Nafis Abeer
HW 3

Question answers made more coherent with aid from GPT 4. Prompt engineering techniques include rewording answers, feeding in code files, feeding in pdf of reading documents, feeding in html files of helpful webpages to dedicated chat agents. Each agent used for coding and question-answering were gpt-4 advanced data analysis webpage instances.

I'll also include the graphs and weights for the various models as a part of my solution as to not run the report too long. The file names are made trivial under the models directory.

3.1

a) The architecture of the Deep Q-Network (DQN) employed is a Convolutional Neural Network (CNN) tailored for the 96x96 frame inputs of the CarRacing environment. The model consists of three convolutional layers, starting with 32 channels, and increasing to 64 channels for the subsequent two layers. Each layer utilizes the ReLU activation function to introduce non-linearity. During the forward pass, it's ensured that sensor values, vital for decision-making, are processed on the appropriate computational device (either CPU or GPU). Additionally, the observation tensor is reordered to fit the conventional input format for neural networks in PyTorch.

   i. The decision to incorporate sensor values explicitly into the model can be viewed from two angles. Although these values are inherently present in the frames, making them explicit might aid the network in grasping their significance more efficiently, potentially improving training efficacy. In terms of the Markov property, the current state, inclusive of these sensor readings, should encapsulate all necessary information for the agent's optimal decision-making. Therefore, accentuating these values can be seen as reinforcing this principle. The sensor states give information about whether breaks should be applied. The abs could help better control the vehicle in certain situations. Because the network architecture generally had few channels per convolutional layer, the ratio between pixels dedicated to the image frames vs. pixels dedicated to sensor values are better reflected in the number of network neurons dedicated to the tasks.

   Ii. Stacking frames is a strategy employed in certain environments to provide temporal context, especially when individual frames might be ambiguous regarding motion direction. The idea behind the frames being stacked averaged out flickering between even and odd pixels [1]. However, in the context of the CarRacing environment from OpenAI Gym, each frame is self-sufficient, providing a comprehensive view of the current state. There is no "flickering" Consequently, there's no necessity to overlay or average multiple frames.

b) i) Using fixed Q-targets with a separate policy and target network is crucial for the stability of the Deep Q-Learning algorithm. In traditional Q-learning, using the same weights to estimate both the current and future Q-values can lead to harmful correlations and introduce a moving target problem. This can, in turn, lead to oscillations or divergence in the learning process. By using a separate target network with weights that are updated less frequently than the policy network, we decouple the targets from the parameters. This helps in stabilizing the learning updates. The DQN Nature paper emphasizes that using fixed Q-targets was one of the key ingredients to make the training of deep Q-networks more stable.

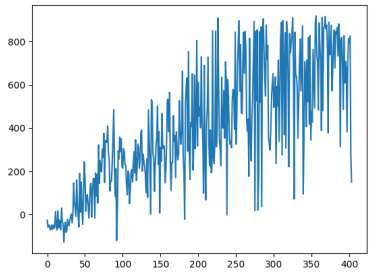
   ii) Sequentially experienced transitions can be highly correlated. Training the network on these in sequence can lead to inefficient learning or harmful feedback loops. By sampling random batches of experiences from the replay memory, these correlations are broken, ensuring that learning is based on a more diverse set of experiences. By storing transitions in the replay memory and sampling from it, past experiences can be reused in multiple updates, making learning more data-efficient. This contrasts with traditional online Q-learning, where past experiences are discarded after a single update. The DQN Nature paper mentions that learning directly from consecutive samples is inefficient due to the high correlation between them, and that using the replay memory allows for more robust learning from temporally uncorrelated samples.

c) The function **select_exploratory_action** employs an ε-greedy strategy to choose an action. Here's how it works: With probability ϵ it selects a random action (exploration). With probability 1 − ϵ, it selects the action that maximizes the Q-values predicted by the policy network (exploitation). ϵ is typically kept small to start and decayed further over time. The function **select_greedy_action** straightforwardly picks the action that maximizes the Q-values as per the policy network, eschewing exploration in favor of pure exploitation.

i) Balancing exploration and exploitation is a foundational challenge in reinforcement learning. Once an agent learns a policy that provides a decent reward, it might be tempted to keep using that known policy to get a surefire reward. This is exploitation. However, there might be better policies (or dishes at the restaurant) that the agent hasn't yet tried, which could potentially provide even greater rewards. To discover these, the agent needs to occasionally try different actions, even if they might initially seem suboptimal. This is exploration. **Too much exploitation**: The agent might get stuck in a local optimum, never discovering potentially better strategies. **Too much exploration**: The agent might never settle on any good policy, continually trying new things and failing to capitalize on what it has already learned.

