Rainbow DQN on Atari Games

Name: Yash Bhalgat

SID: 862465699

Email: ybhal001@ucr.edu

Introduction

This project focuses on implementing the Rainbow algorithm, an advanced reinforcement learning framework that consolidates multiple enhancements to optimize performance in Atari gameplay. Rainbow integrates several cutting-edge techniques, including Double Q-Learning, Prioritized Experience Replay, Dueling Network Architectures, Noisy Nets, Multi-Step Learning, and Distributional Q-Learning. These improvements collectively address the limitations of traditional Deep Q-Networks (DQN), providing a robust and efficient approach to reinforcement learning in complex environments.

```
!pip install gymnasium[atari,accept-rom-license]
```

```
Collecting gymnasium[accept-rom-license,atari]
  Downloading gymnasium-1.0.0-py3-none-any.whl.metadata (9.5 kB)
WARNING: gymnasium 1.0.0 does not provide the extra 'accept-rom-license'
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-rom-license,a
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-rom-lice
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-ro
Collecting farama-notifications>=0.0.1 (from gymnasium[accept-rom-license,atari])
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Collecting ale-py>=0.9 (from gymnasium[accept-rom-license,atari])
  Downloading ale_py-0.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (7.6 kB)
Downloading ale_py-0.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 20.5 MB/s eta 0:00:00
Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Downloading gymnasium-1.0.0-py3-none-any.whl (958 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 958.1/958.1 kB 19.5 MB/s eta 0:00:00
Installing collected packages: farama-notifications, gymnasium, ale-py
Successfully installed ale-py-0.10.1 farama-notifications-0.0.4 gymnasium-1.0.0
```

```python
# Import all the required libraries
import os
import cv2
import random
import numpy as np
import gymnasium as gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import math
```

NoisyLinear Layer

The **NoisyLinear** layer introduces trainable noise into the weights and biases of a neural network during training. This stochasticity enables the agent to explore more effectively in environments where traditional exploration methods like epsilon-greedy might struggle.

Key Features

1. **Trainable Noise**:
   - The layer adds noise sampled from a Gaussian distribution to the weights and biases during the forward pass.
   - This noise adapts during training, allowing the network to balance exploration and exploitation.
2. **Reset Noise**:
   - The `reset_noise` function generates new noise samples for each forward pass, ensuring that exploration is dynamic.
3. **Parameter Initialization**:
   - The weights (`weight_mu`, `weight_sigma`) and biases (`bias_mu`, `bias_sigma`) are initialized with values to ensure stable training.

4. **Training vs. Inference**:
    - During training (`self.training = True`), the layer applies noisy weights and biases.
    - During inference, only the deterministic part of the weights and biases (`weight_mu`, `bias_mu`) is used.

How It Works

- The noise is scaled using a specialized function (`_scale_noise`) that generates random samples with specific statistical properties.
- The forward pass computes a linear transformation of the input using the noisy parameters during training.

This layer is especially useful in **Rainbow DQN** for stochastic exploration, allowing the agent to sample actions with added variability during training. It replaces traditional exploration strategies, making the model more adaptive to complex environments.

```
class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, std_init=0.5):
        super(NoisyLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.std_init = std_init
        self.weight_mu = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.weight_sigma = nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.register_buffer('weight_epsilon', torch.FloatTensor(out_features, in_features))
        self.bias_mu = nn.Parameter(torch.FloatTensor(out_features))
        self.bias_sigma = nn.Parameter(torch.FloatTensor(out_features))
        self.register_buffer('bias_epsilon', torch.FloatTensor(out_features))
        self.reset_parameters()
        self.reset_noise()

    def reset_parameters(self):
        mu_range = 1 / np.sqrt(self.in_features)
        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(self.std_init / np.sqrt(self.in_features))
        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(self.std_init / np.sqrt(self.out_features))

    def reset_noise(self):
        epsilon_in = self._scale_noise(self.in_features)
        epsilon_out = self._scale_noise(self.out_features)
        self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
        self.bias_epsilon.copy_(self._scale_noise(self.out_features))

    def _scale_noise(self, size):
        x = torch.randn(size)
        return x.sign().mul_(x.abs().sqrt_())

    def forward(self, input):
        if self.training:
            return F.linear(input, self.weight_mu + self.weight_sigma * self.weight_epsilon,
                            self.bias_mu + self.bias_sigma * self.bias_epsilon)
        else:
            return F.linear(input, self.weight_mu, self.bias_mu)
```

PrioritizedReplayBuffer

The **PrioritizedReplayBuffer** is an advanced replay buffer implementation that prioritizes transitions based on their importance. Unlike standard replay buffers, this approach focuses on sampling experiences that have higher learning potential, measured by the Temporal Difference (TD) error.

Key Features

1. **Prioritized Sampling**:
    - Transitions are sampled based on their priority, which is proportional to their TD error raised to the power of `alpha`.
    - This ensures that transitions with higher TD errors (indicating greater learning potential) are sampled more frequently.
2. **Importance Sampling Weights**:
    - The weights correct the bias introduced by prioritized sampling to maintain unbiased updates.
    - Controlled by the parameter `beta`, which adjusts the degree of bias correction.
3. **Dynamic Priority Updates**:
    - The `update_priorities` method updates the priority values of specific transitions after they have been used for training.

   - A small constant (`1e-5`) is added to avoid zero priorities, ensuring all transitions remain accessible.

