Reinforcement learning algorithm for gymnasium environment

Gabriele Lerani, ID: 2093689

June 27, 2023

I Introduction

This paper explores and compares two different reinforcement learning techniques for solving a simple/medium gymnasium environment as MountainCar. The first approach uses a Q-table to store the Q values for the best policy, the second one approximates the Q-function with a deep neural network. In this second attempt many network architectures have been tested before finding the one that gives a good estimation of the optimal policy. In Section II there is a short description of the gym environment, Section III provides implementation details for the Q-table approach and the deep Q-network model. Section IV discusses critical aspects faced during the development.

II Gymnasium environment

1 Description

As reported on gym official site Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill.

2 Observation space

The observation is a ndarray with shape $(2,)$:

- position of the car along the x-axis $[-1.2, 0.6]$
- velocity of the car $[-0.07, 0.07]$

3 Action space

There are 3 discrete deterministic actions:

- 0: accelerate to the left
- 1: don't accelerate
- 2: accelerate to the right

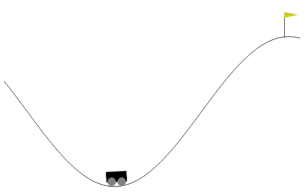


Figure 1: image of the environment.

4 Reward and termination

The agent is penalised with -1 for each timestep and the episode end when either the car reaches the goal position or more than 200 timestep have passed (and the reward is -200). Therefore it is important to observe that the agent's reward only depends on how long it takes before the episode is finished or truncated even if it reaches a position close to the flag, this can slow down the learning process, because an episode with a reward of -200 and with a position of 0.49 has the same reward of an episode truncated with a position of -1. I found online that the environment can be considererd solved if after 100 episodes the average reward is -110, but the documentation doesn't specify nothing about that.

III Q - learning

The main concept of the reinforcement learning is represented by the Q-value that is updated during training with the following general rule:

$$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right)$$

- α: learning rate, it determines how much new experiences override past experiences
- γ: discount factor, it determines weights of future reward
- $r+\gamma\cdot\max_{a'} Q(s',a')$: the immediate reward plus the reward you get if you go on state s' executing action a'

The Q-function maps a state-action pair to a value, representing the expected cumulative reward the agent can achieve by starting in that state, taking the specified action, and following a particular policy thereafter. The policy dictates the agent's behavior, determining which actions to select in different states. The goal for the agent is to make decisions that maximize its expected cumulative reward over time and find an optimal policy that given a state returns the best possible actions. The Q-function can be represented with a table or approximated with a neural network when the state and action space becomes too large.

1 Python libraries

All the work is done in python, the main libraries I used are numpy, tensorflow 2.0, matplotlib and gymnasium. I executed all the code inside google colab environment that provides a virtual machine with 12 GB RAM and 100 GB disk memory.

2 Q-table implementation and results

2.1 Agent initialization and discretization

To realize the Q-learning process through the Q-table I created a class that implements the agent behavior. The last three parameters in the `init()` are used to obtain a discrete representation of the table, indeed in MountainCar every state is a pair (position,velocity) and the number of possible states is potentially infinite. Therefore, attempting to represent the Q-values for every possible state-action pair would require an infinite-sized Q-table, which is not feasible in practice. Thus the idea of discretization is to make neighboring values have the same entry in the table, hence making it fixed size. It is then initialized with random values between -2 and 0.

```
class RLAgent:
    def __init__(
        self,
        learning_rate: float = 0.1,
        initial_epsilon: float = 0.5,
        epsilon_decay: float = 0.00002,
        final_epsilon: float = 0.1,
        action_space: int = 3,
        discount_factor: float = 0.95,
        env: gym.Env = None,
```

```
        env_size: int = 50,

    ):
        """Initialize a Reinforcement Learning agent with an empty dictionary
        of state-action values (q_values), a learning rate and an epsilon.

        Args:
            learning_rate: The learning rate
            initial_epsilon: The initial epsilon value
            epsilon_decay: The decay for epsilon
            final_epsilon: The final epsilon value
            action_space: The number of action for the environment
            discount_factor: The discount factor for computing the Q-value
        """
        self.env = env
        self.actions = action_space
        self.lr = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.training_error = []
        self.discrete_size = [env_size] * len(env.observation_space.high)
        self.discrete_win_size = (env.observation_space.high - env.observation_space.
    low) / self.discrete_size
        self.q_values = np.random.uniform(low=-2, high=0, size=(self.discrete_size + [
    env.action_space.n]))
```

