Autonomous Driving Challenge

Reinforcement Learning

Utilizing Reinforcement Learning algorithms to teach an agent to drive autonomously in a simulated highway environment

https://github.com/andrefilipefmsilva/RL_AutonomousDriving_NovaIMS

Authors:

André Filipe Silva	20230972
André Moreira Lopes	20230570
Luís Queiroz	20230598
João Gonçalves	20230560
Pedro Cerejeira	20230442

NOVA IMS

2023/2024

Contents

1 Introduction

This introduction will branch into two topics: the issues with the number of state spaces, and our overall approach to this problem. As such, it will be quite large, but necessarily so.

Autonomous driving is a very trending topic, with many companies fighting to get ahead in this segment. Our challenge, to simulate autonomous driving decision-making in a computational environment, shows exactly why this is such a difficult task.

Starting by mentioning the biggest challenge in this problem: the number of state spaces is incredibly large and hard to grasp, especially with continuous actions, where 2 out of our 4 solutions focus on. There is a real curse of dimensionality in this problem. The addition of more and more state variables (e.g. position, speed, acceleration, sensors, options of turning radius, …) quickly explodes the number of possible states, in turn making the state space incredibly large. This makes it extremely difficult for the agent to comprehensively explore and learn optimal policies, because of the vast number of states that need to be visited in order to learn their values.

There are also other issues: the sparsity of rewards (the larger the state space, the sparser the rewards) making it challenging for the agent to acquire the association between states and rewards; generalization becomes hard to achieve, as even with a lot of training time steps, the agent may still have not seen a significant part of the states, limiting the robustness and reliability of the learning; computational complexity is ever-increasing, and that is an issue we faced as our solutions took a very long time to run and to fine-tune we often had to wait 10+ hours; and balancing exploration (trying new actions to discover their effects) with exploitation (choosing the best-known actions). This might lead to premature convergence on a suboptimal policy.

In our approach to the problem, we tested both meta-discrete and continuous actions (two solutions for each). This was done to provide a more robust and diverse approach to the problem, but it also came with extra work – a second environment was built to handle the extra complexity of implementing solutions with continuous actions (you can find it here). This second environment respects the guidelines of the project, but is enhanced with custom parameters. Both environments are tweaked with custom parameters to help solve our problem.

We also create our own function for evaluating the performance of the solutions.

As a final note: the video results were uploaded to GitHub in a compressed format, because of GitHub's upload size limitations. You can find the original, uncompressed videos by clicking here.

2 Models and Methodology

We trained all our models using the original project guidelines - 10 lanes and 120 steps of duration.

2.1 Solution 1 – DiscreteMeta_Kinematics

Environment Observation Type: Kinematics
Agent Action Type: Discrete Meta-Actions
Algorithm Used: DQN

The **Kinematics** observation type provides detailed state information about the vehicle's physical properties, such as position, velocity, and heading.

Discrete-Meta Actions offer a higher level of control compared to low-level continuous actions. The available actions typically include lane changes (left or right), speed adjustments (faster or slower), and maintaining the current lane. This abstraction simplifies the action space and allows the agent to make strategic decisions more effectively.

Deep Q-Network (DQN) is an algorithm that combines Q-Learning with deep neural networks to handle environments with high-dimensional state spaces. It was popularized by the success of DeepMind's work on playing Atari games. It builds on Fitted Q-Iteration (FQI) and makes use of different tricks to stabilize the learning with neural networks: it uses a replay buffer, a target network and gradient clipping.

We start to describe this algorithm by approaching Q-Learning: a model-free algorithm that aims to learn the optimal action-selection policy for an agent by maximizing the cumulative reward. The Q-value *Q(s,a)* represents the expected future reward for taking action a in state s. DQN uses a neural network to approximate the Q-value function, $Q(s,a;\theta)$, where θ are the parameters (weights) of the network. This allows it to handle high-dimensional state spaces.

