Implementing a DQN Agent for the Mountain Car Challenge in OpenAI Gym

MALIK E ASHTAR (23044004) AND DANIEL MAGUIRE (23222425)

Video: https://www.youtube.com/watch?v=h00U8TdfgDw

Table of Contents

ABSTRACT

In the classic control problem, Mountain Car, the objective is to control a car situated between two hills, navigating it to the top of the right mountain where a flag marks the destination. This project employs two Reinforcement Learning approaches, a Deep Q-Network (DQN) [1] and a Double Deep Q-Network [2], to effectively learn and execute a policy that navigates the car to its target. The agent interacts with the environment, which is represented by a discrete action space consisting of three possible actions: accelerate to the left, remain stationary, or accelerate to the right.

The findings from this project reveal that both DQN and DDQN, equipped with a visual understanding of the car's relative position, can successfully learn to solve the Mountain Car problem. This demonstrates the versatility and capability of both the DQN and the DDQN in handling numerical state representations as well as visual inputs.

INTRODUCTION

When designing and training a reinforcement learning (RL) agent for a specific Gym environment, the primary goals typically revolve around enabling the agent to learn optimal or near optimal strategies to maximize its cumulative rewards in that environment. The OpenAI Gym library contains a set of environments that can be used within Python to train agents in a variety of games, ranging from simple i.e. Mountain Car, to complex i.e. SeaQuest.

The main objective is for the RL agent to become proficient within a particular environment by understanding the underlying mechanics of the environment, such as state transitions and reward structures. It does so by optimising its policy that attempts to maximise the agent's rewards over time. This involves choosing actions in given states that lead to the highest possible returns. The agent's success is typically measured by the amount of cumulative reward it can secure over an episode or period of time.

An effective RL agent should perform well under known conditions and have the ability to adapt well to new scenarios within the environment. If an agent learns well it will perform across a range of conditions and respond appropriately to changes within the environment.

WHY RL IS PREFERRED FOR THE MOUNTAIN CAR PROBLEM

Reinforcement Learning (RL) is particularly suited for solving problems like the Mountain Car problem, a standard test environment in OpenAI Gym where the objective is to drive a car up a steep hill. The car does not have enough acceleration on it's own to overcome gravity and ascend the slope directly. Instead, the car must learn to leverage potential energy by building momentum from swinging back and forth between the hills. Below explains why RL is the preferred machine learning paradigm for this type of problem and how it contrasts with other paradigms.

SEQUENTIAL DECISION MAKING

RL excels in environments requiring a sequence of decisions, where each decision affects future states. In the Mountain Car problem, each action taken by the car (accelerating forward, backward, or no acceleration) not only influences immediate outcomes (like slight movements) but also sets the stage for future possibilities (like reaching the goal).

Compared with Supervised Learning: Supervised learning is less effective here because it typically requires labelled data for each situation. Labelling correct actions at each possible state in Mountain Car isn't feasible because the optimal action depends on the context of previous actions (to build momentum).

EXPLORATION AND EXPLOITATION

RL inherently balances between exploring new actions to find more efficient solutions and exploiting known actions that yield high rewards. This is crucial in Mountain Car for discovering the necessary back-and-forth momentum strategy.

Compared with Unsupervised Learning: Unsupervised learning doesn't inherently address the need to optimize a specific objective over time, such as reaching the mountain's peak.

DEALING WITH DELAYED REWARDS

RL algorithms, especially those with temporal difference (TD) learning (like DQN), are designed to handle situations where rewards are delayed—a common scenario in the Mountain Car problem, where immediate actions often don't lead directly to the goal, but setup for future success.

Compared with Supervised Learning: In supervised learning, each input is expected to have an immediate label (output), which doesn't suit problems where the utility of actions is realized in the future.

Dynamic Environment Interaction

RL involves learning policies based on continuous interaction with the environment, which is ideal for the Mountain Car's dynamic requirements where the optimal strategy involves dynamically responding to the car's momentum and position.

Compared with Supervised and Unsupervised Learning: Both these paradigms typically operate on static datasets. They do not interact with an environment in a way that adjusts based on the actions of the learning algorithm.

Policy and Value Function Approximation

RL's Methods: Modern RL methods use neural networks (as in DQN) to approximate the value functions or policies directly from high-dimensional sensory inputs like raw pixels, which can be beneficial if the Mountain Car problem were visually based or had high-dimensional states.

Contrast with Other Methods: Classical machine learning methods would struggle with raw pixel data without manual feature engineering.

Specific Applications and Effectiveness

Direct Policy Learning: RL methods, particularly policy gradient methods, can directly learn a policy that maximizes rewards by adjusting actions based on sampled gradients. This is effective in the Mountain Car problem for learning complex maneuvers.

Adaptability and Robustness: RL algorithms can adapt their strategies based on the feedback from the environment. This adaptability is crucial in non-stationary environments or those with high uncertainty, common in real-world analogs of the Mountain Car problem (like actual driving scenarios with variable conditions).