Listing 1: Agent init

The following methods are used to implement the epsilon greedy policy, so as to encourages the agent to explore as much as possible during the first few episodes. I experimented exponential decay but I got better performance using a linear decay as defined in `decay_epsilon(self)`. To obtain the discrete representation of a state `get_discrete_state(self,state)` is used.

```

    def policy(self,state) -> int:
        state = self.get_discrete_state(state)
        return int(np.argmax(self.q_values[state]))


    def get_action(self, obs: np.ndarray) -> int:
        """
        Returns the best action with probability (1 - epsilon)
        otherwise a random action with probability epsilon to ensure exploration.
        """
        greedy = random.random() > self.epsilon

        # exploitation
        if greedy:
            # use the train net to get the action value given a state
            return self.policy(obs)

        # exploration
        else:
             return np.random.choice(self.actions)

    def decay_epsilon(self):
        """ Decay epsilon value by a constant """
        self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)

    def get_discrete_state(self,state):
        """
        Return a discrete representation of the state, this simplify the learning
    process
        by reducing the complexity of the state and allow fixed size q-table
        """
        discrete_state = (state - self.env.observation_space.low) / self.
    discrete_win_size
        return tuple(discrete_state.astype(np.int))
```

Listing 2: epsilon greedy policy

The most important method used during the training is `update`, it basically computes the Q-value given the current state, the next state and the action performed. In addition the agent tracks the temporal difference using a python list and the end it plots its trend during the training.

```
    def update(
        self,
        obs: np.ndarray,
        action: int,
        reward: float,
        terminated: bool,
        next_obs: np.ndarray,
    ):
        """Updates the Q-value of an action."""

        # convert np.ndarray to hashable object
        obs = self.get_discrete_state(obs)
        next_obs = self.get_discrete_state(next_obs)


        # get the future q_value for the current observation
        future_q_value = (not terminated) * np.max(self.q_values[next_obs])

        # get the difference between current q_value and next observation
        temporal_difference = (
            reward + self.discount_factor * future_q_value - self.q_values[obs + (
    action,)]
        )

        # update the q values for the current observation and action
        self.q_values[obs + (action,)] = (
            self.q_values[obs + (action,)] + self.lr * temporal_difference
        )

        # store the training error, the goal is to reduce it
        self.training_error.append(temporal_difference)
```

Listing 3: update q-table

These functions are used to plot the average reward and the policy adopted during the training loop.

```
    def plot_stats(self,env):
        rolling_length = 500
        fig, axs = plt.subplots(ncols=2, figsize=(12, 5))
        axs[0].set_title("Episode rewards")
        # compute and assign a rolling average of the data to provide a smoother graph
        reward_moving_average = (
            np.convolve(
                np.array(env.return_queue).flatten(), np.ones(rolling_length), mode="
    valid"
            )
            / rolling_length
        )
        axs[0].plot(range(len(reward_moving_average)), reward_moving_average)

        axs[1].set_title("Training Error")
        training_error_moving_average = (
            np.convolve(np.array(self.training_error), np.ones(rolling_length), mode="
    same")
            / rolling_length
        )
        axs[1].plot(range(len(training_error_moving_average)),
    training_error_moving_average)
        plt.tight_layout()
        plt.show()

    def plot_policy(self,actions):
        temp_action_x = list(actions.keys())

        action_labels = {0: "left", 1: "stay", 2: "right"}
        action_x = [action_labels[a] for a in temp_action_x]
        action_y = list(actions.values())
```

```
        colors = ['blue', 'green', 'orange']

        fig, ax = plt.subplots()
        ax.bar(action_x, action_y, color=colors)
        ax.set_ylabel('Ocurrences')
        ax.set_title('Actions')
        ax.legend(title='Actions policy')

        plt.show()
```

Listing 4: helper functions to plot reward temporal difference and the policy used

The following are used to periodically save the qtable and to load it after the training loop in order to test the result achieved.

```
    def save_qtable(
        self,
        model_name: str,
        episode: int,
    ):
        """
        Saving Q-Tables
        """
        np.save(f'./q_tables/{model_name}/{episode}-qtable.npy', self.q_values)

    def load_qtable(
        self,
        model_name: str,
        episode: str
    ):
        """
        Load Q-Table
        """

        self.q_values = np.load(f'./q_tables/{model_name}/{episode}-qtable.npy')
```

