INM707 Deep Reinforcement Learning (PRD2 A 2023/24)

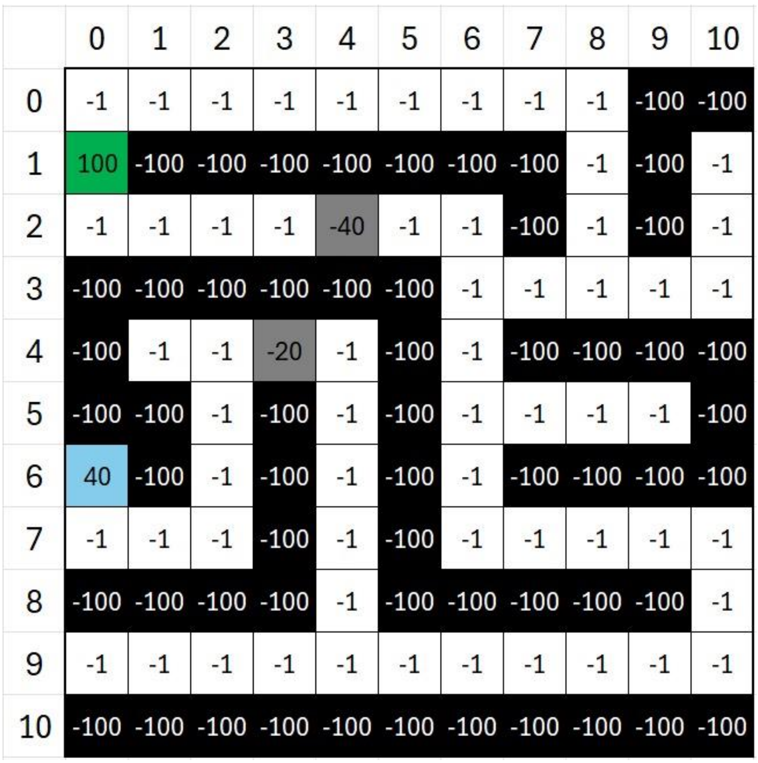
Muhammad Salman Aslam Siddiqui – 230040581
Ashwin Meth Dias Bandaranaike - 230009944
MSc Data Science (FT)

Basic task

Define an environment and the problem to be solved

In video games, when playing with AI allies, it is important that the AI is helpful to the player to facilitate a better gaming experience. In this basic task we will implement a very simplified version of an agent interacting with their environment in order to complete an objective as efficiently as possible. The purpose will be to show how to train an agent to optimise certain tasks, so that it can be a useful AI companion to human players.

In our context, the goal is to train an AI agent to be able to search for a gold coin in a maze. This very basic representation can be scaled up to more complex environments for further work. In this environment, there will be a few challenges the agent has to face. Given below is a visual, top-down representation of the environment that it will be interacting with.



	0	1	2	3	4	5	6	7	8	9	10
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100	-100
1	100	-100	-100	-100	-100	-100	-100	-100	-1	-100	-1
2	-1	-1	-1	-1	-40	-1	-1	-100	-1	-100	-1
3	-100	-100	-100	-100	-100	-100	-1	-1	-1	-1	-1
4	-100	-1	-1	-20	-1	-100	-1	-100	-100	-100	-100
5	-100	-100	-1	-100	-1	-100	-1	-1	-1	-1	-100
6	40	-100	-1	-100	-1	-100	-1	-100	-100	-100	-100
7	-1	-1	-1	-100	-1	-100	-1	-1	-1	-1	-1
8	-100	-100	-100	-100	-1	-100	-100	-100	-100	-100	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

This 11x11 grid layout was heavily inspired by this article [1]. We adapted this to suit our report, changing the environment and modifying it to make it a bit more challenging for the agent.

One challenge is that there is a silver coin in the same maze (blue square), which is not ideal (sub-optimal state) as we want the gold coin (green square, optimal state), hence the agent will have to differentiate between the two and prioritise the gold coin. The black square represents an absence of tiles (a steep drop), and if the agent enters a black square, it will fall and lose the game. This will be referred to as a terminal state.

Another challenge is that we can see in the tiles with coordinates [2,4] and [4, 3], is not black but grey. This indicates a wall that can be cleared by jumping over it. The only problem is that every move/action the agent takes, uses up energy. This is denoted as (-1) in each tile, and we can refer to these as energy points. The wall [2,4] indicates a lot of energy spent (-40) and the wall [4, 3] shows that a medium amount of energy was spent (-20). Since the agent must get to the goal as efficiently as possible, it must find the best path that minimises the amount of energy points spent, ideally avoiding the wall.

The final and more obvious challenge is for the agent to find it's way around the maze without falling into the black tiles, and hopefully avoiding the walls.

Define a state transition function and the reward function

This is an 11x11x4, 3-dimensional grid environment where there are 2 goal states, terminal states, valid movement states, and starting states. There are two different strategies for the starting states.

- Starting from the bottom right corner of the grid: [9,10].
- Starting at a random valid position (white square) each episode.

The agent has 4 actions that it can take at any given time.
0 = up | 1 = right | 2 = down | 3 = left
The action states had to be introduced as a third dimension in our Q matrix with 4 values, since for every tile, there are 4 actions the agent can take. So, every time the agent learns, the respective Q matrix for that action is updated.

For the reward matrix, when the agent moves on white tiles, it incurs a loss of 1 energy point. This is to motivate the agent to converge to some ideal state without being stuck in a particular area. If the agent moves to the black tile, it incurs a loss of 100 energy points. If the agent moves to the blue tile it receives a reward of 40, and moving to green will give it a reward of 100. These 3 states which are not white tiles are terminal states. If the agent reaches any of these states, it will update the Q matrix with the previous episode results and then end it.

To do this, a Boolean check is performed to see if it has entered a terminal state which would be any tile other than white, and it will keep playing till it enters one. A function is built to grab the starting location for an episode by choosing a white tile. We can either supply a starting tile ourselves, or let the function randomly search for a starting tile. In either case it will still check if that tile is white or not. In the event it isn't, it will randomly search for a new starting position.

Another function is built to dictate what the next location will be when the agent moves. This function, depending on the action taken by the agent, will return the next location of the tile. It has built in if-else statements to see if it hits any of the four corners of the maze. If it does, it cannot move beyond it, and is simply returned to its current position.

Set up the Q-learning parameters (gamma, alpha) and policy

The hyperparameters that were optimised were the epsilon values (exploration coefficient), epsilon decay rates, discount rate (gamma), learning rate (alpha), number of episodes, and temperature (Boltzmann policy). The policies that were considered were random, epsilon-greedy and Boltzmann. Some additional parameters that were considered were the different epsilon decay rate and starting position strategies.

The epsilon value is what controls the exploration rate of the agent. This comes into play for the random and epsilon greedy algorithms that we use. The higher the epsilon, the more the agent favours exploration.

However, as it learns, it is not always a good strategy to keep prioritising exploration, but instead move towards exploitation overtime, which takes the best actions depending on the q values thus far. This can be addressed by the epsilon decay rate, which gradually decreases the exploration overtime.
We tried two methods of decaying the epsilon rate: linear and non-linear. For linear decays, we simply multiplied the epsilon by the decay rate if epsilon were above a certain threshold. For non-linear decays, we looked at 2 formulas (Finally decided on one based on results), which can be seen below by the pseudo-code:

- min_epsilon/((max_epsilon-min_epsilon)*np.exp(-epsilon_decay_rate))
- min_epsilon + (max_epsilon - min_epsilon) * np.exp(-1. * episode / epsilon_decay_rate)

The discount rate decides the magnitude of effect that future rewards have on the learning process. A higher value for this will setting will make the agent favour long term rewards while smaller values mean the agent will look for more short terms rewards.
The learning rate determines how fast the agent converges a completely trained model. A high learning rate means the agent will converge faster. Simply inflating both the discount and learning rates will not guarantee better results as it depends on a lot of factors like the environment and other hyperparameters.

The episodes are the number of times we train the agent in order for it to iteratively get better.

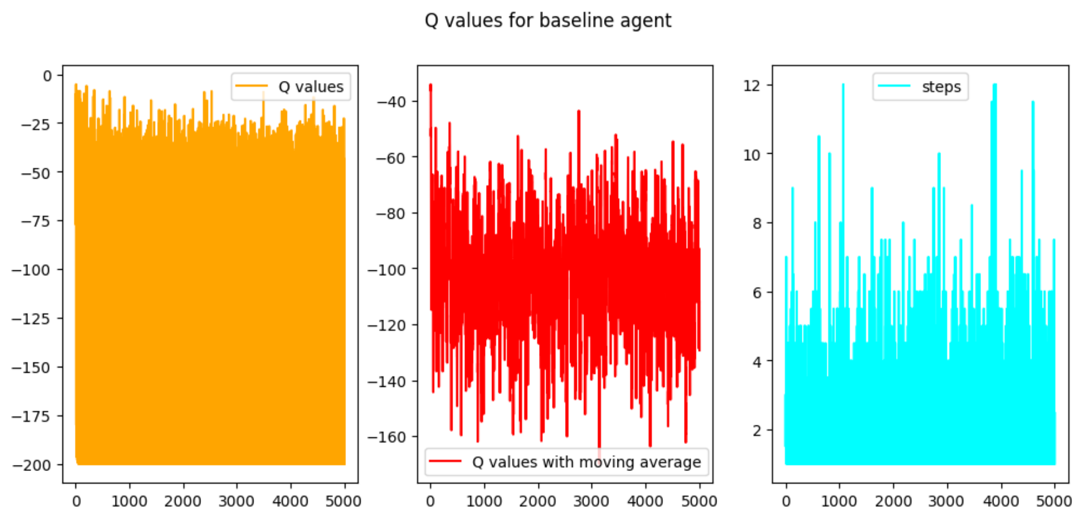
The epsilon greedy policy is such that we take a random uniform number between 1 and 0, and then check if that number is lower than an epsilon value of our choosing. If that is the case, we randomly choose an action for the next step (explore), but if epsilon is lower than that random number, we choose the best possible action (exploit) depending on the Q matrix. The random policy is that we always favour choosing our next action at random. The Boltzmann policy is that we choose the next action based on relative Q-value probability. If Q values are the same, each action is as likely as the next. This can also be controlled with a parameter called temperature. A high temperature will make all actions equally likely.

Run the Q-learning algorithm and represent its performance

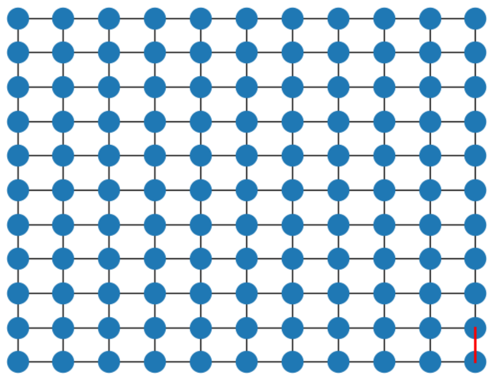
We first set up the script in a way where it will output each episode's action and corresponding q value. After that we set up a test run where we can choose where our starting point is on the maze and get the trained agent to navigate its way through the maze as efficiently as it can. After this we visualise the training scores and steps taken to converge over a particular number of episodes. This process was iteratively carried out with multiple different hyperparameter settings.

The Q learning algorithm is as follows:
Q[s_old,a] + alpha*(R[s_old,a] + gamma*(max(Q[s])) - Q[s_old,a])
where "Q" is the Q matrix (quality matrix), "s_old" is the previous state, "alpha" is the learning rate, "R" is the reward matrix, "gamma" is the discount rate, and "a" is the action.

In this section we will observe a baseline. For the baseline, the random policy was used along with a discount factor and learning rate of 0.5. The starting position was fixed to [9,10], which is the furthest away from the optimal point we can get.



In the figure above, we see the training runs across 5000 episodes, and it's clear that there is a lot of variance in performance and the model does not find neither the gold nor silver coin, and takes a random amount of steps and does not learn to be efficient.

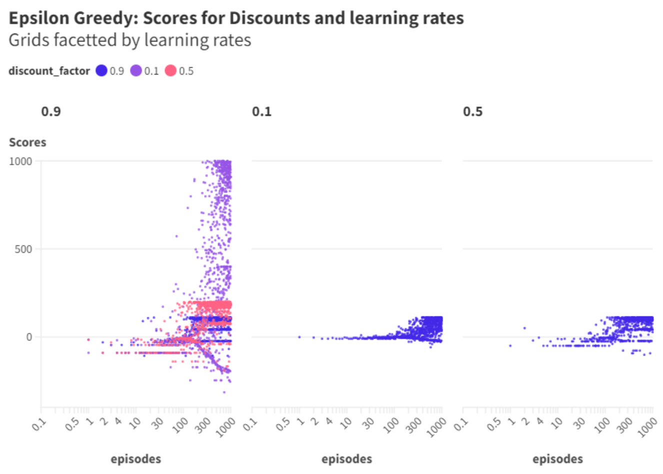


In the figure above, we see a test run of our trained agent starting from the furthest point [9,10]. It immediately falls off the maze by going downwards, showcasing its randomness.

