Final Project Report

Name: Yash Bhalgat
SID: 862465699
Email: ybhal001@ucr.edu

1. Overview of the project

The objective of this project was to implement the Rainbow DQN algorithm to train reinforcement learning agents to play Atari games such as Space Invaders and CartPole. Rainbow DQN combines several enhancements over traditional Deep Q-Networks (DQN), including Double Q-learning, Prioritized Experience Replay (PER), and Dueling Network Architectures, among others. These modifications enable improved learning efficiency, better convergence, and enhanced performance in complex environments.

The project aimed to evaluate the performance of the Rainbow DQN on different environments, focusing on stability and reward maximization over training episodes.

2. Description of Implementation

The Rainbow DQN implementation integrates several advanced reinforcement learning techniques, creating a comprehensive framework for training agents in complex environments. Below is a detailed technical breakdown of the implementation.

a. Environment setup:
The project utilizes the `gymnasium` library to access Atari games and simpler environments like CartPole.
Two environments were chosen:

- **Space Invaders (ALE/SpaceInvaders-v5)**: A complex environment requiring spatial and temporal learning.
- **CartPole (CartPole-v1)**: A simpler environment for balance and control tasks.

For Atari games:

- Observations are RGB images of shape `(210, 160, 3)`.
- These are converted to grayscale using `cv2.cvtColor` to reduce dimensionality.
- Images are resized to `(84, 84)` using `cv2.resize` for compatibility with the network.
- A single channel is added using `np.expand_dims` to match the input requirements of convolutional layers.

For CartPole:

- Observations are already numerical vectors, requiring no preprocessing.

b. Replay Buffer:

Prioritized Replay Buffer:

- A key component of Rainbow DQN, the **Prioritized Experience Replay (PER)** ensures that transitions with higher temporal-difference (TD) errors are sampled more frequently.
- **Priorities and Weights:**
  - Transitions are assigned priorities based on their TD errors: $priority = |\delta| + \epsilon,$ where δ is the TD error and ϵ is a small constant to avoid zero priority.
  - Sampling probabilities are calculated as $p(i) = priority^{\alpha} / \Sigma\, priority_j^{\alpha}$ where α controls the degree of prioritization.
  - Importance-sampling weights $w(i)$ are used to correct the bias introduced by prioritized sampling.
- Multi-step returns (nnn-step) are implemented to account for delayed rewards, improving long-term credit assignment.

c. Rainbow DQN Architecture:

The network architecture is designed to capture spatial and temporal features efficiently:

1. **Input Layer:**
   - Input: A single grayscale frame of shape `(1, 84, 84)`.
2. **Convolutional Layers:**
   - Three convolutional layers extract spatial features:
     - Layer 1: 32 filters $(8 * 8, stride = 4)$
     - Layer 2: 64 filters $(4 * 4, stride = 2)$
     - Layer 3: 64 filters $(3 * 3, stride = 1)$
3. **Dueling Network Architecture:**
   - Splits the network into two streams:
     - **State-Value Stream:** Estimates the value of being in the current state.
     - **Advantage Stream:** Estimates the advantage of each action.

- Combines them to compute Q-values:
  $Q(s,\ a)\ =\ V(S)\ +\ (A(s,\ a)\ -\ \frac{1}{|A|}\Sigma A(s,\ a^{'})$

4. **Output Layer:**
   - Outputs Q-values for each possible action.
5. **Optimization:**
   - Loss: **Huber Loss** (smooth L1 loss) is used for stability.
   - Optimizer: Adam with a learning rate of 10^{-4}.

d. Agent Implementation:

1. **Epsilon-Greedy Exploration:**
   - Balances exploration and exploitation:
     - Starts with high exploration $(\epsilon\ =\ 1.0)$
     - Decays exponentially to a minimum of $(\epsilon\ =\ 0.01)$
2. **Action Selection:**
   - With probability ϵ, selects a random action for exploration.
   - Otherwise, selects the action with the highest predicted Q-value.
3. **Target Network:**
   - A separate network is used to compute target Q-values, updated periodically to improve training stability.
4. **Gradient Updates:**
   - Updates are performed in mini-batches of size 64.
   - TD errors are calculated as: $\delta\ =\ r\ +\ \gamma\ *\ Q_{target}(s^{'},\ a^{'})\ -\ Q(s,a)$
   - Priorities are updated using TD errors.

e. Training and Evaluation:

1. **Training Process:**
   - At each episode:
     - The agent interacts with the environment, collects transitions, and stores them in the replay buffer.
     - After enough transitions, mini-batches are sampled, and the policy network is updated.
     - Every 10 episodes, the target network is synchronized with the policy network.
   - Metrics such as rewards, losses, and epsilon values are logged for visualization.
2. **Evaluation:**
   - The trained model is loaded, and the agent is tested for a fixed number of episodes.
   - The environment is rendered to visually observe the agent's behavior.

3. **Status of the code**

What is Working:

- The Rainbow DQN implementation successfully trains agents for:
    - **Space Invaders:** Achieved a peak score of 155 in evaluation after training for 20 episodes.
    - **CartPole:** Demonstrated basic functionality but requires further tuning for better rewards.

Bugs and Limitations:

1. **Reward Stagnation:**
    - Space Invaders showed fluctuations in rewards but occasionally stagnated due to insufficient replay buffer prioritization.
    - CartPole rewards remained low, indicating the need for simpler algorithms or better hyperparameter tuning.
2. **Training Stability:**
    - Training was occasionally unstable due to insufficient training episodes or suboptimal learning rates.
3. **Performance in CartPole:**
    - Rainbow DQN may be overkill for simpler environments like CartPole.

4. **Third-Party library and code**

Libraries Used:

gymnasium: To set up Atari and CartPole environments.

torch: For deep learning and neural network implementation.

numpy: For numerical operations and data manipulation.

opencv: For preprocessing game frames.

matplotlib: For plotting training rewards and losses.

IPython.display: To render and display gameplay videos.

Third-Party Code:

Some components, such as the Prioritized Replay Buffer and Dueling Network Architecture, were adapted from open-source repositories and official Rainbow DQN papers.

5. Results

Space Invaders (ALE/SpaceInvaders-v5):

- Training: Achieved significant reward improvements, peaking at 720 in episode 16.
- Evaluation: The trained agent demonstrated basic competence, avoiding obstacles and attacking effectively in training of just 20 episodes. These results will be improved with further tuning and longer episode running.

CartPole (CartPole-v1):

- Training: Rewards stagnated around ~10 due to overly complex architecture for the simple environment.
- Evaluation: Demonstrated suboptimal balancing, suggesting the need for further optimization.

6. Future Work

Hyperparameter Tuning:

- Explore different learning rates, batch sizes, and gamma values for improved training stability.
- Optimize replay buffer capacity and alpha for better PER performance.

Environment-Specific Adjustments:

- Use simpler DQN for environments like CartPole.
- Extend training episodes for Space Invaders to achieve higher rewards.

Advanced Visualization:

- Incorporate real-time reward plots during training.
- Analyze Q-values to evaluate action selection.

Extend to More Environments:

- Train the Rainbow DQN on other challenging Atari environments like Pong and Breakout.

Integrate Noisy Networks:

- Replace epsilon-greedy with Noisy Networks for more robust exploration.

Visualisations:

- Gain more insights about the actions by plotting episode vs loss_history and reward_history along with comparison with other DQN networks like Vanilla etc.

Conclusion:

The project successfully demonstrated the potential of Rainbow DQN in Atari environments, achieving reasonable performance in Space Invaders and basic functionality in CartPole. With further optimizations and adjustments, this implementation can serve as a robust framework for reinforcement learning experiments.