4. **Efficient Buffer Management**:
   - Implements a circular buffer to efficiently manage memory by overwriting old transitions when the capacity is reached.

Key Functions

- `push` : Adds a new transition (state, action, reward, next_state, done) to the buffer. Assigns it the highest priority if it's a new entry.
- `sample` : Samples a batch of transitions based on their priorities. Also returns indices and importance sampling weights.
- `update_priorities` : Updates the priorities of sampled transitions to reflect their latest TD errors.

Parameters

- **`capacity`** : Maximum number of transitions the buffer can hold.
- **`alpha`** : Controls the degree of prioritization. A value of 0 means no prioritization (uniform sampling).
- **`beta`** : Controls the strength of importance sampling corrections. A value of 1 fully corrects the bias.

Advantages

- Focuses training on the most relevant transitions, accelerating convergence.
- Balances exploration and exploitation by dynamically adjusting priorities.

This buffer is essential in reinforcement learning algorithms like **Rainbow DQN**, where prioritizing significant experiences improves learning efficiency.

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.buffer = []
        self.priorities = np.zeros((capacity,), dtype=np.float32)
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        max_priority = self.priorities.max() if self.buffer else 1.0
        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.position] = (state, action, reward, next_state, done)
        self.priorities[self.position] = max_priority
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        if len(self.buffer) == self.capacity:
            priorities = self.priorities
        else:
            priorities = self.priorities[:self.position]

        probs = priorities ** self.alpha
        probs /= probs.sum()

        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]

        weights = (len(self.buffer) * probs[indices]) ** (-self.beta)
        weights /= weights.max()

        return samples, indices, weights

    def update_priorities(self, indices, priorities):
        for idx, priority in zip(indices, priorities):
            self.priorities[idx] = priority + 1e-5  # Add small constant to avoid zero priority
```

RainbowDQN Model

The **RainbowDQN** class implements the neural network architecture for the Rainbow DQN algorithm, combining several enhancements to traditional DQNs. This architecture is designed to handle both 1D state spaces (like CartPole) and image-based state spaces (like Atari games).

Key Features

1. **Feature Extraction**:

- **1D State Spaces**: Uses fully connected layers with noisy linear layers for environments like CartPole.
- **Image-Based State Spaces**: Uses convolutional layers to extract features from image inputs in environments like Atari games.

2. **Dueling Network Architecture**:
   - **Advantage Stream**:
     - Models the relative importance of each action.
     - Outputs `action_dim * num_atoms` logits representing the Q-value distribution for each action.
   - **Value Stream**:
     - Models the baseline value of the current state.
     - Outputs `num_atoms` logits representing the Q-value distribution of the state.
3. **Distributional Q-Learning**:
   - Outputs a probability distribution over Q-values using a fixed number of `num_atoms`.
   - Uses the **Softmax function** to normalize the distribution.
4. **Noisy Networks**:
   - Uses `NoisyLinear` layers in both streams to enable stochastic exploration.
   - The `reset_noise` method resets the noise in the NoisyLinear layers, ensuring dynamic exploration during training.
5. **Dynamic Network Construction**:
   - Automatically adjusts the architecture based on the state dimensions (`state_dim`), supporting both 1D and 2D inputs.

Forward Pass

- The forward method performs the following:
  1. Extracts features from the input state using convolutional or fully connected layers.
  2. Separates the features into:
     - **Advantage Stream**: Computes the relative importance of actions.
     - **Value Stream**: Computes the baseline value of the current state.
  3. Combines the streams using: [Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \right)] This ensures the Q-values are stable and accurate.
  4. Normalizes the output Q-value distribution using `F.softmax`.

Key Parameters

- **`state_dim`**: Dimensions of the state space (1D for CartPole, 3D for image-based states).
- **`action_dim`**: Number of actions available to the agent.
- **`num_atoms`**: Number of discrete atoms representing the Q-value distribution.
- **`v_min` and `v_max`**: Minimum and maximum Q-values for the distributional representation.

Advantages

1. **Combines Multiple Rainbow Enhancements**:
   - Dueling networks, noisy networks, and distributional Q-learning improve training efficiency and stability.
2. **Handles Diverse State Spaces**:
   - Automatically adapts to different types of input spaces (1D and 2D).
3. **Exploration-Exploitation Balance**:
   - Noisy networks replace epsilon-greedy exploration for better scalability in complex environments.

This model forms the core of the Rainbow DQN algorithm, enabling agents to effectively learn policies for complex tasks.

```
class RainbowDQN(nn.Module):
    def __init__(self, state_dim, action_dim, num_atoms, v_min, v_max):
        super(RainbowDQN, self).__init__()
        self.num_atoms = num_atoms
        self.v_min = v_min
        self.v_max = v_max
        self.action_dim = action_dim

        if isinstance(state_dim, int) or len(state_dim) == 1:
            # For 1D state spaces (like CartPole)
            self.features = nn.Sequential(
                NoisyLinear(state_dim[0] if isinstance(state_dim, tuple) else state_dim, 128),
                nn.ReLU(),
                NoisyLinear(128, 128),
                nn.ReLU()
            )
            feature_output = 128
        else:
            # For image-based state spaces (like Atari games)
            c, h, w = state_dim
            self.features = nn.Sequential(
                nn.Conv2d(c, 32, kernel_size=8, stride=4),
                nn.ReLU(),
                nn.Conv2d(32, 64, kernel_size=4, stride=2),
                nn.ReLU(),
                nn.Conv2d(64, 64, kernel_size=3, stride=1),
                nn.ReLU(),
                nn.Flatten()
            )
            feature_output = self._get_conv_output(state_dim)