Listing 5: save and load

2.2 Training loop

The training loop is performed for 90000 episodes, the Mountain car environment is wrapped into `wrappers.RecordEispdeStatistics` that keep track of cumulative rewards and episode lengths and it is useful for plotting the reward. Then the average reward of the last 100 episodes is printed every 200 episode, and every 10000 the Q-table is saved. During the loop the agent plays an episode and chooses an action according to the epsilon greedy policy, then it updates the Q-table and decays epsilon. As soon as the value of epsilon decreases the agent starts using the Q-table to choose the next action and the reward will increase. At the end the agent will plot the graphs.

```
EPISODES = 90_000                   # number of episode to play
UPGRADE_STEP =10_000                # frequency of target network upgrade
GAMMA = 0.95                        # discount factor
LR = 0.1                            # q-table learning rate
EPS_MAX = 0.5                       # Initial exploration probability
EPS_MIN = 0.1                       # Final exploration probability
DECAY = EPS_MAX / (EPISODES - 1)    # decay factor
TABLE_SIZE = 50                     # used to compute the size of the q table


env = gym.make('MountainCar-v0')
agent = RLAgent(
    learning_rate=LR,
    initial_epsilon=EPS_MAX,
    epsilon_decay=DECAY,
    final_epsilon=EPS_MIN,
    action_space=env.action_space.n,
    discount_factor=GAMMA,
    env=env,
```

```
    env_size=TABLE_SIZE,
)

reward_sum = 0
time_scores = deque(maxlen=100)
env = wrappers.RecordEpisodeStatistics(env, deque_size=EPISODES)
for episode in range(1,EPISODES + 1):
    obs, info = env.reset()
    done = False

    step = 1
    # play one episode
    while not done:

        # get an action according to epsilon greedy policy
        action = agent.get_action(obs)

        # execute the action
        next_obs, reward, terminated, truncated, info = env.step(action)

        # update the agent
        agent.update(obs, action, reward, terminated, next_obs)

        # update if the environment is done and the current obs
        done = terminated or truncated

        # upgrade to next obs
        obs = next_obs

        if done:

            # store current time for that episode
            time_scores.append(step)

            # compute avg score
            scores_avg = np.mean(time_scores) * -1

            if episode % 200 == 0:
                print(f"Episode {episode}/{EPISODES}, e {agent.epsilon:.6f}, avg
    reward {scores_avg:.2f}, state {next_obs}, time {step}")
            break

        # increment step
        step+=1

    # Save progress every UPGRADE STEP
    if episode % UPGRADE_STEP == 0:
        agent.save_qtable(model_name,episode)

    agent.decay_epsilon()

# plot stats
agent.plot_stats(env)
```

Listing 6: training loop

2.3 Results and hyperparameters tuning

To test the learned policy I made the following script that loads the last q-table from the directory where it is stored and it plays 2000 games. To verify the effectiveness of the policy, I considered the average reward obtained during the episodes and the accuracy, calculated as the ratio between the number of episodes won and the total number of episodes

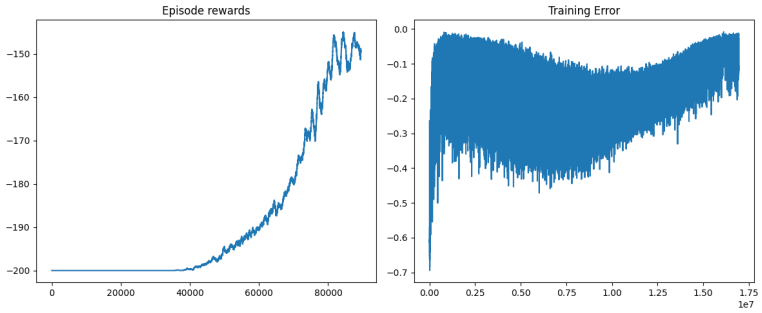
```
def training_result(model_name, episode):

  env = gym.make('MountainCar-v0')

  total_reward = 0.0
  win = 0
  episodes = 2000
```

```
  actions = {0:0,
             1:0,
             2:0}

  agent = RLAgent(env=env)
  agent.load_qtable(model_name, episode)

  for i in tqdm(range(episodes)): # Play 10 episode and take the average
    state, _ = env.reset()
    done = False
    truncated = False
    episode_reward = 0.0
    while not (done or truncated):
      action = agent.policy(state)

      next_state, reward, done,truncated, info = env.step(action)

      # increment action
      actions[action] += 1


      # Count number of win
      if next_state[0] >= 0.5:
        win += 1

      episode_reward += reward
      state = next_state

    if i % 20 == 0:
        print(f"{i}/{episodes}")

    total_reward += episode_reward

  average_reward = total_reward / episodes
  accuracy = win / episodes

  print(f"Average reward: {average_reward}, Accuracy {accuracy:.4f}")
  agent.plot_policy(actions)

if __name__=="__main__":


    model_name = f'LR: 0.1 - DISCOUNT: 0.95 -' \
                 f' EPISODES: 90000'\
                 f' EPSILON: 0.5'\
                 f' TABLE_SIZE: 50'

    episode = "90000"

    training_result(model_name,episode)
```

