Reinforcement Learning for Highway Driving

Marcelo García Escalante
mrgaresc@stanford.edu

Jun 6, 2023

1 Introduction

Autonomous driving is a rapidly growing field of research. The goal of this project is to develop a Reinforcement Learning agent that can drive a car on a highway. The agent will be trained in a simulated environment and will be evaluated on its ability to drive safely (without collisions) and efficiently (at the highest speed allowed). The environment consists of a highway with multiple lanes, where the agent will be evaluated on its ability to drive safely and efficiently.

The environment that we will use is built on top of OpenAI Gym.[1] The environment is called "highway-fast-v0"[2] We will use this environment to train and evaluate the agent using a Reinforcement Learning algorithm we have learned in class such as *Q-learning*, compare its outcome to a random agent and discuss the results.

2 Literature Review

Our goal is to follow the implementation of Deep Q-learning in[2] to solve the RL task stated in section 1. We might consider also other variants of Deep Q-Learning, like Double DQN, which is also stated in[2].
Other interesting works are included in [3] and [4], where DQN techniques become also the choice of approaches to these problems. In these research articles, the question of safe decision making is discussed. There are two main difficulties: Collisions should never happen and the algorithms need to take into account new observable states (e.g. unpredictable behavior of other agents such as cars) in the environment.
In [3] a Deep Reinforcement Learning agent is designed by training an ego vehicle, which learns a driving policy by interaction with diverse simulated traffic. The work in this article is based mainly on a modified version of the double Q-network (DDQN) algorithm.
Moreover, there were three ingredients, which were essential for the stability and efficiency of the agent: *Short-horizon safety check*, *two buffers* (a simpler version of prioritized experience replay), and a *safety controller*. The short-horizon safety check is necessary because Q-learning relies on an exploration policy (e. g. ϵ-greedy). This would lead to scenarios like collisions and thus to simulation resets. By implementing a short-horizon safety check that evaluates chosen actions and provides an alternative safe action, this can be avoided. The safety controller was important during the inference phase and played a key role in finding a good policy.

As mentioned before, we implemented a Deep Reinforcement Learning Algorithm, namely DQN, some of the ideas we got from the cited works above are the following:

- **Replay buffer** to store the transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ and sample from it to train the agent. This will help us to break the correlation between the samples and make the training more stable.
- **Soft updates** of the target network parameters to the online network parameters. This will help us to stabilize the training.

3 Dataset

Since we are using a simulated environment, we will not need to use any external dataset. The environment will provide us with the necessary data to train and evaluate the agent. The environment provides us with a continuous state space and a discrete action space.

The shape of the input is as follows: `-inf, inf, (5, 5), float32`

- **presence**: 1.0 if a vehicle is present, 0.0 otherwise.
- **x**: World offset of ego vehicle or offset to ego vehicle on the x axis.
- **y**: World offset of ego vehicle or offset to ego vehicle on the y axis.
- **vx**: Velocity on the x axis of vehicle.
- **vy**: Velocity on the y axis of vehicle.

Note: the coordinates are relative to the ego-vehicle, except for the ego-vehicle which stays absolute. The world frame is at the top left-corner of the highway. The ego-vehicle is always centered on the lane, and its coordinates are relative to the world coordinate system, as shown below:

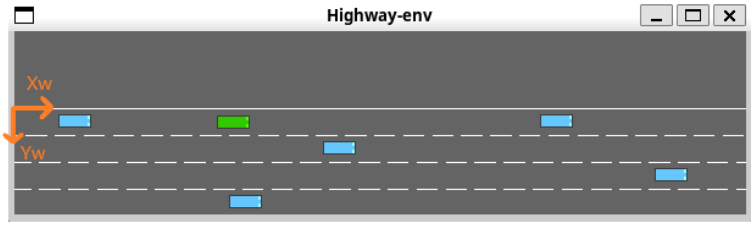

Figure 1: Environment with a world coordinate system at the top left corner of the highway.
ego-vehicle is represented by the green car and the other vehicles are represented by the blue cars.

Note that the coordinate system is fixed relative to the image, so it always remains at the top left corner of the highway. The ego-vehicle has always a x-coordinate more than 0 and a y-coordinate from 0.0 to 1 depending on the lane it is in. Thus, each lane is 0.33 units wide for 3 lanes.

Since our model used an MLP as a function approximator, we needed to flatten the input to a 1D array of size 25 (5x5).