        self.advantage_stream = nn.Sequential(
            NoisyLinear(feature_output, 512),
            nn.ReLU(),
            NoisyLinear(512, action_dim * num_atoms)
        )

        self.value_stream = nn.Sequential(
            NoisyLinear(feature_output, 512),
            nn.ReLU(),
            NoisyLinear(512, num_atoms)
        )

    def _get_conv_output(self, shape):
        o = self.features(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, state):
        features = self.features(state)
        advantage = self.advantage_stream(features).view(-1, self.action_dim, self.num_atoms)
        value = self.value_stream(features).view(-1, 1, self.num_atoms)
        q_dist = value + advantage - advantage.mean(dim=1, keepdim=True)
        return F.softmax(q_dist, dim=-1)

    def reset_noise(self):
        for module in self.modules():
            if isinstance(module, NoisyLinear):
                module.reset_noise()
```

RainbowAgent Class

The **RainbowAgent** class implements the agent's logic for interacting with the environment and learning from it. This class leverages the **Rainbow DQN** architecture, including advanced features such as multi-step learning, prioritized experience replay, and distributional Q-learning.

Key Features

1. **Initialization**:
    - Sets up the **policy network** and **target network** using the RainbowDQN architecture.
    - Defines hyperparameters such as the learning rate (`lr`), discount factor (`gamma`), and number of atoms for distributional Q-learning (`num_atoms`).
    - Initializes the **Prioritized Replay Buffer** for sampling important experiences.
2. **State Preprocessing**:
    - Converts raw environment states (e.g., RGB Atari frames) into normalized grayscale images with fixed dimensions (84x84).
    - Supports both 1D and image-based state spaces.
3. **Action Selection**:
    - Implements an epsilon-greedy exploration strategy:
        - With a small probability (`epsilon`), the agent chooses a random action.

   - Otherwise, the agent selects the action with the highest expected value based on the Q-value distribution.

4. **Update Step**:
   - Samples a batch of transitions from the prioritized replay buffer.
   - Computes the projected Q-value distribution using the **categorical projection** method.
   - Calculates the loss using the Kullback-Leibler (KL) divergence between the projected and predicted distributions.
   - Optimizes the policy network and updates priorities in the replay buffer.
5. **Categorical Projection**:
   - Projects the Q-value distribution onto the fixed `support` atoms for distributional Q-learning.
   - Adjusts for rewards and terminal states using Bellman updates.
6. **Target Network Updates**:
   - Periodically synchronizes the weights of the target network with the policy network.
   - Uses separate noisy layers in both networks for better exploration during training.

Key Functions

- **`preprocess_state(state)`** :
  - Converts the environment's raw state into a form suitable for the neural network.
  - Normalizes image-based inputs and resizes them to (84x84).
- **`select_action(state, epsilon=0.01)`** :
  - Chooses an action based on the current policy, incorporating exploration via epsilon-greedy.
- **`update()`** :
  - Performs the learning step by:
    1. Sampling transitions from the replay buffer.
    2. Computing the Bellman update and categorical projection.
    3. Optimizing the network using the computed loss.
    4. Updating priorities in the replay buffer.
- **`_categorical_projection(next_q_dist, rewards, dones)`** :
  - Projects the Q-value distribution onto the fixed `support` range, ensuring stable learning with distributional Q-learning.

Key Parameters

- **`state_dim`** : Dimensions of the state space.
- **`action_dim`** : Number of actions available to the agent.
- **`lr`** : Learning rate for the optimizer.
- **`gamma`** : Discount factor for future rewards.
- **`num_atoms`** : Number of discrete atoms representing the Q-value distribution.
- **`v_min`, `v_max`** : Minimum and maximum Q-values for the distributional representation.
- **`buffer_size`** : Maximum capacity of the replay buffer.
- **`batch_size`** : Number of samples drawn for training.

Advantages

1. **Efficient Exploration**:
   - Combines noisy layers with prioritized replay for effective exploration and learning.
2. **Faster Reward Propagation**:
   - Uses categorical projection and multi-step returns for stable and efficient reward updates.
3. **Robust Training**:
   - Prioritized experience replay focuses on transitions with the most learning potential.

This class encapsulates the complete workflow for training and deploying the Rainbow DQN model, making it a critical component of the reinforcement learning pipeline.

```
class RainbowAgent:
    def __init__(self, state_dim, action_dim, lr=0.0001, gamma=0.99, num_atoms=51, v_min=-10, v_max=10, buffer_size=100000, batch
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.action_dim = action_dim
        self.gamma = gamma
        self.num_atoms = num_atoms
        self.v_min = v_min
        self.v_max = v_max
        self.batch_size = batch_size
        self.state_dim = state_dim

        self.policy_net = RainbowDQN(state_dim, action_dim, num_atoms, v_min, v_max).to(self.device)
        self.target_net = RainbowDQN(state_dim, action_dim, num_atoms, v_min, v_max).to(self.device)
        self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
        self.memory = PrioritizedReplayBuffer(buffer_size)

        self.support = torch.linspace(v_min, v_max, num_atoms).to(self.device)
        self.delta_z = (v_max - v_min) / (num_atoms - 1)