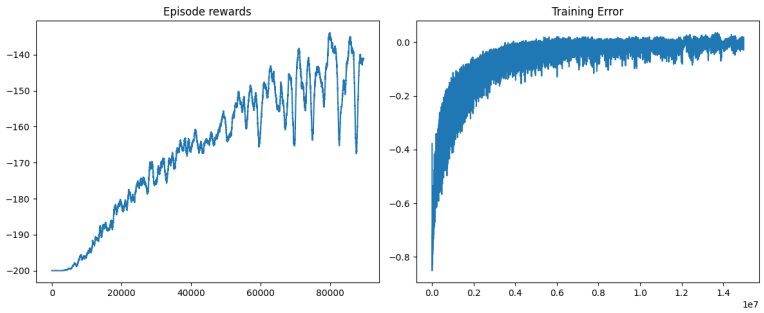
Listing 7: testing

Figure 2(a) shows the execution of the training loop with starting epsilon of 1.0, it is evident that the agent converges earlier with a smaller epsilon of 0.5 as in Figure 2(b) and (c).
I tested different env size that determines different dimensions of the Q-table, in particular for an `env_size` size of 50 the Q-table has a shape of (50,50,3), instead with an `env_size` of 100 the shape is (100,100,3). Therefore we expect that the first Q-table requires lesser time than the second one, this behavior is shown in figure 2. So as expected, in the first solution the average reward increases rapidly and after the first 20,000 episodes it is already around -180 but then it is not stable and fluctuates between -165 and -135 during the last 20,000 episodes. Instead in the second solution it takes more time to raise the average reward but in the end it is more stable and does not vary much with respect to the average value. Using the `training_result()` I played 2000 episodes with the Q-table obtained after 90000 episodes and env size of 100 and it gives an average reward of -120.539 and an accuracy of 1.00.
Figure 3(a) shows random action policy while (b) shows learned policy after 90000 episodes.
Summarizing, the use of the Q-table allows to solve the environment by exploiting the discretization to reduce the size of the table, surely this can lead to a poor precision and can impact the agent's ability

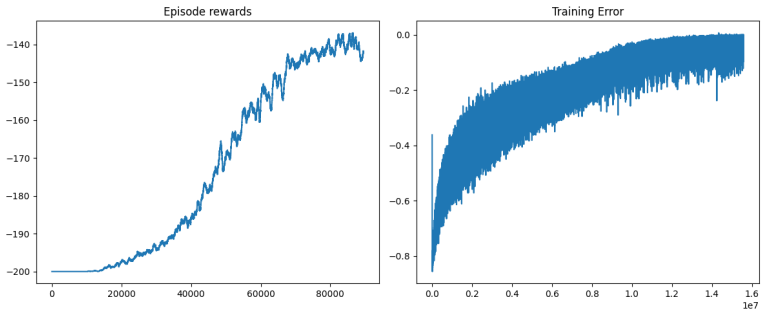
to accurately learn and generalize its Q-values. The following section describes the usage of a neural network to improve the ability of generalization of the Q-values and explains its implementation.



(a) env size = 50, epsilon = 1.0

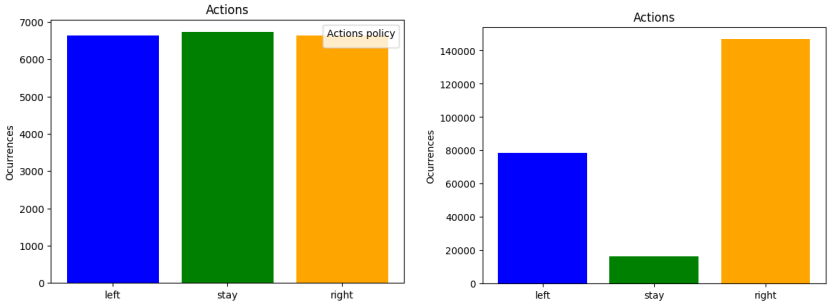


(b) env size = 50, epsilon = 0.5



(c) env size = 100, epsilon = 0.5

Figure 2: Rewards and training error for different env size



(a) Random actions (b) Learned policy

Figure 3: Different policy

3 Deep Q-Learning

The idea of the Q-learning is to use a deep neural network to approximate the Q-function, the training process is similar to the Q-table approach but instead of updating the table we train a neural network. Before reaching a good policy approximation I experimented networks with different layers and depth, also changing hyperparameters and strategy.

