it

University of Tripoli

Faculty of Engineering

Electrical and Electronic Engineering Department

EE569 – Deep Neural Networks

Fall 2024

Brief report for

Assignment 2

Reinforcement Learning Task

Team members:

Mohamed Mustafa Kamil   ID: 2180205565

Mohamed Abdulbaset Alshareef   ID: 2180205640

Instructor: Dr. Nuri Ben Barka

1. Introduction

This project aims to train two agents (cars) to race effectively in the MultiCarRacing-v0 environment using a custom-built Deep Q-Learning (DQN) algorithm. The core objective is to implement every component of the DQN system from scratch—including neural network construction, experience replay, and epsilon-greedy action selection—using supporting libraries (e.g., TensorFlow/Keras) without relying on any pre-built DQN models. The project follows the assignment steps exactly, covering environment setup, preprocessing, network construction, training, and evaluation.

2. Objective

- Implement DQN from scratch: Build all components (neural network, experience replay, epsilon-greedy action selection) without using pre-built models.
- Multi-agent training: Train two independent agents concurrently in the MultiCarRacing-v0 environment.
- Effective preprocessing: Convert raw images to grayscale, use frame stacking, and apply frame skipping for efficient state representation.
- Custom CNN design: Design a convolutional neural network to approximate Q-values for a discrete action space (optimized to 5 actions).
- Stabilize learning: Use target network updates and separate replay buffers to enhance training stability.
- Performance evaluation: Test the trained models in deterministic mode (epsilon = 0) to measure and validate agent performance.

3. Steps of Workflow

Step 1: Understanding the Environment

- Environment: MultiCarRacing-v0 from the multi_car_racing repository.
- Observation Space: Each agent receives an image with dimensions (96, 96, 3) representing a top-down view of the track.
- Action Space: Each car is controlled by three continuous values corresponding to [steering, gas, brake].
- Rewards and Termination:
  - Agents are rewarded for moving forward on the track.
  - Penalties are applied for leaving the road or driving backward.
  - An episode terminates when all agents leave the track or finish the race.

Step 2: Setup Process

- Cloning and Installation: We cloned the multi_car_racing repository and installed it in editable mode.
- Environment Initialization: In our training and testing scripts, we initialize the environment.

Step 2: Preprocessing Observations

To provide the neural network with efficient and lower-dimensional inputs, we implemented two key functions:

- **process_state_image(state)**
  Converts a raw RGB image to grayscale and normalizes the pixel values.
- **generate_state_frame_stack_from_queue(deque)**
  Stacks multiple consecutive frames along the channel dimension to capture temporal dynamics.

Step 3: DQN Agent Implementation

The core of the project is the CarRacingDQNAgent class. This class encapsulates all functionalities needed for a DQN agent.

- **CarRacingDQNAgent Class**
  To define a DQN agent that can interact with the environment, store experiences, and update its Q-network through learning.
  Constructor (__init__) Parameters:
  - action_space: A list defining the available discrete actions. Initially, a larger set (12 actions) was provided, but experiments later led to using 5 actions for better performance.
  - frame_stack_num: Number of consecutive frames to stack.
  - memory_size: Maximum capacity of the replay buffer.
  - gamma: Discount factor for future rewards.
  - epsilon, epsilon_min, epsilon_decay: Parameters for the epsilon-greedy strategy.
  - learning_rate: Learning rate for training the neural network.

- **build_model() function**
  Builds the convolutional neural network (CNN) that approximates the Q-values for each discrete action.

  Input Layer: Accepts input with shape (96, 96, frame_stack_num).

  Convolutional Layers:

  - First Conv2D layer with 6 filters, kernel size (7, 7), stride 3, and ReLU activation.
  - MaxPooling layer.
  - Second Conv2D layer with 12 filters, kernel size (4, 4), and ReLU activation.
  - Followed by another MaxPooling layer.

  Flatten Layer: Converts 2D feature maps to a 1D vector.
  Dense Layers:
  - One hidden Dense layer with 216 units and ReLU activation
  - Output Dense layer with a number of units equal to the size of the action space (e.g., 5 or 12) with a linear activation

  Compilation: Uses mean squared error (MSE) as the loss function and Adam optimizer with the specified learning rate.