        self.state_dim = state_dim

    def preprocess_state(self, state):
        if len(self.state_dim) == 3:
            # For image-based states (like Atari)
            if len(state.shape) == 3 and state.shape[2] == 3:  # If the state is in RGB
                state = cv2.cvtColor(state, cv2.COLOR_RGB2GRAY)
            elif len(state.shape) == 3:  # If the state is already grayscale but has 3 dimensions
                state = state[:, :, 0]
            # Resize the image
            state = cv2.resize(state, (84, 84), interpolation=cv2.INTER_AREA)
            state = np.expand_dims(state, axis=0)  # Add channel dimension
        return state.astype(np.float32) / 255.0  # Normalize the state

    def select_action(self, state, epsilon=0.01):
        if random.random() < epsilon:
            return random.randrange(self.action_dim)
        with torch.no_grad():
            state = self.preprocess_state(state)
            state = torch.FloatTensor(state).to(self.device)
            if state.dim() == 3:
                state = state.unsqueeze(0)  # Add batch dimension if not present
            dist = self.policy_net(state).data.cpu()
            dist = dist * self.support.cpu()
            action = dist.sum(2).max(1)[1].item()
        return action

    def update(self):
        if len(self.memory.buffer) < self.batch_size:
            return

        transitions, indices, weights = self.memory.sample(self.batch_size)
        batch = list(zip(*transitions))

        state_batch = torch.FloatTensor(np.array(batch[0])).to(self.device)
        action_batch = torch.LongTensor(np.array(batch[1])).to(self.device)
        reward_batch = torch.FloatTensor(np.array(batch[2])).to(self.device)
        next_state_batch = torch.FloatTensor(np.array(batch[3])).to(self.device)
        done_batch = torch.FloatTensor(np.array(batch[4])).to(self.device)

        # Compute current Q-values
        current_q_dist = self.policy_net(state_batch)
        current_q_dist = current_q_dist[range(self.batch_size), action_batch]

        # Compute next Q-values
        with torch.no_grad():
            next_q_dist = self.target_net(next_state_batch)
            best_actions = (next_q_dist * self.support).sum(2).max(1)[1]
            next_q_dist = next_q_dist[range(self.batch_size), best_actions]

            # Compute projected distribution
            projected_dist = self._categorical_projection(next_q_dist, reward_batch, done_batch)

        # Compute loss
        loss = -(projected_dist * current_q_dist.log()).sum(1)
        priorities = loss.detach().cpu().numpy()  # This is now an array
        loss = (loss * torch.FloatTensor(weights).to(self.device)).mean()
        # Update priorities
        self.memory.update_priorities(indices, priorities)

        # Optimize the model
        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 10)
        self.optimizer.step()

        # Reset noisy layers
        self.policy_net.reset_noise()
        self.target_net.reset_noise()

    def _categorical_projection(self, next_q_dist, rewards, dones):
        batch_size = len(rewards)
        projected_dist = torch.zeros(batch_size, self.num_atoms).to(self.device)

        rewards = rewards.unsqueeze(1).expand_as(projected_dist)
        dones = dones.unsqueeze(1).expand_as(projected_dist)
        support = self.support.unsqueeze(0).expand_as(projected_dist)

        tz = rewards + (1 - dones) * self.gamma * support
        tz = tz.clamp(min=self.v_min, max=self.v_max)
        b = (tz - self.v_min) / self.delta_z
        l = b.floor().long()
        u = b.ceil().long()

        l[(u > 0) * (l == u)] -= 1
        u[(l < (self.num_atoms - 1)) * (l == u)] += 1

        offset = torch.linspace(0, (batch_size - 1) * self.num_atoms, batch_size).long().unsqueeze(1).expand(batch_size, self.num

        projected_dist.view(-1).index_add_(0, (l + offset).view(-1), (next_q_dist * (u.float() - b)).view(-1))
        projected_dist.view(-1).index_add_(0, (u + offset).view(-1), (next_q_dist * (b - l.float())).view(-1))

        return projected_dist
```

Training Function: `train`

The **`train`** function orchestrates the training process for the Rainbow DQN agent in a given environment. It handles environment interaction, model updates, and periodic model saving. This function is designed to work with both discrete action spaces (e.g., Atari games) and 1D state spaces (e.g., CartPole).

Function Parameters

- **`env_name`**: The name of the environment (e.g., "CartPole-v1", "SpaceInvaders-v5").
- **`num_episodes`**: Total number of episodes for training.
- **`save_interval`**: Interval (in episodes) at which the model is saved.
- **`save_dir`**: Directory path where the models will be saved.

Function Workflow

1. **Environment Initialization**:
    - Creates the environment using `gym.make(env_name)`.
    - Determines the state dimension:
        - For image-based environments (e.g., Atari games), state is resized to `(1, 84, 84)` during preprocessing.
        - For simpler environments (e.g., CartPole), state dimensions are directly taken from the environment's observation space.
2. **Agent Initialization**:
    - Initializes the RainbowAgent with the state and action dimensions derived from the environment.
3. **Training Loop**:
    - For each episode:
        1. Resets the environment and preprocesses the initial state.
        2. Interacts with the environment:
            - Chooses an action using the policy network.
            - Steps in the environment and observes the next state, reward, and termination signal.

        - Stores the transition in the replay buffer.
        - Updates the policy network using samples from the replay buffer.
    3. Updates the target network every 10 episodes to stabilize learning.

4. **Reward Logging**:
    - Logs the total reward for each episode, providing feedback on the agent's performance.
5. **Model Saving**:
    - Saves the policy network's weights periodically based on the `save_interval`.
    - Saves the final model after training is complete.
6. **Environment Cleanup**:
    - Closes the environment using `env.close()` to release resources.

Key Features

- **State Preprocessing**:
    - Converts raw environment states into formats suitable for the neural network, ensuring consistency in input dimensions.
- **Target Network Updates**:
    - Synchronizes the target network with the policy network every 10 episodes, which helps stabilize Q-value updates.
- **Periodic Model Saving**:
    - Saves the model at regular intervals and at the end of training, allowing for easy evaluation or resumption of training.

Advantages

1. **Generalized Design**:
    - Supports both 1D state spaces and image-based environments, making it versatile across different types of tasks.
2. **Efficient Training**:
    - Integrates replay buffer updates and target network synchronization for stable learning.
3. **Progress Monitoring**:
    - Logs rewards for each episode, enabling tracking of the agent's performance over time.

Usage Example