3.1 Agent implementation

The following code is used to implement the agent, the network is created with `build_model()` method. During the training I used different learning rate and after many executions I observed that a dynamic learning rate improves model stability and make the algorithm faster. `get_optimizer()` is used for this purpose, it sets `decay_step` to 500 , i.e. more or less after 35 episodes the value of lr decreases of a factor of `decay_rate`. As a loss function I used the mean squared error, relu activation function for every node in the hidden layers and the linear activation function for the output layer, in addition the `kernel_initializer` is set to `he_uniform` that initializes the weights with random numbers drawn from a uniform distribution.

```
class RLAgent:
    def __init__(
        self,
        env: gym.Env,
        initial_epsilon: float,
        epsilon_decay: float,
        final_epsilon: float,
        gamma: float,
        lr: float,
        dropout_rate: float = 0.3
    ):

        self.state_shape = env.observation_space.shape
        self.actions = env.action_space.n
        self.gamma = gamma
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon
        self.lr = lr
        self.train_net = self.build_model()
        self.loss_values = list()
        self.dropout_rate = dropout_rate


    @staticmethod
    def get_optmizer(lr: int):
```

```
        # apply learning rate EsponentialDecay
        lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
            lr,
            decay_steps=500,
            decay_rate=0.96,
            staircase=True
        )

        # define an optimizer with a learning rate schedule to decrease it over time
        opt = tf.optimizers.Adam(learning_rate=lr_schedule)
        return opt


    def build_model(self):
        """
        Builds a deep neural net which predicts the Q values for all possible
        actions given a state. The input should have the shape of the state
        (which is 2 in MountainCar), and the output should have the same shape as
        the action space (which is 2 in MountainCar) since we want 1 Q value per
        possible action.

        :return: the Q network
        """


        model = Sequential()
        model.add(Dense(24, input_shape=self.state_shape, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dropout(0.0))
        model.add(Dense(48, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dropout(0.0))

        model.add(Dense(self.actions, activation='linear',
                        kernel_initializer='he_uniform'))

        model.compile(optimizer=self.get_optmizer(self.lr), loss='mse', metrics=["mse"
    ])
        return model
```

Listing 8: Agent

The epsilon greedy policy is implemented in the same way of the Q-table solution, but now the action is chosen using the trained network. With respect to the Q-table solution here epsilon decrease exponentially because this approach has shown better results.

```
        def policy(self,state) -> int:
        action_q = self.train_net(np.atleast_2d(state))
        return np.argmax(action_q[0], axis=0)

    def get_action(self, obs) -> int:
        """
        Get an action with an epsilon greedy policy
        """
        greedy = random.random() > self.epsilon

        # exploitation
        if greedy:

            # use the train net to get the action value given a state
            return self.policy(obs)

        # exploration
        else:
             return np.random.choice(self.actions)

    def decay_epsilon(self):
        """ Decay epsilon value by a constant"""
        self.epsilon = max(self.final_epsilon, self.epsilon * self.epsilon_decay)
```

Listing 9: Agent

As before the core part of the Agent is implemented in the `train` method that is periodically called during the training to update network's weights. It learns from a batch of episode the Q-value and fit the model with the classic rule $Q(s,a) \leftarrow r + \gamma \cdot \max_{a'} Q(s',a')$ for each state in the batch.

```
    def update_dropout(self):
        # Enable dropout after the specified episode
        for layer in self.train_net.layers:
            if isinstance(layer, Dropout):
                layer.rate = self.dropout_rate


    def train(self, batch):
        """
        Train the network with a batch sample using a train net
        """

        state, next_state, action, reward, terminated = batch

        # get the current q value for that state, it will be a value for both actions
        current_q = self.train_net(state)

        # copy that value of the current q-value into a target variable
        target_q = np.copy(current_q)

        # using the train network get the q-value of the next state
        next_q = self.train_net(next_state)

        # among the q-values returned by the target network select the best
        max_next_q = np.amax(next_q, axis=1)

        for i in range(state.shape[0]):

            target_q[i][action[i]] = reward[i] + self.gamma * (1 - terminated[i]) *
    max_next_q[i]

        # fit the train model
        history = self.train_net.fit(x=state, y=target_q, epochs=1,verbose=0)

        # add to list
        self.loss_values.append(history.history["loss"])

        # return the loss and learning rate
        return round(self.train_net.optimizer.lr.numpy(), 5)
```

Listing 10: Agent

The following methods are used to plot the max, avg, min reward and the policy used

```
    def plot_rew(self,aggr_ep_rewards,model_name):

        plt.plot(aggr_ep_rewards.get('ep'), aggr_ep_rewards.get('avg'), label="avg
    rewards")
        plt.plot(aggr_ep_rewards.get('ep'), aggr_ep_rewards.get('min'), label="min
    rewards")
        plt.plot(aggr_ep_rewards.get('ep'), aggr_ep_rewards.get('max'), label="max
    rewards")
        plt.legend(loc=4)
        plt.xlabel('Episode')
        plt.ylabel('Reward')
        plt.ylim(-200, None)
        plt.show()


    def plot_policy(self,actions):
        temp_action_x = list(actions.keys())

        action_labels = {0: "left", 1: "stay", 2: "right"}
        action_x = [action_labels[a] for a in temp_action_x]
        action_y = list(actions.values())

        colors = ['blue', 'green', 'orange']

```

```
        fig, ax = plt.subplots()
        ax.bar(action_x, action_y, color=colors)
        ax.set_ylabel('Ocurrences')
        ax.set_title('Actions')
        ax.legend(title='Actions policy')

        plt.show()
```

