

Assignment 2 - Deep Q-Networks

▼ Status Reinforcement Learning

CSE 546: Reinforcement Learning

Instructor: Alina Vereshchaka

27th, March 2024

Name: Charvi Kusuma

UBID: charviku@buffalo.edu

UB Person Number: 50540958

Name: Tarun Reddi

UBID: bhanucha@buffalo.edu

UB Person Number: 50545060

Team Contribution

Team Member	Assignment Part	Contribution (%)
Charvi Kusuma	I, II, III, Bonus	50, 50, 50, 50
Tarun Reddi	I, II, III, Bonus	50, 50, 50, 50

CSE 546: Reinforcement Learning

[Part I \[20 points\] - Introduction to Deep Reinforcement Learning](#)

[Part II \[50 points\] - Implementing DQN & Solving Various Problems](#)

1. Discuss the benefits of:

2. Briefly describe 'CartPole-v1', the second complex environment, and the grid-world environment that you used (e.g. possible actions, states, agent, goal, rewards, etc). You can reuse related parts from your Assignment 1 report.

3. Show and discuss your results after applying your DQN implementation on the three environments. Plots should include epsilon decay and the total reward per episode.

4. Provide the evaluation results. Run your agent on the three environments for at least 5 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.

5. Provide your interpretation of the results. E.g. how the DQN algorithm behaves on different envs.

6. Include all the references that have been used to complete this part

[Part III \[30 points\] - Improving DQN & Solving Various Problems](#)

1. Discuss the algorithm you implemented.

2. What is the main improvement over the vanilla DQN?

3. Show and discuss your results after applying your the two algorithms implementation on the environment. Plots should include epsilon decay and the total reward per episode.

4. Provide the evaluation results. Run your environment for at least 5 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.

5. Compare the performance of both algorithms (DQN & Improved version of DQN) on the same environments (e.g. show one graph with two reward dynamics) and provide your interpretation of the results. Overall three rewards dynamics plots with results from two algorithms applied on:

- Grid-world environment
- 'CartPole-v1'
- Gymnasium envs or Multi-agent environment or any other complex environment

6. Provide your interpretation of the results. E.g., how the same algorithm behaves on different environments, or how various algorithms behave on the same environment.

7. Include all the references that have been used to complete this part

[Bonus Part \[max +10 points\]](#)

Part I [20 points] - Introduction to Deep Reinforcement Learning

- Only the Jupyter Notebook.

Part II [50 points] - Implementing DQN & Solving Various Problems

1. Discuss the benefits of:

- Using experience replay in DQN and how its size can influence the results
- Introducing the target network
- Representing the Q function as $\hat{q} = (s, w)$

Using Experience Replay in DQN:

Experience replay is a technique in DQN where we will store the agent's experiences at each time step as $e_t = (s_t, a_t, r_t, s_{t+1})$, in a data set $D = e_1, \dots, e_N$, pooled over episodes into a replay buffer. During the learning process, we will sample random mini batches from this buffer to update the network weights.

- Things that happen one after another are connected, like scenes in a story. But if we only learn from the story in order, we might not see the bigger picture. By remembering random scenes and learning from them, we can understand the story from different angles. So DQN experience replay helps in breaking up routines.
- Instead of learning from each experience just once and then forgetting it, experience replay lets DQNs go over their memories many times. This is great when getting new experiences is tough or takes a lot of effort. Hence helps in learning more from less.
- Mixing old and new memories helps the DQN to keep its learning steady and balanced, making sure it doesn't forget its previous lessons while picking up new ones. This helps in smoothing out learning bumps.

The size of the experience replay buffer can significantly influence the results:

- If the buffer is too small, it might not contain a diverse enough set of experiences leading to overfitting and poor generalization.
- If the buffer is too large, it could contain outdated experiences that may no longer be relevant especially if the policy changes rapidly.

Introducing the Target Network:

The target network is a copy of the DQN network that is used to generate the Q-value targets for the updates. These target network weights are held fixed between individual updates.

- By using this target network, the DQN has a consistent goal to aim for. It's like having a reliable guide who gives steady advice while learning a new skill, keeping from getting confused by changing instructions. Hence helps in stability.
- Reduces Oscillations and Divergence: Without a target network, the Q-value update can significantly change the policy and the value function, leading to destructive oscillations or divergence of the learning process.

Representing the Q function as $\hat{q} = (s, w)$:

In a DQN, the Q-function approximator is parameterized as $\hat{q}(s, a; w)$, where w represents the weights of the neural network, s is the state, and a is the action.

- Function Approximation: It allows for generalization across similar states, enabling the DQN to handle large state spaces that would be infeasible for a tabular Q-learning approach.
- Scalability: This representation lets the DQN scale to problems with high-dimensional input spaces, like images from video frames.

- Learning Representations: Neural networks can learn to represent the state in a way that is beneficial for the task, potentially learning features that capture the underlying structure of the state space.

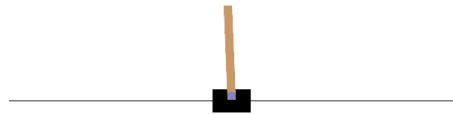
2. Briefly describe 'CartPole-v1', the second complex environment, and the grid-world environment that you used (e.g. possible actions, states, agent, goal, rewards, etc). You can reuse related parts from your Assignment 1 report.

CartPole Environment

Main Characteristics:

- **Cart:** The environment consists of a cart that can move horizontally along a track.
- **Pole:** A pole is attached to the cart via a hinge joint. The pole can rotate freely in a vertical plane.
- **Observations:** The agent receives observations consisting of four continuous variables representing the state of the environment: cart position, cart velocity, pole angle, and pole angular velocity.
- **Actions:** The agent can take one of two discrete actions: push the cart to the left or push it to the right.
- **Rewards:** The agent receives a reward of +1 for each time step the pole remains upright within a certain threshold angle limit. The episode terminates if the pole angle exceeds the threshold or the cart moves outside the track bounds.

The dynamics of the environment involve the physical interactions between the cart and the pole. The agent's actions (pushing the cart left or right) affect the position and velocity of the cart, which in turn influence the dynamics of the pole. The goal of the agent is to learn a policy that can balance the pole by making appropriate control decisions based on the observed states.



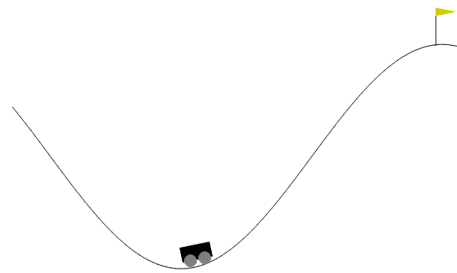
The episode terminates if the pole angle exceeds a certain threshold, or if the cart moves outside the bounds of the track. The goal of the agent is to balance the pole for as long as possible, maximizing its cumulative reward over multiple episodes.

Mountain Car Environment

Main Characteristics:

- **Car:** The environment consists of a car represented as a point mass that moves within a 2-dimensional continuous space.
- **Mountain Terrain:** The car is placed in a valley between two mountains, depicted as a concave landscape with one peak higher than the other.
- **Observations:** The agent receives observations consisting of two continuous variables representing the position and velocity of the car.
- **Actions:** The agent can take one of three discrete actions: accelerate forwards, accelerate backwards, or do nothing (maintain current velocity).
- **Rewards:** The agent receives a reward of -1 for each time step until the goal state is reached. The episode terminates when the car reaches the goal state, which is defined as reaching the top of the higher mountain.

The dynamics of the environment involve the gravitational force acting on the car and its ability to accelerate or decelerate. Due to gravity, the car tends to roll back down the mountain if it doesn't have enough momentum to climb the higher mountain. The agent's actions (acceleration forwards, backwards, or doing nothing) affect the velocity and position of the car, influencing its ability to overcome the terrain.



The goal of the agent is to learn a policy that can navigate the car to the top of the higher mountain by making appropriate control decisions based on the observed states.

