```python
# Source: https://github.com/vwxyzjn/cleanrl/blob/master/cleanrl/dqn_atari.py
# docs and experiment results can be found at https://docs.cleanrl.dev/rl-algorithms/dqn/#dqn_ataripy

import os
import random
import time
from dataclasses import dataclass

import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import tyro
from stable_baselines3.common.atari_wrappers import (
    ClipRewardEnv,
    EpisodicLifeEnv,
    FireResetEnv,
    MaxAndSkipEnv,
    NoopResetEnv,
)
from stable_baselines3.common.buffers import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter


##########################################
# Instantiate a config object (for tyro) #
##########################################

@dataclass
class Args:
    exp_name: str = os.path.basename(__file__)[: -len(".py")]
    """the name of this experiment"""
    seed: int = 1
    """seed of the experiment"""
    torch_deterministic: bool = True
    """if toggled, `torch.backends.cudnn.deterministic=False`"""
    cuda: bool = True
    """if toggled, cuda will be enabled by default"""
    track: bool = False
    """if toggled, this experiment will be tracked with Weights and Biases"""
    wandb_project_name: str = "cleanRL"
    """the wandb's project name"""
    wandb_entity: str = None
    """the entity (team) of wandb's project"""
    capture_video: bool = False
    """whether to capture videos of the agent performances (check out `videos` folder)"""
    save_model: bool = False
    """whether to save model into the `runs/{run_name}` folder"""
    upload_model: bool = False
    """whether to upload the saved model to huggingface"""
    hf_entity: str = ""
    """the user or org name of the model repository from the Hugging Face Hub"""

    # Algorithm specific arguments
    env_id: str = "BreakoutNoFrameskip-v4"
    """the id of the environment"""
    total_timesteps: int = 10000000
    """total timesteps of the experiments"""
    learning_rate: float = 1e-4
    """the learning rate of the optimizer"""
    num_envs: int = 1
    """the number of parallel game environments"""
    buffer_size: int = 1000000
    """the replay memory buffer size"""
    gamma: float = 0.99
    """the discount factor gamma"""
    tau: float = 1.0
    """the target network update rate"""
    target_network_frequency: int = 1000
    """the timesteps it takes to update the target network"""
    batch_size: int = 32
    """the batch size of sample from the replay memory"""
    start_e: float = 1
    """the starting epsilon for exploration"""
    end_e: float = 0.01
    """the ending epsilon for exploration"""
    exploration_fraction: float = 0.10
    """the fraction of `total-timesteps` it takes from start-e to go end-e"""
    learning_starts: int = 80000
```

```
    """timestep to start learning"""
    train_frequency: int = 4
    """the frequency of training"""


def make_env(env_id, seed, idx, capture_video, run_name):
  """
  Make environment

  Args:
    env_id (str): ID of the environment (refer to the environment documentations). \n
    seed (int): Seed in generating the environment. \n
    idx (int) : Index of the environment. \n
    capture_video (boolean): A flag that decides whether to create an environment that will record episodes. \n
    run_name (str): The name of the folder where the recorded videos will be saved to. \n

  Returns:
    thunk (func): A function that returns a created environments.
  """
    def thunk():
        # Create an environment with render_mode
        # if capture_video is True and only for the first environment
        # Record video: Record video intermittently at episode intervals
        #(https://gymnasium.farama.org/api/wrappers/misc_wrappers/#gymnasium.wrappers.RecordVideo)
        if capture_video and idx == 0:
            env = gym.make(env_id, render_mode="rgb_array")
            env = gym.wrappers.RecordVideo(env, f"videos/{run_name}")
        else:
            env = gym.make(env_id)