Understanding the MountainCar-v0 Environment in OpenAI Gym

The Mountain Car environment, a classic reinforcement learning problem implemented in the OpenAI Gym toolkit, serves as an excellent example to illustrate the concepts and functionality of reinforcement learning. Here is a detailed overview of this environment:

1. Goal

The primary objective in the Mountain Car environment is to drive an underpowered car up a steep hill to reach the flag at the top (position 0.5). The challenge arises because the car's engine is not powerful enough to climb the hill in a direct ascent. Therefore, the car must learn to leverage potential energy by rocking back and forth to build enough momentum to reach the goal.

2. States

The Mountain Car environments state representation consists of two components; the cars position (x) and velocity (v) represented as a tuple. The position, labelled as x indicates where the car is positioned along a track ranging from 1.2 to 0.6. Notably the leftmost point, on the track is at x = 1.2 while the rightmost point, housing the goal flag sits approximately x = 0.5. Looking at velocity, denoted by v, determines how fast or slow the car is moving. Negative values imply left motion and positive values signify right movement, within a range of 0.07 to 0.07.

A key issue in navigating the Mountain Car environment arises from its state space nature due to both position and velocity being represented by real numbers—leading to an infinite array of potential states that complicates problem solving.

3. Actions

Regarding actions, in this environment there are three options

0: Accelerates the car left, to decrease its x value.

1: No Acceleration

2: Accelerates the car right, increasing its x value.

The action selected has an impact on how the car moves and as a result, where it ends up in the next moment following the physics of that environment.

4. Rewards

The reward structure is simple but it effectively shapes the learning behavior needed to solve the task:

-1: The car receives a reward of -1 for every time step that the car does not reach the goal. This incentivizes the agent to reach the goal as quickly as possible since doing so minimizes the cumulative negative reward.

The agent's episode continues until it reaches the goal (position 0.5 or greater), or after a maximum number of steps (usually 200) is reached. If the car reaches the goal, the episode ends, and the agent receives no further rewards. This terminal reward structure (no positive reward at the goal) strictly focuses the agent on speed/efficiency rather than exploring beyond the goal.

5. Simulation Dynamics

The dynamics of the car's movement are governed by a simple set of equations that simulate gravitational pull and motor force. At each timestep, the velocity and position of the car are updated based on the current state and the action taken:

- Velocity Update: $v' = v + 0.001*A - 0.0025*\cos(3*x)$

- Position Update: $x' = x + v'$

Where A corresponds to the acceleration action (-1, 0, or 1), and the cosine term models the gravitational effects (which vary with the position). The environment also includes conditions to limit the position and velocity within their respective ranges (e.g., the velocity is clipped between -0.07 and 0.07).

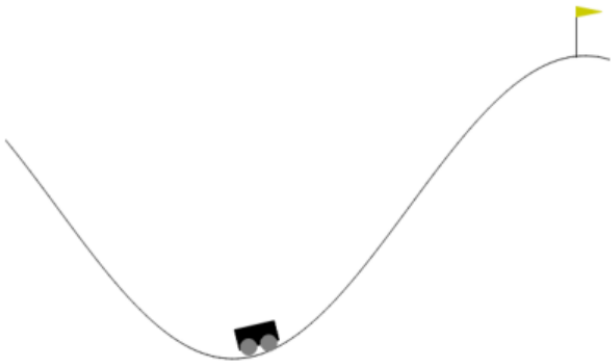
METHODOLOGY

Environment Initialisation

```
import gym
env = gym.make("MountainCar-v0", render_mode="rgb_array")
input_shape = env.observation_space.shape
n_outputs = env.action_space.n
```

Here we initialise a "MountainCar-v0" environment in OpenAI's Gym, rendering the environment as an RGB array. The observation space (state representation of position and velocity) shape is stored in input_shape, and n_outputs is set to the number of discrete actions available (3 actions: push left, no push, push right)

Visualising the Initial State of the 'MountainCar-v0' Environment



Preparing for DQN Implementation:

Here we initialise a Deep Q-Network (DQN) using Keras and TensorFlow, which includes resetting the computational graph, setting random seeds for reproducibility (np.random.seed(42) and tf.random.set_seed(42)), and constructing a neural network model with two hidden layers of 30 neurons each using ELU activation, an input layer tailored to the environment's state space shape (input_shape=input_shape), and an output layer designed to predict Q-values for each possible action (n_outputs equal to the number of actions in the "MountainCar-v0" environment). The network's structure is created using keras.models.Sequential() which simplifies stacking layers linearly.

Network Structure

```
#Preparing for DQN implementation
keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)

#Building the DQN model
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", input_shape=input_shape),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

- Sequential Model: The model is defined using Sequential, which facilitates the linear stacking of layers.
- Input Layer: The first dense layer takes an input shaped as input_shape (defined earlier i.e 2), which should correspond to the dimensionality of the state space in the environment.
- Hidden Layers: Two hidden layers, each with 30 neurons using Exponential Linear Unit (ELU) activation functions, provide sufficient complexity and non-linearity to capture relationships in the input data.
- Output Layer: The final dense layer has a number of neurons equal to n_outputs (defined earlier i.e. 3), which is typically the size of the action space in the environment, outputting the Q-value for each possible action.

Implementing Epsilon-Greedy Policy for Action Selection in DQN

