```python
# Note:
# The push to hub is taken from this tutorial: https://huggingface.co/learn/deep-rl-course/unit8/hands-on-cleanrl
# The PPO implementation is taken directly from the CleanRL repository as the one provided in the above notebook is outdated

# ============================================================================
# !apt install swig cmake
# !pip install swig
# !pip install stable-baselines3==2.0.0a5
# !pip install gymnasium[box2d]
# !pip install imageio-ffmpeg
# !pip install huggingface_hub
# !pip install tyro
# ============================================================================


# ============================================================================
# from huggingface_hub import notebook_login
# notebook_login()
# !git config --global credential.helper store
# ============================================================================


# docs and experiment results can be found at https://docs.cleanrl.dev/rl-algorithms/ppo/#ppopy
import os
import random
import time
from dataclasses import dataclass
from distutils.util import strtobool

import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tyro
from torch.distributions.categorical import Categorical
from torch.utils.tensorboard import SummaryWriter

from pathlib import Path
import datetime
import tempfile
import json
import shutil
import imageio


from huggingface_hub import HfApi, upload_folder
from huggingface_hub.repocard import metadata_eval_result, metadata_save

from wasabi import Printer
msg = Printer()


@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)"""

    # Algorithm specific arguments
    env_id: str = "CartPole-v1"
    """the id of the environment"""
    total_timesteps: int = 500000
    """total timesteps of the experiments"""
    learning_rate: float = 2.5e-4
    """the learning rate of the optimizer"""
    num_envs: int = 4
    """the number of parallel game environments"""
    num_steps: int = 128
    """the number of steps to run in each environment per policy rollout"""
    anneal_lr: bool = True
    """Toggle learning rate annealing for policy and value networks"""
    gamma: float = 0.99
    """the discount factor gamma"""
    gae_lambda: float = 0.95
    """the lambda for the general advantage estimation"""
    num_minibatches: int = 4
    """the number of mini-batches"""
    update_epochs: int = 4
    """the K epochs to update the policy"""
    norm_adv: bool = True
    """Toggles advantages normalization"""
    clip_coef: float = 0.2
    """the surrogate clipping coefficient"""
    clip_vloss: bool = True
    """Toggles whether or not to use a clipped loss for the value function, as per the paper."""
    ent_coef: float = 0.01
    """coefficient of the entropy"""
    vf_coef: float = 0.5
    """coefficient of the value function"""
    max_grad_norm: float = 0.5
    """the maximum norm for the gradient clipping"""
    target_kl: float = None
    """the target KL divergence threshold"""

    # to be filled in runtime
    batch_size: int = 0
    """the batch size (computed in runtime)"""
    minibatch_size: int = 0
    """the mini-batch size (computed in runtime)"""
    num_iterations: int = 0
    """the number of iterations (computed in runtime)"""

    # Adding HuggingFace argument
    repo_id: str = "ThomasSimonini/ppo-CartPole-v1"
    """id of the model repository from the Hugging Face Hub {username/repo_name}"""

def package_to_hub(repo_id,
                model,
                hyperparameters,
```

```python
                eval_env,
                video_fps=30,
                commit_message="Push agent to the Hub",
                token= None,
                logs=None
                ):
  """
  Evaluate, Generate a video and Upload a model to Hugging Face Hub.
  This method does the complete pipeline:
  - It evaluates the model
  - It generates the model card
  - It generates a replay video of the agent
  - It pushes everything to the hub
  :param repo_id: id of the model repository from the Hugging Face Hub
  :param model: trained model
  :param eval_env: environment used to evaluate the agent
  :param fps: number of fps for rendering the video
  :param commit_message: commit message
  :param logs: directory on local machine of tensorboard logs you'd like to upload
  """
  msg.info(
        "This function will save, evaluate, generate a video of your agent, "
        "create a model card and push everything to the hub. "
        "It might take up to 1min. \n "
        "This is a work in progress: if you encounter a bug, please open an issue."