        # Enable the environment to keep track of cumulative rewards and episode lengths.
        # Save into "info" at the end of the episodes
        # https://gymnasium.farama.org/api/wrappers/misc_wrappers/#gymnasium.wrappers.RecordEpisodeStatistics
        env = gym.wrappers.RecordEpisodeStatistics(env)

        # Instead of starting games immediately for each episodes
        # The env sample a few random number of "no-op" as a way to introduce randomness.
        env = NoopResetEnv(env, noop_max=30)

        # Return only every skip-th frame (frameskipping)
        # And return the max between the two last frames.
        # https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html#stable_baselines3.common.atari_wrappers.MaxA
        env = MaxAndSkipEnv(env, skip=4)

        # Make end-of-life == end-of-episode, but only reset on true game over.
        # https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html#stable_baselines3.common.atari_wrappers.Epis
        env = EpisodicLifeEnv(env)

        # Used for Atari environments that remain static until a "FIRE" action is taken.
        # Without this, the environment remains static until the agent takes FIRE actions.
        # As a result, the agent may spend many timestep at the beginning with a static environment until it fires.
        # Which lead to wasting time steps.
        # With this, the environment reset as the agent fires. Therefore, the agent are placed in meaningful states without wasting
        # https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html#stable_baselines3.common.atari_wrappers.Fire
        if "FIRE" in env.unwrapped.get_action_meanings():
            env = FireResetEnv(env)

        # Clip the reward to {+1, 0, -1} by its sign.
        # Simplifying rewards helps the agent focus on the direction of improvement (good/bad/neutral) rather than the exact magnitu
        # https://stable-baselines3.readthedocs.io/en/master/common/atari_wrappers.html#stable_baselines3.common.atari_wrappers.Clip
        env = ClipRewardEnv(env)

        # Reduce state information - Resize observations to a smaller size
        env = gym.wrappers.ResizeObservation(env, (84, 84))

        # Reduce state information - Convert observation into grayscale
        env = gym.wrappers.GrayScaleObservation(env)

        # Return 4 frame as 1 state - allowing overcoming temporal limitations
        # Essentially means that the observation or state that it will be returned in the shape of (4, H, W)
        # 4 because each frame is grayscale, and only has a channel of 1
        env = gym.wrappers.FrameStack(env, 4)

        # Set the seed for the action space
        env.action_space.seed(seed)

        return env

    return thunk


# ALGO LOGIC: initialize agent here:
class QNetwork(nn.Module):
```

```
    """
    A Q-Learning Network.

    Args:
      env (gym.Env): An environment.
    """
    def __init__(self, env):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(4, 32, 8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, 4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, env.single_action_space.n),
        )

    def forward(self, x):
        """
        Forward propagation for the Q-Learning Network.

        Args:
          x (float tensor): An observation or state, expected in the shape of (B,C,H,W), in the range of 0 - 255

        Returns:
          out (float tensor): Probability of taking each actions, in the shape of (B, n) where n is the number of unique actions,
        """
        return self.network(x / 255.0)


def linear_schedule(start_e: float, end_e: float, duration: int, t: int):
    """
    A linear scheduler that reduces the decaying of epsilon for controlling exploration / exploitation.

    Args:
      start_e (float): Starting epsilon. \n
      end_e (float): Ending epsilon. \n
      duration (int): Total number of episodes.
      t (int): Current number of episodes.

    Returns:
      epsilon (float): The epsilon for the current number of episode.