```
def epsilon_greedy_policy(state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)
    else:
        #This insures state is a numpy array
        #before passing it to model.predict
        if isinstance(state, tuple):
            state = state[0]
        Q_values = model.predict(state[np.newaxis], verbose=0)
        return np.argmax(Q_values[0])
```

The *epsilon_greedy_policy* function is a strategy that balances exploration and exploitation when selecting actions. It takes into account the state and a specified epsilon value. If a generated number is lower, than epsilon the policy opts for an action (exploration). Otherwise it utilises the model to predict the Q values for the state and picks the action, with the highest Q value (exploitation). This function requires input of a state and epsilon, ensuring that the state is appropriately formatted for the model by verifying if it is in the correct format and converts it as needed before making predictions. The output is the index of the selected action.

The necessary parameters include:

- *state* (representing the current environment state)
- *epsilon* (indicating exploration rate)
- *n_outputs* (representing actions)
- *model* used to predict Q values.

Setup and Sampling Mechanism for Replay Memory in DQN Training

```
replay_memory = deque(maxlen=2000)
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_memory), size=batch_size)
    batch = [replay_memory[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

- The *sample_experiences* function samples a batch of experiences from replay_memory to train the model. The batch_size parameter specifies how many experiences to sample.
- indices: Randomly selects indices corresponding to experiences in replay_memory using np.random.randint. This is crucial for random sampling, which helps in breaking the correlation between consecutive experiences.
- batch: Constructs a list of experiences by indexing replay_memory using the randomly generated indices.
- states, actions, rewards, next_states, dones: Each is an array extracted from batch. The data is organized such that:
- states contains the state from each experience.
- actions contains the action taken in each state.
- rewards contains the reward received after taking the action.
- next_states contains the states that follow the actions.
- dones contains boolean values indicating whether each next state is terminal (end of an episode).

These arrays are prepared by iterating over each field index (0 to 4) for all experiences in the sampled batch, effectively restructuring the batch data into format suitable for training

THE DEEP Q NETWORK

One-Step Dynamics and Reward Modification for DQN Training in MountainCar

Epsilon-Greedy Action Selection:

```
def play_one_step(env, state, epsilon, car_max_velocity):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, terminated, info, truncated = env.step(action)
    done = terminated
```

- Chooses an action based on the epsilon-greedy policy.
- Executes the chosen action in the environment to get the new state and reward.
- The done status is determined by whether the episode has terminated.

Velocity and Reward Adjustments:

```
#Directional velocity
current_velocity = next_state[1]
if abs(current_velocity) > abs(car_max_velocity):
    #Reward bonus for new highest velocity
    reward += 1
    car_max_velocity = current_velocity

#Reward based on progress and velocity
#Adding bonus for getting closer to the goal, around position 0.5 in
#Scaling to enhance the effect with positive values only
progress_bonus = max(0, (next_state[0] - (-0.5)) * 2)
#Rewarding positive velocity more as it approaches the goal
velocity_bonus = current_velocity if current_velocity > 0 else 0

modified_reward = reward + progress_bonus + velocity_bonus

if done:
    #Large bonus for actually reaching the goal
    modified_reward += 10
```

- Checks if the current velocity is a new maximum and adjusts the reward accordingly.
- Calculates bonuses based on progress towards the goal and the car's velocity, modifying the reward to incentivize desirable behaviors like moving towards the goal and maintaining high speeds.

Experience Storage:

```
replay_memory.append((state, action, modified_reward, next_state, done))
return next_state, modified_reward, done, info, car_max_velocity
```

- Here we append the experience to the replay memory, which includes the current state, action taken, modified reward, next state, and whether the episode has terminated.

Configuration of Training Hyperparameters for Deep Q Network

```
batch_size = 32
discount_rate = 0.95
optimizer = keras.optimizers.Adam(learning_rate=1e-2)
loss_fn = keras.losses.mean_squared_error
```

- Set the batch size to 32
- Set the discount factor (also known as gamma in the context of reinforcement learning) to 0.95
- Initializes an Adam optimizer with a learning rate of 0.01 for training the DQN model
- Set the loss function to Mean Squared Error (MSE), which is standard for regression problems. In the context of DQN, MSE is used to measure the squared difference between predicted Q-values and the target Q-values (calculated using Bellman equation updates).

DEEP Q NETWORK TRAINING

SAMPLING EXPERIENCES:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
```

- This function randomly samples a specified number of experiences from the replay memory. Each experience typically includes the state, action taken, reward received, the next state, and a done flag indicating whether the episode has ended

PREDICTING NEXT Q-VALUES:

```
next_Q_values = model.predict(next_states, verbose=0)
max_next_Q_values = np.max(next_Q_values, axis=1)
```

- Uses the current DQN model to estimate the Q-values for the next states. The verbose=0 argument suppresses the log messages during this operation.
- Computes the maximum Q-value for each next state. This is based on the assumption of optimal future actions, a key concept in Q-learning.

CALCULATING TARGET Q-VALUES:

```
target_Q_values = (rewards +
    (1 - dones) * discount_rate * max_next_Q_values)

target_Q_values = target_Q_values.reshape(-1, 1)
```

The target Q values for each state-action pair are calculated using the Bellman equation: the immediate reward plus the discounted maximum Q-value of the next state, adjusted by whether the state is terminal (done). Terminal states have a next Q value of 0.

Bellman Equation: The target Q-values for the DQN are calculated using the Bellman update rule:

$$Q_{target} = R + \gamma \times (1 - D) \times max_{a'} Q(s'a')$$

where R is the immediate reward, γ is the discount factor, $max_{a'} Q(s'a')$ are the maximum future Q-values as predicted by the model for the next states, and $1-D$ is a factor that nullifies the future Q-values at terminal states [3].

CREATING A MASK FOR CHOSEN ACTIONS:

```
mask = tf.one_hot(actions, n_outputs)
```

LOSS CALCULATION AND GRADIENT UPDATE:

```
with tf.GradientTape() as tape:
    all_Q_values = model(states)
    Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
    loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
```

- Context manager that records the gradients of the loss with respect to the model parameters. This is necessary for automatic differentiation.
- Predicts the Q-values for the current batch of states.
- Applies the mask to the predicted Q-values to extract the Q-value of each chosen action.
- Computes the mean squared error loss between the predicted Q-values (for the selected actions) and the target Q-values.

```
grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

- **Gradient Computation**: Within the context of tf.GradientTape(), TensorFlow automatically calculates the gradients of the loss with respect to the model parameters (model.trainable_variables), which indicates how the loss would change with small changes in each parameter.

- **Weights Update**: The computed gradients are then applied to the model's parameters using the specified optimizer (optimizer). This step adjusts the weights to minimize the loss, thereby correcting the model's Q-value predictions towards the target Q-values.

PREPARATION AND INITIALIZATION:

```
#Variables for tracking best performance and rewards
best_score = -np.inf
rewards = []
epsilons = []
best_weights = None
num_episodes = 500
car_max_velocity = 0

#dictionary to store weights every 100 episodes
saved_weights = {}

write_log("DQN: Starting training...")
```

TRAINING LOOP PER EPISODE:

```
for episode in range(num_episodes):
    obs = env.reset()
    episode_reward = 0
    #Initialise new list to store frames for this episode
    frames = []

    for step in range(500):
        epsilon = max(1 - episode / num_episodes, 0.01)
        #Capture frame
        frames.append(env.render())
        obs, reward, done, info, car_max_velocity =
            play_one_step(env, obs, epsilon, car_max_velocity)
        episode_reward += reward
        if done:
            break
```

- Each episode starts by resetting the environment and initializing reward and frame list for the episode.

- Epsilon Calculation: Decreases linearly with the number of episodes to shift from exploration to exploitation.
- Frame Capture: For potentially creating a video of the episode.
- Play One Step: Executes an action in the environment using the play_one_step function which applies the epsilon-greedy policy.
- End Episode on Terminal State: Stops the episode if the environment returns a terminal state.

POST-EPISODE PROCESSING:

- Track Rewards and Epsilons: For analysis and plotting performance over episodes.
- Check for Best Performance: Updates best performance metrics and saves the video and weights if the current episode's reward exceeds the previous best.
- Training Step: Calls the training_step function to update the model using a batch of experiences from the replay memory if it is sufficiently large.

```
if episode_reward > best_score:
    best_score = episode_reward
    best_weights = model.get_weights()
    save_video(frames,
               f'ep_{episode}_score_{episode_reward:3f}.mp4')
    saved_weights[episode] = model.get_weights()

if len(replay_memory) > batch_size:
    training_step(batch_size)
```

LOGGING AND SAVING PROGRESS:

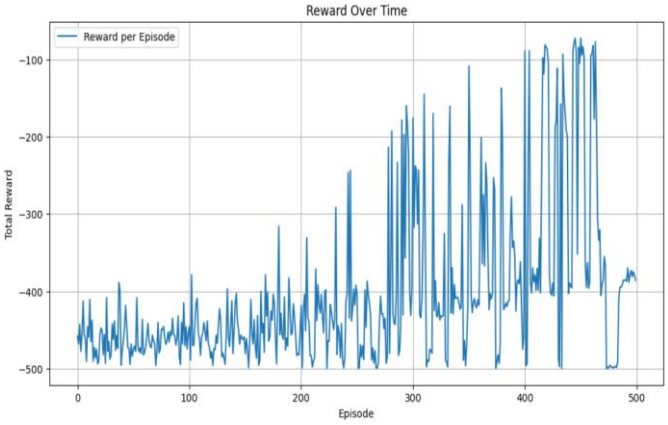
```
episode_details = f"\rEpisode: {episode},
                          Reward: {episode_reward},
                            Best score: {best_score},
                              Epsilon: {epsilon:.3f}"
write_log(episode_details)
print(f"{episode_details}", end="")
```

POST-TRAINING:

```
if best_weights:
    model.set_weights(best_weights)
```

Load Best Weights: After training completes, if improved weights were found, they are loaded back into the model to ensure the best performance is used for any subsequent operations.