```
train(env_name="CartPole-v1", num_episodes=500, save_interval=50, save_dir="models")
```

```
def train(env_name, num_episodes=1000, save_interval=100, save_dir='saved_models'):
    env = gym.make(env_name)

    # Determining the state dimension based on the environment
    if len(env.observation_space.shape) == 3:
        state_dim = (1, 84, 84)  # We'll preprocess to this size for Atari games
    else:
        state_dim = env.observation_space.shape

    action_dim = env.action_space.n

    agent = RainbowAgent(state_dim, action_dim)

    # Creating directory for saving models if it doesn't exist
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    for episode in range(num_episodes):
        state, _ = env.reset()
        state = agent.preprocess_state(state)
        done = False
        total_reward = 0

        while not done:
            action = agent.select_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            next_state = agent.preprocess_state(next_state)
            done = terminated or truncated
            total_reward += reward

            agent.memory.push(state, action, reward, next_state, done)
            agent.update()

            state = next_state

        if episode % 10 == 0:
            agent.target_net.load_state_dict(agent.policy_net.state_dict())

        print(f"Episode {episode}, Total Reward: {total_reward}")

        # Save model periodically
        if (episode + 1) % save_interval == 0:
            save_path = os.path.join(save_dir, f"{env_name}_rainbow_episode_{episode+1}.pth")
            torch.save(agent.policy_net.state_dict(), save_path)
            print(f"Model saved to {save_path}")

    # Save final model
    final_save_path = os.path.join(save_dir, f"{env_name}_rainbow_final.pth")
    torch.save(agent.policy_net.state_dict(), final_save_path)
    print(f"Final model saved to {final_save_path}")

    env.close()


# Test on CartPole
print("Training on CartPole-v1")
train("CartPole-v1", num_episodes=100)
```

```
Training on CartPole-v1
Episode 0, Total Reward: 10.0
Episode 1, Total Reward: 8.0
Episode 2, Total Reward: 10.0
Episode 3, Total Reward: 12.0
Episode 4, Total Reward: 10.0
Episode 5, Total Reward: 9.0
Episode 6, Total Reward: 8.0
Episode 7, Total Reward: 10.0
Episode 8, Total Reward: 8.0
Episode 9, Total Reward: 8.0
Episode 10, Total Reward: 11.0
Episode 11, Total Reward: 10.0
Episode 12, Total Reward: 10.0
Episode 13, Total Reward: 9.0
Episode 14, Total Reward: 9.0
Episode 15, Total Reward: 10.0
Episode 16, Total Reward: 9.0
Episode 17, Total Reward: 9.0
Episode 18, Total Reward: 10.0
Episode 19, Total Reward: 9.0
Episode 20, Total Reward: 9.0
Episode 21, Total Reward: 10.0
Episode 22, Total Reward: 8.0
Episode 23, Total Reward: 9.0
Episode 24, Total Reward: 10.0
Episode 25, Total Reward: 9.0
Episode 26, Total Reward: 9.0
Episode 27, Total Reward: 9.0
Episode 28, Total Reward: 9.0
Episode 29, Total Reward: 10.0
Episode 30, Total Reward: 10.0
Episode 31, Total Reward: 9.0
Episode 32, Total Reward: 9.0
Episode 33, Total Reward: 10.0
Episode 34, Total Reward: 10.0
Episode 35, Total Reward: 9.0
Episode 36, Total Reward: 11.0
Episode 37, Total Reward: 9.0
Episode 38, Total Reward: 9.0
Episode 39, Total Reward: 12.0
Episode 40, Total Reward: 8.0
Episode 41, Total Reward: 10.0
Episode 42, Total Reward: 9.0
Episode 43, Total Reward: 8.0
Episode 44, Total Reward: 9.0
Episode 45, Total Reward: 9.0
Episode 46, Total Reward: 10.0
Episode 47, Total Reward: 8.0
Episode 48, Total Reward: 8.0
Episode 49, Total Reward: 9.0
Episode 50, Total Reward: 9.0
Episode 51, Total Reward: 9.0
Episode 52, Total Reward: 9.0
Episode 53, Total Reward: 10.0
Episode 54, Total Reward: 9.0
Episode 55, Total Reward: 10.0
Episode 56, Total Reward: 11.0
```

```
import gymnasium as gym

# Check available environments
envs = sorted(env_spec.id for env_spec in gym.envs.registry.values())
print("Available Environments:", envs)