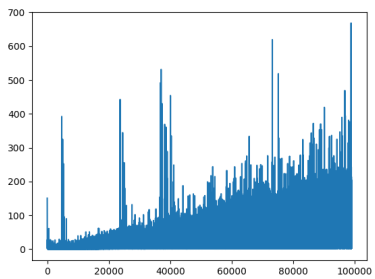
The ε-greedy algorithm strikes a balance by ensuring that, most of the time , the agent chooses the best action it knows (exploitation). But occasionally, it tries a random action (exploration). As learning progresses, ϵ is typically decayed, so the agent explores less and exploits more, reflecting its growing knowledge of the environment. This simple yet effective method allows the agent to learn about the environment while still benefiting from its accumulated knowledge.

d) The agent starts to achieve positive rewards after an initial learning period. The rewards appear to stabilize, indicating that the agent has found a somewhat optimal policy for the environment. This seems to settle rather high, indicating that the parameters chosen for the baseline might be a good fit to find an optimal policy using DQN. I did not have to do much processing. The agent takes about 200 episodes to settle at a likable range:



The ε-greedy exploration schedule ensures that the agent occasionally explores the environment by taking random actions. As training progresses, ε typically decreases, leading the agent to rely more on its learned policy. In the early stages of training, the exploration might lead to suboptimal rewards as the agent is still figuring out the environment. However, as ε decreases and the agent starts to exploit its learned knowledge more, we should see an increase in cumulative rewards. This pattern is evident in the reward curve where, after an initial period of fluctuation, the rewards increase and stabilize.

The loss curve for the DQN exhibits fluctuations throughout training. This behavior is different from a typical supervised learning problem, where we'd expect the loss to decrease steadily as the model becomes better at fitting the training data. The reason for this difference is the nature of reinforcement learning: the DQN continually updates its Q-values based on new experiences, and as the policy improves, the Q-value targets also change. This dynamic target results in a

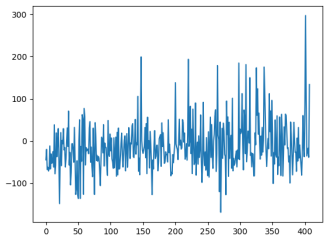


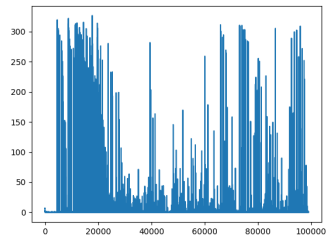
loss curve that can have ups and downs, reflecting the agent's ongoing learning process.

e) The agent actually does really well around corners it could see ahead of time. It does sort of a drifting motion which helps it turn the corner a lot better than I ever could. I think the significance of the abs sensor values is that the model learns the impact from properly reading the sensor values and having smaller parameters and a small set of possible actions helps the agent learn the optimal strategy. It does struggle significantly once it goes off course as I don't think it spent enough iterations on the grass (due to low rewards from grass time) to even learn how to operate properly on the grass. It also does not have set waypoints and a goal so sometimes it turns around and drives the other way. This performs significantly better than dagger because the agent is driving itself and it has one goal of maximizing the results. Instead of imitating us, who make decisions that might not necessarily be reflected in the data due to data imbalance, the model here is able to optimally perform actions that will maximize results, therefore forcing itself to only learn on data that helps it further. The advantage of exploration and exploitation in this procedure is already described above.

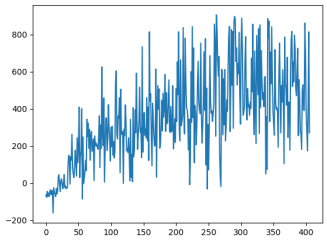
3.2

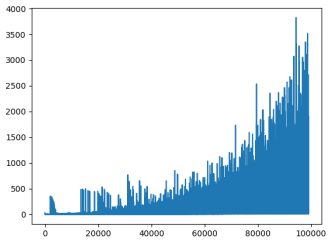
a) For γ=0.5, the agent prioritizes immediate rewards and may not always make decisions that are beneficial in the long run. This can lead to suboptimal policies where the agent may miss out on sequences of actions that yield higher cumulative rewards over time.





For γ=1, the agent treats future rewards equivalently to immediate ones. This can be problematic in non-episodic tasks or tasks with uncertain long-term outcomes, as the agent may pursue a never-ending series of actions without convergence. For our outcomes, it seems the agent behaved more consistently, but had much higher losses as the iterations progressed.





i) The discount factor is used in reinforcement learning to model the agent's consideration of future rewards. It allows the agent to balance immediate versus future gains. Without a discount factor the agent would consider all future rewards as equally important as immediate ones, which might not be suitable for all scenarios. Especially in situations where the environment is stochastic or the optimal strategy requires short-term sacrifices for long-term gains, not

discounting future rewards can lead the agent astray. In infinite or very long horizon tasks, not using a discount can cause infinite expected returns, making the learning process infeasible.

b) The action repeat parameter dictates how often the agent makes decisions. At its default setting of 4, it means the agent's chosen action is executed for 4 consecutive time steps in the environment before picking a new action.

