TASK 2 & 4 - Continuous Control with REINFORCE

Malek Bouhadida, Ammar Mariem, Mohammed El Barhichi

`https://github.com/elbarhichi/Reinforcement-Learning-Project`

Contents

1 Selected Environment

The environment we selected for these two tasks is `racetrack-v0` from `highway-env`. This is a continuous control environment, where, the agent needs to learn two skills: **follow the tracks** and **avoid collisions with other vehicles**.

We chose to keep the default configuration of the environment. We only changed the duration because the simulations took too long so we chose to truncate the simulation when it exceeds 60 time steps. In this environment, the reward is designed as follows:

- The **lane centering reward** $= \frac{1}{1+c_{lc}\cdot\text{lateral}^2}$ penalizes being far from the lane center, where c_{lc} is a cost and lateral is the lateral distance from the lane center.
- The **action reward** $= \|\text{action}\|$ penalizes large control inputs so that the agent does not zigzag on the road.
- The **collision reward** $= 1$ if the agent has crashed and 0 otherwise.
- The **on road reward** $= 1$ if the agent is on the road and 0 otherwise.

The weighted sum of these rewards is then computed, mapped to [0,1] and multiplied by the **on road reward** to obtain the final reward: the agent gets a positive reward only if it is on the road or did not crash.

2 Selected Approach

In continuous action spaces, value-based methods like Q-learning struggle, as estimating values over an infinite range of actions is both impractical and conceptually limiting. Instead, we use policy gradient methods to directly learn the policy, by maximizing expected return via gradient ascent, using the log-likelihood of sampled actions weighted by their returns. In the continuous case, the policy is represented as a Gaussian distribution, with the network outputting the state-dependent mean and standard deviation.

2.1 REINFORCE with Continuous Actions

In both tasks 2 and 4, we use the REINFORCE algorithm, adapted for continuous control by sampling actions from the Gaussian policy and updating the parameters based on episode returns.

The core idea behind REINFORCE is the following: we want to maximize the expected return $J(\theta)$ under the current policy π_θ. Using the log-likelihood trick, the gradient of the expected return can be expressed as:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\cdot\gamma^t G_t\right]$$

In the continuous case, $\pi_\theta(a|s)$ is modeled as a normal distribution $\mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$, and the log-probability of the action is computed accordingly.

To train our agent using REINFORCE algorithm, our approach was to:

1. Collect full episodes by interacting with the environment using the current policy. We chose to run 400 episodes in total.
2. For each episode:
    - Store the state, action, and reward at each time step.
    - At the end of the episode, compute the discounted returns G_t for each time step.
    - Normalize the returns to reduce variance in the gradient estimates.
    - Pass each state through the policy network to obtain the mean $\mu_\theta(s_t)$ and standard deviation $\sigma_\theta(s_t)$ of the Gaussian policy.
    - Use these to define the action distribution: $\pi_\theta(a_t|s_t) = \mathcal{N}(a_t \mid \mu_\theta(s_t), \sigma_\theta(s_t))$.
    - Compute the log-probability of each taken action under its corresponding Gaussian distribution.
    - Multiply the log-probabilities by the normalized returns to obtain the policy gradient estimates.
3. After a batch of episodes, compute the average loss and perform a gradient update on the policy network.
4. After 20 episodes, we evalutate the so far trained agent using Monte Carlo evaluation.

2.2 Adjustments to the Lab Code

We used the Lab6 code as a foundation and introduced the folowing modifications:

- We moved the calculations to the GPU to speed up training.
- In some episodes, all rewards were zero, making the standard deviation zero or near-zero. To avoid division by very small values during normalization, we introduced a threshold, so that this normalization is only applied if the standard deviation exceeds this threshold.
- Evaluation was a bottleneck. To address this, we used a vectorized environment, allowing multiple episodes to be run in parallel, speeding up the evaluation significantly.
- We performed hyperparameter-tuning to improve the agent's behaviour and learning.

