epsilon)A, & A \leq 0 \end{cases}$$

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$$L\left(s, a, \theta_k, \theta\right) = min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s|a), (1+\epsilon)\Big) A^{\pi_{\theta_k}}(s|a),$$

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$$L\left(s, a, \theta_k, \theta\right) = min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s|a), (1-\epsilon)\Big) A^{\pi_{\theta_k}}(s|a),$$

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

Soft Actor-Critic Optimization

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

Key Equations

Let x be a random variable with probability mass or density function P. The entropy H of x is computed from its distribution P according to

$$H(P) = \mathbf{E}_{x \sim P}[-logP(x)].$$

In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = arg \max_\pi \mathbf{E}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t))\Big)\right]$$

where $\alpha > 0$ is the trade-off coefficient.We can now define the slightly-different value functions in this setting. V^π is changed to include the entropy bonuses from every timestep:

$$V^{\pi}(s) = \mathbf{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}\Big(R(s_t,a_t,s_{t+1}) + \alpha H(\pi(\cdot|s_t))\Big)|s_0 = s\right]$$

Q^{π} is changed to include the entropy bonuses from every timestep except the first:

$$Q^{\pi}(s,a) = \mathbf{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}\Big(R(s_t,a_t,s_{t+1}) + \alpha H(\pi(\cdot|s_t))\Big)|s_0 = s, a_0 = a\right]$$

With these definitions, V^{π} and Q^{π} are connected by:

$$V^{\pi}(s) = \mathbf{E}_{\tau\sim\pi}[Q^{\pi}(s,a)] + \alpha H(\pi(\cdot|s_t))$$

and the Bellman equation for Q^{π} is

$$Q^{\pi}(s,a) = \mathbf{E}_{s'\sim P}[R(s,a,s') + \gamma V^{\pi}(s')]$$

SAC trains a stochastic policy with entropy regularization, and explores in an on-policy way. The entropy regularization coefficient α explicitly controls the explore-exploit tradeoff, with higher α corresponding to more exploration, and lower α corresponding to more exploitation. The right coefficient (the one which leads to the stablest / highest-reward learning) may vary from environment to environment, and could require careful tuning.

2.2 Unity 3D Engine

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. As of 2018, the engine had been extended to support more than 25 platforms. The engine can be used to create three-dimensional, two-dimensional, virtual reality, and augmented reality games, as well as simulations and other experiences. The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering and construction. Unity gives users the ability to create games and experiences in both 2D and 3D, and the engine offers a primary scripting API in C♯, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality.

We generated random routes and obstacles' locations every time. And assigned a series of sensors(ray casting sensors, trigger collider sensors, and spherical sensors) to the car agent made it agile to drive.

2.2.1 Unity Machine Learning Agents Toolkit (ML-Agents)

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. The author provide implementations (based on PyTorch) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. Researchers can also use the provided simple-to-use Python API to train Agents using reinforcement learning, imitation learning, neuroevolution, or any other methods. These trained agents can be used for multiple purposes, including controlling NPC behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release. The ML-Agents Toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unitys rich environments and then made accessible to the wider research and game developer communities.

Chapter 3

Enviroment Building and Training

3.1 Hardware and Sofrware

This project deployed on local personal computers, and hardware configuration is listed below. See **??**.

3.2 Project Architecture

Item	**Minimal Requirements**	**Recommdered Requirements**
Memory	32G	16G
CPU	AMD/Intel 8 Core(2.9GHz)	2.6GHz
GPU	NVIDIA RTX 2060 (Optional)	NVIDIA RTX 2060 (Optional)

TABLE 3.1: Hadrware Requirements

Chapter 4

Numerical Results and Discussion

4.1 Training Performance

compare

4.2 Rewards

4.3 Loss and Varience

Chapter 5

Conclusion and Future Directions

5.1 Conclusion

So far, this project shows that given a rule set of reward and penalty, RL algorithms could work well on car self-driving tasks, no matter how the environment is. And we hope this could be a base for future self-driving cars coming into real life and even VR+AI driving school in the Hong Kong market.

5.2 Future Works

Up to now, due to time and computational resource limitations, this project has several shortcomings that can be improved:

- Because we need more than 2 days to train a single test, so we don't have enough time to train various RL algorithms and make further comparisons(PPO was done, but SAC is still in progress).
- There is only one agent(i.e., the car) in the driving environment. In the future, we hope to add more agents to co-operate in the simulated world. For example, at present, all the red balls are defined by a uniform random function, which we hope can replace by pedestrian agents.
- As for environmental construction, we think we can do more exquisitely to be closer to the real-world layout.