The episode terminates when the car reaches the goal state, indicating successful completion of the task. The agent aims to minimize the number of time steps required to reach the goal state, thus maximizing its cumulative reward over multiple episodes.

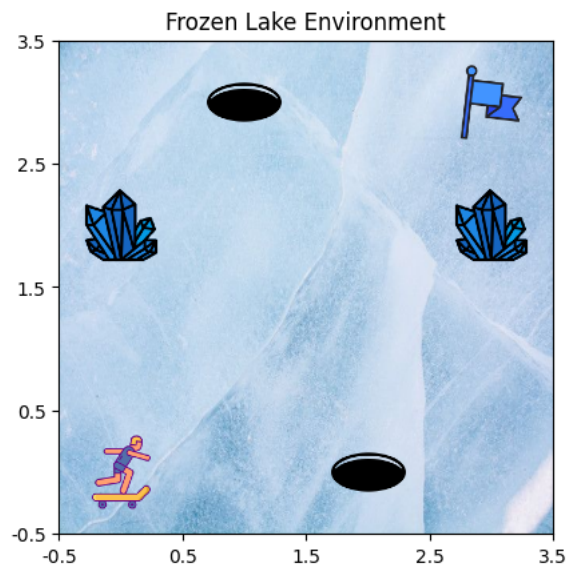
Grid-World Environment

- In our environment, we have a skater as our agent that has to reach the goal destination with maximum rewards.
- We have added two gems locations which counts for positive rewards and two holes locations which drowns the skater as negative rewards.

- **Set of States:**

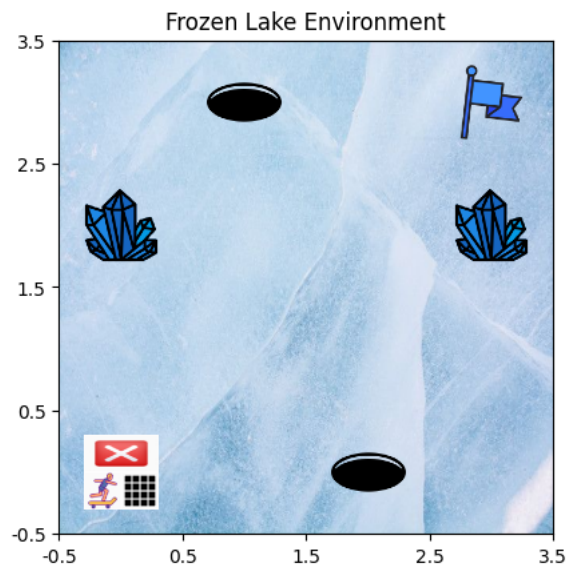
- We have 6 different states that a skater agent can end up in:

1) Agent Initial State: This is where the skater starts his race to reach the goal position. The goal is at the bottom right corner of the grid, as shown below:



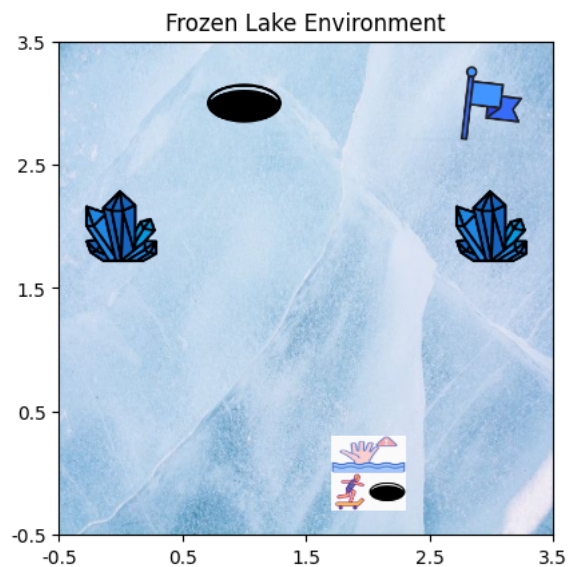
2) Agent going out of the Grid State: If the skater tries to go out of the grid he ends up in the state.

- Consider from the initial position, if the agent tries to go from $[0,0]$ to $[-1, 0]$ or $[0, -1]$ then the agent ends up in this going out of the grid state.

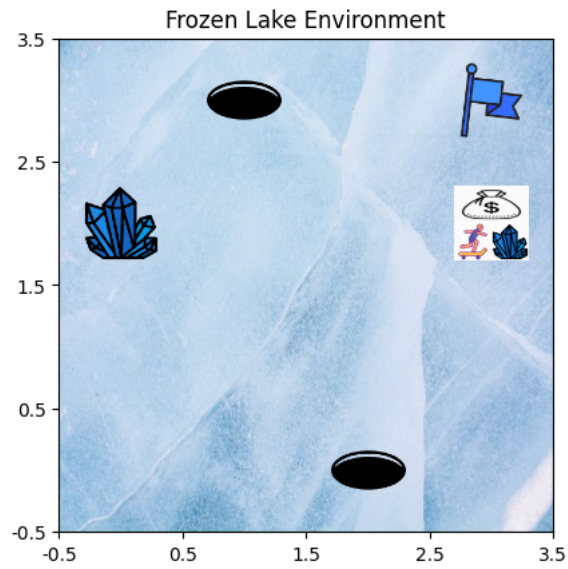


3) In Hole State: If the agent ends up in a hole then he is in this state.

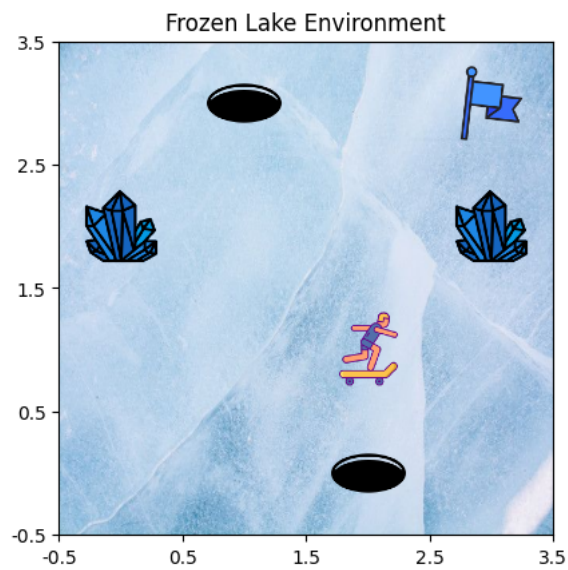
- Agent slips in the hole and drowns if he lands on the locations of the holes with negative rewards.



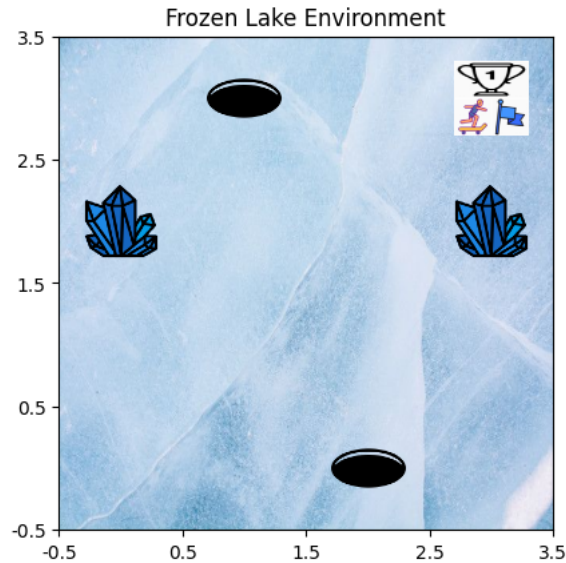
4) On Gems States: If the skater reaches the Gems he has won a lottery and then reach the goal with some gems.



5) Intermediate State: Any other random location that the agent skates the lake grid is the intermediate state.



6) Goal State: The final goal state indicates the termination of the agents race as he has reached the goal.