- **update_target_model() function**
  Copies the weights from the primary Q-network to the target Q-network. This technique stabilizes training by keeping the target values fixed for several iterations.

- **memorize(state, action, reward, next_state, done) function**
  Stores a single experience tuple in the replay buffer. Converts the action into an index by finding its position in action_space and appends the tuple (state, action_index, reward, next_state, done) to the replay memory.

- **act(state) function**
  Determines the next action using an epsilon-greedy policy.
  - With probability $1 - \epsilon$, selects the action with the highest Q-value (exploitation).
  - With probability ϵ, selects a random action from the action space (exploration).

- **replay(batch_size) function**
  Samples a random mini-batch from the replay buffer to update the Q-network. For each experience in the mini-batch:
  - If the episode is terminated (done == True), the target for the action is the immediate reward.
  - Otherwise, compute the target as:

  $$\boldsymbol{target}[\boldsymbol{a}] = \boldsymbol{r} + \boldsymbol{\gamma} \max_{\boldsymbol{a'}} \boldsymbol{Q}_{\boldsymbol{target}}(\boldsymbol{s'}, \boldsymbol{a'})$$

  Train the network using the mini-batch with one epoch. Decay the epsilon value towards its minimum after the training step.

- **load(name) function**
  Loads saved model weights and updates the target network accordingly.

- **save(name) function**
  Saves the weights of the target model to a file.

Step 4: Training Procedure

The training process is managed by a dedicated script (dqn_multicar.py) that performs the following steps:

Environment and Agent Initialization:

- Environment Reset: The environment is reset at the beginning of each episode.
- State Preprocessing and Frame Stacking: Each initial state is processed using process_state_image and then stacked into a deque (with a fixed length determined by frame_stack_num). This provides a temporal window for each agent.
- Agent Creation: Two instances of CarRacingDQNAgent are created, one for each agent.

Main Training Loop:

For each episode:

- **Action Selection**:
  For each agent, the current stacked state is generated via generate_state_frame_stack_from_queue. Each agent selects an action using its act(state) method.

- **Environment Step**:
  The list of actions is passed to env.step(actions), which returns the next states, rewards, a done flag, and any additional info.

- **Reward Accumulation and State Update**:
  Rewards are accumulated per agent. Each agent processes its next state and updates its frame deque.

- **Experience Storage**:
  Each agent stores the experience tuple (state, action, reward, next_state, done) using memorize(...).

- **Training Update**:
  If the replay buffer for an agent exceeds the predefined batch size, the agent updates its network by sampling a mini-batch from the replay buffer and calling replay(batch_size).

- **Target Network Update and Model Saving**:
  At regular intervals, the target network is updated via update_target_model().The model weights are periodically saved to file.

Step 4: Evaluation Procedure

A separate evaluation script (e.g., MCR_test_model.py) is used to assess the performance of the trained agents:

- **Deterministic Behavior**:

  Epsilon is set to 0 to force the agent to always exploit its learned policy.
- **Model Loading**:

  Each agent loads its corresponding pre-trained model weights.
- **Running Episodes**:

  The environment is rendered and the agents' performance (total rewards, time frames) is printed.
- **Performance Comparison**:

  The total rewards are compared against a baseline (e.g., random actions) to verify improvement.

Results:

Episode	Track Generation	Tiles	Time Frames	Agent 0 Total Reward	Agent 1 Total Reward
1	1168..1464	296	1000	316.95	545.76
2	1125..1416	291	1000	15.01	-19.37
3	1164..1459	295	1000	491.84	323.47
4	915..1154	239	1000	366.39	631.09
5	1396..1749	353	1000	179.83	60.51
6	1271..1593	322	1000	326.79	685.05
7	1026..1287	261	1000	288.46	653.85
8	1100..1379	279	1000	738.13	371.22
9	1177..1475	298	1000	420.20	524.58
10	1093..1370	277	1000	485.14	226.09
Averages			**957.4**	**362.87**	**400.23**

The test results indicate that the trained models are generally capable of driving for extended periods (most episodes reached 1000 time frames) on various track configurations. Here are some detailed observations:

- Track Variability:
  Each episode reports a different track generation, with the number of tiles varying (e.g., 296-tiles, 291-tiles, 353-tiles). This variability shows that the agents are tested on diverse track lengths and configurations, which is useful for assessing generalization.