    """
    slope = (end_e - start_e) / duration
    return max(slope * t + start_e, end_e)


if __name__ == "__main__":
    import stable_baselines3 as sb3

    if sb3.__version__ < "2.0":
        raise ValueError(
            """Ongoing migration: run the following command to install the new dependencies:

poetry run pip install "stable_baselines3==2.0.0a1" "gymnasium[atari,accept-rom-license]==0.28.1"  "ale-py==0.8.1"
"""
        )

    ##################################################
    # Obtain the arguments with the help of tyro #
    ##################################################
    args = tyro.cli(Args)

    # Only allow 1 environment
    assert args.num_envs == 1, "vectorized envs are not supported at the moment"

    # Create run names
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"

    # Setting up wandb
    if args.track:
        import wandb

        wandb.init(
            project=args.wandb_project_name,
            entity=args.wandb_entity,
            sync_tensorboard=True,
            config=vars(args),
```

```
            config=vars(args),
            name=run_name,
            monitor_gym=True,
            save_code=True,
        )
    writer = SummaryWriter(f"runs/{run_name}")
    writer.add_text(
        "hyperparameters",
        "|param|value|\n|-|-|\n%s" % ("\n".join([f"|{key}|{value}|" for key, value in vars(args).items()])),
    )

    # TRY NOT TO MODIFY: seeding
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = args.torch_deterministic

    device = torch.device("cuda" if torch.cuda.is_available() and args.cuda else "cpu")

    # env setup
    # With SyncVectorEnv - the environment reset once truncated or terminated
    # This was done under the hood and not need explicitly coded.
    envs = gym.vector.SyncVectorEnv(
        [make_env(args.env_id, args.seed + i, i, args.capture_video, run_name) for i in range(args.num_envs)]
    )

    # Check if the action_space are discrete
    assert isinstance(envs.single_action_space, gym.spaces.Discrete), "only discrete action space is supported"

    # Create q_network
    q_network = QNetwork(envs).to(device)

    # Create optimizer
    optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)

    # Create a copy of the q_network with the same weight and bias
    target_network = QNetwork(envs).to(device)
    target_network.load_state_dict(q_network.state_dict())

    # Create a Replay Buffer
    rb = ReplayBuffer(
        args.buffer_size,
        envs.single_observation_space,
        envs.single_action_space,
        device,
        optimize_memory_usage=True,
        handle_timeout_termination=False,
    )
    start_time = time.time()

    # TRY NOT TO MODIFY: start the game
    obs, _ = envs.reset(seed=args.seed)
    for global_step in range(args.total_timesteps):
        # ALGO LOGIC: put action logic here

        # Update epsilon
        epsilon = linear_schedule(args.start_e, args.end_e, args.exploration_fraction * args.total_timesteps, global_step)

        # Sample an action based on epsilon
        # Exploration
        if random.random() < epsilon:
            actions = np.array([envs.single_action_space.sample() for _ in range(envs.num_envs)])

        # Exploitation
        else:
            q_values = q_network(torch.Tensor(obs).to(device))
            actions = torch.argmax(q_values, dim=1).cpu().numpy()

        # TRY NOT TO MODIFY: execute the game and log data.
        # Take a step in that direction
        next_obs, rewards, terminations, truncations, infos = envs.step(actions)

        # TRY NOT TO MODIFY: record rewards for plotting purposes
        # for wandb
        if "final_info" in infos:
            for info in infos["final_info"]:
                if info and "episode" in info:
                    print(f"global_step={global_step}, episodic_return={info['episode']['r']}")
                    writer.add_scalar("charts/episodic_return", info["episode"]["r"], global_step)
                    writer.add_scalar("charts/episodic_length", info["episode"]["l"], global_step)

        # TRY NOT TO MODIFY: save data to reply buffer; handle `final_observation`
        real_next_obs = next_obs.copy()
```

```
        # When the episode ends with truncation, the next_obs will not be the real next observation [I assume its because its reset
        # Instead, it will be stored in infos["final_observation"]
        # The reason why infos['final_observation'][idx] is used is because it is assuming a vectorised environment

        # No need to perform for the termination state
        # Because it is still in a meaningful state even after termination (this is before reset)

        # In short
        # terminated -> in a meaningful state (basically means at the destination) -> then only get reset
        # truncated -> in a random state (not really meaningful, like step into a wall) -> then only get reset
        # Therefore, only for truncation that we need to find out and replace the true and meaningful next state
        for idx, trunc in enumerate(truncations):
            if trunc:
                real_next_obs[idx] = infos["final_observation"][idx]

        # Add the information to the replay buffer
        # When adding these information into the buffer,
        # For instance if obs has the shape of (num_envs, 4, 32, 32), it will be broken into (4, 32, 32) x num_envs
        # Then later on, when called from replay buffer, it will return (Batch_size, 4, 32, 32)
        rb.add(obs, real_next_obs, actions, rewards, terminations, infos)

        # TRY NOT TO MODIFY: CRUCIAL step easy to overlook
        # ignore the case if terminaed or truncated, i am assuming due to SyncVecEnv, it gets reset in other ways later on
        obs = next_obs