- **Set of Actions** defined in our case:
 - We have 4 main actions which include: Left, Right, Up and Down.
 - We also ensure to keep the agent within the grid by clipping over the boundaries and restoring his location from where he tried to move out of the grid.
 - If the agent is in the same position as the previous step, choose a different action.
- **Set of Rewards**
 - **Goal Flag Reached (Positive Reward):**
If the goal position is reached, the agent receives a reward of 10. This encourages the agent to reach the goal.
 - **Skating on Hole (Negative Rewards):**
If the agent skates on the first hole, it receives a penalty of -5. If the agent slides on the second hole, the agent receives a penalty of -6. These negative rewards discourage the agent from skating over holes as it will drown.
 - **Gem Collections (Positive Rewards):**
If the agent reaches the location of the first gem, the agent receives a reward of 5. If the agent reaches the location of the second gem, the agent receives a reward of 6. These positive rewards encourage the agent to collect gems and win a lottery along the way.
 - **Distance to Goal (Dynamic Rewards):**
If the current distance to the goal is less than the previous distance to the goal, the agent receives a reward of 1. This provides a positive reinforcement for moving closer to the goal.

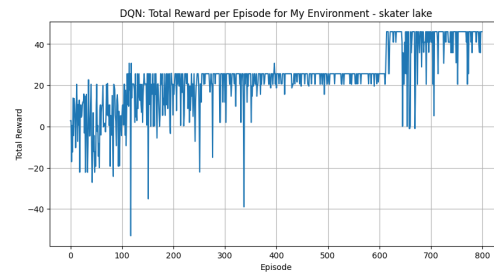
If the current distance to the goal is greater than the previous distance to the goal, the agent receives a penalty of -1. This penalizes the agent for moving away from the goal.

If there is no change in distance, the agent receives a slight negative reward of -0.1. This also handles the case when he is trying to go out of the grid.

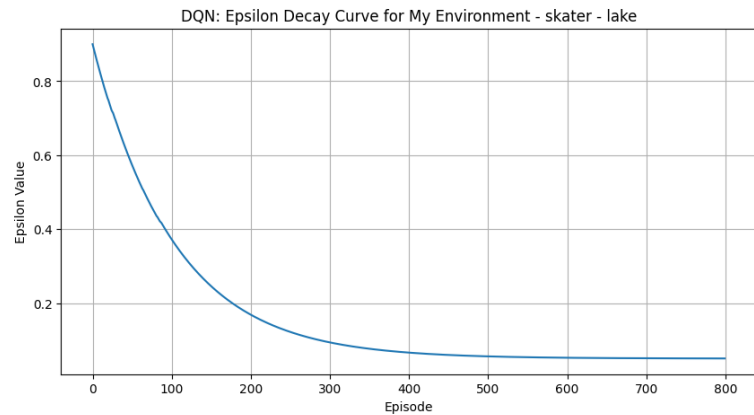
3. Show and discuss your results after applying your DQN implementation on the three environments. Plots should include epsilon decay and the total reward per episode.

My Environment - Frozen Lake

The rewards per episode are highly variable, suggesting that the agent's policy might still be unstable but eventually after 600 epochs learns the policy well enough to keep the rewards near the optimal value. While between 200 to 600 it got stuck in a local maximum, but as the training epochs increased it learnt the optimal value.



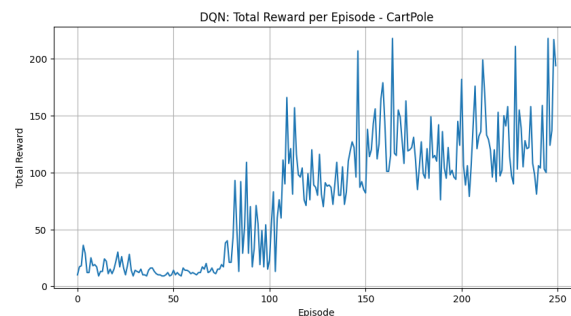
The agent is exploring the environment to learn about the rewards associated with different actions initially. Over time, as epsilon decreases, the agent relies more on the learned policy to make decisions.



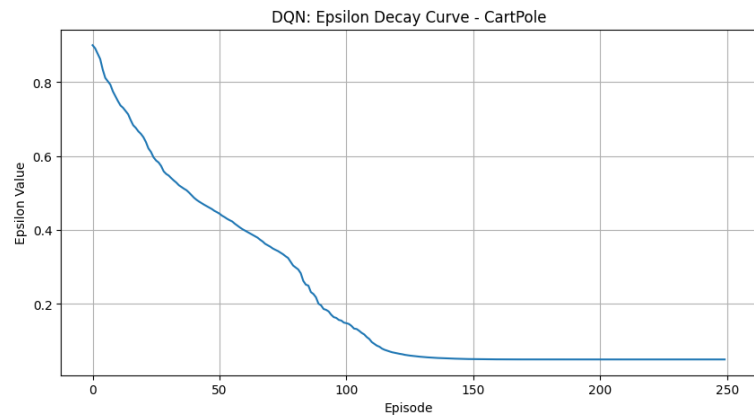
- **Learning Over Time:** By examining the trends in both graphs together, we say that as the agent transitions from exploration to exploitation, there is a clear trend towards an increase in the total reward as the agent is learning an optimal policy.
- **Episodes with Negative Rewards:** The presence of episodes with negative total rewards suggests that the agent frequently encounters states that lead to penalties at the beginning, eventually it is all positive and non decreasing.

CartPole Environment

The total reward per episode plot for the CartPole environment shows a gradual improvement over time, indicative of learning. However, reward fluctuations, including some spikes, suggest ongoing exploration and a learning process that has yet to stabilize, pointing to a need for further learning or policy refinement. This could be improved with more epochs.



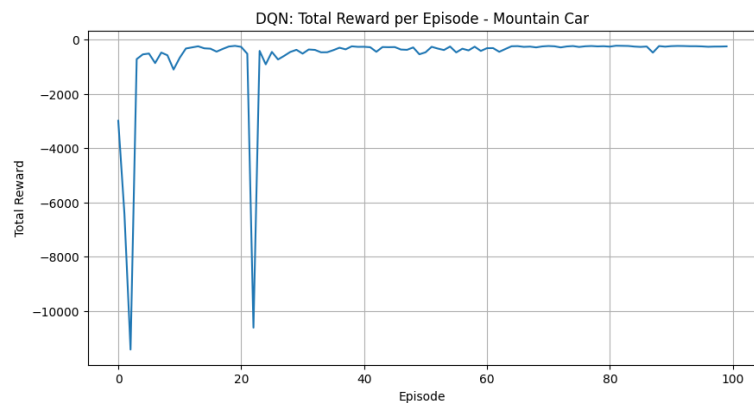
The epsilon decay graph shows a shift from high initial exploration towards greater exploitation, reflecting the agent's increasing reliance on its developing policy as it learns the CartPole task.



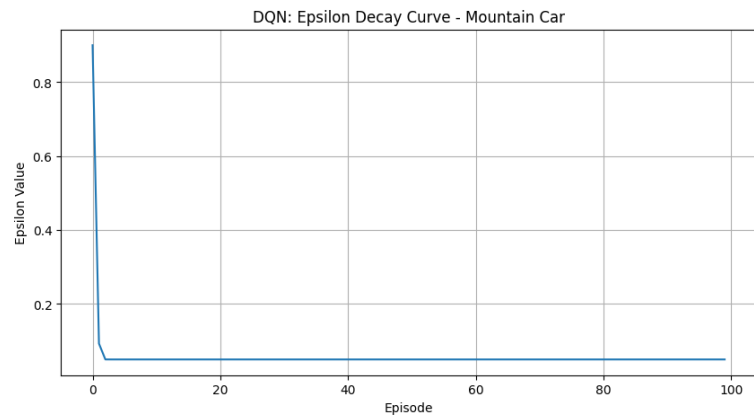
- **Learning Progress:** The agent shows improvement in obtaining rewards as it gains experience over episodes, indicative of learning progress within the CartPole environment.
- **Reward Fluctuations:** Despite the overall upward trend in rewards, fluctuations persist, indicating that the agent may still be discovering new strategies.
- **Optimization:** The optimization function and the update of the target network with a mix of the policy network weights contribute to the gradual learning process.