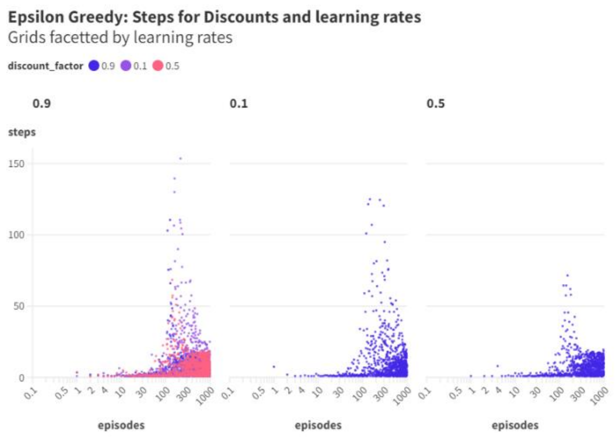
Repeat the experiment with different parameter values, and policies

In this section we will improve upon the baseline shown above. Before we searched through the different parameters, we iterated went over best strategies for starting point and epsilon decay rates. For the discount and learning rates, the values 0.1, 0.5, and 0.9 were considered. This was in order to cover a wide range of scenarios.

Epsilon greedy policy

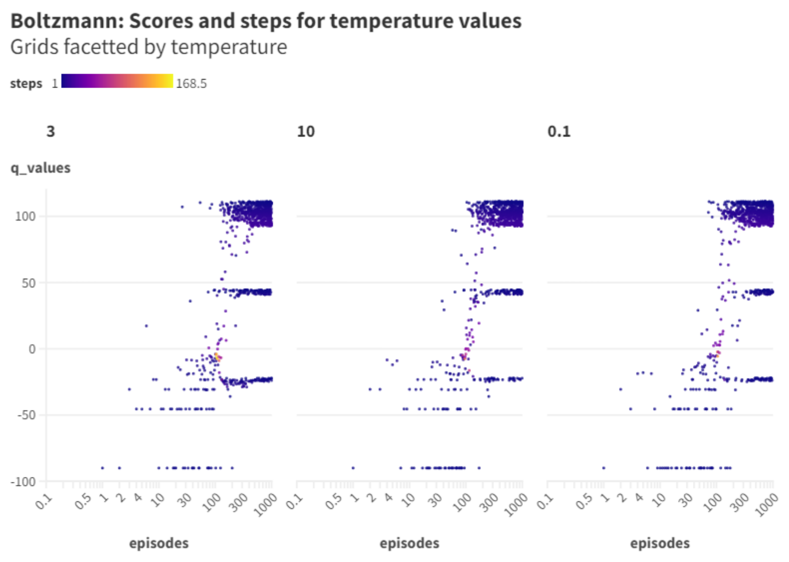


From the above graph we can see how the scores for each episode vary against different discount and learning rates. The graphs are separated into 3 facets by their learning rates, coloured according to the discount rates.



Here we can see how long the agent took to converge under different setting of discount and learning rates, similar to the graph above.

Boltzmann policy

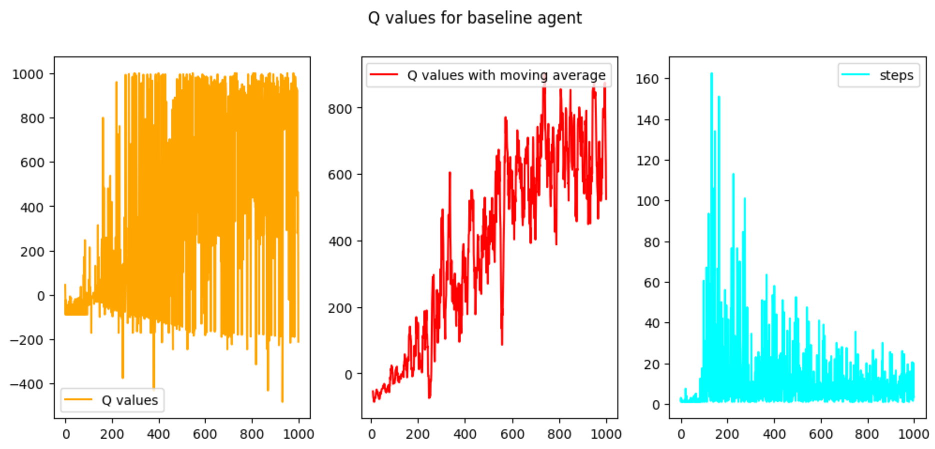


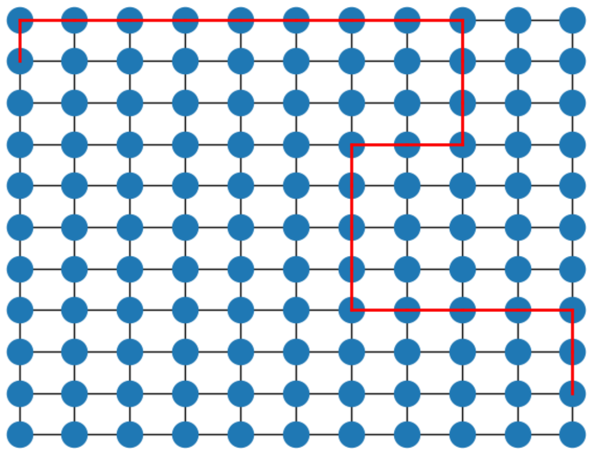
In this graph above, we see how the agent behaves with respect to steps taken. The graphs are facetted according to the different temperature values.

Analyse the results quantitatively and qualitatively

- The best search strategy was to randomly pick a starting point for each episode, instead of starting from the same place. Due to this method, when we put the agent to the test by making it start from different points in the maze, it always converges to the gold coin and avoiding the walls.
- The epsilon decay formula used was non-linear: *min_epsilon/((max_epsilon-min_epsilon)*np.exp(-epsilon_decay_rate))*. Using the linear decay formula showed a lot of variance in the results and sometimes did not help the agent to converge to an optimum state.
- The epsilon values that were fine tuned were between 0 and 1. After checking different parameter combinations, the optimal settings arrived at which were a decay rate of 0.9 and a starting epsilon value of 0.9.
- We noticed that going beyond 1 for the epsilon decay rate will make epsilon go to infinity.
- Through iterative analysis we found that setting the discount and learning rates to 0.9 worked best for Boltzmann.
- The learning rate for an epsilon greedy agent seems to be weakly correlated with number of steps taken.

Given below is the output of the training session for the best agent, which is an epsilon greedy agent with a low discount factor but high learning rate. It also managed to efficiently and consistently converge with a few number of steps. We can see in the path finding graph how it managed to scope out the gold coin, even from the furthest point.





Advanced

Introduction: Mountain Car Environment for Reinforcement Learning with DDQN and PER

This project explores applying Deep Q-Networks (DQN) with two enhancements, Double DQN (DDQN) and Prioritized Experience Replay (PER), to the classic Mountain Car environment in OpenAI Gym. The Mountain Car problem is a well-suited environment for Deep Reinforcement Learning due to the following factors:

- **Continuous Control:** The agent needs to learn to control the car's velocity in a continuous manner to navigate the environment effectively. Deep neural networks excel at approximating complex, non-linear relationships like those in this continuous control task.
- **Sparse Rewards:** The agent receives a reward of -1 for most steps and a high reward only when it reaches the goal. DQN effectively leverages past experiences through experience replay to overcome sparse reward structures.

Mountain Car Environment Description

The Mountain Car environment simulates a car trapped in a valley between two hills. The goal is to control the car (using discrete left, no action, or right acceleration actions) to reach the peak of the right hill. The state space consists of two continuous values: car position and velocity.

Here's a breakdown of the environment:

- **Observation Space:** (2,) array representing position and velocity.
- **Action Space:** Discrete (3): 0 (left), 1 (no action), 2 (right).
- **Reward:** -1 for each step (penalty for not reaching the goal), high reward for reaching the goal.
- **Episode Termination:** Reaches the goal or reaches a maximum number of steps.

Double DQN (DDQN):

This addresses the overestimation issue in standard DQN, where the action selection and target network use the same estimate, leading to overoptimistic Q-values. DDQN decouples these processes by using the main network to select the action and the target network to evaluate it, resulting in more stable and accurate learning.

A Double Deep Q-Network (DDQN) agent is implemented using PyTorch to tackle the Mountain Car environment from OpenAI Gym. The model architecture features a neural network with three linear layers: an input layer matching the state size, followed by two hidden layers with 100 and 64 units respectively, both using ReLU activation. The final output layer has a size equal to the number of available actions, representing the Q-values for each action.

Training utilizes a combination of hyperparameters, including a batch size of 64, a discount factor (γ) of 0.8, and an epsilon-greedy exploration strategy with decay. The target network, crucial for DDQN's stability, is updated every 10 episodes by copying weights from the policy network. Experiences are stored in a replay memory of size 10,000 and used for minibatch training with a learning rate of 0.01. The training process is executed for 100 episodes, with progress printed every 10. Additionally, functionalities for visualizing Q-values, comparing network parameters, and plotting training scores with smoothing are included.

We expect DDQN to converge faster and achieve higher performance compared to standard DQN by reducing overestimation bias in Q-value learning.

Following parameters were explored and their performances compared.

gamma = [0.99, 0.8, 0.5]

epsilon_decay = [0.995, 0.95, 0.85]

epsilon (linear decay) = [0.9]

learning_rate = [0.0001, 0.00001, 0.001, 0.01]

memory_size = [10000, 50000]

These parameters were used to explore this model and it seemed like the model learned to play the game quickly considering the way the reward function was defined for it to have incentive for even moving in the right direction to avoid it going in the wrong direction. The way max number of episodes were optimized to 100 to avoid longer training time considering the results did not change for longer runs. The best set for each of the params were then plugged into the same model and tried to compare the performance. Gamma=0.8, lr=0.001, epsilon

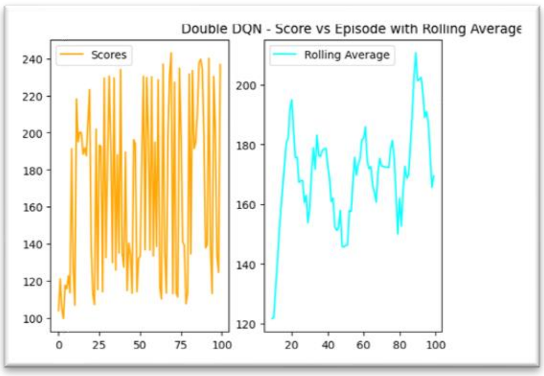


Figure 2: Scores for the optimum set of params

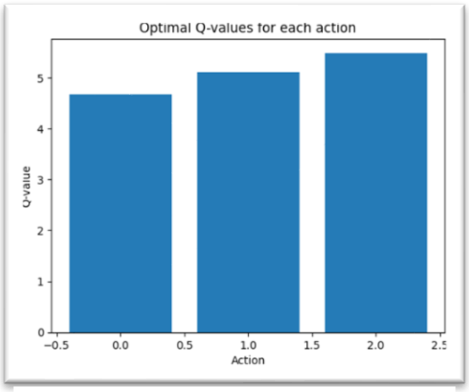


Figure 1: Optimal Q-Values

decay=0.995 and memory size= 10000. This shows that the agent learned the game well and was quickly able to learn the env and how to reach the target position and maintained its position as well.

An investigation was conducted to explore the influence of various hyperparameters on the model's learning efficiency. The reward function provided an incentive for the agent to move in the correct direction, penalizing movement in the wrong direction. To optimize training time and ensure convergence, the maximum number of episodes was set to 100. Following this exploration, the most promising hyperparameter configurations were identified and evaluated for comparative performance analysis.

The configuration that yielded the best performance consisted of:

Gamma (discount factor): 0.8; Learning rate: 0.001; Epsilon decay: 0.995; Memory size: 10,000

This configuration demonstrated rapid learning, suggesting the agent effectively acquired knowledge of the environment and the actions necessary to reach and maintain the target position. Exploration identified a hyperparameter set that facilitated fast learning. The reward function structure likely contributed to this rapid learning by rewarding positive actions. Limiting the episode count to 100 ensured efficient training without compromising convergence.

Prioritized Experience Replay (PER):

This addresses the issue of prioritizing recent experiences in standard experience replay. PER assigns priorities to experiences based on their replay value (how much they can improve the current policy), leading to more efficient learning by focusing on informative experiences.

This code implements a Prioritized Experience Replay DQN (PER DQN) agent for the Mountain Car environment using PyTorch. The DQN model has three linear layers: input (state size), hidden layer 1 (hidden_sizes[0] units with ReLU), and hidden layer 2 (hidden_sizes[1] units with ReLU). The output layer has action_size units representing Q-values.

Key hyperparameters include a batch size of 64, discount factor (γ) of 0.8, learning rate of 0.0001, and target network updates every 10 episodes. Epsilon-greedy exploration with decay is used.

The PrioritizedReplayBuffer prioritizes experiences in the replay memory (size: 10000) based on TD-error for more efficient training. The PER DQN algorithm utilizes separate policy and target networks, updated periodically for stability. Experiences are sampled with priority bias correction, and the policy network is updated using target Q-values. Visualization functionalities (commented out) are included for Q-values and network parameters. Training scores are plotted with smoothing.

We expect PER to improve learning efficiency by prioritizing informative experiences that are more likely to lead to larger updates in the Q-network.

Following parameters were explored and their performances compared.

gamma = [0.99, 0.9, 0.8]

learning_rate = [0.0001, 0.001, 0.01, 0.00001]

epsilon_decay = [0.995, 0.95, 0.85]

epsilon (linear decay) = [0.9]

PER DQN's effectiveness in mastering the Mountain Car environment hinges on several key features. First, Prioritized Experience Replay (PER) prioritizes transitions with larger temporal-difference (TD) errors for more frequent sampling during training. This, with the chosen alpha value of 0.4, improves the agent's learning efficiency. Second, Double Q-Learning, implemented with separate policy and target networks updated every 10 episodes (target_update_freq), reduces overestimation of Q-values, leading to more stable learning. Finally, the exploration-exploitation trade-off is managed by an epsilon-greedy strategy with a decay rate of 0.85 (epsilon_decay). This ensures the agent starts with high exploration (epsilon_start = 1.0) and gradually transitions to exploiting learned actions (epsilon_end = 0.01) throughout training. These elements, combined with well-tuned hyperparameters like a gamma of 0.8 and a learning rate of 0.0001, empower PER DQN to achieve efficient and stable learning in the Mountain Car environment.