3 Task 2

3.1 Results and Analysis

Table 1 shows the final configuration of hyperparameters we selected.
With this configuration, we managed to get good results despite using the REINFORCE algorithm, which is considered a vanilla policy gradient method. The agent learned to successfully navigate the environment and followed the road without crashing.
From figure 1, we observe that while individual episode returns (in light blue) display significant variance, both the smoothed training return (blue) and the evaluation return (green) show a clear

Hyperparameter	Value	Description
learning_rate	0.001	Step size for each iteration
batch_size	5	Number of samples in a batch
N_episodes	400	Total number of episodes
gamma	0.8	Discount factor used to compute the returns
net_hidden_size	128	Size of the hidden layer in neural network

Table 1: Hyperparameters and their descriptions

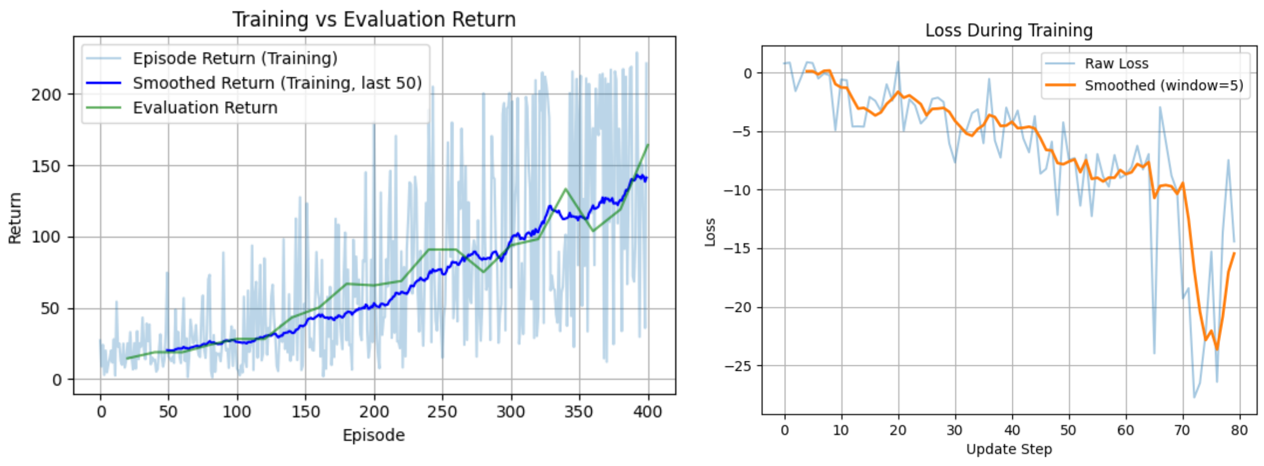

Figure 1: Training performance of the REINFORCE agent. On the left, the evolution of the training and evaluation returns, steadily improving, despite the high variance in per-episode returns. On the right, the evolution of the loss, showing a downward trend and an effective policy updates.

upward trend. This indicates consistent improvement in the agent's performance over time. Initially, the evaluation return hovers around 20, but after 400 episodes, it reaches nearly 170.

This improvement is further supported by the loss plot. Although the raw loss signal is noisy, the smoothed loss curve shows a steady decline, reflecting effective policy optimization. In fact, the smoothed loss decreases from approximately 0 to -15 over 80 update steps (each corresponding to a batch of 5 episodes).

It is important to note that in this context, lower loss values are not necessarily closer to zero. Here, the loss corresponds to the negative expected return weighted by the log-probability of the taken actions. As a result, we obtain significantly negative values. In this setup, a more negative loss corresponds to better performance, as it indicates stronger alignment between the sampled actions and high-return trajectories.

3.2 Limitations & Improvements

As explained in the previous subsection, the agent learned to navigate the environment and followed the road without crashing. However, it is important to mention that it exhibits some zigzagging behavior. One potential improvement could be reducing the weight of the action reward from -0.3 to -0.6, which would encourage the agent to take action only when necessary.