MountainCar Environment

The rewards are consistently negative, indicative of the Mountain Car environment where the agent incurs a step penalty until the goal is reached. There's a sharp initial drop, then the total reward levels off quickly, staying relatively consistent throughout the episodes. This leveling off suggests that the agent quickly learns a policy that minimizes the penalty but does not show significant improvement over time.



The epsilon value starts near 1, showing a preference for exploration at the beginning. The decay is rapid, indicating a swift shift from exploration to exploitation. As epsilon approaches a lower bound, it suggests that the agent starts relying more on the learned policy rather than random actions.

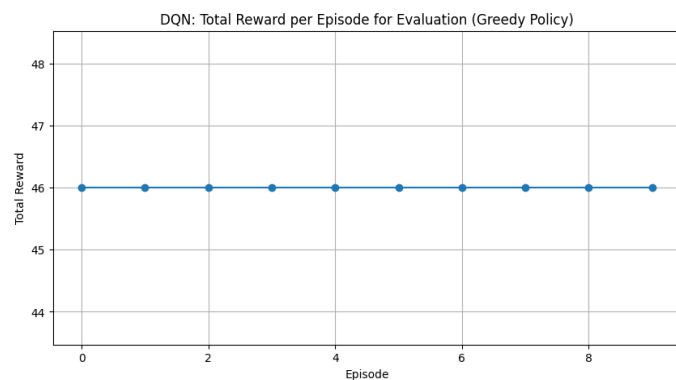


- The agent learns a policy early on that does not evolve substantially, as indicated by the flat total rewards curve after the initial episodes.
- The epsilon decay graph reflects a standard epsilon-greedy strategy, where exploration is heavily favored initially and rapidly tapers off, allowing the learned policy to dictate actions.
- The flat reward graph in combination with the epsilon decay suggests that either the agent's policy is near-optimal, the environment does not allow for much improvement, or the agent could be stuck in a suboptimal policy due to insufficient exploration after the initial episodes.

4. Provide the evaluation results. Run your agent on the three environments for at least 5 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.

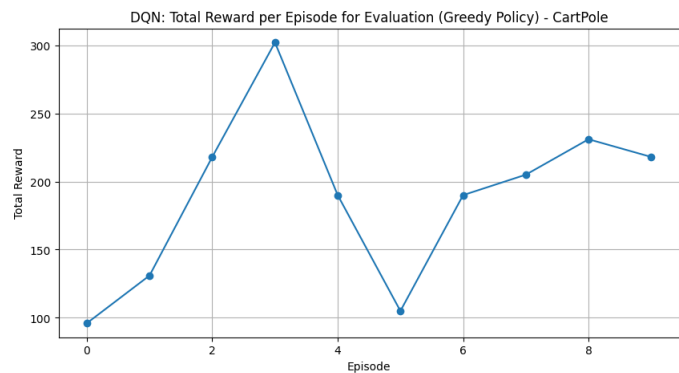
Total Reward per Episode for Evaluation (Greedy Policy) - Frozen Lake Environment:

- The performance is consistent through out, the episodes from 1 to 10 as this is a deterministic environment setup. We can also note that the optimal value is constantly used.



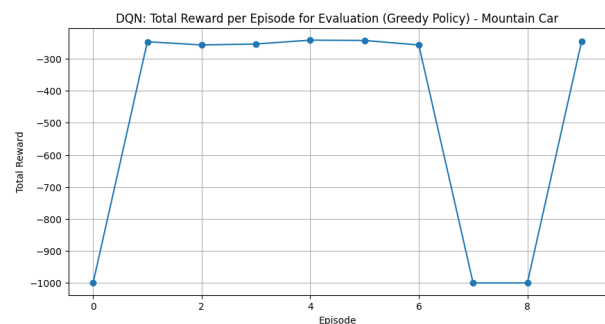
Total Reward per Episode for Evaluation (Greedy Policy) - CartPole:

- Rewards for the CartPole environment fluctuate from episode to episode, showing inconsistency in the policy's effectiveness. Peaks in the graph suggest that sometimes the policy aligns well with the task, but valleys suggest episodes where it does not.



Total Reward per Episode for Evaluation (Greedy Policy) - Mountain Car:

- The Mountain Car environment shows negative rewards throughout, which is characteristic of this environment's reward structure. However, the rewards vary less compared to the other environments, indicating a more consistent policy performance, showing room for improvement.




The evaluation graphs illustrate that while the learned policies can achieve success in certain episodes, there are inconsistencies and fluctuations in performance, likely due to the complexity of each task and limitations in the policies that have been learned. These results emphasize the need for further training or refinement to achieve more stable and consistently high performance across all episodes.

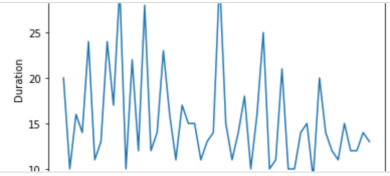
5. Provide your interpretation of the results. E.g. how the DQN algorithm behaves on different envs.

- Across all environments, the DQN shows it can learn to some degree, as shown by episodes of high rewards. It is important that we consider that the environments have varying degrees of difficulty and may require different hyperparameter settings or architectural modifications to the DQN model for optimal learning.
- However, since the epsilon decay curves are gradually decreasing it is safe to consider that it is able to pick the best action over time.
- In the simplest environment Grid World, it achieves a stable policy. In the more complex and dynamic CartPole environment, the algorithm improves over time but still shows inconsistency. Finally, in the Mountain Car environment, which requires a more strategic approach, the DQN succeeds but fails to do so initially. This comparison illustrates that while DQN can learn to navigate different environments, its performance can greatly vary with the complexity and specific challenges of each task.
- It also suggests that further enhancements, like DDQN or other advanced variants, might be necessary to improve learning efficiency and consistency in more complex scenarios.

6. Include all the references that have been used to complete this part


Deep Q Learning (DQN) using Pytorch
Reinforcement Learning

 <https://medium.com/@vignesh.g1609/deep-q-learning-dqn-using-pytorch-a31f02a910ac>



Gymnasium Documentation


A standard API for reinforcement learning and a diverse set of reference environments (formerly Gym)

 https://gymnasium.farama.org/content/basic_usage/



Gymnasium Documentation

A standard API for reinforcement learning and a diverse set of reference environments (formerly Gym)

 https://gymnasium.farama.org/environments/classic_control/cart_pole/



Gymnasium Documentation


A standard API for reinforcement learning and a diverse set of reference environments (formerly Gym)

 https://gymnasium.farama.org/environments/classic_control/mountain_car/




Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.2.1+cu121 documentation

This tutorial shows how to use PyTorch to train a Deep Q Learning (DQN) agent on the CartPole-v1 task from Gymnasium.

 https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Reward Engineering for Classic Control Problems on OpenAI Gym | DQN | RL

Custom reward functions for faster learning!

 <https://towardsdatascience.com/open-ai-gym-classic-control-problems-rl-dqn-reward-functions-16a1bc2b007>



<https://github.com/msachin93/RL/blob/master/MountainCar/MountainCar.ipynb>

Part III [30 points] - Improving DQN & Solving Various Problems

1. Discuss the algorithm you implemented.

- The Double Deep Q-Network (DDQN) algorithm enhances the standard DQN by decoupling the selection of actions from the evaluation of their associated value. This is implemented by having two neural networks: the policy network and the target network.
- In our DDQN, the policy network is used to choose the action that appears best in a given state, while the target network independently evaluates the action's value.
- The replay memory system stores experiences in a buffer, which helps in breaking the correlation between consecutive learning steps, and samples from this buffer are used for training the network in a more stable and efficient way.

- The learning process is regulated through an epsilon-greedy strategy to balance exploration and exploitation, with the epsilon value decaying over time to shift towards more greedy actions.