As mentioned earlier, DQN uses a replay buffer to store transitions (set to 500 000 in our case). During training, random batches of experiences are sampled from this buffer. This breaks the correlation between consecutive experiences, leading to more stable training and efficient use of past experiences.

Moving to the Target Network, this separate target network is used to compute target Q-values, which helps stabilize training by reducing oscillations and divergence.

Finally, the gradient clipping helps prevent the problem of exploding gradients during backpropagation, causing huge updates to network weights. It caps the gradients at a maximum threshold, ensuring the updates remain manageable and improving learning stability dynamics.

This algorithm is particularly useful in high-dimensional state spaces, as the one we are challenged with, as it is able to handle and learn effective policies here through experience replay.

In summary, the Kinematics observations provide the necessary state information, while the

Discrete-Meta Actions simplify the decision-making process, making it feasible for DQN to learn effective strategies in a complex driving environment.

The very first model we trained was based on a suggestion from the highway-env documentation. The results from this model showed that the car was crashing a lot. Our first instinct was to go and change the collision reward to a lower number. This proved to be right.

Our best model is quite simple in training, with a simple change in collision reward from -4 to -1. We experimented with a lot of other parameter changes, even custom rewards, but in the end this simple one was what produced the best results. We could say this model is still not the best at generalizing but this would require training under a lot more time steps and trying a lot of combinations of parameters which was not feasible with the computational resources at our disposal. Please find a video of the final result by clicking here.

2.2 Solution 2 – Continuous_Kinematics

Environment Observation Type: Kinematics
Agent Action Type: Continuous Actions
Algorithm Used: PPO

In this case, we used Kinematics with 5 cars being observed. This provides a good balance between reducing the state space and maintaining performance. Additionally, we incorporated metrics such as sine and cosine functions to give the model information about its orientation relative to the street. This is particularly important for continuous actions, as it helps the model understand its orientation. To better explain the contrast, this is not an issue for discrete or meta-discrete action spaces because the environment handles orientation automatically. However, still in Kinematics, we should point out that it is not immediately clear which vehicles the agent is currently seeing with *vehicle_counts*, as sometimes the behavior it displayed was: as soon as the agent overtook a vehicle, it seemed like it stopped knowing said vehicle's location. This is due to a parameter in Kinematics, *see_behind*,that if set to False, means the agent only observes the closest cars around it and ignores cars that are behind it.

In **continuous action spaces**, agents can select actions from a continuous range rather than a discrete set of predefined actions. It also has more parameters to control than its discrete counterparts (steering, throttle, …). This has its own advantages (allows for a very fine control over the vehicles movements) and disadvantages (far more complex to learn compared to discrete or meta-discrete action spaces).

The **Proximal Policy Optimization (PPO)** algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor). Developed by OpenAI, it is a state-of-the-art policy gradient method designed to optimize policies in a stable and efficient manner. The main idea is that after an update, the new policy should be not too far from the

old policy. For that, PPO uses clipping to avoid too large updates. In addition, it makes use of generalized advantage estimation (GAE) to reduce the variance of the policy gradient estimates, leading to more efficient learning. It performs multiple epochs of training on the same batch of data, improving data efficiency. Proximal Policy Optimization (PPO) is part of the actor-critic family of models. In this approach, we have one neural network, the actor, which predicts actions based on the state, and another neural network, the critic, which evaluates the predicted actions by providing a value (indicating how good the action is) based on the state and the action suggested by the actor. What makes PPO unique is its method of updating these internal neural networks. PPO uses a clipped objective function to limit policy updates, ensuring stability and preventing large, destabilizing changes.

The use of PPO in a Continuous Actions setting is suitable due to its assurance of stable updates (mostly obtained through its gradient clipping implementation). Plus, the Kinematics Environment provides the necessary low-level details for accurate maneuvering. The main challenge lies in the difficult hyperparameter tuning, but that is largely offset by the benefits provided by the algorithm.