Increased Action Repeat (8 times): Increasing the action repeat results in the agent making decisions at a slower pace. This means every chosen action has a prolonged impact on the environment. In some scenarios, this could smooth out the agent's trajectory, but it also has its drawbacks. When the agent chooses a less-than-ideal action, the consequences last longer, possibly causing more harm. In my experiments, the agent often braked too much and hesitated, leading to wasted time and subpar performance.

Decreased Action Repeat (1 time): On the other hand, decreasing the action repeat makes the agent more reactive, deciding more frequently. While this might seem like it would make the agent more adaptable, it actually led to erratic behaviors. The agent often oscillated between decisions, resulting in grainy movement patterns. It's like the agent was second-guessing itself too much.

i) Reusing an action can lead to smoother behaviors. This is especially crucial in environments like CarRacing where abrupt actions can be counterproductive. Making fewer decisions translates to fewer neural network evaluations, which speeds up both training and inference. Some decisions naturally span multiple time steps in dynamic environments. By repeating actions, the agent can operate on this coarser time scale, which can simplify learning. In noisy settings, making decisions less frequently can help average out the noise, leading to steadier decisions. In essence, the default action repeat of 4 seems optimal, balancing decision frequency with stability.

c) The action space of an agent fundamentally dictates its range of possibilities and behaviors in an environment. In the CarRacing scenario, the agent's baseline action set comprises left-turn, right-turn, brake, and acceleration.

Introducing a "do-nothing" action might seem advantageous, giving the agent a choice to remain inert, especially in scenarios where no action might be the best action. However, in practice, this often led to abrupt stops, causing the agent to lose momentum. This behavior can be particularly detrimental in racing scenarios, where maintaining speed and momentum is crucial. Consequently, the frequent use of the null action led to suboptimal trajectories and reduced performance.

Enhancing the granularity of the action space with actions like slight-left and slight-right turns made the action space more saturated, leading to the overshadowing of some pivotal actions. Specifically, the agent started losing its ability to effectively drift using the brake function, resulting in it going off-track more frequently than in the baseline scenario. This points to the complexity of balancing the action space: while more actions provide nuanced control, they can also make the decision-making process murkier, leading to unforeseen behaviors.

i) More choices mean the agent takes longer to explore and understand the implications of each action. As the action space grows, pivotal actions can become overshadowed, leading to

unintended behaviors. Some actions might end up being too similar, adding noise rather than genuine utility.
ii) Deep Q-Networks (DQNs) inherently operate on discrete action spaces. This limitation stems from the way Q-values are computed: the Q-network outputs a value for each possible action from a state. For continuous actions, this paradigm becomes impractical. However, there are methods to handle continuous action spaces:

Deep Deterministic Policy Gradients (DDPG): An algorithm that combines Q-learning with policy gradients, designed for continuous action spaces.

Action Discretization: This involves segmenting the continuous action space into discrete chunks, but it's often a crude workaround and might not capture the nuances of truly continuous actions.

Twin Delayed DDPG (TD3): An enhanced version of DDPG that addresses issues like overestimation of Q-values.

d) The implementation of the standard Deep Q-Learning algorithm has an inherent bias that can lead to the overestimation of Q-values. This is predominantly because when estimating the value of Q for a given state and action, we use the maximum Q-value for the next state's actions as our best guess. Given the inherent noise in any neural network's predictions, this can sometimes lead to an overly optimistic estimate of the Q-value, especially during the early stages of training when our Q-network's predictions aren't very accurate.
The Double Q-Learning algorithm addresses this problem by decoupling the action selection from the Q-value evaluation. Instead of using a single network to both select the best action and evaluate its Q-value for the next state, we use two networks: one for action selection (the online network) and another for Q-value evaluation (the target network). This way, the overestimations and underestimations introduced by the inherent noise in the predictions of the two networks can offset each other, leading to a more accurate estimate of the Q-values.

i) In a standard DQN, there's a tendency to overestimate Q-values. This arises from the max operator used in the Q-learning update, which inherently picks the highest value, combined with the noise in the network's predictions. Given that we're always selecting the maximum Q-value for the next state's actions as our best guess, any positive noise in the Q-value predictions can lead to consistently overoptimistic Q-value estimates.

ii) The Double Q-Learning algorithm mitigates this issue by introducing a second network to the process. In DDQN, one network (the online network) is used to select the best action for the next state, while the other network (the target network) evaluates the Q-value of that chosen action. By decoupling action selection from Q-value evaluation, we reduce the risk of systematic overestimation, as the noise in the predictions of the two networks can cancel each other out. This results in a more robust and stable learning process with more accurate Q-value estimates.

e) I ended up rolling with the DQNN as my final. It did seem to be consistent with the rewards of 450+, but to be honest, its implementation is nearly identical to the baseline. It is trained pretty much the same as the baseline. The suboptimal behavior still lies in a lack of regards gained from being in the grass, leading to an inability to learn how to drift. This can be overcome by implementing a world model.