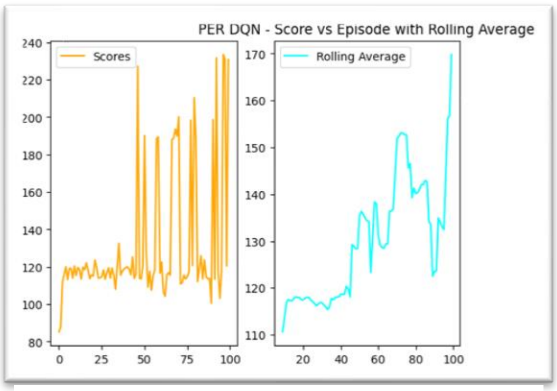


Figure 4: Scores for the optimum set of params

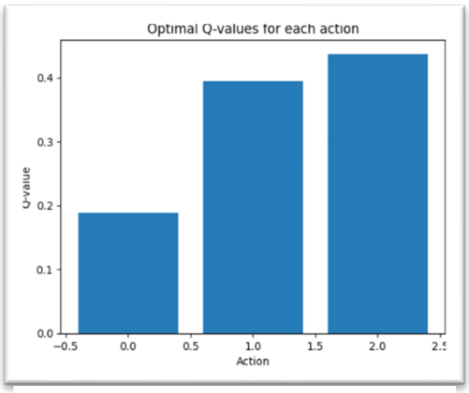
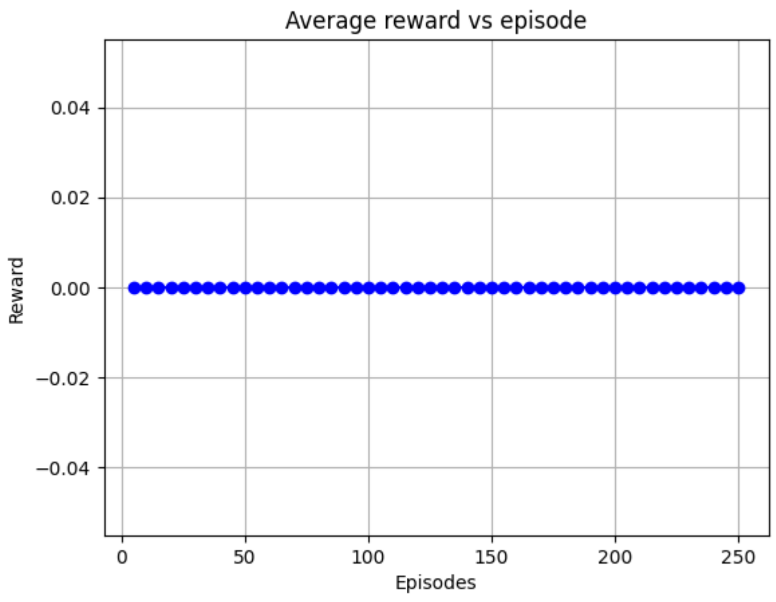


Figure 3: Optimal Q-Values

Individual task

For the individual task I attempted to model a DQN agent to play the Atari game Breakout. A DQN is comprised of two parts. One is that it estimates the Q-values using a network and uses another target network to prevent overestimating the Q-values. It also has a replay memory buffer to store past experiences. It is an off-policy method which is good at generalising and capturing long term information.

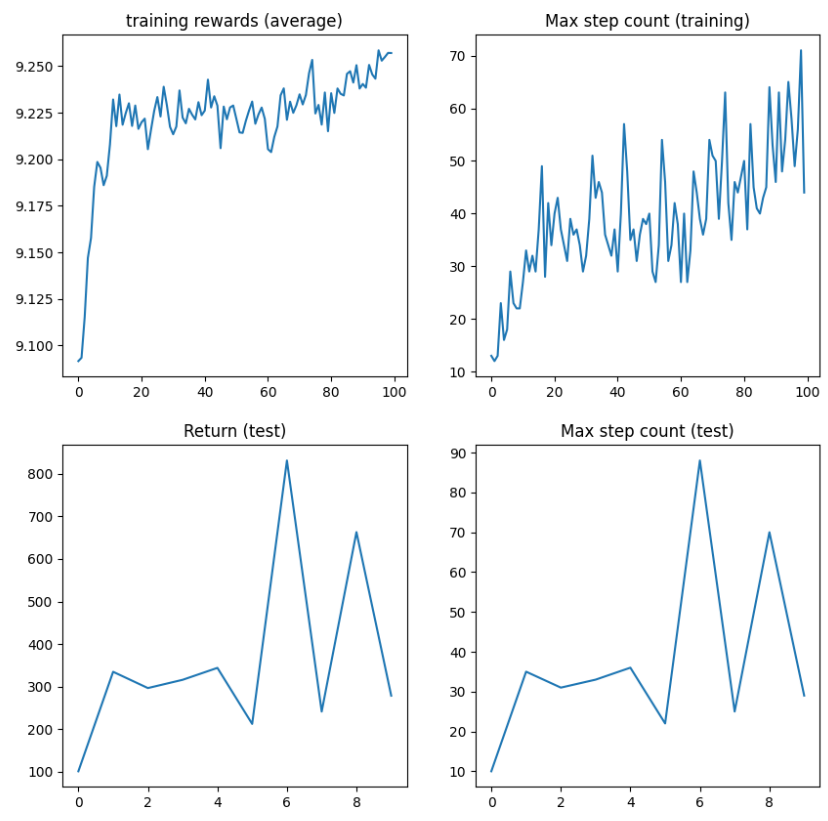
Unfortunately, I was unsuccessful in training the agent properly since it was taking a very long time to converge to optimum results, and I was met with out of memory error. I sized down the model to 250 episodes and with small replay buffer. Although my model did not have the best conditions to converge, I was able to see how much of training and episodes it would take to see a noticeable result, since the environment is very complex. The model I was used had 2 hidden layers, which would suggest that more layers would be required to converge faster and exploit useful strategies.



Extra task

I have applied PPO for the InvertedDoublePendulum environment. PPO is an on-policy gradient algorithm and is used as an improvement to TRPO and REINFORCE. It can learn from current learned policy, and has the functionality to not deviate too far away from past optimal policies.

I tried different combinations of lambda, gamma values, clip epsilon, and entropy epsilon values. I noticed that increasing the entropy decreases performances despite taking fewer steps to converge. A small clip epsilon is very important to achieving high performances, coupled with low entropy. However, I tested the model in the pendulum environment with only limited frames. For further work would it would be useful to look at high frames and episodes.



Contribution to Team task:

Ashwin Meth Dias Bandaranaike:

Basic: We defined and improved an already existing environment, and worked together to iterate through different hyperparameters

Advanced: Did an initial analysis of what models to use and then worked to implement and improve them.

Link to Github repo: https://github.com/Siripala-98/DRL_INM707_230009944

References

[1] L. Panneerselvam, “Q – Learning Algorithm with Step by Step Implementation using Python,” Analytics Vidhya, Apr. 24, 2021. https://www.analyticsvidhya.com/blog/2021/04/q-learning-algorithm-with-step-by-step-implementation-using-python/#wait_approval (accessed May 12, 2024).

Taken from: https://colab.research.google.com/drive/1E2RViy7xmor0mhqskZV14_NUj2jMpJz3?authuser=1#scrollTo=9AdpFVfy6ya9

```
#import libraries
import numpy as np
import time
from datetime import datetime
import pandas as pd
import random
import networkx as nx
import matplotlib.pyplot as plt


#define the shape of the environment (i.e., its states)
environment_rows = 11
environment_columns = 11

#Create a 3D numpy array to hold the current Q-values for each state and action pair: Q(s, a)
#The array contains 11 rows and 11 columns (to match the shape of the environment), as well as a third "action" dimension.
#The "action" dimension consists of 4 layers that will allow us to keep track of the Q-values for each possible action in
#each state (see next cell for a description of possible actions).
#The value of each (state, action) pair is initialized to 0.
q_m = np.zeros((environment_rows, environment_columns, 4))
# q_values = np.random.randint(5, size = (11, 11))
q_m
```

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
q_m[(5,3)]
```

```
array([0., 0., 0., 0.])
```

Actions

The actions that are available to the AI agent are to move the robot in one of four directions:

- Up
- Right
- Down
- Left

Obviously, the AI agent must learn to avoid driving into the item storage locations (e.g., shelves)!

```
#define actions
#numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left
actions = ['up', 'right', 'down', 'left']
```

Rewards

The last component of the environment that we need to define are the **rewards**.

To help the AI agent learn, each state (location) in the warehouse is assigned a reward value.

The agent may begin at any white square, but its goal is always the same: ***to maximize its total rewards***!

Negative rewards (i.e., **punishments**) are used for all states except the goal.

- This encourages the AI to identify the *shortest path* to the goal by *minimizing its punishments*!

To maximize its cumulative rewards (by minimizing its cumulative punishments), the AI agent will need find the shortest paths between the item packaging area (green square) and all of the other locations in the warehouse where the robot is allowed to travel (white squares). The agent will also need to learn to avoid crashing into any of the item storage locations (black squares)!

```
# #Create a 2D numpy array to hold the rewards for each state.
# #The array contains 11 rows and 11 columns (to match the shape of the environment), and each value is initialized to -100.
# rewards = np.full((environment_rows, environment_columns), -100.)
# rewards[0, 5] = 100. #set the reward for the packaging area (i.e., the goal) to 100

# #define aisle locations (i.e., white squares) for rows 1 through 9
# aisles = {} #store locations in a dictionary
# aisles[1] = [i for i in range(1, 10)]
# aisles[2] = [1, 7, 9]
# aisles[3] = [i for i in range(1, 8)]
# aisles[3].append(9)
# aisles[4] = [3, 7]
# aisles[5] = [i for i in range(11)]
# aisles[6] = [5]
# aisles[7] = [i for i in range(1, 10)]
# aisles[8] = [3, 7]
# aisles[9] = [i for i in range(11)]

# #set the rewards for all aisle locations (i.e., white squares)
# for row_index in range(1, 10):
#   for column_index in aisles[row_index]:
#     rewards[row_index, column_index] = -1.

# #print rewards matrix
# for row in rewards:
#   print(row)


# rewards
```

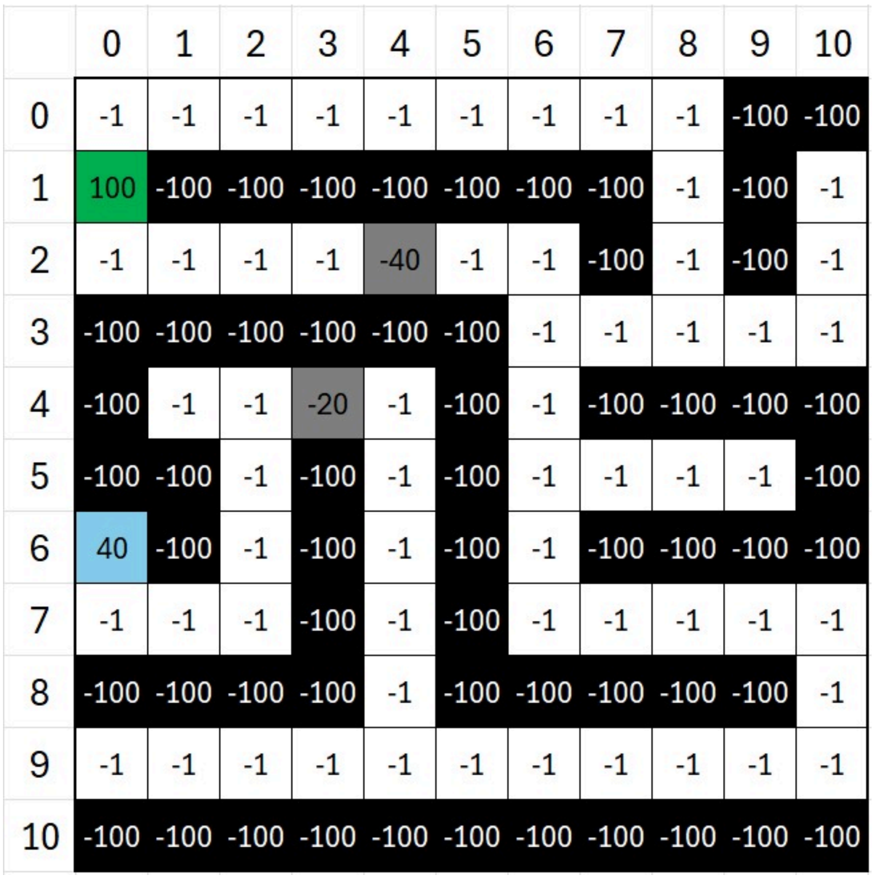
Alternate env

```
#Create a 2D numpy array to hold the rewards for each state.
#The array contains 11 rows and 11 columns (to match the shape of the environment), and each value is initialized to -100.
rewards = np.full((environment_rows, environment_columns), -100.)
# rewards[0, 5] = 100. #set the reward for the packaging area (i.e., the goal) to 100
rewards[1, 0] = 100. #set the reward for the packaging area (i.e., the goal) to 100
rewards[6, 0] = 40. # Set sub-optimal reward


#define aisle locations (i.e., white squares) for rows 1 through 9
aisles = {} #store locations in a dictionary
aisles[0] = [i for i in range(0, 9)]
aisles[1] = [8, 10]
aisles[2] = [i for i in range(0, 7)]
aisles[2].append(8)
aisles[2].append(10)
aisles[3] = [i for i in range(6, 11)]
aisles[4] = [i for i in range(1, 5)]
aisles[4].append(6)
aisles[5] = [2, 4, 6, 7, 8, 9]
aisles[6] = [2, 4, 6]
aisles[7] = [0, 1, 2, 4, 6, 7, 8, 9, 10]
aisles[8] = [4, 10]
aisles[9] = [i for i in range(11)]

#set the rewards for all aisle locations (i.e., white squares)
for row_index in range(0, 10):
  for column_index in aisles[row_index]:
    rewards[row_index, column_index] = -1.

