CS 5955/6955 Advanced Artificial Intelligence Spring 2025

Homework 4: Q-Learning
Student: Simón Pedro González
UID: 1528314

1 Tabular Q-Learning to Win Big (sort of) at Blackjack

The Q-function representation consists of a numpy array of shape:

$|\{\text{Player sums}\}| \times |\{\text{Dealer Card}\}| \times |\{\text{usable ace, no usable ace}\}| \times |\{\text{hit, stand}\}|$
$= 21 \times 11 \times 2 \times 2$

Note that we don't need to calculate Q estimates for states in which the player sum is over 21, since the game ends at that point. The Q-function was initialized with 0. Both the ϵ-greedy parameter ϵ and the learning rate α were obtained through an exponentially decaying schedule. The discount factor used is $\gamma = 0.99$. To compare Q-learning with a random policy, I run 100000 training rounds and calculated the average reward over a 10000 rounds window. The results can be seen in Fig. 1.

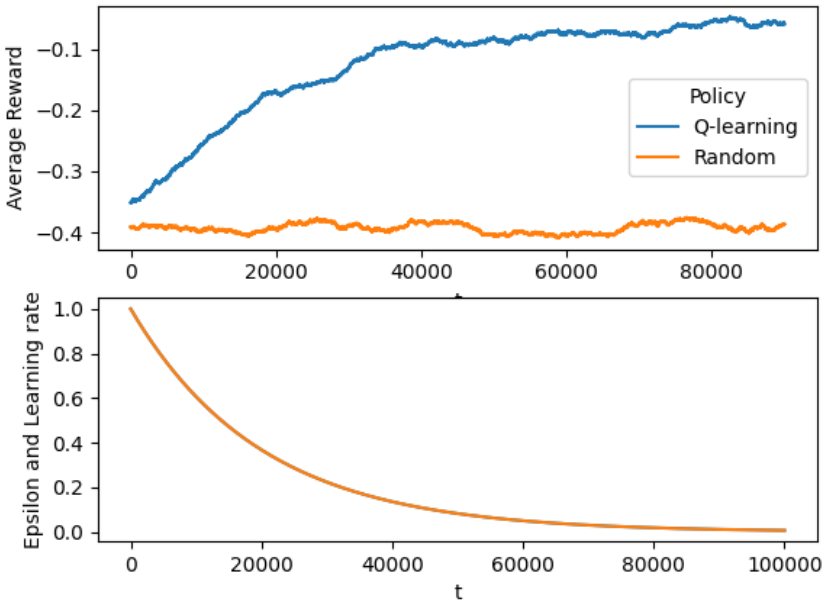

Figure 1: Top: Average reward over time. Bottom: learning rate and epsilon schedule.