    )
  # Step 1: Clone or create the repo
  repo_url = HfApi().create_repo(
        repo_id=repo_id,
        token=token,
        private=False,
        exist_ok=True,
    )

  with tempfile.TemporaryDirectory() as tmpdirname:
    tmpdirname = Path(tmpdirname)

    # Step 2: Save the model
    torch.save(model.state_dict(), tmpdirname / "model.pt")

    # Step 3: Evaluate the model and build JSON
    mean_reward, std_reward = _evaluate_agent(eval_env,
                                           10,
                                           model)

    # First get datetime
    eval_datetime = datetime.datetime.now()
    eval_form_datetime = eval_datetime.isoformat()

    evaluate_data = {
        "env_id": hyperparameters.env_id,
        "mean_reward": mean_reward,
        "std_reward": std_reward,
        "n_evaluation_episodes": 10,
        "eval_datetime": eval_form_datetime,
    }

    # Write a JSON file
    with open(tmpdirname / "results.json", "w") as outfile:
      json.dump(evaluate_data, outfile)

    # Step 4: Generate a video
    video_path =  tmpdirname / "replay.mp4"
    record_video(eval_env, model, video_path, video_fps)

    # Step 5: Generate the model card
    generated_model_card, metadata = _generate_model_card("PPO", hyperparameters.env_id, mean_reward, std_reward, hyperparameters)
    _save_model_card(tmpdirname, generated_model_card, metadata)

    # Step 6: Add logs if needed
    if logs:
      _add_logdir(tmpdirname, Path(logs))

    msg.info(f"Pushing repo {repo_id} to the Hugging Face Hub")

    repo_url = upload_folder(
            repo_id=repo_id,
            folder_path=tmpdirname,
            path_in_repo="",
            commit_message=commit_message,
            token=token,
        )

    msg.info(f"Your model is pushed to the Hub. You can view your model here: {repo_url}")
  return repo_url

def _evaluate_agent(env, n_eval_episodes, policy):
  """
  Evaluate the agent for ``n_eval_episodes`` episodes and returns average reward and std of reward.
  :param env: The evaluation environment
  :param n_eval_episodes: Number of episode to evaluate the agent
  :param policy: The agent
  """
  episode_rewards = []
  for episode in range(n_eval_episodes):
    state = env.reset()[0]
    step = 0
    done = False
    total_rewards_ep = 0

    while done is False:
      state = torch.Tensor(state).to(device)
      action, _, _, _ = policy.get_action_and_value(state)
      new_state, reward, terminated, truncated, info = env.step(action.cpu().numpy())
      done = terminated or truncated
      total_rewards_ep += reward
      if done:
        break
      state = new_state
    episode_rewards.append(total_rewards_ep)
  mean_reward = np.mean(episode_rewards)
  std_reward = np.std(episode_rewards)

  return mean_reward, std_reward


def record_video(env, policy, out_directory, fps=30):
  images = []
  done = False
  state = env.reset()[0]
  img = env.render()
  images.append(img)
  while not done:
```

```python
    state = torch.Tensor(state).to(device)
    # Take the action (index) that have the maximum expected future reward given that state
    action, _, _, _  = policy.get_action_and_value(state)
    state, reward, terminated, truncated, info = env.step(action.cpu().numpy()) # We directly put next_state = state for recording logic
    done = terminated or truncated
    img = env.render()
    images.append(img)
  imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)], fps=fps)


def _generate_model_card(model_name, env_id, mean_reward, std_reward, hyperparameters):
  """
  Generate the model card for the Hub
  :param model_name: name of the model
  :env_id: name of the environment
  :mean_reward: mean reward of the agent
  :std_reward: standard deviation of the mean reward of the agent
  :hyperparameters: training arguments
  """
  # Step 1: Select the tags
  metadata = generate_metadata(model_name, env_id, mean_reward, std_reward)

  # Transform the hyperparams namespace to string
  converted_dict = vars(hyperparameters)
  converted_str = str(converted_dict)
  converted_str = converted_str.split(", ")
  converted_str = '\n'.join(converted_str)

  # Step 2: Generate the model card
  model_card = f"""
  # PPO Agent Playing {env_id}

  This is a trained model of a PPO agent playing {env_id}.