For training the model we used a lane centering reward, a custom reward to encourage the agent to stay centered in the lane. We also adjusted the reward speed range so the agent receives negative rewards when its velocity falls below that of the surrounding NPCs, encouraging it to overtake rather than fall behind. Additionally, we implemented an action penalty to discourage large corrective actions, promoting smoother driving. We also tweaked the policy frequency to allow the agent to make more decisions per second, which is crucial for making the numerous micro-adjustments needed in continuous actions. It's important to note that the agent was trained in multiple phases, which can be seen as fine-tuning.

Considering the total number of timesteps, it underwent training for 1,300,000 timesteps to become a decent agent. To expedite the process, a vectorized environment was used, with five environments running simultaneously. This technique was also applied in Solution 4 because continuous action training is computationally even more expensive.

Please find a video of the final result by clicking here.

2.3 Solution 3 - DiscreteMeta_Grayscale

Environment Observation Type: Grayscale
Agent Action Type: Discrete Meta-Actions
Algorithm Used: DQN

The **Grayscale** observation type provides observations in the form of gray scale images. The agent receives a visual input where each pixel's intensity represents different shades of gray, capturing the environment's visual state without color information. This is often used to reduce the complexity and computational load compared to color images. To capture temporal information, multiple

consecutive gray scale frames are often stacked together. This allows the agent to infer motion and changes in the environment over time.

Joining all the parts of our model together, we get something similar to Solution 1, except for the Environment observation type. Gray scale images provide a richer context, capturing more environmental details than kinematic data alone. Kinematic observations, however, are lower-dimensional, leading to faster processing and potentially more stable learning, and the data is directly relevant for vehicle control tasks, potentially leading to more accurate and responsive control policies. It is hard to say *a priori* which one is the best choice, as there's advantages in both, which is why we have a section that evaluates our solutions in a comparable way.

The main tweaks done in this solution were to the collision reward (higher penalization) and the reward speed range (rewards only for higher speeds than the default).

Please find a video of the final result by clicking here.

2.4 Solution 4 – Continuous_Grayscale

Environment Observation Type: Grayscale
Agent Action Type: Continuous Actions
Algorithm Used: SAC

The **Soft Actor-Critic (SAC)** algorithm, is particularly well-suited for environments with continuous action spaces. It builds on the foundations of traditional actor-critic methods but introduces several key innovations to enhance performance. Unlike traditional algorithms that focus solely on maximizing cumulative rewards, SAC incorporates an entropy term into the objective. This promotes exploration by encouraging the policy to remain stochastic, especially during early stages of learning. It is off-policy, having high sample efficiency since this means it learns from a replay buffer of past experiences rather than requiring new experiences for each update. It also utilizes a "trick " known as Double Q-Learning, aimed at mitigating the positive bias in the policy improvement step, by using two parallel Q-learning networks. But the central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy. 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. On the downside, it is extremely computationally intensive due to the multiple networks (two Q-networks, two target Q-networks, and a policy network). The hyperparameter tuning is also very sensitive, with small changes in tuning leading to large changes in results. Those need to be accounted for very carefully.

Even though what was theoretically described above about the SAC algorithm is true, this was the only model that would bug out and exploit the environment. We do not know exactly why it happened, but it always happens when the agent started at the top lane. The SAC model

figured out that it could still get a decent reward, when starting at the top lane, if it just stayed behind driving at very low speeds in the same lane. This will leave him abandoned and defeat the purpose of autonomous driving in a traffic-filled environment. This is, indeed, exploitation as opposed to the results of both DQN models provided, where the car would not overtake much and just stick to accompanying traffic, which still follows the purpose of autonomous driving solutions in a traffic-filled environment.

When the SAC model was trained, the reward was set to 0 if the speed was slower than other vehicles (25m/s). This contrasts with the penalization for solution 2, where we give negative rewards in this situation. Other than that, the training was quite similar, but with fewer time steps (1 000 000, compared to Solution 2's 1 300 00).