2. What is the main improvement over the vanilla DQN?

The main improvement that DDQN has over vanilla DQN is in addressing the overestimation of action values. In standard DQN, the same network estimates the values for both selecting and evaluating an action, which can lead to overoptimistic value predictions. DDQN avoids this by using the policy network to choose the action and the target network to evaluate its value, thus providing a more conservative and generally more accurate estimate. This tends to result in more reliable training and often leads to better policies, as it helps avoid the feedback loop where overestimated values reinforce themselves.

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

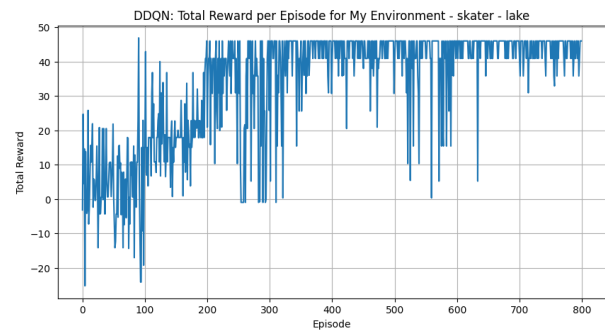
    # Double DQN updates here
    # First, select the best action in the next state from the policy network
    policy_net_best_action = policy_net(non_final_next_states).max(1)[1].unsqueeze(1)
    # Then, evaluate the action using the target network
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).gather(1, policy_net_best_action).squeeze()