# Add puddles (penalties)
rewards[2, 4] = -40
rewards[4, 3] = -20

# #print rewards matrix
# for row in rewards:
#   print(row)
pd.DataFrame(rewards)
```

	0	1	2	3	4	5	6	7	8	9	10
0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-100.0	-100.0
1	100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-1.0	-100.0	-1.0
2	-1.0	-1.0	-1.0	-1.0	-40.0	-1.0	-1.0	-100.0	-1.0	-100.0	-1.0
3	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-1.0	-1.0	-1.0	-1.0	-1.0
4	-100.0	-1.0	-1.0	-20.0	-1.0	-100.0	-1.0	-100.0	-100.0	-100.0	-100.0
5	-100.0	-100.0	-1.0	-100.0	-1.0	-100.0	-1.0	-1.0	-1.0	-1.0	-100.0
6	40.0	-100.0	-1.0	-100.0	-1.0	-100.0	-1.0	-100.0	-100.0	-100.0	-100.0
7	-1.0	-1.0	-1.0	-100.0	-1.0	-100.0	-1.0	-1.0	-1.0	-1.0	-1.0
8	-100.0	-100.0	-100.0	-100.0	-1.0	-100.0	-100.0	-100.0	-100.0	-100.0	-1.0
9	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
10	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0

```
rewards[8, 0]
```

```
-100.0
```



	0	1	2	3	4	5	6	7	8	9	10
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100	-100
1	100	-100	-100	-100	-100	-100	-100	-100	-1	-100	-1
2	-1	-1	-1	-1	-40	-1	-1	-100	-1	-100	-1
3	-100	-100	-100	-100	-100	-100	-1	-1	-1	-1	-1
4	-100	-1	-1	-20	-1	-100	-1	-100	-100	-100	-100
5	-100	-100	-1	-100	-1	-100	-1	-1	-1	-1	-100
6	40	-100	-1	-100	-1	-100	-1	-100	-100	-100	-100
7	-1	-1	-1	-100	-1	-100	-1	-1	-1	-1	-1
8	-100	-100	-100	-100	-1	-100	-100	-100	-100	-100	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

Train the Model

Our next task is for our AI agent to learn about its environment by implementing a Q-learning model. The learning process will follow these steps:

1. Choose a random, non-terminal state (white square) for the agent to begin this new episode.
2. Choose an action (move *up*, *right*, *down*, or *left*) for the current state. Actions will be chosen using an *epsilon greedy algorithm*. This algorithm will usually choose the most promising action for the AI agent, but it will occasionally choose a less promising option in order to encourage the agent to explore the environment.
3. Perform the chosen action, and transition to the next state (i.e., move to the next location).
4. Receive the reward for moving to the new state, and calculate the temporal difference.
5. Update the Q-value for the previous state and action pair.
6. If the new (current) state is a terminal state, go to #1. Else, go to #2.

This entire process will be repeated across 1000 episodes. This will provide the AI agent sufficient opportunity to learn the shortest paths between the item packaging area and all other locations in the warehouse where the robot is allowed to travel, while simultaneously avoiding crashing into any of the item storage locations!

```
# Number of white squares
sum(rewards == -1).sum()
```

```
59
```

Define Helper Functions

```
#define a function that determines if the specified location is a terminal state
def is_terminal_state(current_row_index, current_column_index):
  """
  This function is to determine whether the current state is a state which will terminate the game or not.
  If this function returns True, then it will terminate the game.
  Landing on a white square continues the game, while landing on a black or the green square will terminate it.
  """
  if rewards[current_row_index, current_column_index] == -1.: # Falling on white
    return False
  else: # Falling on green or black
    return True

#define a function that will choose a random, non-terminal starting location
def get_starting_location(row = None, col = None):
  current_row_index = np.random.randint(environment_rows) if row is None else row
  current_column_index = np.random.randint(environment_columns) if col is None else col

  # # get a random row and column index
  # current_row_index = np.random.randint(environment_rows)
  # current_column_index = np.random.randint(environment_columns)

  # # Get specific row and column
  # current_row_index = 9
  # current_column_index = 10

  #continue choosing random row and column indexes until a non-terminal state is identified
  #(i.e., until the chosen state is a 'white square').
  while is_terminal_state(current_row_index, current_column_index):
    current_row_index = np.random.randint(environment_rows)
    current_column_index = np.random.randint(environment_columns)
  return current_row_index, current_column_index

# Epsilon greedy policy
def eps_greedy(current_row_index, current_column_index, epsilon):
  #if a randomly chosen value between 0 and 1 is less than epsilon,
  #then choose the most promising value from the Q-table for this state.
  if np.random.random() < epsilon:
    return np.random.randint(4)
  else: #choose a random action
    return np.argmax(q_m[current_row_index, current_column_index])

# Random policy
def random_pol(current_row_index, current_column_index):
  return np.random.randint(4)

# Botlzmann policy (ref: https://github.com/8Gitbrix/Reinforcement-Learning/blob/master/qlearn.py and https://automaticaddison.com/bolt
def boltz_policy(current_row_index, current_column_index, tau):
    if tau > 0:
      p = np.array([q_m[current_row_index, current_column_index, x]/tau for x in range(4)], dtype=np.float128)
      prob_actions = np.exp(p) / np.sum(np.exp(p))
      cumulative_probability = 0.0
      choice = random.uniform(0,1)
      for a,pr in enumerate(prob_actions):
          cumulative_probability += pr
          if cumulative_probability > choice:
              return a
    else:
      return np.argmax(q_m[current_row_index, current_column_index])


# Transition function
def get_next_location(current_row_index, current_column_index, action_index):
  """
  This will instruct how the agent will move. As long as the agent is not near one of the four walls, it can move about freely.
  The moment it is against a wall, it will not be able to move beyond it, and the next location will be the same as the current
  location.
  """
  new_row_index = current_row_index
  new_column_index = current_column_index
  if actions[action_index] == 'up' and current_row_index > 0:
    new_row_index -= 1
  elif actions[action_index] == 'right' and current_column_index < environment_columns - 1:
    new_column_index += 1
  elif actions[action_index] == 'down' and current_row_index < environment_rows - 1:
    new_row_index += 1
  elif actions[action_index] == 'left' and current_column_index > 0:
    new_column_index -= 1
  return new_row_index, new_column_index

#Define a function that will get the shortest path between any location within the warehouse that
#the robot is allowed to travel and the item packaging location.
def get_shortest_path(start_row_index, start_column_index):
  #return immediately if this is an invalid starting location
  if is_terminal_state(start_row_index, start_column_index):
    return []
  else: #if this is a 'legal' starting location
    current_row_index, current_column_index = start_row_index, start_column_index
    shortest_path = []
    shortest_path.append([current_row_index, current_column_index])
    #continue moving along the path until we reach the goal (i.e., the item packaging location)
    while not is_terminal_state(current_row_index, current_column_index):

      # Choose policy
      action_index = eps_greedy(current_row_index, current_column_index, .0) # Epsilon greedy policy
      # action_index = boltz_policy(row_index, column_index, tau) # Boltzmann policy
      # action_index = random_pol(row_index, column_index) # Random policy

      #move to the next location on the path, and add the new location to the list
      current_row_index, current_column_index = get_next_location(current_row_index, current_column_index, action_index)
      shortest_path.append([current_row_index, current_column_index])
      # print(current_row_index, current_column_index)
    return shortest_path


# #display a few shortest paths
# start = time.time()
# print(f"Optimal path finding started: {datetime.fromtimestamp(start)}")

# print(f"\nPath: {get_shortest_path(0, 8)} (Moves: {len(get_shortest_path(3, 9))})") #starting at row 3, column 9
# print(f"\nPath: {get_shortest_path(5, 9)} (Moves: {len(get_shortest_path(5, 0))})") #starting at row 5, column 0
# print(f"\nPath: {get_shortest_path(9, 10)} (Moves: {len(get_shortest_path(9, 5))})") #starting at row 9, column 5

# end = time.time()
# print(f"\nOptimal path finding ended: {datetime.fromtimestamp(end)}")
# elapsed_time = end - start
# print(f"Elapsed time: {np.round(elapsed_time/60, 3)} minutes\n")
```

We can see before training, in this particular starting positions, the agent takes very long to converge, hence we had to interrupt it half way.

After training however, the agent manages to get there instantly

Train the AI Agent using Q-Learning

```
# #define training parameters
# epsilon = [0.1, 0.5, 0.9] # exploration
# max_epsilon = 1
# min_epsilon = 0.01
# epsilon_decay_rate = [0.1, 0.5, 0.9] # If you go beyond 1 it will make epsilon go to inf
# discount_factor = [0.1, 0.5, 0.9] #discount factor for future rewards
# learning_rate = [0.1, 0.5, 0.9] #the rate at which the AI agent should learn
# training_episodes = 1000
# tau = 5 # This is temperature for blotzmann policy. Higher tau means more equal probability of taking an action. Lower tau is choosin

# param_search_q_values = []
# param_search_episodes = []
# param_search_steps = []
# param_search_params = []
# param_index = []

# for param_i, param in enumerate(epsilon):
#   q_values = []
#   episodes = []
#   steps = []
#   q_m = np.zeros((environment_rows, environment_columns, 4))

#   #run through training episodes
#   for episode in range(training_episodes):
#     # Define starting location. Keep it empty to randomise start location during training
#     row_index, column_index = get_starting_location()
#     step = 0

#     #continue taking actions (i.e., moving) until we reach a terminal state
#     #(i.e., until we reach the item packaging area or crash into an item storage location)
#     while not is_terminal_state(row_index, column_index):

#       # Choose the policy
#       action_index = eps_greedy(row_index, column_index, param) # Epsilon greedy policy
#       # action_index = boltz_policy(row_index, column_index, tau) # Boltzmann policy
#       # action_index = random_pol(row_index, column_index) # Random policy

#       # Decay the epsilon rate linearly
#       if param > 0.01:
#         param *= epsilon_decay_rate[-1]

#       # # Temp decay
#       # if tau > 0.1:
#       #   tau -= 0.5

#       # # Decay the epsilon rate exponentially based on minimum and max values
#       # epsilon = min_epsilon/((max_epsilon-min_epsilon)*np.exp(-epsilon_decay_rate[-1])) # https://medium.com/@nancyjemi/level-up-un
#       # # epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-1. * episode / epsilon_decay_rate[-1]) # Non-lienar decay

#       #perform the chosen action, and transition to the next state (i.e., move to the next location)
#       old_row_index, old_column_index = row_index, column_index #store the old row and column indexes
#       row_index, column_index = get_next_location(row_index, column_index, action_index)

#       # Bellman's equation: Q[s_old,a] + alpha*(R[s_old,a] + gamma*(max(Q[s])) - Q[s_old,a])
#       q_m[old_row_index, old_column_index, action_index] = q_m[old_row_index, old_column_index, action_index] + learning_rate[-1]*(re
#       print(f"\nAction taken: Go {actions[action_index]} (index = {action_index})\nMoving from {old_row_index, old_column_index} to {
#       print(f"Q value is: {q_m[old_row_index, old_column_index, action_index]}")
#       q_values.append(q_m[old_row_index, old_column_index, action_index])
#       episodes.append(episode)
#       step += 1
#       steps.append(step)

#       # print(f"\nQ matrix for going {actions[action_index]} is:\n {q_values[:, :, action_index]}")
#       # #receive the reward for moving to the new state, and calculate the temporal difference
#       # reward = rewards[row_index, column_index]
#       # old_q_value = q_values[old_row_index, old_column_index, action_index]
#       # temporal_difference = reward + (discount_factor * np.max(q_values[row_index, column_index])) - old_q_value

#       # #update the Q-value for the previous state and action pair
#       # new_q_value = old_q_value + (learning_rate * temporal_difference)
#       # q_values[old_row_index, old_column_index, action_index] = new_q_value
#     param_search_q_values.append(q_values)
#     param_search_episodes.append(episodes)
#     param_search_steps.append(steps)
#     param_search_params.append(param)
#     param_index.append(param_i)

#   print(f"\nTraining complete for {training_episodes} training episodes!")
```

numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left

```
# pd.DataFrame(param_search_steps[100])
```

The reason that the q matrix is 3D, is that the 3rd dimension contains the q values for each action at any given point.

For example at (2,0), going up (action = 0) has a q value of 100, which makes sense because going up from (2,0) will win the game.

So then we look at the q matrix corresponding to action = 0, from the 4 actions. This means we access the q matrix that corresponds to action = 0 from the 4 matrices in the 3rd dimension of q_values. Here we will see that (2,0) has a q value of 100, nudging the agent to move upwards (take action = 0).