4 Baseline

For the Baseline we used a simple agent that randomly chooses an action from the action space at each step with a uniform distribution. The action space is as follows:

Action index	0	1	2	3	4
Action name	lane change left	idle	lane change right	accelerate	decelerate

To evaluate the agent we used the following metrics that are best described in the Evaluation Metric section 6:

- **Collision rate**: 97% of the episodes ended in a collision.
- **Mean cumulative rewards**: Between 0.6 and 0.8 depending on the episode.

The results can better be visualized in the Results & Analysis section 7, where we also discuss the results.

5 Main approach

We will follow [5] and [6]. In the lecture, we already introduced RL algorithms, but let's introduce some notation before diving deeper. A Markov Decision process (MDP) is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S} \in \mathbb{R}^n$ is the state space, $\mathcal{A}$ the action space, $\mathcal{T}\colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the state transformation function and $\mathcal{R}\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ the reward function. A policy is defined by $\pi\colon \mathcal{S} \to \mathcal{A}$, where $a_t = \pi(s_t)$ denotes the action at timestep t. Every action a_t leads to a state transition from s_t to s_{t+1} with a reward r_{t+1}. The goal is to find an optimal policy, such that the total accumulated reward is maximized:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

where $0 < \gamma \leq 1$. Notice that we are working with a continuous state space and a discrete action space, given by

$$\mathcal{A} = \{0 : \text{lane change left}, 1 : \text{idle}, 2 : \text{lane change right}, 3 : \text{accelerate}, 4 : \text{decelerate}\}$$

We will introduce a model-free approach, namely Deep Q-Network (or DQN). This algorithm is based on Q-Learning with function approximation, where the state-action value function Q is represented by a neural network. Since the approximation of the state-action value function leads to instability, experience replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: DQN (see Mnih et al.).

comes into play. Another observation is that DQN uses a second neural network, namely the target network. This is to ensure to obtain an unbiased estimator of the mean-squared Bellman error used in training the Q-network.

Besides this a soft update of the target network parameters to the online network parameters was used. This help the algorithm to stabilize the training.

The soft update is defined as follows:

$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{online}} + (1 - \tau)\theta_{\text{target}},$$

Where τ is a hyperparameter, which controls the rate of the update. This is useful because it allows the target network to track the online network, but not too fast, so that the target network is stable. This was a key ingredient to stabilize the training since at the beginning we were using a hard update, which was updating the target network parameters to the online network parameters. This was causing the training to be unstable and the agent was not able to learn anything. Besides by using a soft update we were able to update with a higher frequency the target network parameters, which we observed that helped the agent to learn faster.

For our use case we had a MLP with 3 hidden layers, with a mix of 32 and 64 neurons and ReLU activation. The output layer had 5 neurons, one for each action. The input layer had 25 neurons, which is the size of the flattened state space.
We used the Adam optimizer and the final parameters are the following:

- Learning rate: 5e-4
- Discount factor: 0.8
- Target network update frequency: 4
- Replay buffer size: 15000
- batch size: 128
- epsilon: 0.8 (epsilon-greedy policy)
- epsilon decay: 0.9995

- epsilon min: 0.01
- tau: 1e-3

For the training phase we used 1000 episodes and for the evaluation phase we used 100 episodes. The results can better be visualized in the & Analysis section 7, where we also discuss the results.

Algorithm Implementation Details

Let's do a step by step of the algorithm for an input example s_t listed in the appendix A Sample 1:

- **Step 1**: We first flatten the input to a 1D array of size 25 (5x5). Thus we got the following 1D tensor:

```
[1.0, 0.86, 0.33, 0.26, 0.0, 1.0, 0.1, 0.33, 0.01, 0.0, 1.0, 0.2,
 -0.33, -0.05, 0.0, 1.0, 0.32, -0.33, -0.01, 0.0, 1.0, 0.44, 0.06, 0.0, 0.0]
```

- **Step 2**: We pass the 1D tensor to the select action function, which returns the action index. We used an epsilon-greedy algorithm to balance exploration and exploitation. In the beginning of the training we wanted to explore more and then exploit more as the training progresses. The epsilon-greedy algorithm is as follows:
  - With probability ϵ we choose a random action from the action space.
  - With probability $1 - \epsilon$ we choose the action with the highest Q-value.

