Reinforcement Learning on Highway-Env

Final Project — Reinforcement Learning Course

DOUDOU Hissein
ADMANTA Kamelia

Teacher: MORAND Victor

March 2026

https://github.com/hissein02/Highway-env-project

1 Introduction

For this project we had to train an agent to drive on a highway without crashing into other cars. We used the Highway-Env simulator and tried two approaches: a DQN that we coded ourselves in PyTorch, and PPO from the Stable-Baselines3 library. We trained both on a faster version of the environment and then tested them on a harder setup with 40 aggressive vehicles. We also tried different hyperparameters to see what worked best.

2 Environment

The environment is `highway-v0` from Farama Foundation. Our agent is a car on a 3-lane highway that needs to drive fast and avoid collisions with other vehicles.

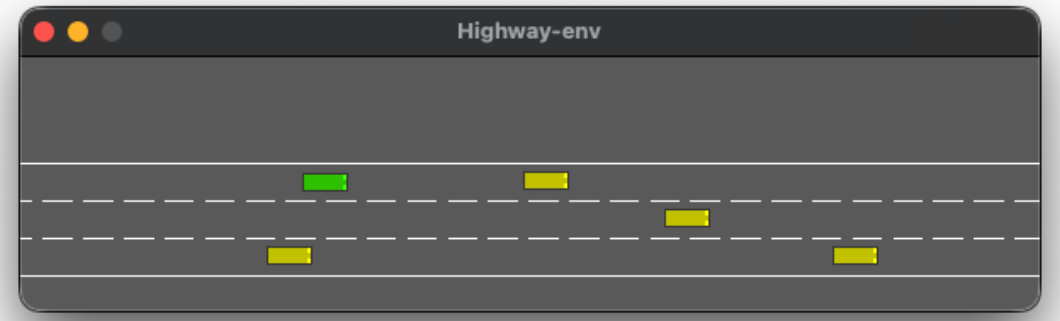

Figure 1: The highway environment

Observation: At each step, the agent gets a 5×5 matrix. Each row is a vehicle (our car + the 4 closest ones), and the columns are: `presence`, `x`, `y`, `vx`, `vy`. Everything is normalized and relative to our car.

Actions: 5 possible actions: go left, stay in lane, go right, speed up, slow down.

Reward: The agent gets rewarded for going fast and staying on the right lanes. If it crashes, it gets -1.

Training setup: We trained on `highway-fast-v0` which runs 15x faster, with 20 vehicles instead of 40. This way training doesn't take forever and the agent can actually learn without crashing every 2 seconds. For evaluation we use the hard config: 40 aggressive vehicles, tight spacing. Since the environment is random (vehicles spawn in different positions each time), running the same model twice doesn't give the exact same score. The results we report are the best ones we got for each experiment.

3 DQN — Our Implementation

3.1 How it works

DQN learns a Q-function: for each state, it predicts how much reward we can expect from each action. Then the agent just picks the action with the highest Q-value. To make sure it doesn't get stuck doing the same thing, we use epsilon-greedy: with some probability ε it takes a random action instead.

Three things are needed to make this work:

- **Replay buffer** –> We save transitions $(s, a, r, s', \text{done})$ and train on random batches. If we trained on consecutive transitions, the network would overfit to whatever's happening right now.
- **Target network** –> A copy of the main network that we update every 50 episodes. We use it to compute targets during training. If we used the same network for both, the targets would keep shifting and training wouldn't converge.
- **Epsilon decay** –> ε starts at 1 (100% random) and goes down to 0.05 over 50k steps. So the agent explores a lot at first, then gradually starts using what it learned.

The training follows the Bellman equation:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q_{\text{target}}(s', a')$$

We compute the difference between what the network predicted and this target, and minimize it with MSE loss. We also clip gradients to a max norm of 1.0 to avoid big jumps during training.

3.2 Architecture

We use a simple MLP with 3 fully connected layers:

$$25 \xrightarrow{\text{ReLU}} 256 \xrightarrow{\text{ReLU}} 256 \to 5$$

Input is the flattened 5×5 observation (so 25 numbers), output is one Q-value per action. Nothing fancy, the observation is small enough that a basic MLP handles it fine.

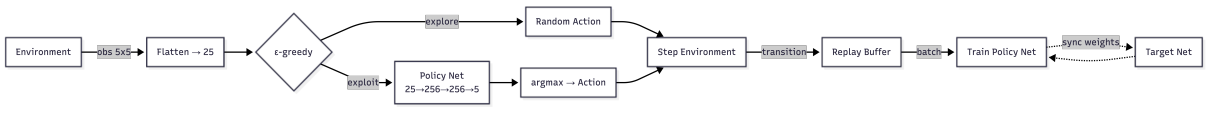

Figure 2: Architecture of the DQN

3.3 Hyperparameters

Parameter	Value
Total steps	100,000
Batch size	64
Discount factor (γ)	0.8
Learning rate	5×10^{-4}
ε start $\rightarrow$ end	$1.0 \rightarrow 0.05$
ε decay	over 50k steps
Replay buffer size	15,000
Target net sync	every 50 episodes
Warmup	1,000 steps
Gradient clipping	max norm 1.0
Optimizer	Adam
Loss	MSE

Table 1: Final DQN hyperparameters.