# Check for Pong
print("Is ALE/SpaceInvaders-v5 available?", "ALE/SpaceInvaders-v5" in envs)
```

```
Available Environments: ['Acrobot-v1', 'Ant-v2', 'Ant-v3', 'Ant-v4', 'Ant-v5', 'BipedalWalker-v3', 'BipedalWalkerHardcore-v3
Is ALE/SpaceInvaders-v5 available? False
```

```
!pip install gymnasium[atari,accept-rom-license]
!pip install ale-py==0.10.1
```

```
Requirement already satisfied: gymnasium[accept-rom-license,atari] in /usr/local/lib/python3.10/dist-packages (1.0.0)
WARNING: gymnasium 1.0.0 does not provide the extra 'accept-rom-license'
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-rom-license,a
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-rom-lice
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-ro
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept
Requirement already satisfied: ale-py>=0.9 in /usr/local/lib/python3.10/dist-packages (from gymnasium[accept-rom-license,ata
Requirement already satisfied: ale-py==0.10.1 in /usr/local/lib/python3.10/dist-packages (0.10.1)
Requirement already satisfied: numpy>1.20 in /usr/local/lib/python3.10/dist-packages (from ale-py==0.10.1) (1.26.4)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from ale-py==0.10.1) (4.12.2)
```