Please find a video of the final result by clicking here.

3 Results

To evaluate our results, we created a custom function. It returns the total cumulative reward of each episode (we simulate 10 episodes), shows the episode duration, and if the episode was truncated or not, represented by a Boolean value - an episode is truncated if it reaches the limit of steps mandated by the guidelines (120).

Essentially, in the 'Returns List' column, these are the meanings of our results:

1. (x, False) - The agent collided / went off-road;
2. (< 105, True) - The agent did not solve the environment - it either stayed behind or kept pace with other cars;
3. (>= 110, True) - The agent solved the environment.

Solution 1 - DiscreteMeta_Kinematics

Episode #	Average Standardized Reward	Episode Duration	Returns List
1	0.15	0.48	(0.15, False)
2	33.68	20.78	(33.68, False)
3	90.43	54.98	(90.43, True)
4	92.09	52.93	(92.09, True)
5	2.95	1.64	(2.95, False)
6	83.47	54.38	(83.47, True)
7	88.04	54.74	(88.04, True)
8	91.16	55.26	(91.16, True)
9	88.17	54.35	(88.17, True)
10	7.38	3.91	(7.38, False)

Table 1: Solution 1 Results

Looking at our evaluation, in this solution the agent did not solve the environment in any of the episodes, but in 6 of them it did not crash or go offroad.

Solution 2 - Continuous_Kinematics

Episode #	Average Standardized Reward	Episode Duration	Returns List
1	55.11	17.09	(55.11, False)
2	21.73	6.49	(21.73, False)
3	119.27	35.65	(119.27, True)
4	27.50	8.32	(27.50, False)
5	11.42	3.29	(11.42, False)
6	31.78	9.57	(31.78, False)
7	15.46	4.58	(15.46, False)
8	119.07	35.53	(119.07, True)
9	9.55	2.72	(9.55, False)
10	22.26	6.57	(22.26, False)

Table 2: Solution 2 Results

Looking at our evaluation, in this solution the agent managed to solve the environment

2 times. Another point to consider is the variability in the results. Sometimes the agent performs exceptionally well, while other times it performs poorly. This variance has been difficult to address, possibly due to overfitting. Additionally, the environment is more challenging at the beginning, with NPCs being more compact. Once the agent gets past the initial phase, its performance tends to improve, unless it fails early in this phase.

Furthermore, when comparing these results with other models, it is important to note that this agent operates at an 8 Hz policy frequency. This means it makes 8 actions per second and receives 8 rewards per second, which can lead to a higher expected return.

Solution 3 - DiscreteMeta_Grayscale

Episode #	**Average Standardized Reward**	**Episode Duration**	**Returns List**
1	1.68	1.34	(1.68, False)
2	32.26	16.86	(32.26, False)
3	0.19	0.42	(0.19, False)
4	90.69	54.20	(90.69, True)
5	4.98	2.96	(4.98, False)
6	0.91	0.80	(0.91, False)
7	11.91	6.10	(11.91, False)
8	89.23	53.77	(89.23, True)
9	1.69	1.22	(1.69, False)
10	0.93	0.83	(0.93, False)

Table 3: Solution 3 Results

For solution 3, we can see that the agent managed twice to not collide or go off-road, but even when it was able to, it still could not solve the environment.

Solution 4 - Continuous_Grayscale

Episode #	Average Standardized Reward	Episode Duration	Returns List
1	70.42	23.11	(70.42, False)
2	73.12	24.12	(73.12, False)
3	15.73	4.97	(15.73, False)
4	29.46	10.07	(29.46, False)
5	34.39	12.06	(34.39, False)
6	17.41	5.93	(17.41, False)
7	28.73	9.06	(28.73, False)
8	46.12	14.56	(46.12, False)
9	50.29	16.39	(50.29, False)
10	113.71	36.71	(113.71, True)

Table 4: Solution 4 Results

On Solution 4, we can see that in 1 episode the agent actually managed to solve the environment.