Beyond theis task-specific limitation, there are fundamental drawbacks associated with the REINFORCE algorithm itself. As a vanilla policy gradient method, it suffers from high variance in gradient estimates and slow convergence, particularly in continuous action spaces. Since it relies on Monte Carlo estimates of returns and updates the policy using data collected from a single policy iteration (i.e., on-policy), the learning process is inherently noisy and sample-inefficient.

Furthermore, our implementation discards past episodes after each update, meaning that only the most recent batch of episodes contributes to learning. This limits data efficiency and exacerbates REINFORCE's on-policy nature. In contrast, off-policy methods such as DDPG or TD3 use experience replay buffers to store and reuse past interactions, significantly improving sample efficiency and training stability. These more advanced algorithms could be valuable alternatives for future work.

3.3 Additional experiment

We tried changing the `longitudinal` value in the environment configuration to `True`, so that the agent can accelerate and decelerate. Our intuition was that to avoid collisions, the agent will learn to speed up / lower its speed.

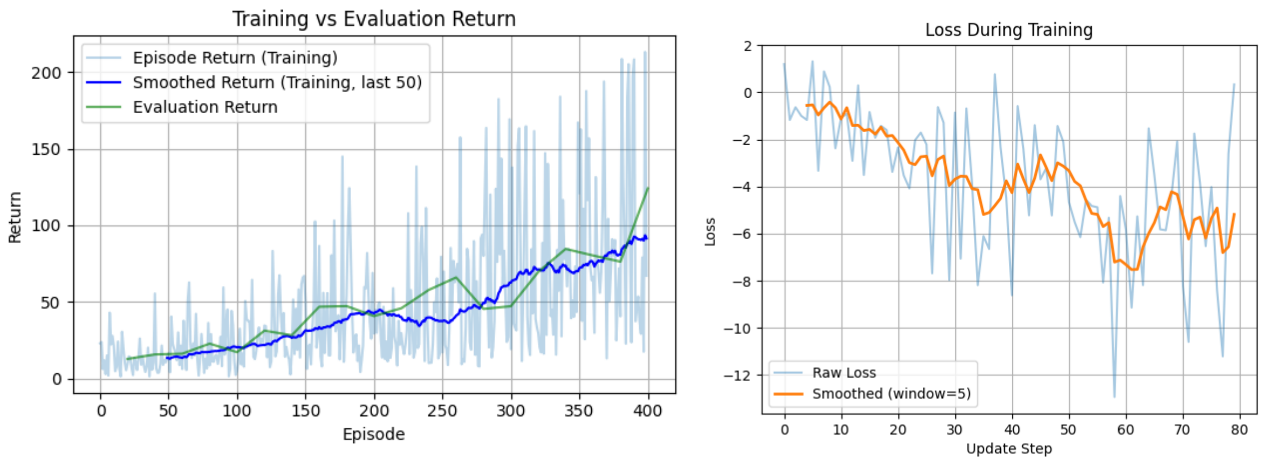

Figure 2: Training performance of the REINFORCE agent, where longitudinal=True

The learning was not as good as with the case without this additional degree of liberty, for the same choice of hyperparameters we have a poor performance as shown in figure 2, even though the return curve continues increasing and the loss curve continues decreasing. Moreover, after running a few simulating, we discovered that the agent just decided to stop in the load without doing anything.

This behavior likely emerged because the current reward structure unintentionally favors inaction as there's no explicit incentive for forward movement or reaching a goal. The agent learns that staying still minimizes its expected penalty. To counter this, we can introduce a positive reward term for speed or distance traveled, and penalize extended periods of inactivity.

3.4 Difference with discrete actions

It is true that the environments of task1 (discrete) and task2 (continuous) are different, however, it is interesting to compare the behaviours and our results in those two environments.

