**CMPS 497 – L04 - Reinforcement Learning**

**Dr. Badawy Ahmed**

**Group ID:** 1  
**Students:**

Abdulla Al-malki – 202009135

Syed Subzwari – 202105006

Abdollah Kandrani – 202104093

**Overview:**

Our project focuses on the Gymnasium CarRacing environment, which is built on Box2D—a 2D physics engine for games. In this environment, a random racing track is generated each episode, and the objective is to train an agent to drive through the entire track successfully. The project aims to apply both **tabular reinforcement learning** and **function approximation methods** to this environment and perform a comparative analysis between their performance.

The action space in CarRacing can be either **continuous** or **discrete**. In the continuous version, the action is a 3-dimensional vector: steering (ranging from -1 to 1), gas (0 to 1), and brake (0 to 1). The discrete version includes five predefined actions: 0 = do nothing, 1 = steer left, 2 = steer right, 3 = gas, and 4 = brake. The observation space is a 96x96 RGB image representing the top-down view of the track and the car.

The final deliverables will include a Jupyter Notebook containing the implementation, a written analysis comparing the two RL approaches, and a discussion with conclusions about their strengths, weaknesses, and applicability in continuous control environments.

**Literature Review:**

**The problem of large state size:**

Through initial research (with ChatGPT) and discussions with our instructor, we have found that Car Racing only produces pixels as an observation space, and this is huge *(27,648)* for tabular and function approximation methods. Therefore, we need to handle this case before proceeding with the project. We must find a way to reduce the state size or handle it differently.

**A blog covering solving Car Racing with PPO (a deep reinforcement method):**

We have walked through a blog made by an Engineer who refers to himself as Mike (https://notanymike.github.io/Solving-CarRacing/). The blog covers how to train the Car Racing model with Proximal Policy Optimizations (PPO), which looks like a deep RL method but nevertheless, his discoveries are insightful for our case.

**Discrete actions Case:**

Car Racing offers a discrete action space which makes the agent only perform one actions at a time (do nothing, turn left, turn right, gas, brake) which provides simplicity but produces a new problem when dealing with track curves. In a real scenario, the car brakes AND turns as one action. This might affect the performance of our case if we did not handle it.

The author handled the case by providing soft discrete actions. It provides more actions that perform hard or soft turning which gives full or partial values for turning and accelerating, or turning and braking.

**Simplifying the observation space:**

The author simplified the observation space by:

1. Removing the bottom panel that shows observable stats for the human observer. Since this is useless for the agent, removing it is better to avoid wasteful training.

2. Converting from RGB to gray scaled frame. This reduces the size of the state from 96x96x3 to 96x96.

3. The author stacked 4 consecutive frames. Policy-gradient methods like PPO need **partially observable environments**, so that's why the author stacked the frames. For our case, this is not needed, and we could ignore doing this.

**Modifying the reward structure and termination cases:**

* The author added a **reward clipping** feature, where in a normal scenario the car eventually gets higher rewards, and consequently it gets too confident increase its speed. This in the short term is beneficial but it will later handle curves recklessly. Therefore, the author suggested to clip the reward to a maximum of 1.
* Some scenarios are useless or wasteful to keep computing on. Such cases are when the agent decides to drive out of the track for a long time. The author added a \*\*timeout\*\* feature that sets a limit of time the agent could drive out of the track. If the time limit has passesd, the episode is terminated and negatively rewarded.

**System Model and Problem Definition:**

In this project, we aim to deploy tabular and functional approximation methods on the CarRacing-v3 environment and comparatively analyze their performance. In this environment, a random racing track is generated each episode, and the objective is to train an agent to drive through the entire track successfully. Gymnasium describes this as a control task, which means that we aim to learn policies that will try to maximize the total expected reward during episodes by staying on the track and reaching its destination quickly.

**Observation Space:**

The Observation Space is 96x96 RGB images of the car and track in a top-down view. It should be noted that the state isn’t explicitly given like in Toy Text environments where states are defined as numbers like s = 8. Rather, the images are given as the observed state, and the agent must be able to derive features from it to classify it as a state. While this may seem inconvenient, we need to understand that this is because Toy Text environments are small, so it is easy for us to get a state number. However, in Car Racing the observable state possibilities are so huge that storing it is infeasible. Also, because the number of states is so high, there is an extremely low chance of encountering the same state again even in later episodes. Due to this, we need to generalize the returned image after ‘taking a step’ in the environment regardless of if we are using tabular or function approximation. Actually, pure tabular methods would theoretically use all the states possible but this is infeasible, and even more so for us as we neither have the computational power, time or storage to achieve this.

For a rough idea, both tabular and functional approximation will need us to extract features. In tabular methods we need to be able to discretize the features as well, most probably by organizing them into bins and forming states from them. This is known as State – Aggregation. In function approximation, there is no need to do this as we just need to update our weights. The more encountered states throughout episodes will end up influencing the weights more, and hopefully that leads the car to perform optimally. We can look at the strategies in the slides for ways to construct our features.

**Action Space:**

For this environment, there are two types of action spaces: Continuous and Discrete. If the action space is defined to be continuous, an action is defined as a 3-dimensional vector [s, g, b] containing steering, gas and brake respectively. Steering ranges from -1 to 1, where -1 is fully left and +1 is fully right. Gas and brake range from 0 to 1. The discrete version includes five predefined actions: 0 = do nothing, 1 = steer left, 2 = steer right, 3 = gas, and 4 = brake. As per online research, we will have to create discrete actions on our own. We can do this by predefining the continuous actions as a discrete action, so we could map the continuous action [0, 1, 0] as the gas, for example.