```
# Individual param testing
epsilon = 0.9 # exploration
max_epsilon = 1
min_epsilon = 0.01
epsilon_decay_rate = 0.9 # If you go beyond 1 it will make epsilon go to inf
discount_factor = 0.1 #discount factor for future rewards (gamma)
learning_rate = 0.9 #the rate at which the AI agent should learn (alpha)
training_episodes = 1000
tau = 0.1 # This is temperature for blotzmann policy. Higher tau means more equal probability of taking an action. Lower tau is choosin

q_values = []
episodes = []
steps = []

#run through training episodes
for episode in range(training_episodes):
  # Define starting location. Keep it empty to randomise start location during training
  row_index, column_index = get_starting_location()
  step = 0

  #continue taking actions (i.e., moving) until we reach a terminal state
  #(i.e., until we reach the item packaging area or crash into an item storage location)
  while not is_terminal_state(row_index, column_index):

    # Choose the policy
    action_index = eps_greedy(row_index, column_index, epsilon) # Epsilon greedy policy
    # action_index = boltz_policy(row_index, column_index, tau) # Boltzmann policy
    # action_index = random_pol(row_index, column_index) # Random policy

    # # Decay the epsilon rate linearly
    # if epsilon > 0.01:
    #   epsilon *= epsilon_decay_rate

    # # Temp decay
    # if tau > 1.:
    #   tau -= 0.1

    # Decay the epsilon rate exponentially based on minimum and max values
    epsilon = min_epsilon/((max_epsilon-min_epsilon)*np.exp(-epsilon_decay_rate)) # https://medium.com/@nancyjemi/level-up-understandin
    # epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-1. * episode / epsilon_decay_rate) # Non-lienar decay

    #perform the chosen action, and transition to the next state (i.e., move to the next location)
    old_row_index, old_column_index = row_index, column_index #store the old row and column indexes
    row_index, column_index = get_next_location(row_index, column_index, action_index)

    # Bellman's equation: Q[s_old,a] + alpha*(R[s_old,a] + gamma*(max(Q[s])) - Q[s_old,a])
    q_m[old_row_index, old_column_index, action_index] = q_m[old_row_index, old_column_index, action_index] + learning_rate*(rewards[ro
    print(f"\nAction taken: Go {actions[action_index]} (index = {action_index})\nMoving from {old_row_index, old_column_index} to {row_
    print(f"Q value is: {q_m[old_row_index, old_column_index, action_index]}")
    q_values.append(q_m[old_row_index, old_column_index, action_index])
    episodes.append(episode)
    step += 1
    steps.append(step)

    # print(f"\nQ matrix for going {actions[action_index]} is:\n {q_values[:, :, action_index]}")
    # #receive the reward for moving to the new state, and calculate the temporal difference
    # reward = rewards[row_index, column_index]
    # old_q_value = q_values[old_row_index, old_column_index, action_index]
    # temporal_difference = reward + (discount_factor * np.max(q_values[row_index, column_index])) - old_q_value

    # #update the Q-value for the previous state and action pair
    # new_q_value = old_q_value + (learning_rate * temporal_difference)
    # q_values[old_row_index, old_column_index, action_index] = new_q_value

print(f"\nTraining complete for {training_episodes} training episodes!")
```

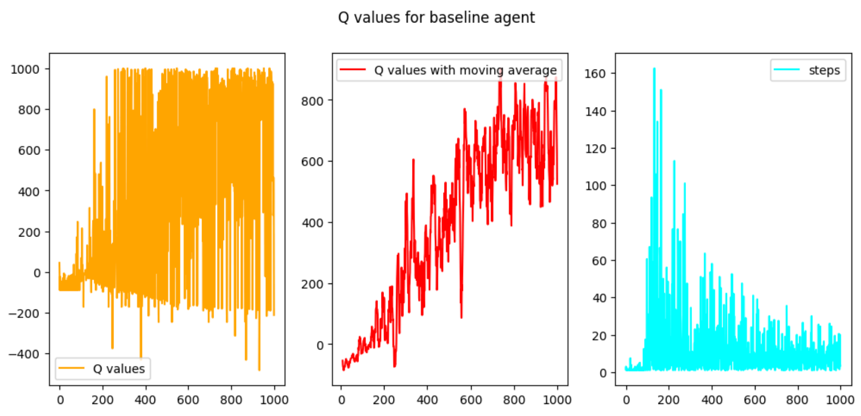
```
Streaming output truncated to the last 5000 lines.
Moving from (0, 2) to (0, 1) with reward -1.0
Q value is: 979.9999999999982

Action taken: Go left (index = 3)
Moving from (0, 1) to (0, 0) with reward -1.0
Q value is: 989.9999999999987

Action taken: Go down (index = 2)
Moving from (0, 0) to (1, 0) with reward 100.0
Q value is: 999.9999999999993

Action taken: Go left (index = 3)
Moving from (7, 8) to (7, 7) with reward -1.0
Q value is: 708.7857739213772

Action taken: Go left (index = 3)
Moving from (7, 7) to (7, 6) with reward -1.0
Q value is: 775.7414237570148

Action taken: Go up (index = 0)
Moving from (7, 6) to (6, 6) with reward -1.0
Q value is: 814.611983474748

Action taken: Go up (index = 0)
Moving from (6, 6) to (5, 6) with reward -1.0
Q value is: 836.2190157288333

Action taken: Go up (index = 0)
Moving from (5, 6) to (4, 6) with reward -1.0
Q value is: 849.2999036919308

Action taken: Go up (index = 0)
Moving from (4, 6) to (3, 6) with reward -1.0
Q value is: 859.8171081825539

Action taken: Go right (index = 1)
Moving from (3, 6) to (3, 7) with reward -1.0
Q value is: 869.9681070132146

Action taken: Go right (index = 1)
Moving from (3, 7) to (3, 8) with reward -1.0
Q value is: 879.995266691308

Action taken: Go up (index = 0)
Moving from (3, 8) to (2, 8) with reward -1.0
Q value is: 889.9994599891417

Action taken: Go up (index = 0)
Moving from (2, 8) to (1, 8) with reward -1.0
Q value is: 899.9998963873518

Action taken: Go up (index = 0)
Moving from (1, 8) to (0, 8) with reward -1.0
Q value is: 909.9999819402808

Action taken: Go left (index = 3)
Moving from (0, 8) to (0, 7) with reward -1.0
```

```
results = pd.DataFrame([q_values, episodes, steps], index = ["q_values", "episodes", "steps"]).T

results_grouped_episodes = results.groupby(by = ["episodes"]).mean().reset_index()
results_grouped_episodes
```

	episodes	q_values	steps
0	0.0	44.550000	1.5
1	1.0	-18.720000	3.0
2	2.0	-90.000000	1.0
3	3.0	-23.175000	2.5
4	4.0	-45.450000	1.5
...	...	...	...
995	995.0	277.874035	1.5
996	996.0	293.182255	4.0
997	997.0	463.842136	20.0
998	998.0	337.188428	3.5
999	999.0	-211.869125	3.5

1000 rows × 3 columns

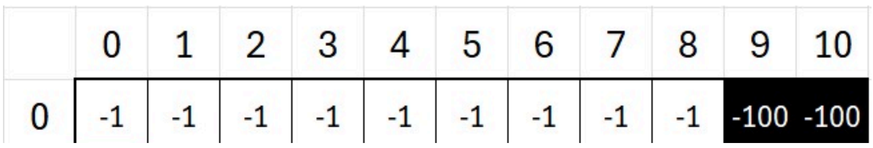
```
import matplotlib.pyplot as plt
import seaborn as sns

window_size = 10  # Adjust the window size as needed
rolling_avg_scores = np.convolve(results_grouped_episodes['q_values'], np.ones(window_size)/window_size, mode='valid')

fig, ax = plt.subplots(1, 3, figsize=(12, 5))
fig.suptitle("Q values for baseline agent")
ax[0].plot(results_grouped_episodes["episodes"], results_grouped_episodes['q_values'], label = "Q values", color = "orange")
ax[1].plot(range(training_episodes)[window_size - 1:], rolling_avg_scores, label = "Q values with moving average", color = "red")
ax[2].plot(results_grouped_episodes["episodes"], results_grouped_episodes['steps'], label = "steps", color = "cyan")
ax[0].legend()
ax[1].legend()
ax[2].legend()
plt.show()
```



Test run

```
# epsilon
tau
```

```
0.1
```



	0	1	2	3	4	5	6	7	8	9	10
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100	-100

```
# get_starting_location()

start = time.time()
print(f"Optimal path finding started: {datetime.fromtimestamp(start)}\n")

# Starting location
# Interesting starting locations: 9,4 | 3,8 | 5,9 | 9,10 | 1,10
# row, col = 9,10
row, col = get_starting_location(9,10)
shortest_path_result = get_shortest_path(row, col)
print()
print(f"Path: {shortest_path_result} (Moves: {len(shortest_path_result)})\n") #starting at row 3, column 9

end = time.time()
print(f"Optimal path finding ended: {datetime.fromtimestamp(end)}")
elapsed_time = end - start
print(f"Elapsed time: {np.round(elapsed_time/60, 3)} minutes\n")

# Draw path

# Define the size of the grid
n = 11
# Create a grid graph
G = nx.grid_2d_graph(n, n)
# Set the position of the nodes using the node coordinates
pos = {(x, y): (y, -x) for x, y in G.nodes()}
nx.draw(G, pos=pos, node_size=300, with_labels=False)

# Define the coordinates of the boxes you want the line to go through
# coords = [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
coords = [tuple(x) for x in shortest_path_result] # https://stackoverflow.com/questions/5506511/python-converting-list-of-lists-to-tupl

# Draw the line through the specified coordinates
for i in range(len(coords) - 1):
    x1, y1 = coords[i]
    x2, y2 = coords[i + 1]
    plt.plot([y1, y2], [-x1, -x2], 'r-', linewidth=2)
# Set the x and y axis ticks
# plt.xticks(range(n), [str(i) for i in range(n)])  # Column indices
# plt.yticks(range(-n, 0), [str(-i) for i in range(n)])  # Row indices
plt.show()
```

```
Optimal path finding started: 2024-05-12 14:42:25.378140
```

```
!pip install gym
```

```
!pip install plotly
```

```
# !pip install pygame
# !pip uninstall pygame
```

```
# !pip install gym[toy_text]==0.26.*
# !pip install gym[toy_text]==0.26.2
!pip install gym[box2d]==0.26.0 pyglet==1.5.27 pyvirtualdisplay
```

```
!pip install pyvirtualdisplay
!pip install pyglet==1.5.27
```

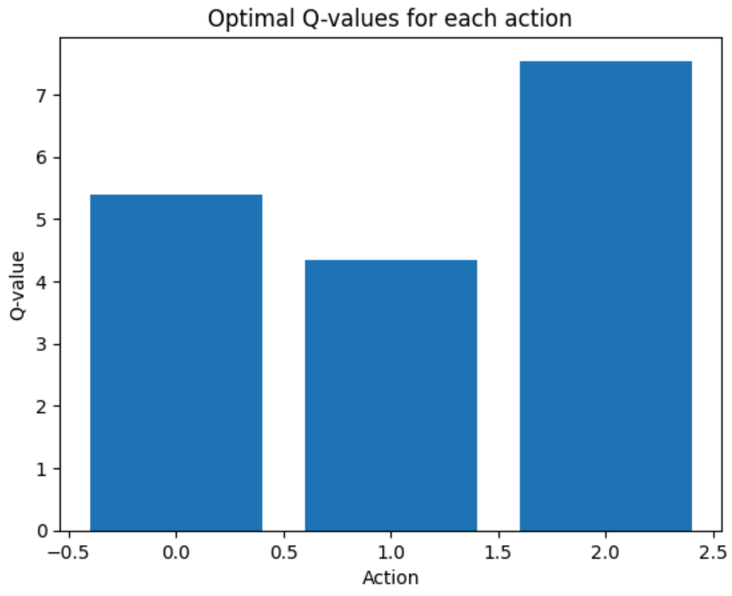
Double DQN

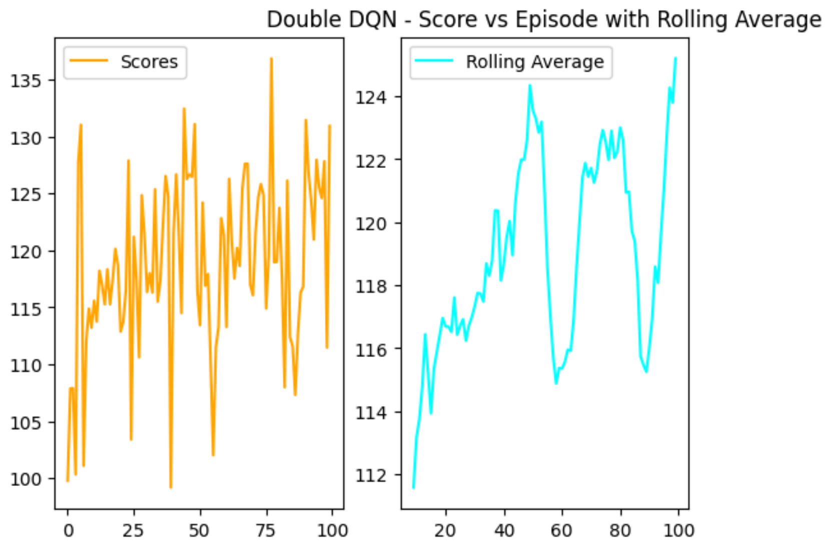
```
# Run this

import gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import deque
import plotly.graph_objects as go
import matplotlib.pyplot as plt

random_seed = 42
random.seed(random_seed)
np.random.seed(random_seed)
torch.manual_seed(random_seed)