  In our case we used $\epsilon = 0.8$ and $\epsilon_{min} = 0.01$. We also used a decay rate of 0.9995.
- **Step 3**: We pass the action index to the step function, which returns the next state, the reward and a boolean variable that indicates if the episode is done or truncated (the episode is truncated if the time limit is reached).
- **Step 4**: We store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in the replay buffer until the buffer has enough samples to start training. In our case we used a replay buffer size of 15000. And to start training we need at least 128 samples due to the batch size we used.
- **Step 5**: We sample a batch of 128 samples from the replay buffer and we pass it to the learn function (optimize model function in the code).
- **Step 6**: We update the target network parameters with the online network parameters. We used a soft update with a *tau* rate of 1e-3. The loss function we used was the smooth L1 loss function (Huber loss) which is defined as follows:

$$\text{loss}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

  We choose this loss instead of the mean squared error because it is less sensitive to outliers and it provides a smoother transition from quadratic loss for small errors to linear loss for larger errors based on the literature cited in section 2.
- **Step 7**: We update the epsilon value by multiplying it by the decay rate. In our case we used a decay rate of 0.9995.
- **Step 8**: We repeat steps 1 to 7 for each new input until the episode is done or truncated. Then we move to the next episode until we have finished all the episodes.

6 Evaluation Metric

For the evaluation of the agent we are using the following metrics:

- **Collision rate**: The percentage of episodes that ended in a collision. The goal is to have a collision rate of 0% because we can't trade-off safety for efficiency. So the lower the collision rate the better.

- **Mean cumulative rewards**: The mean of the cumulative rewards of all the episodes. That is we collect the rewards for each step in the episode and then we take the mean of all the rewards for that specific episode. The goal is to maximize the mean cumulative rewards.

  The **reward function** depends on 4 factors that are described below:

  - **Collision reward**: -1 if there is a collision, 0 otherwise. The episode ends when there is a collision.
  - **High speed reward**: 0.4 if the speed is between 20 and 30 m/s, 0 otherwise.
  - **Close to right lane reward**: 0.1 if the ego-vehicle is on the right lane, 0 otherwise. We prefer the ego-vehicle to be on the right lane because it is the safest lane and the road infrastructure is designed to be driven on the right lane such as signs, lane markings, and exit ramps. (assuming that the ego-vehicle is in a country where people drive on the right side of the road)

- **Rewards normalization**: This is important as all our episode rewards are normalized between 0 and 1 by dividing the rewards by the maximum possible reward in 30 seconds (the maximum time allowed for each episode).

- **Standard deviation of the cumulative rewards**: The standard deviation of the cumulative rewards for all the episodes. The goal is to minimize the standard deviation of the cumulative rewards so that we can have a consistent agent.

7 Results & Analysis

Random Agent

As described in section 4 we selected a Random agent as baseline. Based on the metrics we have defined in the Evaluation Metric section 6, we get the following results presented in figure 3 after running the agent for 100 episodes:

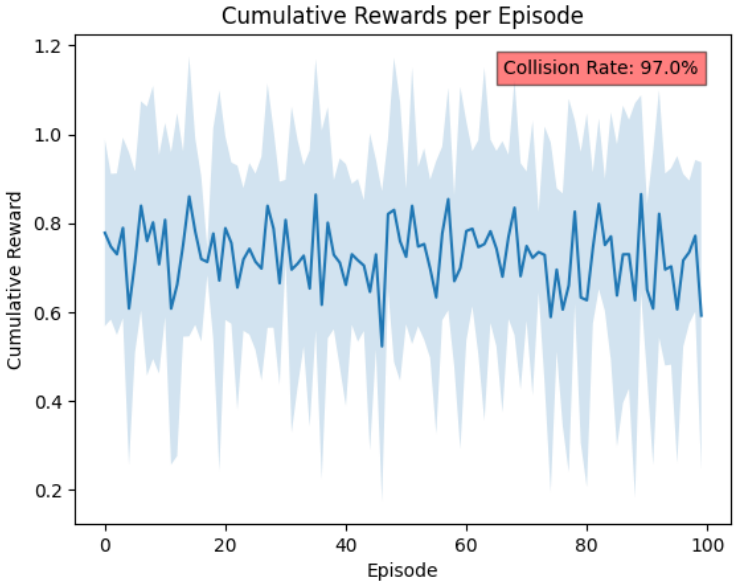

Figure 3: Results for the baseline agent.