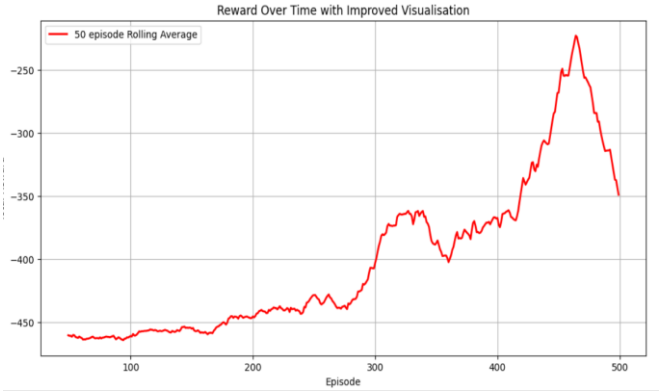
DEEP Q NETWORK RESULTS



The reward increases as the episodes progress, though it does contain a large amount of variability which is to be expected. The peaks are an indicator that the agent has discovered some successful strategy and earned a large reward. Overall we can see an increase in the reward over time which is a good indicator that the agent learning.

SMOOTHED VISUALISATION

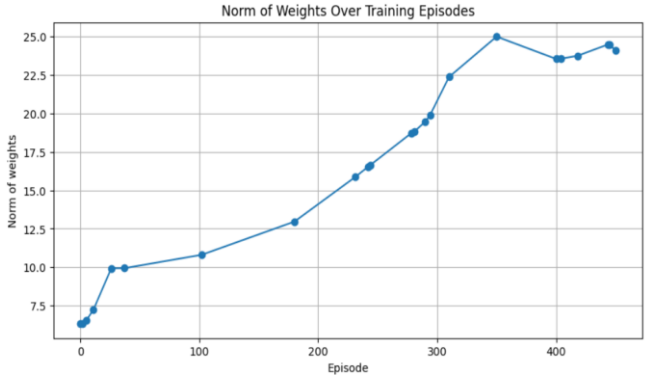
Below we compute the rolling average of the rewards with a window size of 50 episodes. This window size determines how many consecutive episodes' rewards are averaged to calculate the rolling average at each point, smoothing over 50 episodes at a time.



By applying the 50 episode rolling average we can see a clear trend that the agent is successfully learning over episodes. Around episode 160 we can start to see a clear improvement. There is a steady and consistent increase in the reward over time which is a good indicator that the agent is learning.

MONITORING THE MAGNITUDE OF DQN MODEL WEIGHTS ACROSS TRAINING EPISODES

Below we are calculating the Euclidean norm (L2 norm) of the weights for each set of saved weights



We can see that the norm of the weights increases as we train but start to see a plateau starting around episode 350 which is a sign that the learning process is stabilising.

Implementation of the test_agent Function

```
def test_agent(env, model, n_episodes=50):
    total_rewards = []
    for episode in range(n_episodes):
        #Reset the environment and get the initial state
        state_tuple = env.reset()
        #Insure state is the actual observation array, not a tuple
        state = state_tuple if not isinstance(state_tuple, tuple) else state_tuple[0]
        done = False
        total_reward = 0
        while not done:
            #Prepare state for model prediction
            q_values = model.predict(state[np.newaxis], verbose=0)
            #Choose the best action based on the model's prediction
            action = np.argmax(q_values[0])
            #Perform the action
            outcomes = env.step(action)
            #Correctly handle different lengths of returned values
            state_tuple, reward, done, _ = outcomes if len(outcomes) == 4 else (outcomes[0],
            #Update state with the new state
            state = state_tuple if not isinstance(state_tuple, tuple) else state_tuple[0]
            total_reward += reward
        total_rewards.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")
    print(f"Average Total Reward over {n_episodes} episodes: {np.mean(total_rewards)}")
    plot_rewards(total_rewards)

test_agent(env, model)
```

Capture and Sampling of the Data:

In the test_agent function, data capture and sampling are executed through systematic interactions with the environment using a model-predicted policy to perform actions and collect states, actions, and rewards, which are then used to update the agent's state and compile rewards for performance evaluation.

- Environment Reset (env.reset()): Initializes the environment to a starting state and provides the initial observation. This state is either directly usable or unpacked from a tuple, depending on the environment's implementation.
- Predict and Act (model.predict(...), env.step(action)): The model predicts Q-values for the current state, and the best action (highest Q-value) is chosen and executed in the environment. This step is where the "sampling" of Q-values (data) occurs, guiding the action selection process.
- State and Reward Update: After performing the action, the environment returns the new state, reward, and completion status (done). These elements are captured and used to update the agent's state and cumulative reward.

This process effectively outlines how the function interacts with the environment to "sample" necessary data (states, actions, rewards) by utilising the trained model to make decisions (action selections) and observing the outcomes, which directly influences the agent's learning and assessment post-episode. This method ensures a comprehensive evaluation of the model across a variety of environmental scenarios, reflecting its practical utility and effectiveness.

The action with the highest Q-value is selected as it represents the model's belief in the most rewarding action from the current state.