Listing 11: Agent

As before the model is periodically saved in h5 extension and loaded at the end to check its performance.

```
    def save_model(
        self,
        model_name:str,
        episode: int
    ):
        """
        Saveing model
        """
        self.train_net.save(f'./{model_name}/trainNetwork{episode}.h5')

    def load(
        self,
        model_name:str,
        episode: int
    ):
        self.train_net = load_model(f'./{model_name}/trainNetwork{episode}.h5')
```

Listing 12: Agent

3.2 Replay buffer

A common approach in deep Q-learning is to use a replay buffer as a memory location for the samples obtained during episode execution, it enables more efficient use of the samples by reusing them multiple times during training. It allows the agent to extract more information from each sample and learn from diverse experiences without needing to interact with the environment at every training step. In addition it can improve the stability and convergence of RL algorithms, it reduces the variance of updates and prevents the agent from getting stuck in suboptimal states. This stability promotes better convergence and more robust learning. I realized it with a deque with max size of 10000 sample, whenever the structure is full oldest item are removed, this reduces correlations in the observation sequence and smooths changes in the data distribution.

```

class ReplayBuffer:
  def __init__(self,exp_max_size,batch_size):
    self.exp_max_size = exp_max_size
    self.batch_size = batch_size
    self.experiences = deque(maxlen=exp_max_size)

  def get_exp_size(self):
    """
    Get experiences length
    """
    return len(self.experiences)

  def add_experience(self,exp):
    """
    Add new experience to buffer
    """
    # oldest item are automatically removed when dimensione is over max_exp_size
    self.experiences.append(exp)


  def sample_game_batch(self):
    """
    Sample game batch for training loop
    """
```

```
    # take a sample of batch size
    sampled_gameplay_batch = random.sample(self.experiences, self.batch_size)

    # define state, next_state, action ,reward, done
    state_batch, next_state_batch, action_batch, reward_batch, done_batch= [], [], [],
     [], [],

    # for each experience in the batch get a sample
    for gameplay_experience in sampled_gameplay_batch:
      state_batch.append(gameplay_experience[0])
      next_state_batch.append(gameplay_experience[1])
      reward_batch.append(gameplay_experience[2])
      action_batch.append(gameplay_experience[3])
      done_batch.append(gameplay_experience[4])

    return np.array(state_batch), np.array(next_state_batch), np.array(action_batch),
    np.array(reward_batch), np.array(done_batch)
```

Listing 13: Replay buffer

3.3 Training loop

The training loop is the same of the Q-table approach but now when there is enough experience the agent train the network.

```
EPISODE = 4000                          # Number of episode to play
EPISODE_MAX_LENGTH = 200                # This number depends on the environment
SAVE_MODEL_STEP = EPISODE // 8          # Frequency of saving model
DROPOUT = 750                           # Point of insertion of dropout layers
GAMMA = 0.99                            # Discount factor
EXP_MAX_SIZE = 10_000                   # Max batch size of past experience previous 10000
#LR = 0.001251                          # Fixed learning rate
LR = 0.01251                            # Different learning rate
EPS_MAX = 1.0                           # Initial exploration probability
EPS_MIN = 0.001                         # Final exploration probability before: 0,00001
DECAY = 0.85                            # Decay value
BATCH_SIZE = 32                         # Sample to get from experiences # 32 previous
PLOT = 500                              # Frequency of plotting graphs

win = 0
scores = list()

model_name = f'LR: {LR} - GAMMA: {GAMMA} -' \
             f' EPISODES: {EPISODE}'\
             f' EPSILON: {EPS_MAX}'\
             f" BATCH: {BATCH_SIZE}"


# create model directory for storing models
if not os.path.exists(model_name):
    os.makedirs(model_name)


env = gym.make('MountainCar-v0')
agent = RLAgent(
        env=env,
        lr=LR,
        initial_epsilon=EPS_MAX,
        epsilon_decay=DECAY,
        final_epsilon=EPS_MIN,
        gamma=GAMMA,
        )


buffer = ReplayBuffer(
                      exp_max_size=EXP_MAX_SIZE,
                      batch_size=BATCH_SIZE
                      )

time_scores = deque(maxlen=100)
```

```
lr_value = 0

aggr_ep_rewards = {'ep': [], 'avg': [], 'min': [], 'max': []}

for episode_cnt in range(1, EPISODE + 1):
  state, _ = env.reset()
  terminated = False

  # play the game and collect experience
  for step in range(1, EPISODE_MAX_LENGTH + 1):
    action = agent.get_action(state)
    next_state, reward, terminated, truncated, _ = env.step(action)

    # add experience tu the buffer
    buffer.add_experience((state, next_state, reward, action, terminated))

    # agent won't start learning if there isn't enough experience
    if buffer.get_exp_size() > BATCH_SIZE and step % 15 == 0:
        gameplay_experience_batch = buffer.sample_game_batch()
        lr_value = agent.train(gameplay_experience_batch)

    # set state to next state
    state = next_state


    if terminated or truncated:

        # store current time for that episode
        time_scores.append(step * -1)

        # compute avg score of the last 100 episodes
        avg_reward = np.mean(time_scores)
        min_reward = min(time_scores)
        max_reward = max(time_scores)
        aggr_ep_rewards['ep'].append(episode_cnt)
        aggr_ep_rewards['avg'].append(avg_reward)
        aggr_ep_rewards['min'].append(min_reward)
        aggr_ep_rewards['max'].append(max_reward)

        # store avg score
        scores.append(avg_reward)


        print(f"Episode {episode_cnt}/{EPISODE}, e {agent.epsilon:.6f}, avg reward {
    avg_reward:.2f}, time {step}, lr :{lr_value:.6f}")
        break

  if episode_cnt % SAVE_MODEL_STEP == 0:
    agent.save_model(model_name, episode_cnt)

  agent.decay_epsilon()

  # show avarage reward
  if episode_cnt % PLOT == 0:
    agent.plot_rew(aggr_ep_rewards,model_name)
    agent.plot_loss()

  if episode_cnt == DROPOUT:
    agent.update_dropout()
    print("Starting dropout")
```