# https://www.bing.com/search?pglt=43&q=nn.Module&cvid=3e8744507f7d432fb59a2a95abeccad8&gs_lcrp=
# EgZjaHJvbWUyBggAEEUYOTIGCAEQABhAMgYIAhAAGEAyBggDEAAYQDIGCAQQABhAMgYIBRAAGEAyBggGEAAYQDIGCAcQABhAMgYICBBFGDzSAQcxNDhqMGoxqAIAsAIA&FORM=A

# Define Double DQN Model Architecture
class DoubleDQN(nn.Module):
    def __init__(self, state_size, action_size, hidden_sizes):
        super(DoubleDQN, self).__init__()
        self.hidden1 = nn.Linear(state_size, hidden_sizes[0])
        self.hidden2 = nn.Linear(hidden_sizes[0], hidden_sizes[1])
        self.output = nn.Linear(hidden_sizes[1], action_size)

    def forward(self, state):
        x = torch.relu(self.hidden1(state))
        x = torch.relu(self.hidden2(x))
        return self.output(x)

# Function to visualize Q-values predicted by the policy network
def visualize_q_values(policy_net, state):
    with torch.no_grad():
        q_values = policy_net(torch.tensor(state, dtype=torch.float32).unsqueeze(0))
    action_values = q_values.numpy()[0]
    actions = np.arange(len(action_values))

    plt.bar(actions, action_values)
    plt.title("Optimal Q-values for each action")
    plt.ylabel("Q-value")
    plt.xlabel("Action")
    plt.show()

    # fig = go.Figure(data=[go.Bar(x=actions, y=action_values)])
    # fig.update_layout(title='Q-values Predicted by Policy Network',
    #                   xaxis_title='Action',
    #                   yaxis_title='Q-value')
    # fig.show()

# # Function to compare parameters of policy network and target network
# def compare_networks(policy_net, target_net):
#     policy_params = np.concatenate([param.data.numpy().flatten() for param in policy_net.parameters()])
#     target_params = np.concatenate([param.data.numpy().flatten() for param in target_net.parameters()])

#     fig = go.Figure()
#     fig.add_trace(go.Scatter(y=policy_params, mode='lines', name='Policy Network', line=dict(color='blue', width=2), opacity=1))
#     fig.add_trace(go.Scatter(y=target_params, mode='lines', name='Target Network', line=dict(color='red', width=2), opacity=0.5))
#     fig.add_trace(go.Scatter(y=target_params, mode='lines', name='Target Network', line=dict(color='red', width=2), opacity=0.5))

#     # Add vertical lines to indicate the separation between different layers
#     layer_sizes = [state_size] + hidden_sizes + [action_size]
#     param_count = 0
#     for size in layer_sizes[:-1]:
#         param_count += size * layer_sizes[layer_sizes.index(size) + 1]
#         fig.add_shape(type='line',
#                       x0=param_count, y0=policy_params.min(),
#                       x1=param_count, y1=policy_params.max(),
#                       line=dict(color='gray', width=1))

#     fig.update_layout(title='Comparison of Policy and Target Networks Parameters',
#                       xaxis_title='Parameter Index',
#                       yaxis_title='Parameter Value',
#                       legend=dict(x=0.8, y=0.9, bgcolor='rgba(255, 255, 255, 0.8)'))
#     fig.show()




# Set up the environment
env = gym.make('MountainCar-v0')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n

# Reward Function
def reward_function(state, next_state):
    position = state[0]
    next_position = next_state[0]
    velocity = state[1]
    next_velocity = next_state[1]

    # Check if the episode is done
    if next_position >= 0.5:
        return 100  # Large positive reward for reaching the target position

    # Reward proportional to the change in position towards the target
    reward = (next_position - position) * 10

    # Additional reward for maintaining positive velocity
    if next_velocity > 0:
        reward += 1

    return reward

# Hyperparameters
batch_size = 64
gamma = 0.8
epsilon_start = 1.0
epsilon_end = 0.01
epsilon_decay = 0.95
# epsilon = 0.9 # Only run if using linear epsilon decay
target_update_freq = 10
learning_rate = 0.01
memory_size = 10000
hidden_sizes = [100, 64]
num_episodes = 100
print_every = 10

# Create the Double DQN network and target network
policy_net = DoubleDQN(state_size, action_size, hidden_sizes)
target_net = DoubleDQN(state_size, action_size, hidden_sizes)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

# Define the optimizer
optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)

# Define the replay memory
# https://docs.python.org/3/library/collections.html
memory = deque(maxlen=memory_size)

# Define the epsilon-greedy policy
def epsilon_greedy_policy(state, epsilon):
    if random.random() > epsilon:
        with torch.no_grad():
            return policy_net(state).argmax(dim=1).item()
    else:
        return random.randrange(action_size)

# Initialize lists to store scores and episode numbers
scores = []
```

```
episode_numbers = []

# Train the Double DQN agent
for episode in range(num_episodes):
    state = env.reset()

    # Epsilon decay stratergy
    epsilon = epsilon_end + (epsilon_start - epsilon_end) * np.exp(-1. * episode / epsilon_decay) # Non-linear decay
    # epsilon *= epsilon_decay # Linear decay

    done = False
    score = 0
    while not done:
        action = epsilon_greedy_policy(torch.tensor(state, dtype=torch.float32).unsqueeze(0), epsilon)
        next_state, _, done, _ = env.step(action)
        reward = reward_function(state, next_state)
        memory.append((state, action, reward, next_state, done))
        state = next_state
        score += reward

        if len(memory) >= batch_size:
            batch = random.sample(memory, batch_size)
            states, actions, rewards, next_states, dones = zip(*batch)
            states = torch.tensor(states, dtype=torch.float32)
            actions = torch.tensor(actions, dtype=torch.long).unsqueeze(1)
            rewards = torch.tensor(rewards, dtype=torch.float32).unsqueeze(1)
            next_states = torch.tensor(next_states, dtype=torch.float32)
            dones = torch.tensor(dones, dtype=torch.float32).unsqueeze(1)

            q_values = policy_net(states).gather(1, actions)
            next_q_values = target_net(next_states).max(1)[0].unsqueeze(1)
            expected_q_values = rewards + (1 - dones) * gamma * next_q_values

            loss = nn.MSELoss()(q_values, expected_q_values)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    if episode % target_update_freq == 0:
        target_net.load_state_dict(policy_net.state_dict())

    scores.append(score)
    episode_numbers.append(episode)

    # Print episode number, epsilon, and score after every 'print_every' episodes
    if (episode + 1) % print_every == 0:
        print(f"Episode: {episode+1}, Epsilon: {epsilon:.4f}, Score: {score:.2f}")

# Visualize Q-values for a sample state
sample_state = env.observation_space.sample()
visualize_q_values(policy_net, sample_state)

# # Compare parameters of policy network and target network
# compare_networks(policy_net, target_net)

# # Plot the scores against episode numbers using Plotly
# fig = go.Figure(data=go.Scatter(x=episode_numbers, y=scores, mode='lines'))
# fig.update_layout(title='Double DQN - Score vs Episode',
#                   xaxis_title='Episode',
#                   yaxis_title='Score')
# fig.show()

# Calculate rolling average
window_size = 10  # Adjust the window size as needed
rolling_avg_scores = np.convolve(scores, np.ones(window_size)/window_size, mode='valid')

# # Plot the scores and rolling average against episode numbers using Plotly
# fig = go.Figure()
# fig.add_trace(go.Scatter(x=episode_numbers, y=scores, mode='lines', name='Scores'))
# fig.add_trace(go.Scatter(x=episode_numbers[window_size - 1:], y=rolling_avg_scores, mode='lines', name='Rolling Average'))
# fig.update_layout(title='Double DQN - Score vs Episode with Rolling Average',
#                   xaxis_title='Episode',
#                   yaxis_title='Score',
#                   legend=dict(x=0.8, y=0.9, bgcolor='rgba(255, 255, 255, 0.8)'))
# fig.show()

fig, ax = plt.subplots(1, 2)
ax[0].plot(episode_numbers, scores, label = "Scores", color = "orange")
ax[1].plot(episode_numbers[window_size - 1:], rolling_avg_scores, label = "Rolling Average", color = "cyan")
ax[0].legend()
ax[1].legend()
plt.title("Double DQN - Score vs Episode with Rolling Average")
plt.show()
```

```
# import plotly.graph_objects as go

# fig = go.Figure(layout=dict(title="Double DQN - Score vs Episode with Rolling Average"))

# # Plot scores
# trace1 = go.Scatter(
#     x=episode_numbers,
#     y=scores,
#     mode="lines",
#     name="Scores",
#     line_color="orange",
# )

# # Plot rolling average scores (starting from window_size)
# trace2 = go.Scatter(
#     x=episode_numbers[window_size - 1:],
#     y=rolling_avg_scores,
#     mode="lines",
#     name="Rolling Average",
#     line_color="cyan",
# )

# # Add traces to the figure
# fig.add_trace(trace1)
# fig.add_trace(trace2)

# # Show legend
# fig.update_layout(legend=dict(yanchor="top", y=1.02, xanchor="right", x=1))

# fig.show()


# Close the environment
env.close()
```

```
Episode: 40, Epsilon: 0.0100, Score: 99.18
Episode: 50, Epsilon: 0.0100, Score: 116.47
Episode: 60, Epsilon: 0.0100, Score: 121.37
Episode: 70, Epsilon: 0.0100, Score: 116.98
Episode: 80, Epsilon: 0.0100, Score: 118.99
Episode: 90, Epsilon: 0.0100, Score: 116.83
Episode: 100, Epsilon: 0.0100, Score: 130.93
```





PER DQN

```python
import gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import namedtuple
import matplotlib.pyplot as plt
import plotly.graph_objects as go

random_seed = 42
random.seed(random_seed)
np.random.seed(random_seed)
torch.manual_seed(random_seed)

# https://pytorch.org/rl/stable/reference/generated/torchrl.data.PrioritizedReplayBuffer.html
# https://www.bing.com/search?pglt=43&q=nn.Module&cvid=3e8744507f7d432fb59a2a95abeccad8&gs_lcrp=
# EgZjaHJvbWUyBggAEEUYOTIGCAEQABhAMgYIAhAAGEAyBggDEAAYQDIGCAQQABhAMgYIBRAAGEAyBggGEAAYQDIGCAcQABhAMgYICBBFGDzSAQcxNDhqMGoxqAIAsAIA&FOR

# Define the Prioritized Experience Replay buffer
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta_start=0.4, beta_frames=100000):
        self.capacity = capacity
        self.alpha = alpha
        self.beta_start = beta_start
        self.beta_frames = beta_frames
        self.buffer = []
        self.pos = 0
        self.priorities = np.zeros((capacity,), dtype=np.float32)

    def add(self, state, action, reward, next_state, done):
        max_prio = self.priorities.max() if self.buffer else 1.0
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.pos] = (state, action, reward, next_state, done)
        self.priorities[self.pos] = max_prio
        self.pos = (self.pos + 1) % self.capacity

    def sample(self, batch_size, beta):
        if len(self.buffer) == self.capacity:
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]
        probs = prios ** self.alpha
        probs /= probs.sum()
        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]
        total = len(self.buffer)
        weights = (total * probs[indices]) ** (-beta)
        weights /= weights.max()
        return samples, indices, np.array(weights, dtype=np.float32)

    def update_priorities(self, batch_indices, batch_priorities):
        for idx, prio in zip(batch_indices, batch_priorities):
            self.priorities[idx] = prio

    def __len__(self):
        return len(self.buffer)

# Define the DQN network architecture
class DQN(nn.Module):
    def __init__(self, state_size, action_size, hidden_sizes):
        super(DQN, self).__init__()
        self.hidden1 = nn.Linear(state_size, hidden_sizes[0])
        self.hidden2 = nn.Linear(hidden_sizes[0], hidden_sizes[1])
        # self.hidden3 = nn.Linear(hidden_sizes[1], hidden_sizes[2])
        # self.hidden4 = nn.Linear(hidden_sizes[2], hidden_sizes[3])
        # self.hidden5 = nn.Linear(hidden_sizes[3], hidden_sizes[4])
        self.output = nn.Linear(hidden_sizes[1], action_size)

    def forward(self, state):
        x = torch.relu(self.hidden1(state))
        x = torch.relu(self.hidden2(x))
        # x = torch.relu(self.hidden3(x))
        # x = torch.relu(self.hidden4(x))
        # x = torch.relu(self.hidden5(x))
        return self.output(x)

# Reward Function
def reward_function(state, next_state):
    position = state[0]
    next_position = next_state[0]
```

```
    velocity = state[1]
    next_velocity = next_state[1]

    # Check if the episode is done
    if next_position >= 0.5:
        return 100  # Large positive reward for reaching the target position

    # Reward proportional to the change in position towards the target
    reward = (next_position - position) * 10

    # Additional reward for maintaining positive velocity
    if next_velocity > 0:
        reward += 1

    return reward

# Function to visualize Q-values predicted by the policy network
def visualize_q_values(policy_net, state):
    with torch.no_grad():
        q_values = policy_net(torch.tensor(state, dtype=torch.float32).unsqueeze(0))
    action_values = q_values.numpy()[0]
    actions = np.arange(len(action_values))
    plt.bar(actions, action_values)
    plt.title("Optimal Q-values for each action")
    plt.ylabel("Q-value")
    plt.xlabel("Action")