```
!wget http://www.atarimania.com/roms/Roms.rar
!unrar x Roms.rar ./
```

```
Streaming output truncated to the last 5000 lines.
Extracting  ./ROMS/Bank Heist (Unknown) (PAL) [a].bin                    OK
Extracting  ./ROMS/Bank Heist (Unknown) (PAL).bin                        OK
Extracting  ./ROMS/Barnstorming (1982) (Activision, Steve Cartwright) (AX-013) ~.bin  OK
Extracting  ./ROMS/Barnstorming (CCE).bin                                OK
Extracting  ./ROMS/Barnstorming (Unknown) (PAL) [a].bin                  OK
Extracting  ./ROMS/Barnstorming (Unknown) (PAL).bin                      OK
Extracting  ./ROMS/Barnstorming - Die tollkeuhnen Flieger (1982) (Activision, Steve Cartwright - Ariola) (EAX-013, PAX-013 -
Extracting  ./ROMS/Base Attack (1983) (Home Vision - Gem International Corporation - VDI) (VCS83113) (PAL) ~.bin  OK
Extracting  ./ROMS/Base Attack (Hack) (Unknown).bin                      OK
Extracting  ./ROMS/Baseball (AKA Super Challenge Baseball) (1988) (Telegames) (5665 A016) (PAL).bin  OK
Extracting  ./ROMS/Baseball (AKA Super Challenge Baseball) (1988) (Telegames) (5665 A016).bin  OK
Extracting  ./ROMS/Basic Math (Math Pack) (1977) (Atari, Gary Palmer) (CX2661) (PAL).bin  OK
Extracting  ./ROMS/Basic Math (Unknown) (PAL).bin                        OK
Extracting  ./ROMS/Basic Math - Math (Math Pack) (1977) (Atari, Gary Palmer - Sears) (CX2661 - 99808, 6-99808) ~.bin  OK
Extracting  ./ROMS/BASIC Programming (Keyboard Controller) (1979) (Atari, Warren Robinett) (CX2620) ~.bin  OK
Extracting  ./ROMS/BASIC Programming (Keyboard Controller) (1979) (Atari, Warren Robinett) (CX2620, CX2620P) (PAL).bin  OK
Extracting  ./ROMS/Basketball (1978) (Atari, Alan Miller - Sears) (CX2624 - 6-99826, 49-75113) ~.bin  OK
Extracting  ./ROMS/Basketball (1978) (Atari, Alan Miller) (CX2624, CX2624P) (PAL).bin  OK
Extracting  ./ROMS/Basketball (32 in 1) (1988) (Atari, Alan Miller) (CX26163P) (PAL).bin  OK
Extracting  ./ROMS/Basketball (Hack) (32 in 1) (Bit Corporation) (R320).bin  OK
Extracting  ./ROMS/Basketball (Hack) (Unknown) (PAL).bin                 OK
Extracting  ./ROMS/Basketball (Unknown) (PAL).bin                        OK
Extracting  ./ROMS/Battlezone (05-02-1983) (Atari - GCC, Michael Feinstein, Patricia Goodson, Brad Rice) (CX2681) (Prototype
Extracting  ./ROMS/Battlezone (05-12-1983) (Atari - GCC, Michael Feinstein, Patricia Goodson, Brad Rice) (CX2681) (Prototype
Extracting  ./ROMS/Battlezone (1983) (Atari - GCC, Michael Feinstein, Patricia Goodson, Brad Rice) (CX2681) ~.bin  OK
Extracting  ./ROMS/Battlezone (1983) (Atari - GCC, Michael Feinstein, Patricia Goodson, Brad Rice) (CX2681, CX2681P) (PAL).b
Extracting  ./ROMS/Beamrider (1984) (Activision - Cheshire Engineering, David Rolfe, Larry Zwick) (AZ-037-04) ~.bin  OK
Extracting  ./ROMS/Beamrider (1984) (Activision - Cheshire Engineering, David Rolfe, Larry Zwick) (EAZ-037-04, EAZ-037-04I)
Extracting  ./ROMS/Beany Bopper (1982) (20th Century Fox Video Games - Sirius Software, Grady Ward) (11002) ~.bin  OK
Extracting  ./ROMS/Beany Bopper (1983) (CCE) (C-835).bin                 OK
Extracting  ./ROMS/Bear Game Demo (Paddle) (1983) (SEGA, Fred Mack) ~.bin  OK
Extracting  ./ROMS/Beat 'Em & Eat 'Em (Paddle) (1982) (Mystique - American Multiple Industries, Joel H. Martin) (1003) ~.bin
Extracting  ./ROMS/Beat 'Em & Eat 'Em (Paddle) (1982) (Mystique - American Multiple Industries, Joel H. Martin) (PAL).bin  O
Extracting  ./ROMS/Beat 'Em & Eat 'Em (Paddle) (1982) (PlayAround - JHM) (204).bin  OK
Extracting  ./ROMS/Beat 'Em & Eat 'Em (Paddle) (1983) (Dynacom).bin      OK
Extracting  ./ROMS/Berenstain Bears (Kid Vid Voice Module) (1983) (Coleco) (2658) ~.bin  OK
Extracting  ./ROMS/Bermuda (AKA River Raid) (1983) (Quelle) (322.913 5) (PAL).bin  OK
Extracting  ./ROMS/Bermuda (AKA River Raid) (Unknown) (PAL).bin          OK
Extracting  ./ROMS/Bermuda (AKA River Raid) (Unknown).bin                OK
Extracting  ./ROMS/Bermuda Triangle (1983) (Data Age, J. Ray Dettling) (112-007) ~.bin  OK
Extracting  ./ROMS/Bermuda Triangle (1983) (Gameworld, J. Ray Dettling) (133-007) (PAL).bin  OK
Extracting  ./ROMS/Bermuda, The (AKA River Raid) (1983) (Rainbow Vision - Suntek) (SS-009) (PAL) [a].bin  OK
Extracting  ./ROMS/Bermuda, The (AKA River Raid) (1983) (Rainbow Vision - Suntek) (SS-009) (PAL).bin  OK
Extracting  ./ROMS/Berzerk (1982) (Atari, Dan Hitchens - Sears) (CX2650 - 49-75168) ~.bin  OK
Extracting  ./ROMS/Berzerk (1982) (Atari, Dan Hitchens) (CX2650) (PAL).bin  OK
Extracting  ./ROMS/Berzerk (CCE).bin                                  OK
Extracting  ./ROMS/Berzerk (Unknown) (PAL) [a].bin                    OK
Extracting  ./ROMS/Berzerk (Unknown) (PAL).bin                        OK
Extracting  ./ROMS/Bi! Bi! (AKA Skindiver) (1983) (Rainbow Vision - Suntek) (SS-013) (PAL).bin  OK
Extracting  ./ROMS/Bi! Bi! (AKA Skindiver) (2600 Screen Search Console) (Jone Yuan Telephonic Enterprise Co) (PAL).bin  OK
Extracting  ./ROMS/Big Bird's Egg Catch (Grover's Egg Catch) (Kid's Controller) (05-02-1983) (Atari - CCW, Christopher H. Om
Extracting  ./ROMS/Big Bird's Egg Catch (Grover's Egg Catch) (Kid's Controller) (05-17-1983) (Atari - CCW, Christopher H. Om
Extracting  ./ROMS/Big Bird's Egg Catch (Grover's Egg Catch) (Kid's Controller) (12-08-1982) (Atari - CCW, Christopher H. Om
Extracting  ./ROMS/Big Bird's Egg Catch (Grover's Egg Catch) (Kid's Controller) (1983) (Atari - CCW, Christopher H. Omarzu)
Extracting  ./ROMS/Big Bird's Egg Catch (Grover's Egg Catch) (Kid's Controller) (1983) (Atari - CCW, Christopher H. Omarzu)
Extracting  ./ROMS/Billard (AKA Trick Shot) (1983) (Quelle) (626.610 0) (PAL).bin  OK
```

```
import ale_py
print(ale_py.__version__)
```

```
0.10.1
```

```
import ale_py
print(ale_py.__file__)
```

```
/usr/local/lib/python3.10/dist-packages/ale_py/__init__.py
```

```
import os
import shutil

# Source directory where ROMs were extracted
source_roms_path = "./ROMS"  # Update if your ROMs are elsewhere

# Destination directory for ale-py ROMs (replace with the actual path if necessary)
ale_roms_path = os.path.join(os.path.dirname(ale_py.__file__), "roms")

# Ensure the directory exists
os.makedirs(ale_roms_path, exist_ok=True)

# Copy the ROMs into ale-py directory
for file_name in os.listdir(source_roms_path):
    full_file_name = os.path.join(source_roms_path, file_name)
    if os.path.isfile(full_file_name):
        shutil.copy(full_file_name, ale_roms_path)

print(f"ROMs successfully copied to {ale_roms_path}")
```

```
ROMs successfully copied to /usr/local/lib/python3.10/dist-packages/ale_py/roms
```

```
import gymnasium as gym
print("Is ALE/SpaceInvaders-v5 available?", "ALE/SpaceInvaders-v5" in gym.envs.registry)
```

```
Is ALE/SpaceInvaders-v5 available? True
```

```
# Test on Space Invaders
print("\nTraining on SpaceInvaders-v4")
train("ALE/SpaceInvaders-v5", num_episodes=20)
```