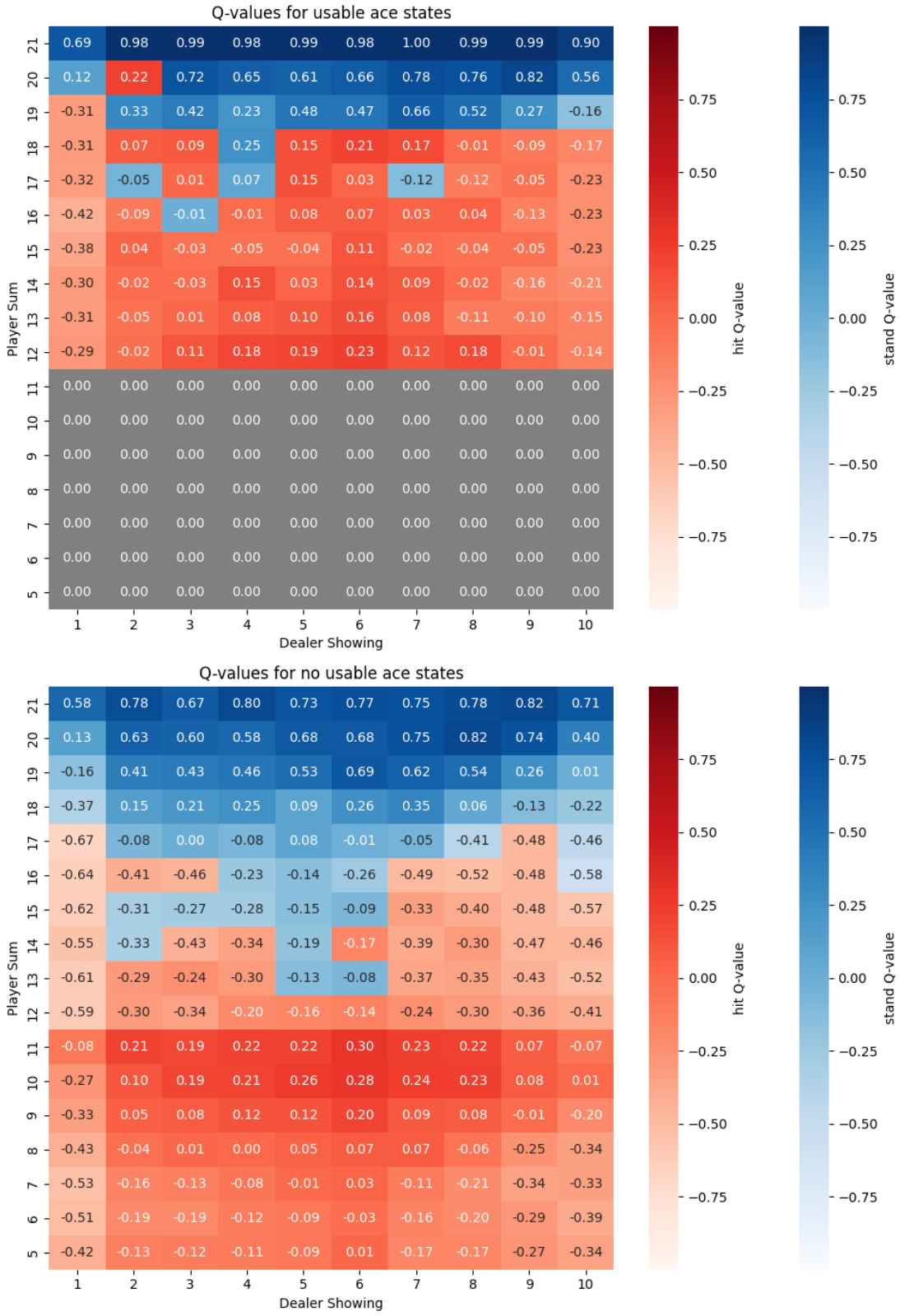

Figure 2: Q-function estimates for the best (argmax) actions in Blackjack. The color hue indicates the action and the brightness indicates the Q estimate value.

The obtained policy is close, but has not reached yet, an average reward of -0.01, which would correspond to a win rate of about %49.5, the theoretical limit when playing perfectly.

The Q-function estimates are shown in Fig. 2. We can see how a weak dealer's upcard (2-6) encourages the agent to stand with lower values (13-17). However, when the upcard is strong (7-10), the agents has to play more aggressive. Also, when a usable ace is in play, the agent tends to hit with higher scores (17-18), since it is less risky. Finally, we can see some puzzling behavior, for example, hitting in the state "(20 player sum, 2 dealer's upcard, usable ace)", which is, possibly, evidence that the policy is not optimal yet.

2 Landing on the Moon using DQN

The results from the random policy and human control are shown in Fig. 3. The environment is very challenging, and my performance was very inconsistent. However, I was able to surpass the random policy, achieving an average of approximately -100, whereas the random policy gets an average of about -180.