    # fig = go.Figure(data=[go.Bar(x=actions, y=action_values)])
    # fig.update_layout(title='Q-values Predicted by Policy Network',
    #                   xaxis_title='Action',
    #                   yaxis_title='Q-value')
    # fig.show()

# # Function to compare parameters of policy network and target network
# def compare_networks(policy_net, target_net):
#     policy_params = np.concatenate([param.data.numpy().flatten() for param in policy_net.parameters()])
#     target_params = np.concatenate([param.data.numpy().flatten() for param in target_net.parameters()])

# #     fig = go.Figure()
# #     fig.add_trace(go.Scatter(y=policy_params, mode='lines', name='Policy Network', line=dict(color='rgba(0, 0, 255, 0.8)', width=2
# #     fig.add_trace(go.Scatter(y=target_params, mode='lines', name='Target Network', line=dict(color='rgba(255, 0, 0, 0.6)', width=2
#     fig = go.Figure()
#     fig.add_trace(go.Scatter(y=policy_params, mode='lines', name='Policy Network', line=dict(color='blue', width=2), opacity=1))
#     fig.add_trace(go.Scatter(y=target_params, mode='lines', name='Target Network', line=dict(color='red', width=2), opacity=0.3))

#     # Add vertical lines to indicate the separation between different layers
#     layer_sizes = [state_size] + hidden_sizes + [action_size]
#     param_count = 0
#     for size in layer_sizes[:-1]:
#         param_count += size * layer_sizes[layer_sizes.index(size) + 1]
#         fig.add_shape(type='line',
#                       x0=param_count, y0=policy_params.min(),
#                       x1=param_count, y1=policy_params.max(),
#                       line=dict(color='gray', width=1, dash='dot'))

#     fig.update_layout(title='Comparison of Policy and Target Networks Parameters',
#                       xaxis_title='Parameter Index',
#                       yaxis_title='Parameter Value',
#                       legend=dict(x=0.8, y=0.9, bgcolor='rgba(255, 255, 255, 0.8)'))
#     fig.show()


# Set up the environment
env = gym.make('MountainCar-v0')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n

# Hyperparameters
batch_size = 64
gamma = 0.8
learning_rate = 0.0001
target_update_freq = 10
epsilon_start = 1.0
epsilon_end = 0.01
epsilon_decay = 0.85
# epsilon = 0.9 # Use for linear decay rate
memory_size = 10000
alpha = 0.4
beta_start = 0.6
beta_frames = 50000
hidden_sizes = [100, 64]
num_episodes = 100
```

```python
print_every = 10

# Create the PER buffer and DQN network
memory = PrioritizedReplayBuffer(memory_size, alpha)
policy_net = DQN(state_size, action_size, hidden_sizes)
target_net = DQN(state_size, action_size, hidden_sizes)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

# Define the optimizer
optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)

# Define the loss function
def loss_fn(batch_states, batch_actions, batch_rewards, batch_next_states, batch_dones, batch_weights):
    batch_q_values = policy_net(batch_states).gather(1, batch_actions)
    batch_next_q_values = target_net(batch_next_states).max(1)[0].unsqueeze(1).detach()
    batch_expected_q_values = batch_rewards + (1 - batch_dones) * gamma * batch_next_q_values
    batch_weights_tensor = torch.FloatTensor(batch_weights).unsqueeze(1)
    loss = (batch_weights_tensor * (batch_q_values - batch_expected_q_values) ** 2).mean()
    return loss, batch_q_values, batch_expected_q_values

# Training loop
scores = []
episode_numbers = []
for episode in range(num_episodes):
    state = env.reset()
    score = 0
    done = False
    beta = min(1.0, beta_start + episode * (1.0 - beta_start) / beta_frames)

    # Epsilon decay rate stratergy
    epsilon = epsilon_end + (epsilon_start - epsilon_end) * np.exp(-1. * episode / epsilon_decay) # Non-lienar decay
    # epsilon *= epsilon_decay # Linear decay

    while not done:
        if random.random() < epsilon:
            action = env.action_space.sample()
        else:
            with torch.no_grad():
                state_tensor = torch.FloatTensor(state).unsqueeze(0)
                q_values = policy_net(state_tensor)
                action = q_values.argmax().item()

        next_state, _, done, _ = env.step(action)
        reward = reward_function(state, next_state)
        memory.add(state, action, reward, next_state, done)
        state = next_state
        score += reward

        if len(memory) >= batch_size:
            batch, batch_indices, batch_weights = memory.sample(batch_size, beta)
            batch_states, batch_actions, batch_rewards, batch_next_states, batch_dones = zip(*batch)
            batch_states = torch.FloatTensor(batch_states)
            batch_actions = torch.LongTensor(batch_actions).unsqueeze(1)
            batch_rewards = torch.FloatTensor(batch_rewards).unsqueeze(1)
            batch_next_states = torch.FloatTensor(batch_next_states)
            batch_dones = torch.FloatTensor(batch_dones).unsqueeze(1)

            loss, batch_q_values, batch_expected_q_values = loss_fn(batch_states, batch_actions, batch_rewards, batch_next_states, bat
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            batch_priorities = (torch.abs(batch_q_values - batch_expected_q_values) + 1e-5).squeeze().detach().cpu().numpy()
            memory.update_priorities(batch_indices, batch_priorities)

    if episode % target_update_freq == 0:
        target_net.load_state_dict(policy_net.state_dict())

    scores.append(score)
    episode_numbers.append(episode)

    if (episode + 1) % print_every == 0:
        avg_score = np.mean(scores[-print_every:])
        print(f"Episode: {episode+1}, Average Score: {avg_score:.2f}, Epsilon: {epsilon:.2f}")

# Visualize Q-values for a sample state
sample_state = env.observation_space.sample()
```

Referenced from: https://github.com/radical-p/Breakout_Deep_Q_Reinforcement_Learning/blob/main/Breakout_DQN.py

```
# !pip install stable-baselines3[extra] pyvirtualdisplay gym[atari] pyglet
# !pip install atari-py==0.2.5
# !apt-get install unrar

# !wget http://www.atarimania.com/roms/Roms.rar
# !unrar x Roms.rar
# !unzip ROMS.zip
# !python -m atari_py.import_roms ROMS

# import os

# os.system('apt-get update')
# os.system('apt-get install -y xvfb')
# os.system('wget https://raw.githubusercontent.com/yandexdataschool/Practical_DL/fall18/xvfb -O ../xvfb')
# os.system('apt-get install -y python-opengl ffmpeg')
# os.system('pip install pyglet==1.5.0')

# os.system('python -m pip install -U pygame --user')

# prefix = 'https://raw.githubusercontent.com/yandexdataschool/Practical_RL/master/week04_approx_rl/'

# os.system('wget ' + prefix + 'atari_wrappers.py')
# os.system('wget ' + prefix + 'utils.py')
# os.system('wget ' + prefix + 'replay_buffer.py')
# os.system('wget ' + prefix + 'framebuffer.py')

# # print('setup complete')

# # XVFB will be launched if you run on a server
# import os
# if type(os.environ.get("DISPLAY")) is not str or len(os.environ.get("DISPLAY")) == 0:
#     !bash ../xvfb start
#     os.environ['DISPLAY'] = ':1'


import random
import numpy as np
import pandas as pd
import gym
import time
from collections import deque
from keras import optimizers
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
import tensorflow as tf


class DQN:
    def __init__(self, env):
        # Initialize the DQN agent
        self.env = env
        self.memory = deque(maxlen=4000)  # Replay memory (Origianl: 400000)
        self.gamma = 0.9  # Discount factor for future rewards
        self.epsilon = .9  # Exploration rate
        self.epsilon_min = 0.01  # Minimum exploration rate
        self.epsilon_decay = self.epsilon_min / 200000  # Decay rate for exploration

        self.batch_size = 16  # Batch size for training
        self.train_start = 100  # Number of experiences required before starting training
        self.state_size = self.env.observation_space.shape[0] * 4  # Size of the state vector
        self.action_size = self.env.action_space.n  # Number of possible actions
        self.learning_rate = 0.0001  # Learning rate for the optimizer

        self.evaluation_model = self.create_model()  # Neural network for evaluation
        self.target_model = self.create_model()  # Neural network as a target for stable training

    def create_model(self):
        # Create a neural network model for the DQN
        model = Sequential()
        model.add(Dense(128 * 2, input_dim=self.state_size, activation='relu'))
        model.add(Dense(128 * 2, activation='relu'))
        # model.add(Dense(128 * 2, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mean_squared_error', optimizer=tf.keras.optimizers.legacy.RMSprop(lr=self.learning_rate, decay=0.99, epsilon=
        return model

    def choose_action(self, state, steps):
        # Choose an action using epsilon-greedy exploration strategy
        if steps > 50000:
            if self.epsilon > self.epsilon_min:
                self.epsilon -= self.epsilon_decay
        if np.random.random() < self.epsilon:
            return self.env.action_space.sample()
        return np.argmax(self.evaluation_model.predict(state)[0])

    def remember(self, cur_state, action, reward, new_state, done):
        # Store the experience in the replay memory
        if not hasattr(self, 'memory_counter'):
            self.memory_counter = 0
        transition = (cur_state, action, reward, new_state, done)
        self.memory.extend([transition])
        self.memory_counter += 1

    def replay(self):
        # Train the DQN by replaying experiences from the replay memory
        if len(self.memory) < self.train_start:
            return
        mini_batch = random.sample(self.memory, self.batch_size)
        update_input = np.zeros((self.batch_size, self.state_size))
        update_target = np.zeros((self.batch_size, self.action_size))

        for i in range(self.batch_size):
            state, action, reward, new_state, done = mini_batch[i]
            target = self.evaluation_model.predict(state)[0]
            if done:
                target[action] = reward
            else:
                target[action] = reward + self.gamma * np.amax(self.target_model.predict(new_state)[0])
            update_input[i] = state
            update_target[i] = target

        self.evaluation_model.fit(update_input, update_target, batch_size=self.batch_size, epochs=1, verbose=0)

    def target_train(self):
        # Update the target model with the weights of the evaluation model
        self.target_model.set_weights(self.evaluation_model.get_weights())
        return

    def visualize(self, reward, episode):
        # Visualize the average reward per episode
        plt.plot(episode, reward, 'ob-')
        plt.title('Average reward vs episode')
        plt.ylabel('Reward')
        plt.xlabel('Episodes')
        plt.grid()
        plt.show()

    def transform(self, state):
        # Transform the state representation if necessary
        if state.shape[1] == 512:
            return state
        a = [np.binary_repr(x, width=8) for x in state[0]]
        res = []
        for x in a:
            res.extend([x[:2], x[2:4], x[4:6], x[6:]])
        res = [int(x, 2) for x in res]
        return np.array(res)


def main():
    # Initialize the environment
    env = gym.make('Breakout-ram-v0')
    env = env.unwrapped

    # Print environment information
    print(env.action_space)
    print(env.observation_space.shape[0])
    print(env.observation_space.high)
    print(env.observation_space.low)

    episodes = 250
    trial_len = 10

    tmp_reward = 0
    sum_rewards = 0
    total_steps = 0
    graph_reward = []
    graph_episodes = []
    time_record = []

    dqn_agent = DQN(env=env)

    for i_episode in range(episodes):
        start_time = time.time()
        total_reward = 0
        cur_state_tuple = env.reset()
        cur_state = np.array(cur_state_tuple[0]).reshape(1, 128)
        cur_state = dqn_agent.transform(cur_state).reshape(1, 128 * 4) / 4

        for _ in range(trial_len):
            # Choose action, take a step in the environment
            action = dqn_agent.choose_action(cur_state, total_steps)
            step_result = env.step(action)
            if len(step_result) == 5:
                new_state, reward, done, _, info = step_result
            else:
                new_state, reward, done, _, info = step_result + (None,) * (4 - len(step_result))

            new_state = new_state.reshape(1, 128)
            new_state = dqn_agent.transform(new_state).reshape(1, 128 * 4) / 4
            total_reward += reward
            sum_rewards += reward
            tmp_reward += reward
            if reward > 0:
                reward = 1  # Testing whether it is good.