Environment Interaction:

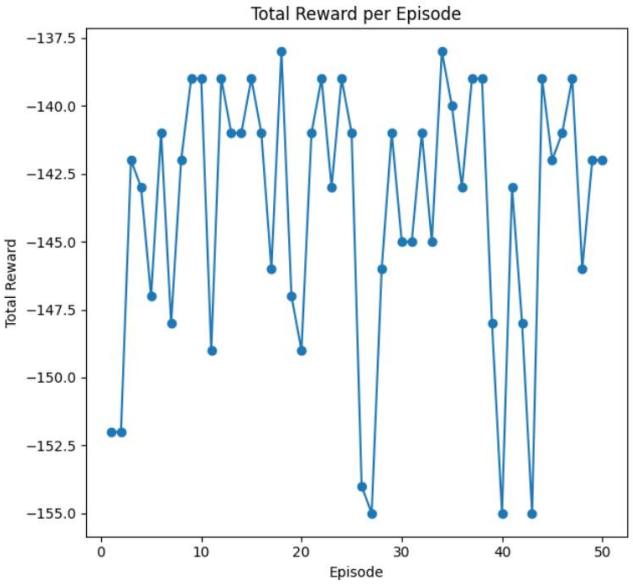
- env.step(action): Executes the chosen action in the environment, which returns the new state, reward, and done flag, among possibly other information.

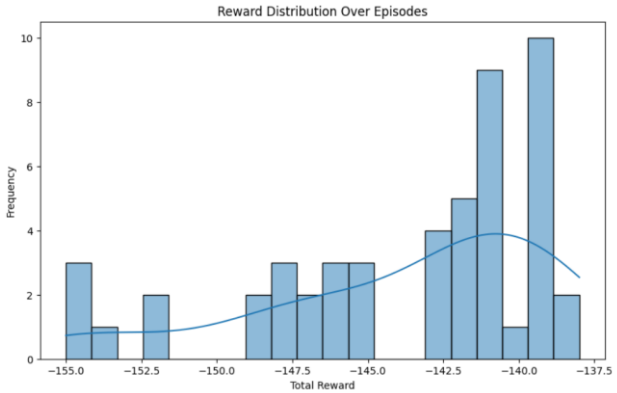
Handling Different Return Types:

- Ensures compatibility with different types of environments which might return additional data or use a tuple to enclose the state.

State and Reward Update:

- The reward from performing the action is added to total_reward for the current episode.





DOUBLE DEEP Q NETWORK

CONFIGURATION OF TRAINING HYPERPARAMETERS FOR DOUBLE DEEP Q-NETWORK

```
batch_size = 32
discount_rate = 0.95
optimizer = keras.optimizers.Adam(learning_rate=1e-2)
loss_fn = keras.losses.mean_squared_error
```

Here we are initializing training hyperparameters for a DDQN model, setting a batch size of 32 for experience sampling, a discount rate of 0.95 for future reward valuation, an Adam optimizer with a learning rate of 0.01 for efficient gradient descent, and mean squared error as the loss function to optimize Q-value predictions.

NETWORK STRUCTURE

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", input_shape=input_shape),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(n_outputs)
])

#Cloning the model to create a target model
target_model = keras.models.clone_model(model)
target_model.set_weights(model.get_weights())
```

Initializes a DQN using Keras, with two hidden layers of 30 neurons each using ELU activation, and an output layer matching the action space size, then creates a stable target model by cloning the primary model's architecture and synchronizing their weights to facilitate stable learning through the dual-network approach typical in DQN implementations.

DOUBLE DEEP Q NETWORK TRAINING

```
def DDQN_training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
```

Samples a mini-batch of experiences from the replay memory, each experience typically comprises the state, action, reward, next state, and done flag indicating whether the episode has ended.

```
#Use the main model to select the best action for the next states
next_actions = np.argmax(model.predict(next_states, verbose=0), axis=1)
```

The main model (online network) predicts the Q-values for the next states and the action with the highest Q-value is chosen. This step involves only action selection, not evaluation.

```
#Use the target model to calculate the Q-values for the best actions
#in the next states as chosen by the main model
next_Q_values = target_model.predict(next_states, verbose=0)[np.arange(batch_size), next_actions]
```

The target model (stable network) is used to predict the Q-values for the next states. The Q-values corresponding to the actions chosen by the main model are selected. This decoupling of action selection and Q-value calculation is key in DDQN to prevent overestimations.

```
#Calculate target Q values
target_Q_values = rewards + (1 - dones) * discount_rate * next_Q_values
target_Q_values = target_Q_values.reshape(-1, 1)
```

Q value Calculation for Best Actions: The target model then predicts the Q values for these next states but only the Q-values corresponding to the best actions (selected by the main model) are used. This use of the target model helps stabilize learning.

The target Q values are computed using the Bellman equation [3], which in its simplest form for Q learning is:

$$Q_{target} = R + \gamma \cdot max(Q(s'a'))$$

where R is the immediate reward, γ is the discount factor, $max(Q(s'a'))$ is the maximum predicted Q value for the next state s' (across all possible actions a'), predicted by the target model.

ERROR CALCULATION (LOSS COMPUTATION):

```
with tf.GradientTape() as tape:
    all_Q_values = model(states)
    Q_values = tf.reduce_sum(all_Q_values
                             * tf.one_hot(actions, n_outputs),
                                axis=1, keepdims=True)
    loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

- Gradient Recording: tf.GradientTape() is used to monitor all computations for automatic differentiation.
- Q-value Filtering: From the main model's predictions (all_Q_values), only those Q-values that correspond to the actions taken are filtered out using one-hot encoding multiplied by the predicted Q-values.
- MSE Calculation: The loss is the mean squared error between these filtered Q-values (Q_values) and the computed target_Q_values.