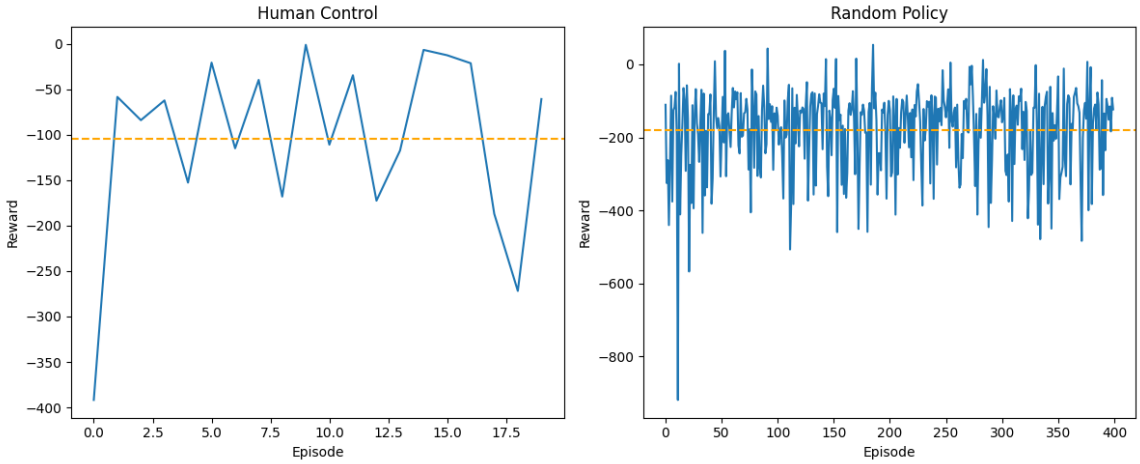

Figure 3: Human and random policy performance on the Lunar Lander environment.

The DQN policy performs significantly better, as shown in Fig. 4. After rendering several rounds with the DQN policy controlling the inputs, I observed that it cycles among thrusters at a very high speed, making it seem like all thrusters are on at the same time. After training for 600 episodes on a GPU, the agent achieves an average score of about 200.

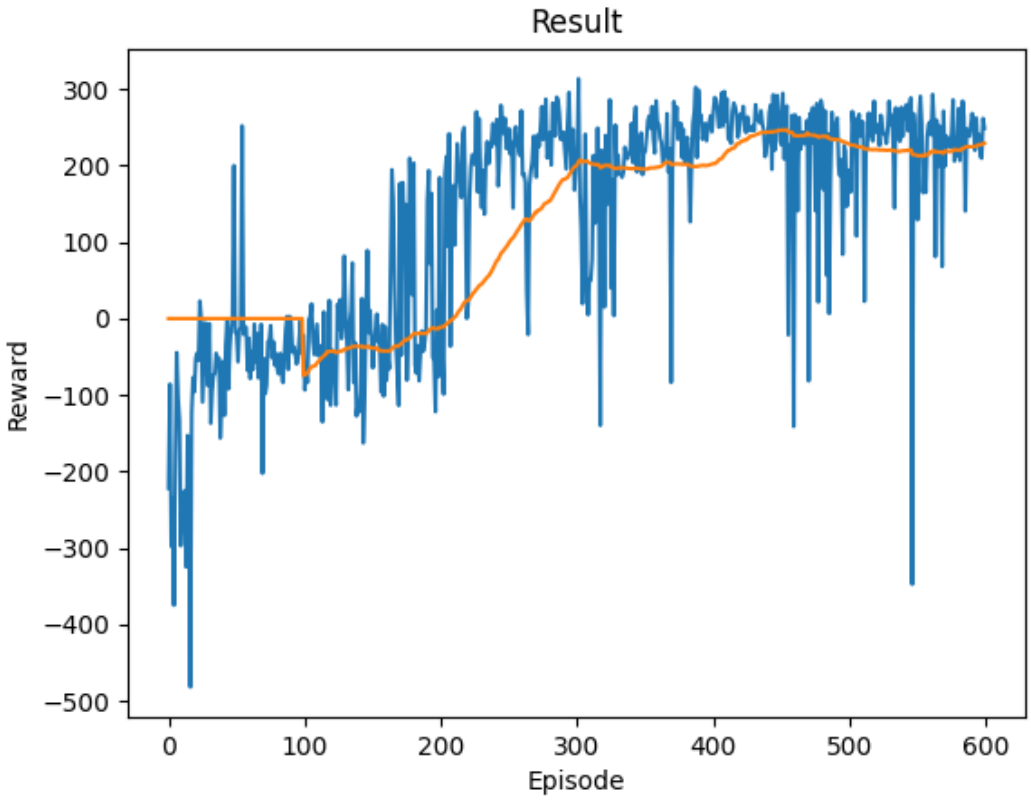

Figure 4: DQN performance on the Lunar Lander environment.