        # ALGO LOGIC: training.
        # First check if the number of step is already more than the learning start steps
        if global_step > args.learning_starts:

            # Check if the step are divisible by the frequency to decide if gradient descent is performed
            if global_step % args.train_frequency == 0:

                # Sample data up to the batch size
                data = rb.sample(args.batch_size)

                # Use the target network to find the next actions based on the recorded next observation / state
                with torch.no_grad():

                    # returns 2 variables because of using .max
                    # data.next_observations is in the shape of (B, C, H, W) where B is the number of batch
                    # target_network(data.next_observations) return (B, n)
                    # target_network(data.next_observations).max(dim=1) returns the target_max (B,) and its corresponding indices (B
                    target_max, _ = target_network(data.next_observations).max(dim=1)

                    # Calculatet the td target
                    # td_target = r + gamma * max(target_network(s', a'), dim = a')
                    # data.dones is the termination flag
                    # if termination - data.dones = 1, hence (1 - data.dones.flatten()) = 1 - 1 = 0
                    # There's no need to predict the action in next state
                    td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())

                # Calculate the current old value
                # old_val = q_network(s, a)
                # Which mean it passes the observation of this sample
                # q_network(data.observations) -> Send it to the q_network to get (B, n)
                # .gather(1, data.actions) -> Based on the index of of the actions, sample the value in the dim=1 (https://docs.pytc
                # .squeeze -> to reduce the shape to (B,)
                old_val = q_network(data.observations).gather(1, data.actions).squeeze()

                # Compute MSE Loss
                loss = F.mse_loss(td_target, old_val)

                if global_step % 100 == 0:
                    writer.add_scalar("losses/td_loss", loss, global_step)
                    writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)
                    print("SPS:", int(global_step / (time.time() - start_time)))
                    writer.add_scalar("charts/SPS", int(global_step / (time.time() - start_time)), global_step)

                # optimize the model
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

            # update target network
            # Update partially with the help of tau
            if global_step % args.target_network_frequency == 0:
                for target_network_param, q_network_param in zip(target_network.parameters(), q_network.parameters()):
                    target_network_param.data.copy_(
                        args.tau * q_network_param.data + (1.0 - args.tau) * target_network_param.data
                    )

    if args.save_model:
```

```
        model_path = f"runs/{run_name}/{args.exp_name}.cleanrl_model"
        torch.save(q_network.state_dict(), model_path)
        print(f"model saved to {model_path}")
        from cleanrl_utils.evals.dqn_eval import evaluate

        episodic_returns = evaluate(
            model_path,
            make_env,
            args.env_id,
            eval_episodes=10,
            run_name=f"{run_name}-eval",
            Model=QNetwork,
            device=device,
            epsilon=0.05,
        )
        for idx, episodic_return in enumerate(episodic_returns):
            writer.add_scalar("eval/episodic_return", episodic_return, idx)

        if args.upload_model:
            from cleanrl_utils.huggingface import push_to_hub

            repo_name = f"{args.env_id}-{args.exp_name}-seed{args.seed}"
            repo_id = f"{args.hf_entity}/{repo_name}" if args.hf_entity else repo_name
            push_to_hub(args, episodic_returns, repo_id, "DQN", f"runs/{run_name}", f"videos/{run_name}-eval")

    envs.close()
    writer.close()
```

Show hidden output