3.4 What we tried

It took us a few tries to get here. Here's a summary of our experiments:

Experiment	Config	Steps	Reward	Notes
DQN (128, γ=0.99)	20 vehicles, sync 50	100k	29.57 ± 1.78	Stable but lower
Double DQN (128, γ=0.99)	30 vehicles, sync 25	200k	27.28 ± 6.77	Worse
DQN (256, γ=0.8, clip)	20 vehicles, sync 50	100k	$\mathbf{33.40 \pm 8.15}$	Best score

Table 2: DQN experiments.

First version (γ=0.99, 128 neurons). Our first DQN used a smaller network and $\gamma = 0.99$. It scored 29.57 with very low variance; the agent played it safe and almost never crashed, but it didn't take enough risks to get a higher score.

Double DQN. We tried Double DQN, where the policy net picks the action and the target net evaluates it. The idea is to avoid overestimating Q-values. We also bumped vehicles to 30 and trained for 200k steps. It actually performed worse; the agent couldn't handle the denser traffic during training.

Final version (γ=0.8, 256 neurons, clipping). The thing that really made the difference was lowering γ from 0.99 to 0.8. With $\gamma = 0.99$ the agent tries to plan far ahead, but on a highway with aggressive drivers that doesn't really work (things change too fast!). With $\gamma = 0.8$ the agent focuses on what's right in front of it: don't crash *now*, dodge *now*. The reward went from 29.57 to 33.40. We also made the network wider (256 instead of 128) and added gradient clipping to keep training stable.

The downside is more variance (± 8.15 instead of ± 1.78). The agent drives more aggressively, so sometimes it gets unlucky and crashes. But on average it does much better (It's very impressive).

3.5 Result

Mean Reward: 33.40 ± 8.15 | **Mean Episode Length:** 36.73 ± 8.20

```
=== DQN Evaluation ===
evaluating Model on 30 episodes ...
100%|██████████| 30/30 [02:14<00:00,  4.49s/it]

------------------------------------------------------
Results :
        - Mean Reward: 33.403 ± 8.15
        - Mean elapsed Time per episode: 36.733 ± 8.20
------------------------------------------------------
```

Figure 3: DQN Evaluation

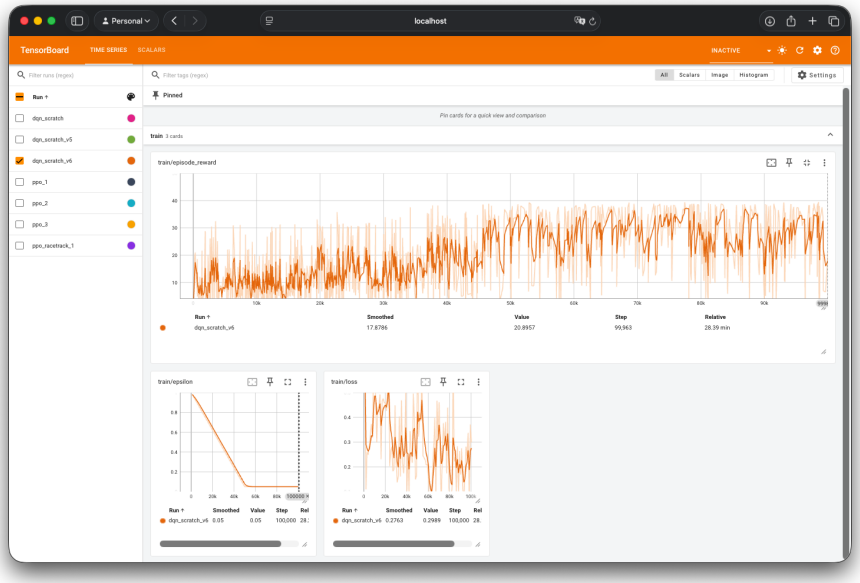

Figure 4: DQN training curve from TensorBoard

When we watched the videos, the agent was clearly driving well : weaving between cars, changing lanes to avoid slow vehicles. Some episodes it scored above 40, it was like watching Baby Driver ;)

4 PPO — Stable-Baselines3

4.1 How it works

PPO works differently from DQN. Instead of learning Q-values, it directly learns a policy : a function that says "in this state, do this action". It updates the policy using gradient ascent on the expected reward, but clips the updates so the policy doesn't change too much at once. This makes training more stable.