We can say that the main goal of this project is that the agent learns to drive autonomously. That does not necessarily mean that the agent solves the environment, but that it keeps up with traffic and avoids collisions. If the agent surpasses everyone else, or actually solves the environment, we consider that an extra.

What we found were two types of solutions: One where the agent attributed more value to driving at high speeds and overtaking all other cars, and a second one where the agent prefers a more defensive driving, slowing down when needed to avoid collisions, and developing no incentives to overtake other cars as the rewards for going faster is not enough to compensate for the possible penalties to be collected due to the higher probability of collisions.

This is actually a logical development, as the solutions in which the agent learns to slow down are the ones with meta-discrete actions, in which the number of actions is approximately 1 per second, and as such the agent will only receive the reward of that single action - and that reward will never be greater than the penalty for colliding. As such, it prefers to slow down, momentarily reducing the reward obtained, but avoid being severely penalized.

On the other hand, when dealing with continuous actions, as the agent is stuck to the same (throttle, steering) values in one action, and this takes 1 second to happen if you keep the *policy_frequency* as default, he will execute that action continuously during that time

frame. This is bad as during that time frame it can be enough to lead him off-road or into much farther away lanes than intended. So, to counterbalance it, in continuous actions the *policy_frequency* is increased. However, as it increases, so does the reward per second, and if the agent is accumulating more rewards than the penalty for collision, it will lead to it not worrying about going at high speeds and drastically increasing the probability of collision. All of this results in the agent not needing to learn how to slow down, and in a larger number of tests, overtaking every other car and solving the environment.

As such, both scenarios can be considered as a metric of success: it depends on what any given person would value most when evaluating these kinds of autonomous driving scenarios. One could value speed and efficiency, another could value safety and tranquility. With that established, we can confidently say that all final models of each solution are able to shine in their respective mentioned scenarios. The discrete solutions, 1 and 3, are more conservative and safe as, without exploiting, they truncate more times than the continuous solutions, 2 and 4, that happen to be more speedy and efficient when it comes to going further and isolating themselves by overtaking every single car.

4 Conclusion

We were tasked with developing four different solutions for autonomous driving in a highway environment. Our solutions covered both continuous and meta-discrete actions, Kinematics and Grayscale Observation types, and DQN, PPO and SAC algorithms. We got relatively good results considering the complexity of some of the implemented solutions and the computational resources available. Further work would involve training with a larger number of timesteps - as we noticed some of our models still did not generalize too well in certain situations -, and try different algorithms to find the best possible combinations. We were mainly limited by computational resources, as these models took very long hours to train, and every fine-tuning meant re-training the models. Another key takeaway from this project is the importance of rewards. Small changes in the rewards can significantly alter the agent's behavior. For the agent to perform well, it needs clear instructions through effective reward design. In some cases, achieving this requires customized rewards and careful balancing of various reward components.

In addition, not only did we develop effective solutions with continuous actions, which are rare to find in this highway environment, but we also enhanced the capabilities of the environment to make it more suitable for these types of applications.

We believe our work shows some of the challenges with autonomous driving, and why it has not arrived "to the real world" just yet.

5 References

1. Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, January 4). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv.org. https://arxiv.org/abs/1801.01290

2. highway-env Documentation. (n.d.). https://highway-env.farama.org/

3. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December 19). Playing Atari with Deep Reinforcement Learning. arXiv.org. https://arxiv.org/abs/1312.5602

4. Sanghi, N. (2021). Deep Reinforcement Learning with Python. In Apress eBooks. https://doi.org/10.1007/978-1-4842-6809-4

5. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, July 20). Proximal Policy optimization Algorithms. arXiv.org. https://arxiv.org/abs/1707.06347

6. Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 2.4.0a3 documentation. (n.d.). https://stable-baselines3.readthedocs.io/

7. Welcome to Spinning Up in Deep RL! — Spinning Up documentation. (n.d.). https://spinningup.openai.com/