As we can see from the results, the agent is not able to drive safely and efficiently. The collision rate is 97% and the mean cumulative rewards are between 0.6 and 0.8 depending on the episode. This means that the agent is colliding with other vehicles most of the time and it is not able to drive at the highest speed allowed, is not able to stay on the road and is not able to stay close to the right lane. Besides that, the standard deviations of the cumulative rewards are high relative to the means, which means that the agent is not consistent.

Based on these results, we got a better understanding of the environment and the agent as well as an intuition of the challenges that our DQN agent might face. It is important to design our agent robust enough

to drive safely and efficiently in the environment so that we can achieve a collision rate of 0% and a high mean cumulative rewards.

DQN Agent

After running the DQN agent for 1000 episodes, we got the following results in a 100 episode evaluation phase presented in figure 4:

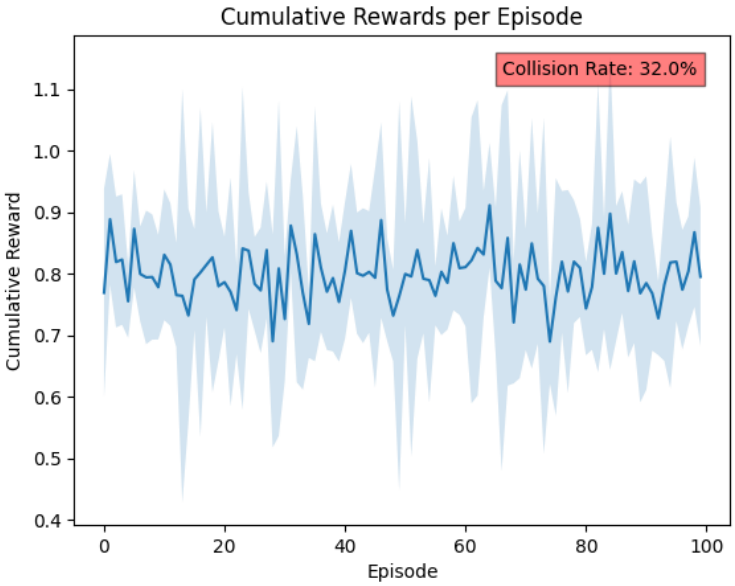

Figure 4: Results for the DQN agent.

Although we couldn't get the collision rate to 0%, we were able to reduce it to 32% and we were able to get a mean cumulative rewards between 0.75 and 0.9 depending on the episode. The variance of the cumulative rewards is also lower than the baseline agent, which means that the agent is more consistent. Also improving the mean cumulative rewards by around 0.1 to 0.2 depending on the episode compared to the baseline agent might not seem like a big improvement, but it is a good improvement because as it was mentioned on the Evaluation Metric section 6, the mean cumulative rewards are normalized between 0 and 1, so an improvement of 0.1 to 0.2 is around 10% to 20% improvement.

8 Error Analysis

The main error that we observed is that the agent is not able to drive safely because it is hard to get the collision rate to 0% which is the main goal. We observed that the agent is not able to drive safely because some cars also have a random behavior, which makes it hard to predict their behavior. This is a challenge that we need to overcome in order to get the collision rate to 0%.

On our hope to improve the agent performance we try different rewards functions to penalize collisions more than the other rewards. However we learned that reward designing is a hard task and it is very easy to get stuck in local minima.

For innstance, when we tried to penalize collisions a lot more than the other rewards, we got below's figure 5 results on a 100 episode evaluation phase:

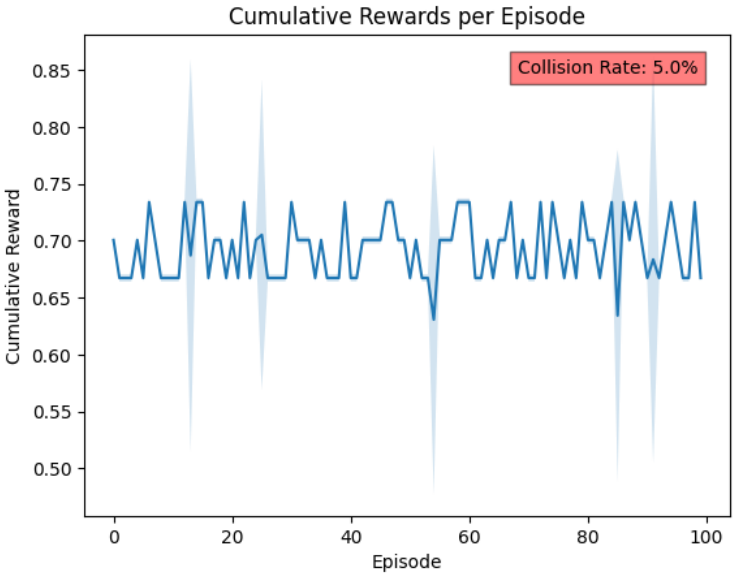

Figure 5: Results for the DQN agent with a reward function that highly penalizes collisions