    expected_state_action_values = (next_state_values * GAMMA) + reward_batch
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))
    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
```

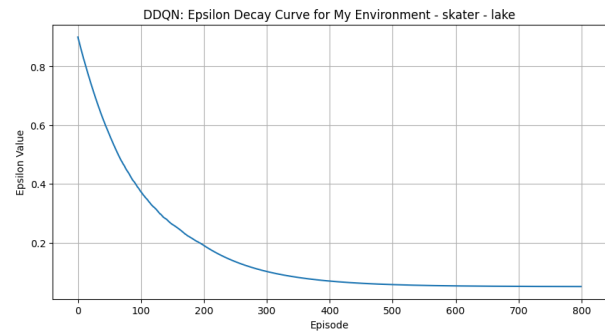
3. Show and discuss your results after applying your the two algorithms implementation on the environment. Plots should include epsilon decay and the total reward per episode.

Frozen Lake Environment:

Total Rewards in case of DDQN for the grid world environment is increasing very similar to the DQN algorithm. The fluctuations indicate varying degrees of success across episodes, with a general trend showing the agent often achieving positive rewards, although some episodes result in negative total rewards.

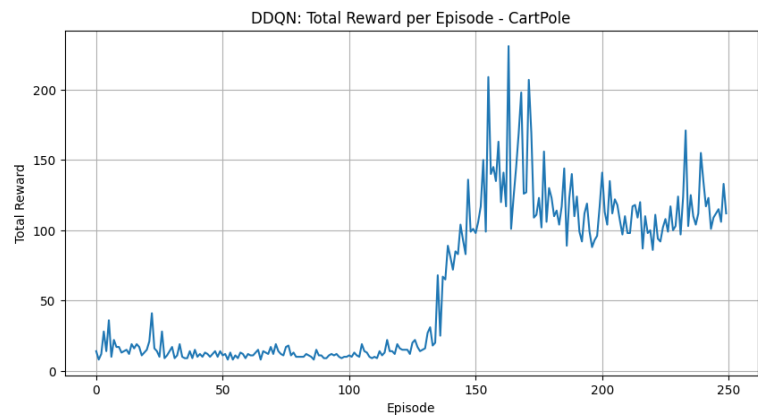


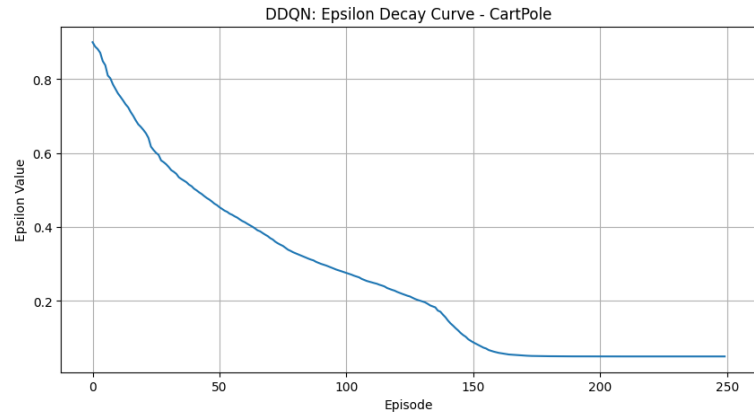
Initially the value of epsilon is high around 0.9, indicating a strong preference for exploration. Over time episodes, epsilon decreases sharply, reducing the likelihood of random actions.



CartPole Environment:

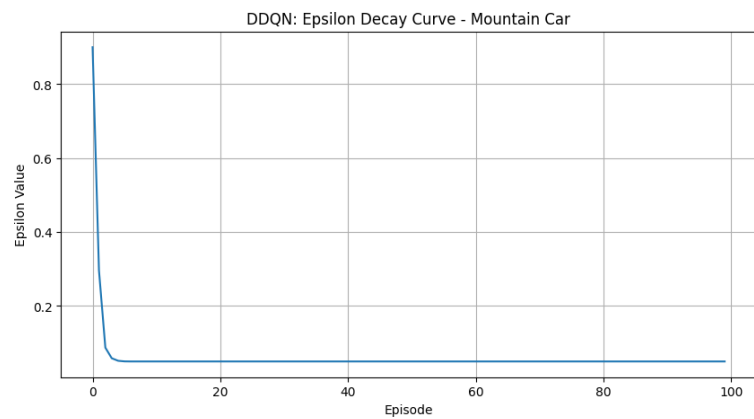
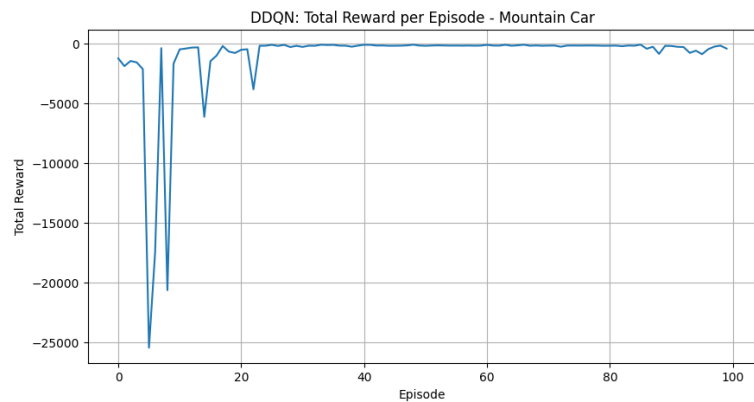
The results for CartPole show an increasing trend in total reward over time for DDQN, which suggests a steady learning process. DDQN appears to be more stable compared to DQN, which has more pronounced spikes in performance. This indicates that DDQN may be better at consistently balancing the pole due to its more reliable value estimation.





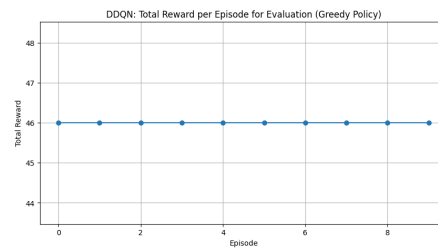
Mountain Car Environment:

In Mountain Car, both DQN and DDQN struggle as indicated by a flat total reward trend. However, DDQN shows occasional episodes with less negative reward compared to DQN, hinting at sporadic successful episodes. Despite these occasional successes, both algorithms fail to reliably solve the environment within the evaluated episode range.

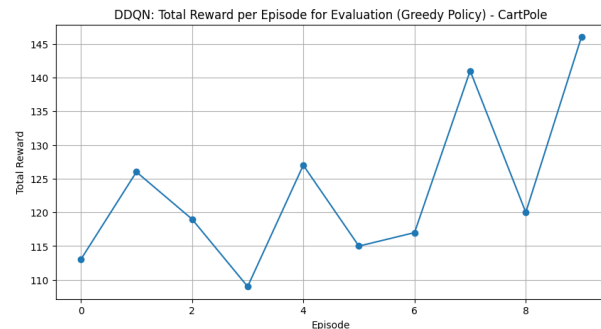


4. Provide the evaluation results. Run your environment for at least 5 episodes, where the agent chooses only greedy actions from the learnt policy. Plot should include the total reward per episode.

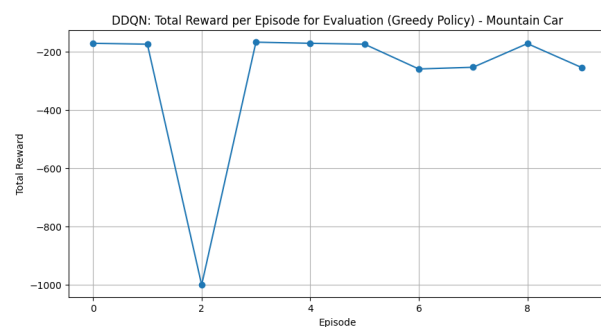
A consistent total reward per episode during the evaluation of a Double Deep Q-Network (DDQN) using a greedy policy, indicating that the agent's performance has stabilized with little variation across episodes.



The performance in CartPole, while also fluctuating, shows an overall trend of higher rewards as the episodes progress, which is a sign of learning and adaptation. However, the variability points to potential areas of improvement, particularly in achieving more consistent results.



The Mountain Car environment presents a challenging scenario for the DDQN algorithm. The total rewards are deeply negative, with only occasional increases, indicating that the agent rarely succeeds in solving the environment within the limited episodes. This highlights the difficulty DDQN faces in environments where the reward structure is sparse and the goal is not straightforward.



These evaluations show that while DDQN has the potential to improve upon DQN's capabilities, particularly in environments with clear and frequent feedback, it still struggles in more challenging environments where strategic exploration and long-term planning are essential.

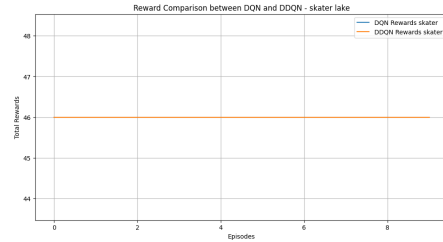
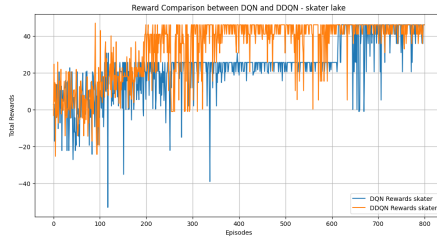
5. Compare the performance of both algorithms (DQN & Improved version of DQN) on the same environments (e.g. show one graph with two reward dynamics) and provide your interpretation of the results. Overall three rewards dynamics plots with results from two algorithms applied on:

- Grid-world environment
- 'CartPole-v1'
- Gymnasium envs or Multi-agent environment or any other complex environment
- When applied to various environments, DDQN can produce different results due to the double change. For environments with direct decision-making, like **the Grid-world, we can see quicker convergence to an optimal policy as shown below in the figure plot on left.**
- In more complex environments with larger state spaces, such as CartPole-v1, the improved stability in learning can lead to achieving the set benchmarks for solving the environment in a more consistent manner. For the Mountain Car

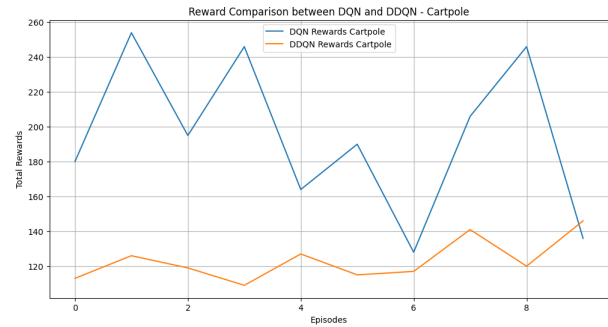
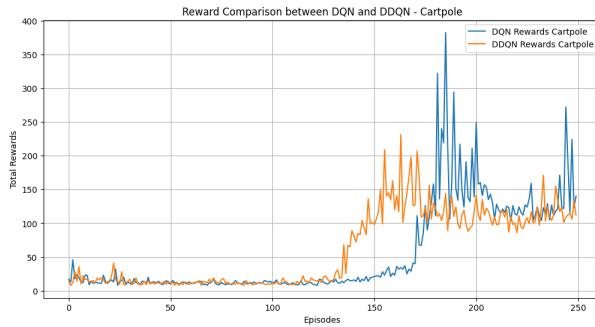
problem, where successful episodes are rare and learning is more challenging, DDQN's accurate value estimation is crucial for learning a policy that can consistently solve the task.

- The architecture of the implemented neural network consists of two fully connected hidden layers with 128 units and 256 each, using ReLU activation functions for non-linearity. The network outputs Q-values for each action given the current state of the environment.
- The replay memory stores transitions that are sampled randomly, helping to break the correlation between consecutive learning updates. This process is crucial for stabilizing the learning algorithm and ensuring effective use of experiences.

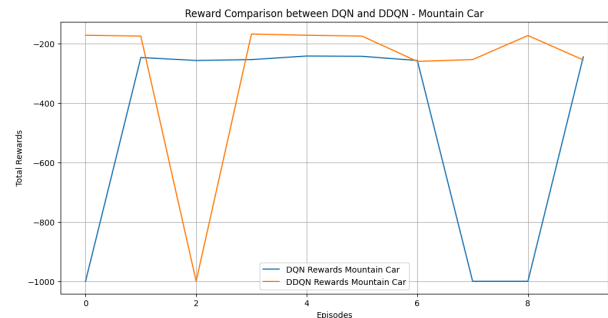
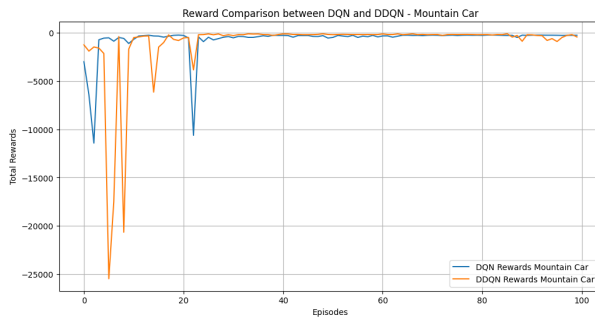
Squirrel Hunter Grid World:



CartPole Environment:



Mountain Car Environment:



6. Provide your interpretation of the results. E.g., how the same algorithm behaves on different environments, or how various algorithms behave on the same environment.

- **Frozen Lake Environment:** The total reward per episode graph for the grid world is consistent. . The epsilon decay curve implies that the agent's policy transitioned from exploration to exploitation over time.
- **CartPole Environment:** In the CartPole environment, the DDQN algorithm demonstrates a learning curve with increasing total rewards over episodes. However, the performance contains spikes, suggesting occasional lapses in the agent's decision-making. This is common in environments where balance and precision are key, and learning can be affected by the agent's initial random actions. The DDQN seems to be learning and improving but hasn't reached a plateau of consistent high performance, which could be a target for further optimization.
- **Mountain Car Environment:** For the Mountain Car environment, the DDQN's total reward per episode is relatively stable and low, with some sharp peaks. This indicates the challenge in achieving the goal state consistently. Given the nature of the Mountain Car environment, where the agent must learn a sequence of actions to reach the goal, these results may suggest the agent has started to grasp the strategy but is not executing it consistently every episode.

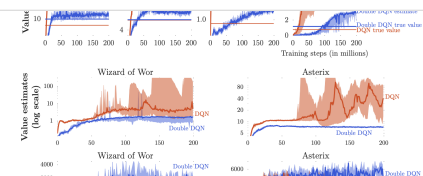