Weights Update:

Finally, the weights of the main model are updated by applying gradients computed from the loss:

```
grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

- Gradient Computation: The gradients of the loss with respect to the model's trainable parameters are calculated.
- Apply Gradients: These gradients are applied using the specified optimizer (Adam in this case), effectively nudging the weights in a direction that minimizes the loss, thereby improving the model's predictions.

COMPREHENSIVE DDQN TRAINING LOOP FOR OPTIMAL POLICY DEVELOPMENT IN REINFORCEMENT LEARNING ENVIRONMENTS

```
#Vriables for tracking best performance and rewards
best_score = -np.inf
rewards = []
epsilons = []
best_weights = None
num_episodes = 500
car_max_velocity = 0

#dictionary to store weights every 100 episodes
saved_weights = {}

write_log("Starting training DDQN...")
for episode in range(num_episodes):
    obs = env.reset()
    episode_reward = 0
    #Initialise new list to store frames for this episode
    frames = []
```

Initializes variables to track the best score, list of rewards per episode, exploration rates (epsilons), and the best model weights.

```
for step in range(500):
    epsilon = max(1 - episode / num_episodes, 0.01)
    #Capture frame
    frames.append(env.render())
    obs, reward, done, info, car_max_velocity = play_one_step(env, obs, epsilon, car_max_velocity)
    episode_reward += reward
    if done:
        break
```

- Each episode can have a maximum of 500 steps.
- epsilon decreases linearly from 1 to 0.01, reducing the amount of exploration over time.
- Captures and appends the rendered state of the environment to frames.
- Captures and appends the rendered state of the environment to frames.
- play_one_step() executes one action based on the current state and policy (epsilon-greedy). It returns the new state, reward, and whether the episode has terminated (done).
- Accumulates the received reward into episode_reward.
- Exits the loop if the episode terminates (done is True).

Post-Episode Processing

```
if episode_reward > best_score:
    best_score = episode_reward
    best_weights = model.get_weights()
    save_video(frames,
               f'ep_{episode}_score_{episode_reward:3f}.mp4')
    saved_weights[episode] = model.get_weights()
```

- Updates the lists rewards and epsilons.
- Checks if the current episode's reward exceeds the best_score. If so, updates best_score, saves the episode's frames as a video, and updates best_weights.

Conditional Model and Target Model Updates:

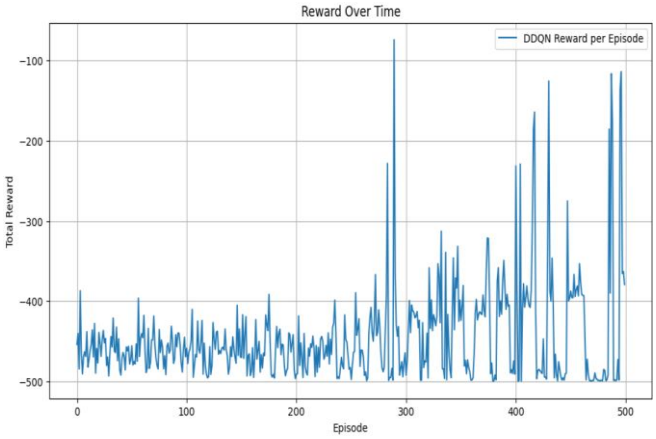
```
if len(replay_memory) > batch_size:
    DDQN_training_step(batch_size)

#Frequency to update the model
if episode % 20 == 0:
    target_model.set_weights(model.get_weights())
```

- Performs a training step using DDQN_training_step() if the replay memory has enough experiences.
- Periodically updates the target model's weights with the main model's weights to stabilize training (every 20 episodes in this case).

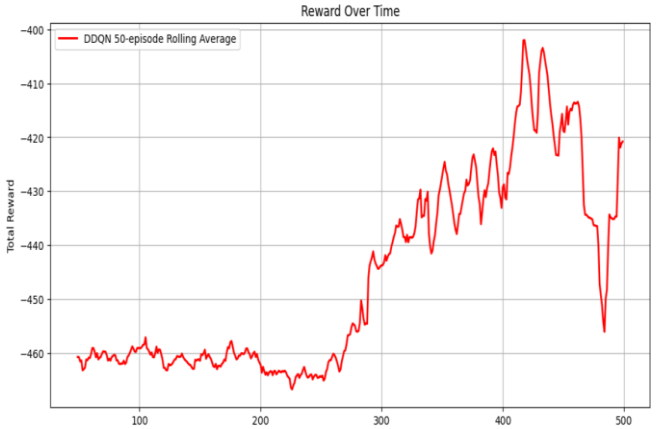
DDQN Results

Visualisation of DDQN Training Rewards Over 500 Episodes



Just like in the case of the DQN, we can see that the reward is increasing over time, though it does contain a large amount of variability due to the exploration/exploitation trade off. The peaks indicate that the agent has discovered some successful strategies. Overall we can see an increase in the reward over time which is a good indicator of learning.

Smoothed Visualisation



Here we can see a steep improvement in the reward accumulated around episode 260. We can clearly see that the agent is successfully learning the environment. However, the

DDQN learning curve is a lot less smooth compared to the DQN which was clearly learning at a smoother rate. If we trained for longer this may have

Performance Evaluation of DQN/DDQN Models Over Multiple Episodes