3 Blackjack exploration methods

The Q-learning algorithm was modified to use different exploration methods. The learning rate was set to $\alpha = 0.01$ in all cases (no learning rate schedule). The tested methods are:

- Fixed ϵ-greedy with different ϵ values.
- Boltzmann with $\beta = 10$.
- UCB1.
- ϵ-greedy with exponentially decaying ϵ (the same used in section 1).

The results, shown in Fig. 5. The overall pattern is similar to what was observed in the multi-armed bandit experiments. UCB1 and Boltzmann exploration exhibit a slower start and a consistent upward trend, while fixed-epsilon policies quickly achieve a high average

but lack a clear growing trend. The exponentially decaying ϵ-greedy method appears to show a stronger growing trend than the other methods.

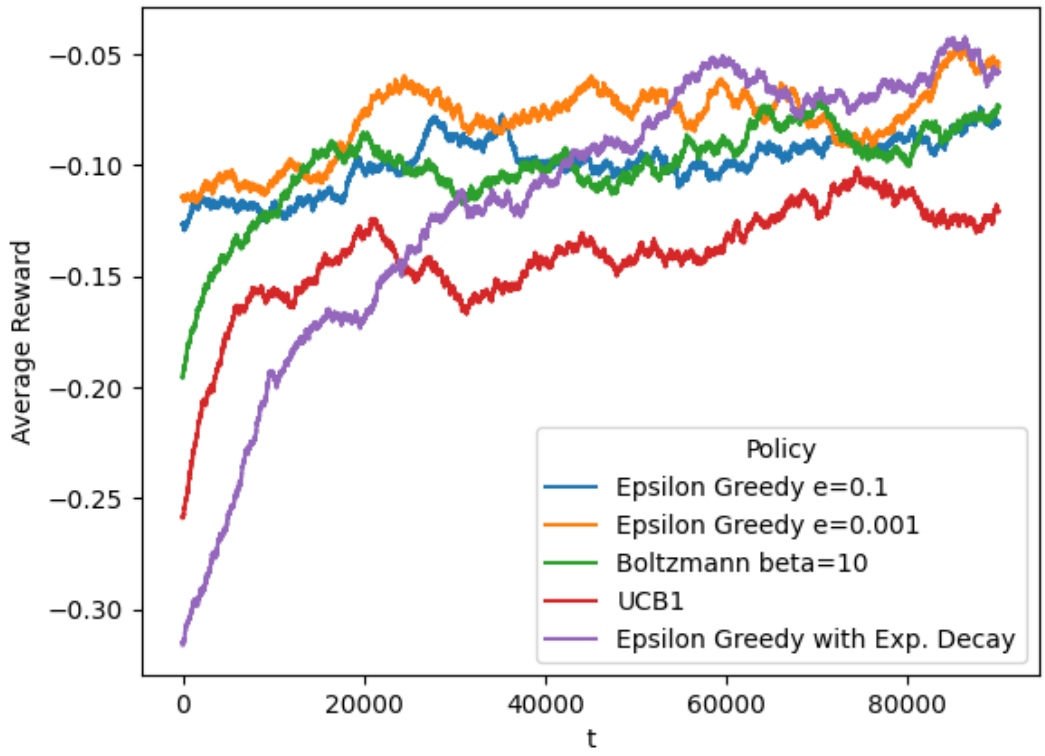

Figure 5: Average reward over time in the Blackjack setting with different exploration methods.