**Reward Structure:**

The agent is punished by -0.1 every frame and is rewarded with +1000/N for every track tile visited. The idea of punishing for every frame aims to encourage the agent to finish as quickly as possible. +1000/N may seem confusing at first but the breakdown of this reward is clever. Initially through the episode, the reward gained is high as “N” is small. However, as we progress through the episode “N” increases, and the reward gained decreases. This is so that the car tries to adopt efficient pathways (take minimum number of steps) to attain maximum reward. The episode terminates if all the tiles are visited (meaning the lap is done perfectly) or if the car goes far off the track where it receives -100 punishment and dies.

**Methodology:**

**Function Approximation Method:**

**Deciding the Control Method:**

In Chapter 9, we have touched upon three prediction methods: Gradient Monte Carlo, TD (0), and LSTD, mainly focusing on the first two. Within these methods, feature construction is important as both methods rely on features (the derivative of the linear function approximator is equal to the features!).

TD (0) is a better method for a couple of reasons. Firstly, we expect the episodes to be long, which TD (0) works well with. This is because TD (0) allows the agent to learn online every step of the way, meaning it is more sample efficient. Gradient MC, on the other hand, would require us to wait for the whole episode to finish before updating weights. The hope is that TD (0) will converge to the optimal policy faster, which is highly desirable to us due to hardware limitations. From this, we can choose SARSA, Expected SARSA, or Q-Learning for control in the approximation method.

            We decided to go with Q-Learning for a couple of reasons. Firstly, Q-Learning learns the optimal policy as it is an off-policy method. The goal is that it will converge faster than SARSA and Expected SARSA in complex environments like CarRacing, where optimal actions are required. Additionally, we have discretized the action space, which works well with Q-learning’s “max” function and would help it. It should be noted that Q-learning is known for its instability, but we feel that its efficient nature would help our particular needs where we have hardware/time limitations.

**Deciding the Feature Extraction Method:**

The other important factor was to decide which feature construction method we would choose within TD (0). Our options were to choose Polynomial, Fourier, and Tile Coding. Polynomial features help capture complex relationships that pure linear (and manual) features can’t extract. However, the issue with this method is that the feature vector can explode with higher number of features and degrees. This can end up making it more computationally expensive because we would have some unneeded features, which is not desirable, as the environment is complex enough as it is. Tile coding, on the other hand, offers a better way to generalize but discriminate when needed because we can efficiently discretize the continuous state space. Hence, we ended up choosing Tile Coding.

**Results and Discussion:**

**– Presentation of results with thorough analysis and critical evaluation:**

**Exploration strategies attempted and rationale behind the final choice:**

We have chosen Epsilon-Greedy for a variety of reasons. Firstly, Epsilon-Greedy is an easy-to-implement method and works well with Q-learning. This is because Q-Learning is an off-policy method, so we can explore with Epsilon-Greedy while Q-learning learns the best action from its “max” function. Also, Epsilon-Greedy provides a balance between exploration and exploitation. One may question why we don’t only explore, and this is because that will slow down the learning, and some exploitation ensures the agent learns from ‘promising’ positions. Decaying Epsilon is also a good method, but it may not have been the best choice for our case. This is because we didn’t know how to tune Decaying Epsilon so that there was enough exploration. This is important because our state space is so huge that prematurely stopping exploration can be harmful to our agent. Hence, we have a fixed Epsilon when training to ensure that the agent eventually learns the optimal weights.

**– Justification for the selected tabular reinforcement learning methods:**

**Justification for the chosen function approximation method:**

As explained in the “Methodology” section of the report, we have chosen the semi-gradient version of Q-learning (TD (0) version) coupled with Tile Coding. Tile coding was selected because it transforms the continuous state space observation into a discretized version. This allows us to balance the level of generalization and discrimination, while being computationally efficient, more than Polynomial or Fourier features. We used Q-learning because it tries to converge quickly to gain optimal weights and its efficiency. Also, our action space is discretized, which would better suit this method. While Q-learning can theoretically be unstable, the off-policy nature, faster learning, and benefits entice us to choose it despite these risks.

**– Analysis of learning rate effects:**

**Description and reasoning behind the reward structures used:**

We have kept the same reward structures as we liked the ones offered by CarRacing-v3. It gives small rewards for visiting new tiles and punishments for going way off-track. The environment also lightly punishes the agent for every frame, and this is done so that the agent doesn’t become comfortable with inefficient driving and starts to drive faster. By doing this, we try to ensure the car completes the track quickly and minimizes the time it is off-road. This reward structure was kept constant in both the tabular and functional approximation methods. It should be noted that reflecting on these objectives will take a long time because the agent will need to continuously explore the right areas and carry out those changes when it finds them. So, if we suppose that the car completes the track correctly with few errors, it will then take time to explore different actions which may yield a better completion time.

**Conclusions and Discussion:**

**Summary of Key Findings:**

**Limitations:**

One of the key limitations we felt was the time needed to train the agent. We have seen in previous works that people tend to use Deep Reinforcement Learning Methods (such as DQN and PPO) which are better suited for CarRacing than the methods in our scope. However, it should be noted that even those methods required large number of episodes (upwards of 30,000!) which seems to be the norm. After such rigorous training, their methods were able to achieve decent results. In our case, it took us several hours just to be able to reach a thousand episodes and coupled with the fact that our methods may be as well suited as DRL methods, it is justifiable that we weren’t able to reach the same level of accuracy as them.

**Potential Directions for Future Work:**

Another area of experimentation could be changing the action space. We could either try the continuous version or perhaps introduce more discrete versions of actions. Some of these actions could look like “Half left turns”, “Half right turns”, “Slower Driving”, etc. which is needed because at the present we only have full-left/full-right turns. This could allow the agent to drive more efficiently, preventing it from going off-road and perhaps allow it to finish faster.