7. Include all the references that have been used to complete this part

<https://github.com/Parsa33033/Deep-Reinforcement-Learning-DQN/blob/master/Double-DQN.py>

Papers with Code - Double DQN Explained

A Double Deep Q-Network, or Double DQN utilises Double Q-learning to reduce overestimation by decomposing the max operation in the target into action selection and action evaluation. We evaluate the greedy policy according to the online network, but we use the target network to estimate its value. The

<https://paperswithcode.com/method/double-dqn>



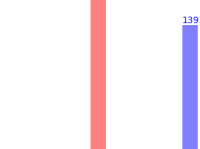
https://github.com/dxyang/DQN_pytorch

Double DQN Implementation to Solve OpenAI Gym's CartPole v-0

This post describes a reinforcement learning agent that solves the OpenAI Gym game, CartPole (v-0). The agent was based off of a family of...

<https://medium.com/@leosimmons/double-dqn-implementation-to-solve-openai-gyms-cartpole-v-0-df554cd0614d>

Episodes Till Solve



<https://github.com/fschur/DDQN-with-PyTorch-for-OpenAI-Gym>

Bonus Part [max +10 points]

• Solving Image-based Environment [7 points]

Use any Atari environments with image representation of the state that requires to utilize CNN (Convolution Neural Network) for the state preprocessing (e.g. OpenAI Breakout). Apply both DQN and an improved version of DQN, compare the results and provide your analysis.

It has been a challenging task, we faced several issues with loading env of Atari games in colab and running the episodes with limited System RAM. Session used to crash for episodes crossing 50. Hence the results are not at par, however the concept of using wrappers and implemneting DQN with PyTorch is the main idea.

We enhanced and customized the environment for training the model on the Atari games using OpenAI's Gym library. We used several wrappers and utility classes that are publicly available to define environment's states, modify rewards, and

control the dynamics to make it more suitable for training deep reinforcement learning models, mainly the Deep Q-Networks (DQN) and its variants like Double DQN (DDQN).

Overview for the wrappers used from

https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py:

- **make_env Function:** Creates a wrapped environment with several enhancements for RL training:
 - **EpisodicLifeEnv:** Makes the game episodic by ending the episode when a life is lost (not just when the game is over). This can help the agent learn more efficiently from failures.
 - **NoopResetEnv:** Applies a random number of no-operation (noop) actions at the start of each episode, providing diverse initial states.
 - **MaxAndSkipEnv:** Skips a fixed number of frames and returns the max pixel value over skipped frames. This helps in reducing the frequency at which the agent needs to make decisions and stabilizes the visual input.
 - **FireResetEnv:** Automatically presses the FIRE button for environments that require it to start the game.
 - **WarpFrame:** Resizes the game frames to 84×84 and converts them to grayscale to reduce the dimensionality of the input space.
 - **FrameStack:** Stacks a specified number of consecutive frames together as the environment state. This helps the agent to perceive motion.
 - **ClipRewardEnv:** Modifies the rewards to be +1, 0, or -1 based on their sign. This simplifies the reward structure and can help in stabilizing training.
- **RewardScaler (Not Used in make_env):** Scales rewards by a constant factor, which can be useful in environments with large reward scales but is not applied by default in the make_env function.
- **LazyFrames Class:** Optimizes memory usage by delaying the creation of a combined numpy array for stacked frames until absolutely necessary. This is particularly useful when using FrameStack.
- WarpFrame, MaxAndSkipEnv, NoopResetEnv, FireResetEnv, EpisodicLifeEnv, and ClipRewardEnv are all wrappers that modify the environment's original behavior in ways beneficial for training RL models. These modifications include changing the observation space (resizing and grayscaling images), modifying the reward structure, and altering the episode dynamics (e.g., treating loss of life as episode end).

Why did we use the wrappers?

The purpose of wrapping the environment with these modifications is to make it more tractable for RL algorithms to learn. For instance, reducing the resolution and color of frames lowers the computational cost and complexity. Stacking frames helps the agent to infer velocity and direction of objects in the game, which is not possible from a single frame. Modifying the reward structure to a simpler form can help in faster convergence of the RL model by focusing on the essential part of learning - maximizing cumulative rewards.

Architecture Defined:

- **Input:** 4 game frames where in_channels=4, each preprocessed to a smaller size of 84×84 pixels
- Three convolutional layers (conv1, conv2, conv3) extract features from the frames. The number of filters increases with depth (32, 64, 64), and the kernel sizes and strides are chosen to reduce dimensionality while capturing relevant spatial information.
- A fully connected layer transforms the high level features into a 512 dim vector.
- The output layer maps this vector to the predicted Q-values for each action in the game n_actions.
- Frames are normalized divided by 255 for learning.
- The ReLU activation function is used after each layer except the last, adding non-linearity.

Environment:

PongNoFrameskip-v4 Atari game environment available in OpenAI Gym simulates the classic Pong game, where two paddles control a bouncing ball, aiming to surpass the opponent.

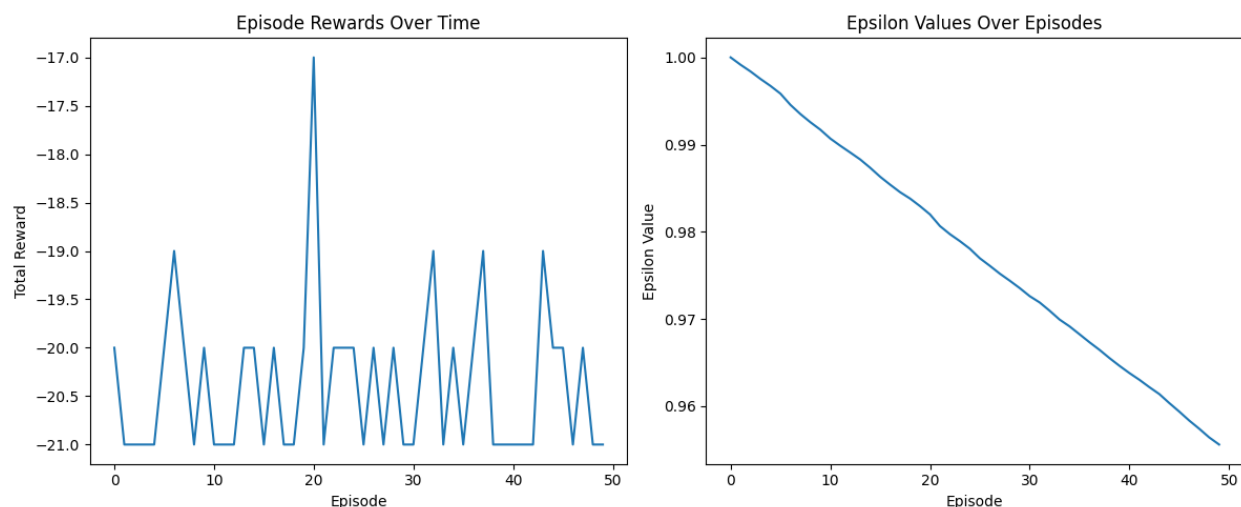
- **States:** The state is represented by raw pixel values from the game screen, typically with a resolution of 210×160 pixels and 3 color channels (RGB). The "NoFrameskip" version doesn't skip frames, providing a more detailed temporal sequence for decision-making.
- **Actions:** The agent can choose among a discrete set of actions to move its paddle up or down or to do nothing.
- **Rewards:** +1 for scoring a point by getting the ball past the opponent, -1 when the opponent scores, and 0 otherwise. This direct and sparse reward structure closely aligns with the game's objective.
- **Episode Termination:** An episode ends when one player reaches a game-winning score, typically 21 points, but in a training context, episodes might be capped at a certain time step for practicality.

DQN

Training:

We train a Deep Q-Network (DQN) model for the environment using PyTorch, with specific configurations for training hyperparameters and environment preprocessing:

- **DQN Model:** Initialized two identical DQN models: `policy_net` for selecting actions and `target_net` for calculating target Q-values. The `target_net`'s weights are periodically updated with `policy_net`'s weights to stabilize training.
- **Optimizer:** Used the Adam optimizer for training the `policy_net`.
- The "PongNoFrameskip-v4" Gym environment with custom preprocessing steps to make it more suitable for DQN training with modifications to frame stacking, reward clipping.
- **Training:** Executed a training for 50 episodes, recording the total reward per episode and the epsilon value at each step of the training as follows:



The left graph shows the total rewards per episode over 50 episodes, which is low for training, however would definitely improve over episodes. values fluctuating around -20 suggests the agent hasn't learned to consistently score points against the opponent.

The right graph illustrates the decay of epsilon, a parameter controlling the trade-off between exploration and exploitation, which steadily decreases, indicating a shift from exploration towards exploitation as learning progresses.

For testing:

Environment Setup with Video Recording: The environment is wrapped with the `RecordVideo` wrapper, which will automatically records each episode. Videos are saved in the `/content/videos/dqn_pong_video` directory.

- **Tracking Total Rewards:** After each episode is done, the total reward for that episode is added to the episode_rewards list.
- **Function Output:** The function returns a list of total rewards, one for each tested episode, allowing for performance evaluation of the policy.
- **Testing:** We evaluated for 5 episodes and the rewards were not very well learnt with the short 50 episode training.

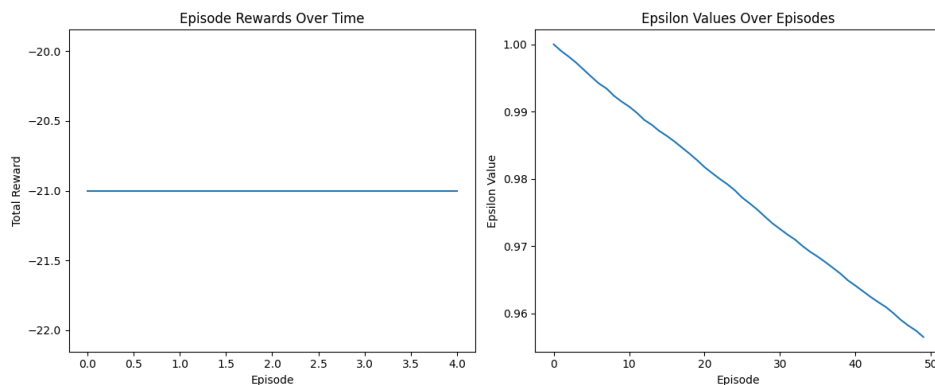


DDQN

Main difference in the optimize method: The optimize_model function updates the Double Deep Q-Network (DDQN). It mitigates overestimation of Q-values by using two networks: one for selecting the best action (policy_net) and another for evaluating that action's value (target_net). By sampling from the replay memory and calculating the loss between predicted and target Q-values, it refines the policy through gradient descent, using the target network to stabilize the learning process.

Training

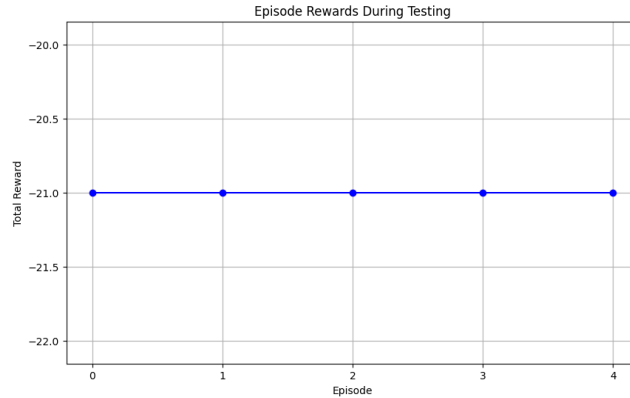
- The agent did not learn for 50 episodes here, however DDQN would have performed better if the episode count increases, learning the optimal policy quicker.



- The agent constantly lost the match for all of its training. however the epsilon decay is going more towards the exploitation.

Testing

- Since it did not learn the game, the testing was in vain, however the setup and usage of the DQN for CNN based Atari games training and evaluation setup is all pipelined.



- The testing videos are attached in the submission

Comparison

Over 50 episodes, comparing DQN and DDQN, the episode rewards suggested improvement mostly inconsistent in DQN, potential for learning an effective policy given more time or episodes for training. DDQN is a framework that performs on the same level all the way, but without clear improvement within these initial episodes.

This may be because the much more complicated learning mechanism within DDQN, often requiring more experiences before the system starts to perform better than DQN.

They would show good and sound foundational setups, with opportunity for further tuning and extended training.

• RL Libraries [3 points]

Use the existing implementation of DQN in Stable-baselines3 or Ray RLLib to solve your Grid World environment, CartPole-v1, and the second complex environment. Compare the results with your scratch implementation. Note: You may use the RL in-built model only to complete this bonus task; using RL in-built models for other parts will not be evaluated.

REFERENCES

GitHub - Farama-Foundation/Arcade-Learning-Environment: The Arcade Learning Environment (ALE) -- a platform for AI research.

The Arcade Learning Environment (ALE) -- a platform for AI research. - Farama-Foundation/Arcade-Learning-Environment

<https://github.com/Farama-Foundation/Arcade-Learning-Environment#rom-management>

Farama
Arcade

The Arcade Learning Environment (ALE) -- a platform for AI research.

44
Contributors

<https://gymnasium.farama.org/environments/atari/pong/>

Learning to play Pong with PyTorch & Tianshou

Training a Deep Q Network with a modular RL library

<https://medium.com/@joachimihakr/learning-to-play-pong-with-pytorch-tianshou-a9b8d2f1b8bd>



dqn-pytorch/wrappers.py at master · jmichaux/dqn-pytorch

DQN to play Atari Pong. Contribute to jmichaux/dqn-pytorch development by creating an account on GitHub.

<https://github.com/jmichaux/dqn-pytorch/blob/master/wrappers.py>

jmichaux/dqn-pytorch

DQN to play Atari Pong

1 Contributor 2 Issues 108 Stars 46 Forks



baselines/baselines/common/atari_wrappers.py at master · openai/baselines

OpenAI Baselines: high-quality implementations of reinforcement learning algorithms - openai/baselines

https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py

openai/baselines

OpenAI Baselines: high-quality implementations of reinforcement learning algorithms

112 Contributors 549 Used by 15k Stars 5k Forks



https://github.com/sudharsan13296/Deep-Reinforcement-Learning-With-Python/blob/master/09_Deep_Q_Network_and_its_Variants/9.03_Playing_Atari_Games_using_DQN.ipynb

https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Reinforcement Learning: Deep Q-Learning with Atari games

In my previous post A First Look at Reinforcement Learning, I attempted to use Deep Q learning to solve the CartPole problem. In this post...

<https://medium.com/nerd-for-tech/reinforcement-learning-deep-q-learning-with-atari-games-63f5242440b1>



<https://github.com/jasonbian97/Deep-Q-Learning-Atari-Pytorch>

DQN-PyTorch-Breakout/Breakout/replay_mem.py at master · KJ-Waller/DQN-PyTorch-Breakout

An attempt at recreating DeepMind's implementation of Deep Q Learning on Atari Breakout using PyTorch - KJ-Waller/DQN-PyTorch-Breakout

https://github.com/KJ-Waller/DQN-PyTorch-Breakout/blob/master/Breakout/replay_mem.py

KJ-Waller/DQN-PyTorch-Breakout

An attempt at recreating DeepMind's implementation of Deep Q Learning on Atari Breakout using PyTorch

1 Contributor 1 Issue 7 Stars 0 Forks



ImportError: cannot import name 'Monitor' from 'gym.wrappers'

I have just created a new environment with gym installation. I am just getting started with Atari games but am getting an import error for my below code -
import gym

<https://stackoverflow.com/questions/71520568/importerror-cannot-import-name-monitor-from-gym-wrappers>