In the continuous setting, the agent learns to adjust steering in a smooth and gradual manner. This leads to more natural driving behavior, with fewer abrupt turns or jerky movements (even though we couldn't reach this level of performance). In contrast, with discrete actions, the agent is limited to a fixed set of options (turn left/right sharply, accelerate/decelerate in steps), which often results in more erratic behaviour.

The continuous policy allows the agent to fine-tune its actions based on small variations in the state, such as adjusting speed slightly when approaching a curve. In the discrete case, the granularity is limited, which can cause the agent to either over- or under-correct, particularly in complex parts of the road.

While the continuous policy provides better control, training was more challenging. The REINFORCE algorithm in continuous spaces suffers from higher variance and requires more samples to converge. In comparison, discrete-action agents often learn faster and more reliably, particularly when using value-based methods like DQN.

4 Task 4

For task 4, we chose to use the same environment as task2, namely the `racetrack-v0`. In this section we will study the impact of certain hyperparameters on performance, namely the batch size and gamma. We will also explore whether our trained algorithm generalize to other environments. And at the end, we will see what happens if we use our trained model to drive two cars at the same time.

4.1 Impact of hyperparameters

4.1.1 Batch size

We began our experiments by varying the batch size to observe its effect on training performance. Figure 4 illustrates the evolution of episode returns across three different configurations.

We clearly see that a larger batch size =20 results in lower performance. This is because, with a fixed number of episodes = 400, a higher batch size means that the policy network is updated less frequently. It is also interesting to note that, in this case, the episode returns exhibit less variability. In contrast, with a smaller batch size of 5, the returns increase more significantly and display greater variance, and the smoothed return curve have a higher variance compared to the others. The setup with a batch size of 10 exhibits behavior that falls between the other two cases, both in terms of performance and variability.

4.1.2 Gamma

We know keep all other hyperparameters fixed and vary gamma. In the case where gamma=0.95, the agent values long-term rewards more and delayed gratification. On the contrast, when gamma=0.8,

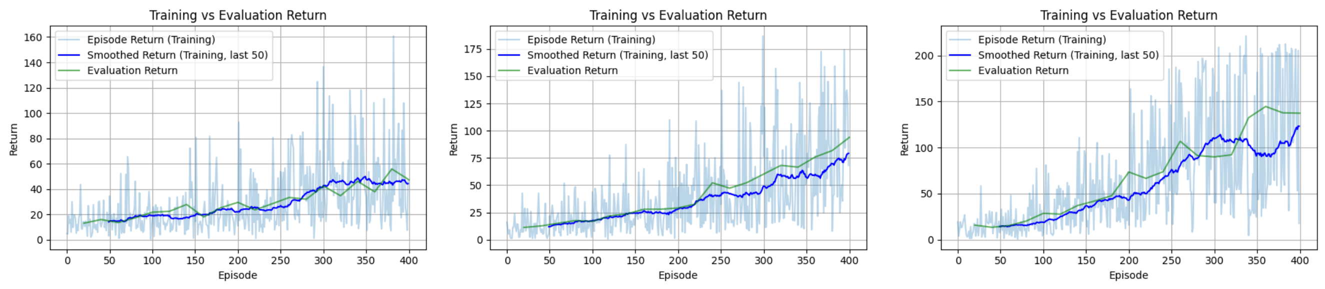

Figure 3: Training performance of the REINFORCE agent for different batch sizes: 20 (left), 10 (center), and 5 (right).

the agent focuses more on immediate rewards, making training potentially faster and more stable for the short term, but suffering from myopic policies.

For the configuration of hyperparameters we selected (batch_size=10, learning_rate=0.001), we see that the learning is faster with gamma=0.9, even though it decreases at the end, suggesting that the policy may be overfitting to short-term gains rather than optimizing for long-term success. On the other hand, $\gamma = 0.95$ leads to more stable and consistent improvements of the evalution returns. In fact, this highlights the trade-off between training speed and policy foresight. Learning fast is not what we are looking for, our choice of the hyperparameters should be guided by the temporal structure of rewards in the environment.