We used the implementation from Stable-Baselines3 with 4 environments running in parallel to collect data faster.

4.2 Hyperparameters

Parameter	**Value**
Total timesteps	100,000
Parallel envs	4
Steps per rollout	512
Batch size	64
Epochs per update	10
Learning rate	5×10^{-4}
Discount factor (γ)	0.99
GAE λ	0.95
Clip range	0.2
Entropy coefficient	0.02
Policy	MlpPolicy

Table 3: PPO hyperparameters.

4.3 What we tried

Our first PPO run used the default SB3 settings (lr=3×10^{-4}, n_steps=256, no entropy bonus). It got to about 27–28 reward around 20k steps and then just stayed there.

We tried changing things: higher learning rate, longer rollouts (512 steps), added entropy to push it to explore more. Nothing really worked... the curve always flattened at the same place.

Since lowering γ to 0.8 worked so well for our DQN, we tried the same thing with PPO. The score dropped to 25.29 ± 10.51 with high variance. That said, the driving style was noticeably more aggressive and fun to watch, but the agent crashed too often for it to pay off score-wise. PPO seems to need a higher γ since it learns over full rollouts and needs to see longer-term consequences.

4.4 Result

Mean Reward: ∼27.8 | Plateaued after ∼30k steps

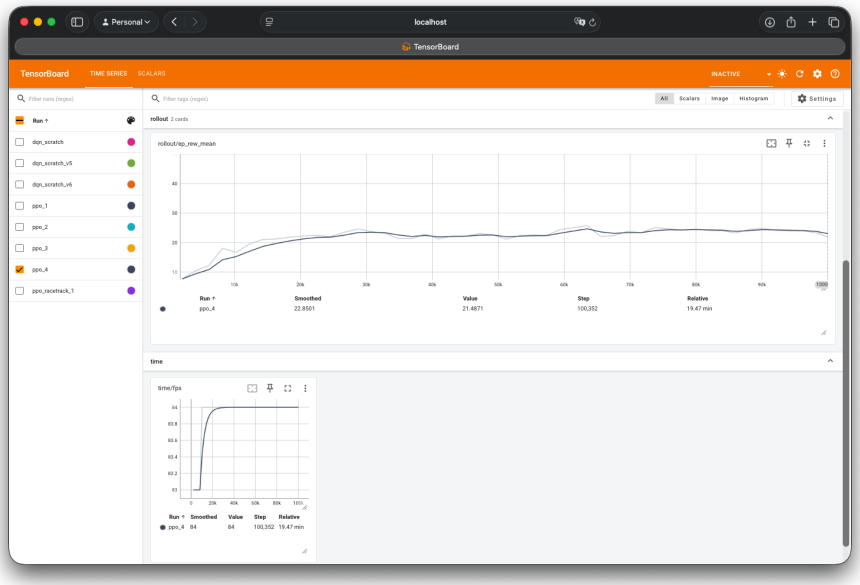

Figure 5: Enter Caption

5 Comparison

	DQN (ours)	PPO (SB3)
Mean Reward	$\mathbf{33.40 \pm 8.15}$	∼27.8
Episode Length	36.73 ± 8.20	—
Training time	∼28 min	∼20 min

Table 4: DQN vs PPO.

DQN did better than PPO here. We think that's because DQN is made for problems with a small number of discrete actions, and we only have 5. The replay buffer also helps a lot, DQN can reuse old experiences to train on, while PPO uses each batch of data once and throws it away.

PPO is meant for harder problems with continuous actions or huge action spaces. For our case it's overkill and doesn't bring anything extra.

Looking at the training curves, DQN kept getting better through the whole 100k steps, while PPO got stuck early on and couldn't improve no matter what we tweaked.

6 Bonus: Racetrack

We also gave the `racetrack-v0` environment a try. Here the agent has to steer around a track while avoiding other cars. The big difference is that the actions are **continuous** (steering angle), so DQN doesn't work; it can only handle discrete choices. PPO handles continuous actions natively so we used it here.

We trained for 200k steps with 4 parallel environments. The agent learned to follow the track very good, and mostly avoids crashing into others.

7 Conclusion

Our best model is the DQN we wrote from scratch, with a score of 33.40 ± 8.15 against 40 aggressive vehicles. The biggest thing we learned is that γ matters a lot. Lowering it to 0.8 made the agent react to what's happening right now instead of trying to plan ahead in a chaotic environment, and that's what got us the best score.

We also saw that fancier doesn't always mean better. Double DQN was supposed to improve things but it didn't. And PPO, which is a more general algorithm, couldn't match our simple DQN on this specific task. It really depends on the problem.

For the racetrack bonus, PPO was the right choice since DQN can't handle continuous actions. So both algorithms have their place; it just depends on what you're trying to solve !