You can see that the agent is able to indeed reduce the collision rate to 5%, but it is not able to drive efficiently at all as it drived at a very low speed earning a very low cumulative reward consistently. We found that the environment is designed in a way that no upcoming vehicle from behind can collide with the ego-vehicle, so the agent learned to drive at the least speed possible to avoid collisions. This can be considered a reward hacking problem, where the agent is able to exploit the reward function and the environment to get a high reward given that we over penalized collisions.

9 Future Work

As future work we would like to try different reward functions to see if we can get a better performance. We would also like to try different hyperparameters to see if we can get a better performance.

We would also like to try other algorithms such as Double DQN, Dueling DQN, and Rainbow DQN that are also stated in [2] to see if we can get a better performance.

Finally it would be great to try out some of the ideas that were presented in the literature review section 2 such as the short-horizon safety check, two buffers, and a safety controller to see if we can get a better performance.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[2] Edouard Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[3] Subramanya Nageshrao, Eric Tseng, and Dimitar Filev. Autonomous highway driving using deep reinforcement learning. *arXiv preprint arXiv:1904.00035*, 2019.

[4] Arash Mohammadhasani, Hamed Mehrivash, Alan Lynch, and Zhan Shu. Reinforcement learning based safe decision making for highway autonomous driving. *arXiv preprint arXiv:2105.06517*, 2021.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

A Appendix: Raw state space samples

Sample 1

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	0.8600	0.3300	0.2600	0.0000
vehicle 1	1.0	0.1000	0.3300	0.0100	0.0000
vehicle 2	1.0	0.2000	-0.3300	-0.0500	0.0000
vehicle 3	1.0	0.3200	-0.3300	-0.0100	0.0000
vehicle 4	1.0	0.4400	0.0600	-0.0000	0.0800

Sample 2

Table 1: Sample 2

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	1.0	0.0800	1.0	0.0
vehicle 1	1.0	0.2100	-0.0400	-0.1800	0.0000
vehicle 2	1.0	0.2400	-0.0800	-0.4800	0.0000
vehicle 3	1.0	0.5700	-0.0800	-0.3400	0.0000
vehicle 4	1.0	0.9500	-0.0800	-0.1600	0.0000

Sample 3

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	1.0	0.0500	0.6200	-0.1400
vehicle 1	1.0	0.0100	-0.0300	-0.0900	0.1400
vehicle 2	1.0	-0.2400	-0.0500	0.2200	0.1400
vehicle 3	1.0	0.2500	-0.0500	0.3500	0.1400
vehicle 4	1.0	0.8000	0.0300	0.3300	0.1400

Sample 4

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	1.0	0.0	1.0	0.0
vehicle 1	1.0	0.0600	0.0800	-0.4200	0.0000
vehicle 2	1.0	0.2500	0.0000	-0.4600	0.0000
vehicle 3	1.0	0.4800	0.0400	-0.5000	0.0000
vehicle 4	1.0	0.7700	0.0800	-0.4300	0.0000

Sample 5

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	1.0	0.0800	1.0	0.0
vehicle 1	1.0	0.1600	-0.0400	0.0700	0.0000
vehicle 2	1.0	0.2200	-0.0800	-0.2000	0.0000
vehicle 3	1.0	0.5700	-0.0800	-0.0600	0.0000
vehicle 4	1.0	0.9300	-0.0100	-0.0500	-0.3300

Sample 6

Vehicle	presence	x	y	vx	vy
ego-vehicle	1.0	1.0	0.0800	1.0	0.0
vehicle 1	1.0	-0.1600	-0.0800	-0.0300	0.0000
vehicle 2	1.0	0.2600	0.0000	-0.0200	0.0000
vehicle 3	1.0	-0.3300	-0.0400	-0.0600	-0.0100
vehicle 4	1.0	0.6800	-0.0800	0.0200	0.0000