- Episode Length:
  Most episodes run for 1000 time frames, suggesting that the agents can sustain control over the entire duration. However, Episode 2 ended early at 574 time frames, which resulted in very low rewards for both agents. This may indicate that this particular track was either more challenging or that one or both agents encountered issues that forced an early termination.

- Reward Distribution:
  - In several episodes (e.g., Episodes 1, 3, 4, 6, 7, 9), both agents obtained reasonably positive rewards, though there is noticeable variability between them.
  - In some cases, one agent outperforms the other (for instance, in Episode 1, Agent 1 got a higher reward than Agent 0, while in Episode 8, Agent 0 achieved a very high reward compared to Agent 1).
  - This variation may be due to differences in how each agent has learned to handle different parts of the track or because of inherent variability in the environment dynamics.

- Overall Performance:
  The performance across the 10 episodes is mixed, with some episodes showing high cumulative rewards (e.g., Episode 8 with Agent 0 achieving 738.13) and others demonstrating low rewards (e.g., Episode 2 with negative rewards for Agent 1). This suggests that while the agents have learned useful policies, their performance can still be inconsistent, possibly due to differences in track difficulty or areas where further tuning might be necessary.

In summary, these results demonstrate that the agents are generally capable of sustained performance on diverse tracks, but further improvements in training or fine-tuning of hyperparameters might be needed to reduce variability and handle challenging tracks more robustly.

4. Experiments, Performance, and Challenges

Experiments:

- Action Space Experimentation:
  Initially, we experimented with a discrete action space of only 3 actions (for example, [Left, Gas, Brake]). However, the limited choices resulted in poor performance. After testing, we expanded the action space to 5 actions (Left, Right, Accelerate, Brake, No-Op), which provided more precise control and improved the agents' performance.

- Hyperparameter Tuning:
  We experimented with various hyperparameters including the learning rate (set to 0.001), gamma (0.95), epsilon decay, and replay buffer size (5000 transitions). These parameters were adjusted based on initial training runs to achieve better convergence and stability.

- Frame Stacking and Skipping:
  The use of frame stacking (stacking 3 or 4 consecutive frames) and frame skipping (processing every 3rd or

4th frame) was essential to capture temporal dynamics and to reduce computational load. These techniques significantly improved the training efficiency and overall performance of the agents.

Performance:

- Training Performance:
  Over the course of 500 to 1000 episodes, the agents showed a steady improvement in their ability to remain on track and complete laps. The average total rewards per episode increased as the agents learned more effective driving strategies.

- Evaluation Metrics:
  In evaluation runs (with epsilon set to 0), the agents achieved significantly higher average rewards compared to random action selection. Detailed performance metrics, such as average reward over 10 evaluation episodes and time frames per episode, demonstrated the stability and effectiveness of the learned policies.

Challenges and Solutions:

- Environment Registration:
  Initially, the MultiCarRacing-v0 environment was not recognized, leading to registration errors. This was resolved by properly cloning and installing the gym-multi-car-racing repository in editable mode.

- Action Space Limitation:
  Our early experiments with a 3-action space proved insufficient for effective control. Expanding to a 5-action space provided a better balance between simplicity and control granularity.

- Training Stability:
  Training multiple agents concurrently introduced additional variability. Techniques such as frame stacking, frame skipping, and the use of a separate target network for each agent were critical in stabilizing the training process.

- Replay Buffer Management:
  Ensuring that each agent maintained its own replay buffer helped prevent the mixing of experiences and allowed each agent to learn independently from its own interactions.

5. Conclusion

This project successfully implements a multi-agent DQN system from scratch, fulfilling all the requirements specified in the assignment. Key achievements include:

- Setting up and understanding the MultiCarRacing-v0 environment.
- Implementing essential preprocessing steps (grayscale conversion, frame stacking, frame skipping) to optimize the input for the neural network.
- Building a custom convolutional neural network to approximate Q-values for a discrete action space.
- Developing a comprehensive DQN agent with methods for action selection, experience replay, and network updates.
- Conducting extensive experiments and hyperparameter tuning to improve performance.
- Overcoming challenges related to environment setup, action space design, and training stability.

This report details every component of the project—from class definitions and function implementations to experimental results and encountered challenges—demonstrating a thorough and systematic approach to solving a multi-agent reinforcement learning task using Deep Q-Learning.