Listing 14: Training loop

3.4 Network architectures and result

I tested four networks, the output layer is with 3 neurons (1 per action) and is the same for all networks:

(i) 2 layers with 10 neurons

(ii) 1 layer with 20 neurons, 1 hidden layer with 25 neurons

(iii) 1 layer with 20 neurons, 1 hidden layer with 25 neurons, 1 hidden layer with 13 neurons

(iv) 1 layer with 24 neurons, 1 hidden layer with 48 neurons

(v) 1 layer with 24 neurons, 1 hidden layer with 48 neurons with dynamic dropout layer

As shown in Figure 4 scenario (i) is not enough to guarantee the convergence of the model because it learns something during the first 2000 episodes but then the rewards goes back to -200. Figure 6 shows that (iii) doesn't improve agent performance. Figure 5 and 7 show that (ii) (iv) lead to more stability even if the reward varies around -140; in this network the model is able to learn a suboptimal policy but it is not able to do better than this. To address the problem I tried to add a dynamic dropout layer, indeed after 750 episodes the dropout rate of the first and second layer is set to 0.3, it means that 30% of the neurons of each leayer are disabled. Dropout reduces the tendency of the network to rely too heavily on any single neuron, preventing overfitting and improving the model's ability to generalize to new data. Figure 8 shows that the agent outperforms all the other architectures, in particular I tested the network model obtained after 2500 episodes and I got an average reward of -110.007 and an accuracy of 1.00 over an execution of 1000 episodes.

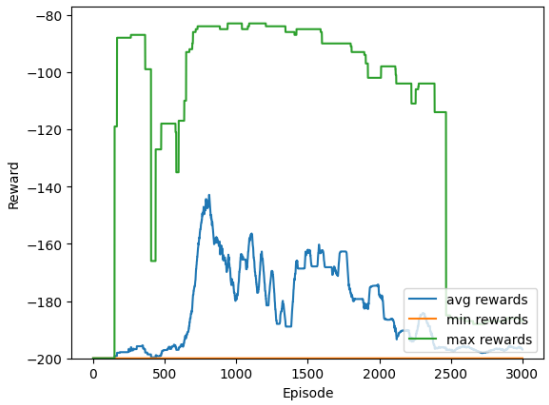


Figure 4: (i)

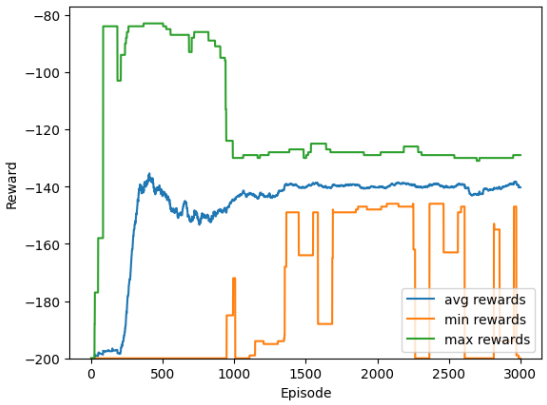


Figure 5: (ii)

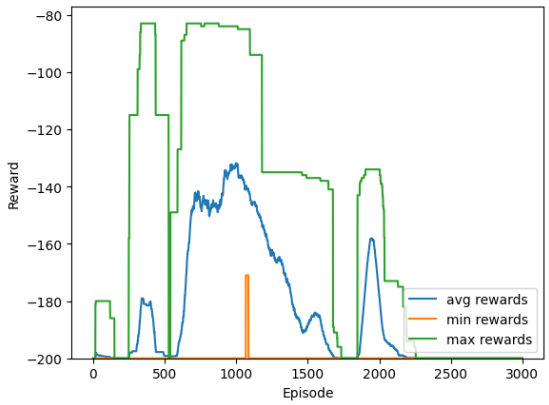


Figure 6: (iii)