```
def test_agent(env, model, n_episodes=50):
    total_rewards = []
    for episode in range(n_episodes):
        #Reset the environment and get the initial state
        state_tuple = env.reset()

        state = state_tuple if not isinstance(state_tuple, tuple) else state_tuple[0]
        done = False
        total_reward = 0
```

Starts the episode loop where the environment is reset to obtain the initial state, and variables for tracking the episode's completion (done) and cumulative reward (total_reward) are initialized.

```
while not done:
    #Preparing state for model prediction
    q_values = model.predict(state[np.newaxis], verbose=0)
    #Choosing best action based on the model prediction
    action = np.argmax(q_values[0])
    #Perform action
    outcomes = env.step(action)
    #Handle different lengths of returned values
    state_tuple, reward, done, _ = outcomes if len(outcomes) == 4 else (outcomes[0],
    #Update state with new state
    state = state_tuple if not isinstance(state_tuple, tuple) else state_tuple[0]
    total_reward += reward
```

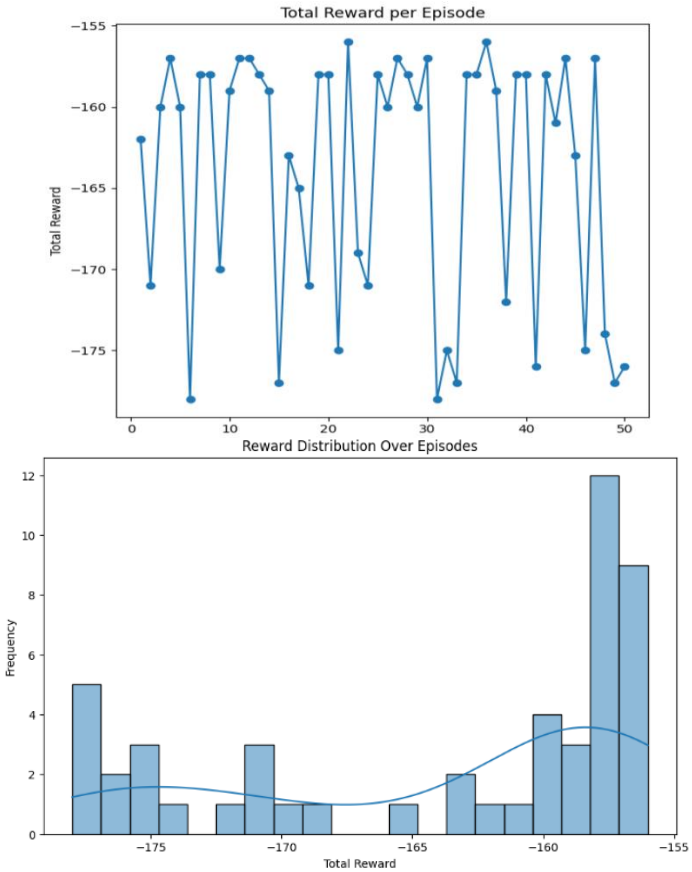
Executes a loop within each episode to select and perform actions based on the model's Q-value predictions until the episode ends (done), updating the state and accumulating rewards.

```
        total_rewards.append(total_reward)
        print(f"Episode {episode + 1}: Total Reward = {total_reward}")

    print(f"Average Total Reward over {n_episodes} episodes: {np.mean(total_rewards)}")
    plot_rewards(total_rewards)

test_agent(env, model)
```

- At the end of each episode, records the total_reward in total_rewards and prints it, providing a summary of the agent's performance for that episode.
- After all episodes are completed, calculates and prints the average total reward across all episodes and visualizes this data using plot_rewards, a function assumed to be defined elsewhere that plots the reward trajectory.



Overall, when comparing the DDQN reward distribution with the DQN reward distribution, the DQN performed better than the DDQN . This could be due to the frequency (20) which we chose to update the target model, or the increased variance introduced by decoupling the two networks.

Conclusion

The successful implementation of a DQN, as well as the DDQN, to solve the Mountain Car problem demonstrates the potential of these deep reinforcement learning strategies in classic control challenges and also raises intriguing questions about the limits of visual-based state representation and the ability of the DQN and DDQN to adapt to diverse problem settings. This project highlights significant considerations in the choice of network architecture, hyperparameters, and the method of representing environmental states, all of which contribute to the agent's learning efficiency and success.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," in *Proc. NIPS Deep Learning Workshop*, 2013.

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," in *Proc. of AAAI Conference on Artificial Intelligence (AAAI)*, 2016, arXiv:1509.06461 [cs.LG].

[3] R. Bellman, "A Markovian Decision Process," Journal of Mathematics and Mechanics, vol. 6, no. 5, pp. 679-684, 1957.