            # Store the experience in the replay memory
            dqn_agent.remember(cur_state, action, reward, new_state, done)

            if total_steps > 100:
                if total_steps % 4 == 0:
                    # Train the DQN by replaying experiences from the replay memory
                    dqn_agent.replay()
                if total_steps % 500 == 0:
                    # Update the target model with the weights of the evaluation model
                    dqn_agent.target_train()

            cur_state = new_state
            total_steps += 1
            if done:
                env.reset()
                break

        if (i_episode + 1) % 5 == 0:
            graph_reward.append(sum_rewards / 5)
            graph_episodes.append(i_episode + 1)
            sum_rewards = 0

        end_time = time.time()
        time_record.append(end_time - start_time)
        tmp_reward = 0

    print("Reward: ")
    print(graph_reward)
    print("Episode: ")
    print(graph_episodes)
    print("Average_time: ")
    print(sum(time_record) / 60)
    dqn_agent.visualize(graph_reward, graph_episodes)


main()
```

```
/usr/local/lib/python3.10/dist-packages/gym/envs/registration.py:555: UserWarning
  logger.warn(
Discrete(4)
128
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
/usr/local/lib/python3.10/dist-packages/keras/src/optimizers/legacy/rmsprop.py:14
  super().__init__(name, **kwargs)
```
Streaming output truncated to the last 5000 lines.
```
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 44ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 53ms/step
1/1 [==============================] - 0s 69ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 51ms/step
```

Set up

Code taken from: https://pytorch.org/tutorials/intermediate/reinforcement_ppo.html?highlight=ppo

Another article used for reference: https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe

```
# !pip3 install torchrl
# !pip3 install gym[mujoco]
# !pip3 install tqdm
```

```
from collections import defaultdict

import matplotlib.pyplot as plt
import torch
from tensordict.nn import TensorDictModule
from tensordict.nn.distributions import NormalParamExtractor
from torch import nn
from torchrl.collectors import SyncDataCollector
from torchrl.data.replay_buffers import ReplayBuffer
from torchrl.data.replay_buffers.samplers import SamplerWithoutReplacement
from torchrl.data.replay_buffers.storages import LazyTensorStorage
from torchrl.envs import (Compose, DoubleToFloat, ObservationNorm, StepCounter,
                          TransformedEnv)
from torchrl.envs.libs.gym import GymEnv
from torchrl.envs.utils import check_env_specs, ExplorationType, set_exploration_type
from torchrl.modules import ProbabilisticActor, TanhNormal, ValueOperator
from torchrl.objectives import ClipPPOLoss
from torchrl.objectives.value import GAE
from tqdm import tqdm
```

```
import multiprocessing
is_fork = multiprocessing.get_start_method() == "fork"
device = (
    torch.device(0)
    if torch.cuda.is_available() and not is_fork
    else torch.device("cpu")
)
num_cells = 256  # number of cells in each layer i.e. output dim.
lr = 0.0005
max_grad_norm = 1.0
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
  and should_run_async(code)
```

```
frames_per_batch = 1000
# For a complete training, bring the number of frames up to 1M
total_frames = 100_000
```

```
sub_batch_size = 64  # cardinality of the sub-samples gathered from the current data in the inner loop
num_epochs = 10  # optimization steps per batch of data collected

# Clipping Range: 0.1, 0.2, 0.3
clip_epsilon = (
    0.2  # clip value for PPO loss
)

# Can be thought of as bias-variance trade off
gamma = 0.9 # Discount factor
lmbda = 0.1
entropy_eps = 0.9 # Regularizer (0 to 0.01)
```

```
base_env = GymEnv("InvertedDoublePendulum-v4", device=device)
```

```
env = TransformedEnv(
    base_env,
    Compose(
        # normalize observations
        ObservationNorm(in_keys=["observation"]),
        DoubleToFloat(),
        StepCounter(),
    ),
)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
  and should_run_async(code)
```

```
env.transform[0].init_stats(num_iter=1000, reduce_dim=0, cat_dim=0)


print("normalization constant shape:", env.transform[0].loc.shape)
print("observation_spec:", env.observation_spec)
print("reward_spec:", env.reward_spec)
print("input_spec:", env.input_spec)
print("action_spec (as defined by input_spec):", env.action_spec)
```

```
normalization constant shape: torch.Size([11])
observation_spec: CompositeSpec(
    observation: UnboundedContinuousTensorSpec(
        shape=torch.Size([11]),
        space=None,
        device=cpu,
        dtype=torch.float32,
        domain=continuous),
    step_count: BoundedTensorSpec(
        shape=torch.Size([1]),
        space=ContinuousBox(
            low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
            high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True)),
        device=cpu,
        dtype=torch.int64,
        domain=continuous), device=cpu, shape=torch.Size([]))
reward_spec: UnboundedContinuousTensorSpec(
    shape=torch.Size([1]),
    space=ContinuousBox(
        low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
        high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True)),
    device=cpu,
    dtype=torch.float32,
    domain=continuous)
input_spec: CompositeSpec(
    full_state_spec: CompositeSpec(
        step_count: BoundedTensorSpec(
            shape=torch.Size([1]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True),
                high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, contiguous=True)),
            device=cpu,
            dtype=torch.int64,
            domain=continuous), device=cpu, shape=torch.Size([])),
    full_action_spec: CompositeSpec(
        action: BoundedTensorSpec(
            shape=torch.Size([1]),
            space=ContinuousBox(
                low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
                high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True)),
            device=cpu,
            dtype=torch.float32,
            domain=continuous), device=cpu, shape=torch.Size([])), device=cpu, shape=torch.Size([]))
action_spec (as defined by input_spec): BoundedTensorSpec(
    shape=torch.Size([1]),
    space=ContinuousBox(
        low=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True),
        high=Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, contiguous=True)),
    device=cpu,
    dtype=torch.float32,
    domain=continuous)
```

```
rollout = env.rollout(3)
print("rollout of three steps:", rollout)
print("Shape of the rollout TensorDict:", rollout.batch_size)
```

```
rollout of three steps: TensorDict(
    fields={
        action: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.float32, is_shared=False),
        done: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        next: TensorDict(
            fields={
                done: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
                observation: Tensor(shape=torch.Size([3, 11]), device=cpu, dtype=torch.float32, is_shared=False),
                reward: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.float32, is_shared=False),
                step_count: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.int64, is_shared=False),
                terminated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
                truncated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False)},
            batch_size=torch.Size([3]),
            device=cpu,
            is_shared=False),
        observation: Tensor(shape=torch.Size([3, 11]), device=cpu, dtype=torch.float32, is_shared=False),
        step_count: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.int64, is_shared=False),
        terminated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([3, 1]), device=cpu, dtype=torch.bool, is_shared=False)},
    batch_size=torch.Size([3]),
    device=cpu,
    is_shared=False)
Shape of the rollout TensorDict: torch.Size([3])
```

Policy

```
actor_net = nn.Sequential(
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(2 * env.action_spec.shape[-1], device=device),
    NormalParamExtractor(),
)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/lazy.py:181: UserWarning: Lazy modules are a new feature under heavy devel
  warnings.warn('Lazy modules are a new feature under heavy development '
```

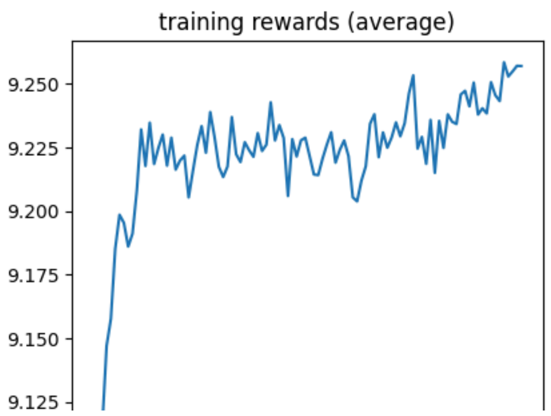
```
policy_module = TensorDictModule(
    actor_net, in_keys=["observation"], out_keys=["loc", "scale"]
)


policy_module = ProbabilisticActor(
    module=policy_module,
    spec=env.action_spec,
    in_keys=["loc", "scale"],
    distribution_class=TanhNormal,
    distribution_kwargs={
        "min": env.action_spec.space.low,
        "max": env.action_spec.space.high,
    },
    return_log_prob=True,
    # we'll need the log-prob for the numerator of the importance weights
)


value_net = nn.Sequential(
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(num_cells, device=device),
    nn.Tanh(),
    nn.LazyLinear(1, device=device),
)

value_module = ValueOperator(
    module=value_net,
    in_keys=["observation"],
)


print("Running policy:", policy_module(env.reset()))
print("Running value:", value_module(env.reset()))
```

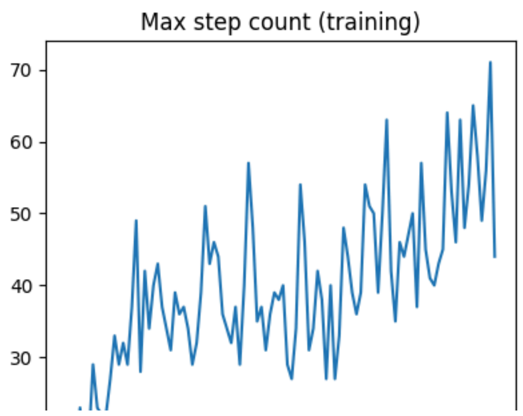
```
Running policy: TensorDict(
    fields={
        action: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
        done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
        loc: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
        observation: Tensor(shape=torch.Size([11]), device=cpu, dtype=torch.float32, is_shared=False),
        sample_log_prob: Tensor(shape=torch.Size([]), device=cpu, dtype=torch.float32, is_shared=False),
        scale: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
        step_count: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, is_shared=False),
        terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
        truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False)},
    batch_size=torch.Size([]),
    device=cpu,
    is_shared=False)
Running value: TensorDict(
    fields={
        done: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
        observation: Tensor(shape=torch.Size([11]), device=cpu, dtype=torch.float32, is_shared=False),
        state_value: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.float32, is_shared=False),
        step_count: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.int64, is_shared=False),
        terminated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False),
            truncated: Tensor(shape=torch.Size([1]), device=cpu, dtype=torch.bool, is_shared=False)},
        batch_size=torch.Size([]),
        device=cpu,
        is_shared=False)


collector = SyncDataCollector(
    env,
    policy_module,
    frames_per_batch=frames_per_batch,
    total_frames=total_frames,
    split_trajs=False,
    device=device,
)


replay_buffer = ReplayBuffer(
    storage=LazyTensorStorage(max_size=frames_per_batch),
    sampler=SamplerWithoutReplacement(),
)


# Generalized Advantage Estimation
advantage_module = GAE(
    gamma=gamma, lmbda=lmbda, value_network=value_module, average_gae=True
)

loss_module = ClipPPOLoss(
    actor_network=policy_module,
    critic_network=value_module,
    clip_epsilon=clip_epsilon,
    entropy_bonus=bool(entropy_eps),
    entropy_coef=entropy_eps,
    # these keys match by default but we set this for completeness
    critic_coef=1.0,
    loss_critic_type="smooth_l1",
)

optim = torch.optim.Adam(loss_module.parameters(), lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optim, total_frames // frames_per_batch, 0.0
)
```

```
logs = defaultdict(list)
pbar = tqdm(total=total_frames)
eval_str = ""

# We iterate over the collector until it reaches the total number of frames it was
# designed to collect:
for i, tensordict_data in enumerate(collector):
    # we now have a batch of data to work with. Let's learn something from it.
    for _ in range(num_epochs):
        # We'll need an "advantage" signal to make PPO work.
        # We re-compute it at each epoch as its value depends on the value
        # network which is updated in the inner loop.
        advantage_module(tensordict_data)
        data_view = tensordict_data.reshape(-1)
        replay_buffer.extend(data_view.cpu())
        for _ in range(frames_per_batch // sub_batch_size):
            subdata = replay_buffer.sample(sub_batch_size)
            loss_vals = loss_module(subdata.to(device))
            loss_value = (
                loss_vals["loss_objective"]
                + loss_vals["loss_critic"]
                + loss_vals["loss_entropy"]
            )

            # Optimization: backward, grad clipping and optimization step
            loss_value.backward()
            # this is not strictly mandatory but it's good practice to keep
            # your gradient norm bounded
            torch.nn.utils.clip_grad_norm_(loss_module.parameters(), max_grad_norm)
            optim.step()
            optim.zero_grad()

    logs["reward"].append(tensordict_data["next", "reward"].mean().item())
    pbar.update(tensordict_data.numel())
    cum_reward_str = (
        f"average reward={logs['reward'][-1]: 4.4f} (init={logs['reward'][0]: 4.4f})"
    )
    logs["step_count"].append(tensordict_data["step_count"].max().item())
    stepcount_str = f"step count (max): {logs['step_count'][-1]}"
    logs["lr"].append(optim.param_groups[0]["lr"])
    lr_str = f"lr policy: {logs['lr'][-1]: 4.4f}"
    if i % 10 == 0:
        # We evaluate the policy once every 10 batches of data.
        # Evaluation is rather simple: execute the policy without exploration
        # (take the expected value of the action distribution) for a given
        # number of steps (1000, which is our ``env`` horizon).
        # The ``rollout`` method of the ``env`` can take a policy as argument:
        # it will then execute this policy at each step.
        with set_exploration_type(ExplorationType.MEAN), torch.no_grad():
            # execute a rollout with the trained policy
            eval_rollout = env.rollout(1000, policy_module)
            logs["eval reward"].append(eval_rollout["next", "reward"].mean().item())
            logs["eval reward (sum)"].append(
                eval_rollout["next", "reward"].sum().item()
            )
            logs["eval step_count"].append(eval_rollout["step_count"].max().item())
            eval_str = (
                f"eval cumulative reward: {logs['eval reward (sum)'][-1]: 4.4f} "
                f"(init: {logs['eval reward (sum)'][0]: 4.4f}), "
                f"eval step-count: {logs['eval step_count'][-1]}"
            )
            del eval_rollout
    pbar.set_description(", ".join([eval_str, cum_reward_str, stepcount_str, lr_str]))

    # We're also using a learning rate scheduler. Like the gradient clipping,
    # this is a nice-to-have but nothing necessary for PPO to work.
    scheduler.step()
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_
  and should_run_async(code)
eval cumulative reward:  278.8475 (init:  101.3024), eval step-count: 29, average reward= 9.2570 (init= 9.0916), step count (max):
```

```
plt.figure(figsize=(10, 10))
plt.subplot(2, 2, 1)
plt.plot(logs["reward"])
plt.title("training rewards (average)")
plt.subplot(2, 2, 2)
plt.plot(logs["step_count"])
plt.title("Max step count (training)")
plt.subplot(2, 2, 3)
plt.plot(logs["eval reward (sum)"])
plt.title("Return (test)")
plt.subplot(2, 2, 4)
plt.plot(logs["eval step_count"])
plt.title("Max step count (test)")
plt.show()
```


training rewards (average)


Max step count (training)