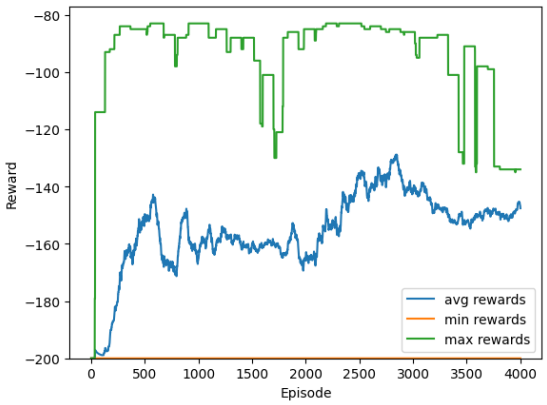


Figure 7: (iv)

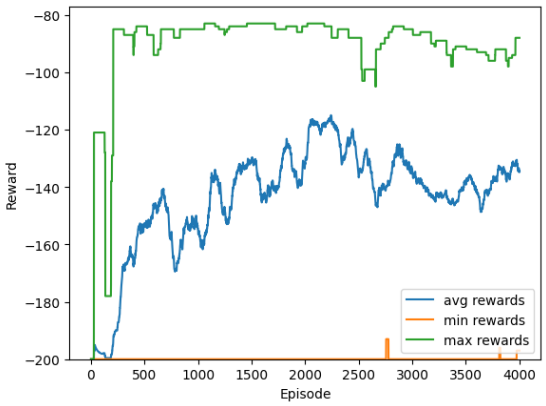


Figure 8: (v)

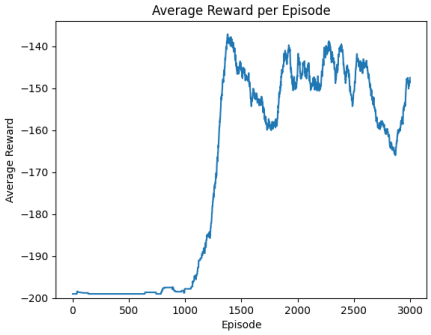


Figure 9: fixed lr = 0.0012

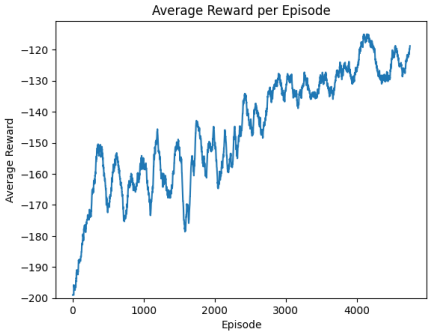


Figure 10: decaying lr = 0.012

IV Critical aspects and what went wrong

The solutions proposed in the previous sections certainly have critical points, in the q-table the hardest part was understanding how to discretize the input and how to set the size of the table so as not to lose too many observations, indeed I report that before making a discretized table I made some attempts without this technique, obviously after a few thousand episodes the table started to grow indefinitely consuming all the available memory and my computer crashed. In addition, as mentioned above, I had to fine-tune some parameters values before getting any acceptable results, in particular I made different executions with epsilon and env size but I never changed the discount factor. The tests have shown how an intermediate epsilon of 0.5 reduces convergence times because it encourages the agent more to immediately exploit what he has learned.
Instead in the case with the neural network the main problem was to find the best network for this type of environment, clearly this required me more time because each training on about 4000 episodes takes 1 hour and a half, but luckily the python libraries I've used are well documented and there are many examples of how to use them. I used different learning rates and the tests showed that a variable learning rate leads to a more stable model as shown in Figure 9 and 10; I also tried using batch sizes of 32, 64, 128, 256 and the first two values seem to behave better because the probability that two batches with the same observations are taken consecutively is reduced and this decreases the probability that the network is too correlated only to certain neurons. Parameters are reported in Table 1 and 2.

Episodes	90000
γ	0.95
α	0.1
ϵ	0.5
min ϵ	0.1
Decay	$\frac{\epsilon}{Episodes-1}$
Table size	$(100, 100, 3)$

Table 1: Q-table

Episodes	4000
γ	0.99
α	0.012
ϵ	1.0
min ϵ	0.001
Decay	$0,85$
Batch size	32
Experience size	10000

Table 2: Neural network

Summarizing, the use of the q-table can be more suitable for certain types of environments where the number of states is finite, in fact in these cases the times to reach an optimal policy are minimal, in my implementation it took about 40 minutes to fill the table in 90000 episodes. Instead, the use of the neural network can lead to longer training times but requires much fewer episodes to achieve an optimal policy and also produces a model with greater ability to generalize.