```

Training on SpaceInvaders-v4
Episode 0, Total Reward: 440.0
Episode 1, Total Reward: 245.0
Episode 2, Total Reward: 165.0
Episode 3, Total Reward: 135.0
Episode 4, Total Reward: 200.0
Episode 5, Total Reward: 120.0
Episode 6, Total Reward: 180.0
Episode 7, Total Reward: 285.0
Episode 8, Total Reward: 285.0
Episode 9, Total Reward: 285.0
Episode 10, Total Reward: 285.0
Episode 11, Total Reward: 230.0
Episode 12, Total Reward: 55.0
Episode 13, Total Reward: 380.0
Episode 14, Total Reward: 300.0
Episode 15, Total Reward: 195.0
Episode 16, Total Reward: 70.0
Episode 17, Total Reward: 595.0
Episode 18, Total Reward: 280.0
Episode 19, Total Reward: 315.0
```

```
Final model saved to saved_models/ALE/SpaceInvaders-v5_rainbow_final.pth
```

```
from gymnasium.wrappers import RecordVideo
import torch

def inference(env_name, model_path, num_episodes=10, render=True):
    env = gym.make(env_name, render_mode="rgb_array")
    env = RecordVideo(env, video_folder="./videos", episode_trigger=lambda ep: True)

    # Determining the state dimension based on the environment
    if len(env.observation_space.shape) == 3:
        state_dim = (1, 84, 84)  # Preprocessed image size for Atari games
    else:
        state_dim = env.observation_space.shape

    action_dim = env.action_space.n

    # Create and load the agent
    agent = RainbowAgent(state_dim, action_dim)
    agent.policy_net.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))  # Map to CPU
    agent.policy_net.eval()  # Set the network to evaluation mode

    for episode in range(num_episodes):
        state, _ = env.reset()
        done = False
        total_reward = 0

        while not done:
            if render:
                # Render the frame (only saves when using RecordVideo)
                frame = env.render()

            action = agent.select_action(state, epsilon=0)  # Use epsilon=0 for greedy action selection
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            total_reward += reward

            state = next_state

        print(f"Episode {episode}, Total Reward: {total_reward}")

    env.close()


save_dir = 'saved_models'


# Inference for CartPole
env_name = "CartPole-v1"
model_path = os.path.join(save_dir, f"{env_name}_rainbow_final.pth")
print("Inferencing CartPole-v1:")
inference(env_name, model_path, num_episodes=10)
```

```
Inferencing CartPole-v1:
<ipython-input-17-061cec8a9369>:18: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default
  agent.policy_net.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))  # Map to CPU
Episode 0, Total Reward: 9.0
Episode 1, Total Reward: 10.0
Episode 2, Total Reward: 10.0
Episode 3, Total Reward: 9.0
Episode 4, Total Reward: 10.0
Episode 5, Total Reward: 9.0
Episode 6, Total Reward: 10.0
Episode 7, Total Reward: 10.0
Episode 8, Total Reward: 9.0
Episode 9, Total Reward: 8.0
```

```
save_dir = 'saved_models'
env_name = "ALE/SpaceInvaders-v5"
model_path = os.path.join(save_dir, f"{env_name}_rainbow_final.pth")

print("\nInferencing SpaceInvaders-v5:")
inference(env_name, model_path, num_episodes=20, render=True)
```

```
Inferencing SpaceInvaders-v5:
/usr/local/lib/python3.10/dist-packages/gymnasium/wrappers/rendering.py:283: UserWarning: WARN: Overwriting existing videos
  logger.warn(
<ipython-input-17-061cec8a9369>:18: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default
  agent.policy_net.load_state_dict(torch.load(model_path, map_location=torch.device('cpu')))  # Map to CPU
Episode 0, Total Reward: 310.0
Episode 1, Total Reward: 310.0
Episode 2, Total Reward: 190.0
Episode 3, Total Reward: 435.0
Episode 4, Total Reward: 375.0
Episode 5, Total Reward: 225.0
Episode 6, Total Reward: 200.0
Episode 7, Total Reward: 225.0
Episode 8, Total Reward: 375.0
Episode 9, Total Reward: 560.0
Episode 10, Total Reward: 370.0
Episode 11, Total Reward: 225.0
Episode 12, Total Reward: 225.0
Episode 13, Total Reward: 225.0
Episode 14, Total Reward: 225.0
Episode 15, Total Reward: 225.0
Episode 16, Total Reward: 720.0
Episode 17, Total Reward: 185.0
Episode 18, Total Reward: 380.0
Episode 19, Total Reward: 225.0
```

```
import glob
from IPython.display import HTML
from base64 import b64encode

# Find the latest video file
video_path = glob.glob("./videos/*.mp4")[0]

# Display the video
with open(video_path, "rb") as video_file:
    video_data = b64encode(video_file.read()).decode("utf-8")
HTML(f'<video width="600" controls><source src="data:video/mp4;base64,{video_data}" type="video/mp4"></video>')
```

0:00 / 0:25

```
import shutil

folder_to_download = 'saved_models'  # Replace with your folder's name
shutil.make_archive(folder_to_download, 'zip', folder_to_download)
```

```
'/content/saved_models.zip'
```

```
folder_to_download = 'videos'  # Replace with your folder's name
shutil.make_archive(folder_to_download, 'zip', folder_to_download)
```

```
'/content/videos.zip'
```

```
Start coding or generate with AI.
```