4 Convolutional DQN for the Car Racing environment

My first implementation, using mostly the same code as in the Lunar Lander experiment, was too slow to train. That is why I took several ideas from this repository [2]. The new implementation consists of the following:

1. An ϵ-greedy policy with an exponentially decaying ϵ schedule, just like the one used in section 1.

2. Environment wrappers to modify the state representation. Instead of a $96 \times 96 \times 3$ channels (RGB), the state is represented as an $84 \times 84 \times 4$ frames. The wrappers apply the following transformations:

- Transform to grayscale and reduce the resolution.
- Skip 4 frames: the passed action is repeated for 4 frames.
- Frame stacking: stack the last 4 frames in a single state representation to provide temporal context.

3. Two DQN networks (policy and target) with the following architecture:

Layer	**Description**
Conv2d	in: 4 channels, out: 16 channels, kernel size: 8, stride: 4
ReLU	Activation
Conv2d	in: 16 channels, out: 32 channels, kernel size: 4, stride: 2
ReLU	Activation
Flatten	Flattens the feature maps
Linear	FC: input 2592, output 256
ReLU	Activation
Linear	FC: input 256, output 5 actions

Table 1: DQN Architecture

4. A wrapper class around the `TensorDictReplayBuffer` from the TorchRL library [1], with `push` and `sample` functions.

5. An `Agent` class that combined the DQN networks, the buffer, the `AdamW` optimizer and the `SmoothL1Loss`. The `Agent` class has the follwoing functions:

   - `act(state)->action` produces actions with ϵ-greedy given the Q estimate from the DQN.
   - `update()` samples the buffer and updates the policy network taking the target network as reference.
   - `sync_target()` copies the policy network into the target network. The direct copying approach was chosen over the smooth update with a τ parameter (as used in the Lunar Lander experiment) because updating the weights every iteration was too slow.

The agent was trained for 600 episodes. The results can be seen in Fig. 6 and Fig. 7. We can observe that performance plateaus around episode 350.

The agent far surpasses my own performance. It consistently achieves nearly 800 cumulative reward, but occasionally goes off track. In those cases, the final reward depends on whether it recovers effectively or ends up driving in the wrong direction. Here is a video showing how the agent recovers successfully: `https://drive.google.com/file/d/1A2ix2boMwEQwzDhWgbMOolo7cuJiPg3f/view?usp=sharing`.

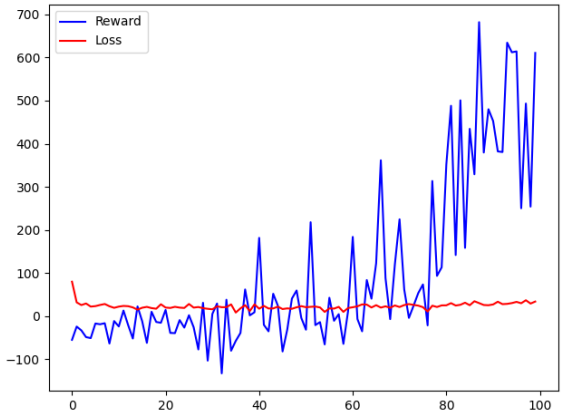

Figure 6: Cumulative reward per episode and Loss for the first 100 training episodes

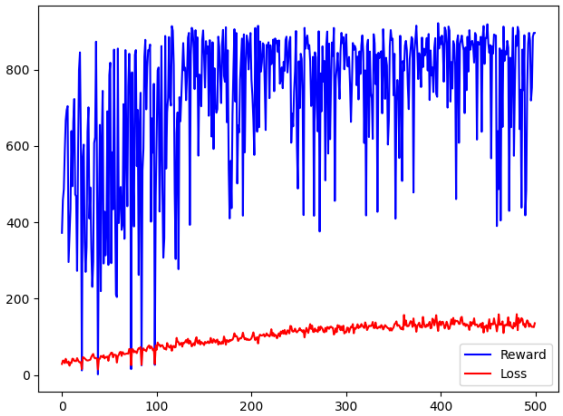

Figure 7: Cumulative reward per episode and Loss for episodes 100 to 600.

References

[1] Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. Torchrl: A data-driven decision-making library for pytorch, 2023. URL: `https://arxiv.org/abs/2306.00577`, `arXiv:2306.00577`.

[2] Vladislav Govor. Dqn car racing. URL: `https://github.com/wiitt/DQN-Car-Racing/tree/main`.