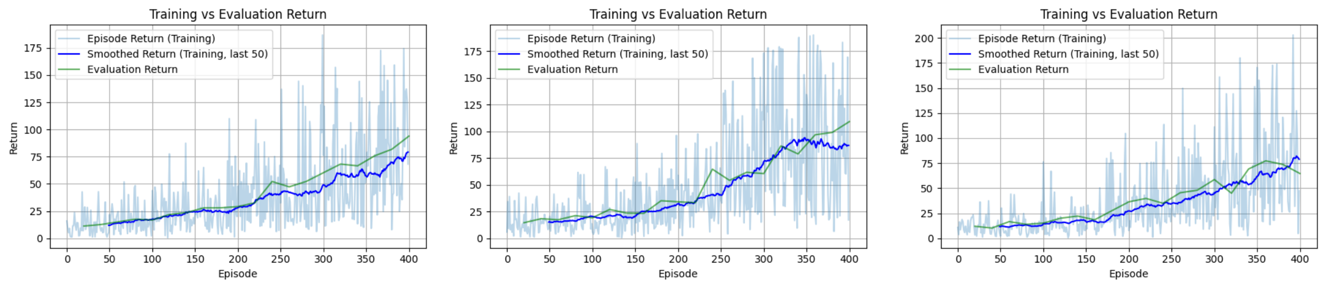

Figure 4: Training performance of the REINFORCE agent for different values of gamma: 0.95 (left), 0.90 (center), and 0.80 (right).

4.2 Generalisation to other environments

Now we move to a new kind of experiments, we will be testing our trained agent in new environments:

- A first `racetrack-v0` environment with a slightly harder configuration, namely with 4 vehicles on the road. So the agent needs to avoid collision more frequently.
- A second `racetrack-v0` environment with 6 vehicles on the road.
- A third `racetrack-v0` environment with 9 vehicles on the road.

We ran 10 simulations in each environment. The results in table 2 show that our agent performs best on the environment with 4 more vehicles, which is not intuitive. We believe this result may be due to the limited number of simulations, which introduces variability in performance. Averaging over a larger number of runs would give us more reliable results. On the other hand, the agent performs worst in the environment with 9 other vehicles, which is expected given the increased complexity and likelihood of collisions in denser traffic.

Environment	Average Return	Collected Returns (rounded)
previous env	106.5	39, 41, 56, 225, 120, 211, 43, 39, 222, 56
previous env with 4 other vehicles	146.4	224, 75, 22, 227, 224, 99, 94, 51, 228, 97
previous env with 6 other vehicles	118,6	83, 136, 90, 52, 179, 195, 219, 85, 44, 221
previous env with 9 other vehicles	99.1	65, 37, 228, 125, 222, 43, 42, 184, 67, 108

Table 2: Results of 10 runs for each environment.

We also wanted to test our agent in totally different environments from `highway env` but we did not succeed as the shapes of action spaces and observation spaces did not match, which needed additional adaptation.

4.3 Testing with Multi-agent setup

We now move to our final experiment, changing the configuration of our environment differently. We keep the previous environment used to train our model and change the number of controlled vehicles. We test with 2 controlled vehicles and 1 other vehicle on the road. We observe instantly that one of the controlled vehicles performs as expected but the other one gets out of road immediately. And in other simulations, we saw both cars performing very badly and both getting out of road, as shown in figure 5.

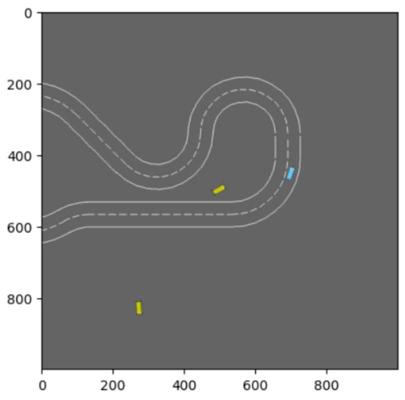

Figure 5: Both controlled cars get out of road in this setup.