  # Hyperparameters
  ```python
  {converted_str}
  ```
  """
  return model_card, metadata

def generate_metadata(model_name, env_id, mean_reward, std_reward):
  """
  Define the tags for the model card
  :param model_name: name of the model
  :param env_id: name of the environment
  :mean_reward: mean reward of the agent
  :std_reward: standard deviation of the mean reward of the agent
  """
  metadata = {}
  metadata["tags"] = [
        env_id,
        "ppo",
        "deep-reinforcement-learning",
        "reinforcement-learning",
        "custom-implementation",
        "deep-rl-course"
  ]

  # Add metrics
  eval = metadata_eval_result(
      model_pretty_name=model_name,
      task_pretty_name="reinforcement-learning",
      task_id="reinforcement-learning",
      metrics_pretty_name="mean_reward",
      metrics_id="mean_reward",
      metrics_value=f"{mean_reward:.2f} +/- {std_reward:.2f}",
      dataset_pretty_name=env_id,
      dataset_id=env_id,
)

  # Merges both dictionaries
  metadata = {**metadata, **eval}

  return metadata

def _save_model_card(local_path, generated_model_card, metadata):
    """Saves a model card for the repository.
    :param local_path: repository directory
    :param generated_model_card: model card generated by _generate_model_card()
    :param metadata: metadata
    """
    readme_path = local_path / "README.md"
    readme = ""
    if readme_path.exists():
        with readme_path.open("r", encoding="utf8") as f:
            readme = f.read()
    else:
        readme = generated_model_card

    with readme_path.open("w", encoding="utf-8") as f:
        f.write(readme)

    # Save our metrics to Readme metadata
    metadata_save(readme_path, metadata)

def _add_logdir(local_path: Path, logdir: Path):
  """Adds a logdir to the repository.
  :param local_path: repository directory
  :param logdir: logdir directory
  """
  if logdir.exists() and logdir.is_dir():
    # Add the logdir to the repository under new dir called logs
    repo_logdir = local_path / "logs"

    # Delete current logs if they exist
    if repo_logdir.exists():
      shutil.rmtree(repo_logdir)

    # Copy logdir into repo logdir
    shutil.copytree(logdir, repo_logdir)

def make_env(env_id, idx, capture_video, run_name):
    def thunk():
        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)
        env = gym.wrappers.RecordEpisodeStatistics(env)
        return env
```

```python
    return thunk


def layer_init(layer, std=np.sqrt(2), bias_const=0.0):
    torch.nn.init.orthogonal_(layer.weight, std)
    torch.nn.init.constant_(layer.bias, bias_const)
    return layer


class Agent(nn.Module):
    def __init__(self, envs):
        super().__init__()
        self.critic = nn.Sequential(
            layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 1), std=1.0),
        )
        self.actor = nn.Sequential(
            layer_init(nn.Linear(np.array(envs.single_observation_space.shape).prod(), 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, 64)),
            nn.Tanh(),
            layer_init(nn.Linear(64, envs.single_action_space.n), std=0.01),
        )

    def get_value(self, x):
        return self.critic(x)

    def get_action_and_value(self, x, action=None):
        logits = self.actor(x)
        probs = Categorical(logits=logits)
        if action is None:
            action = probs.sample()
        return action, probs.log_prob(action), probs.entropy(), self.critic(x)


if __name__ == "__main__":
    args = tyro.cli(Args)
    args.batch_size = int(args.num_envs * args.num_steps)
    args.minibatch_size = int(args.batch_size // args.num_minibatches)
    args.num_iterations = args.total_timesteps // args.batch_size
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"
    if args.track:
        import wandb

        wandb.init(
            project=args.wandb_project_name,
            entity=args.wandb_entity,
            sync_tensorboard=True,
            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
    envs = gym.vector.SyncVectorEnv(
        [make_env(args.env_id, i, args.capture_video, run_name) for i in range(args.num_envs)],
    )
    assert isinstance(envs.single_action_space, gym.spaces.Discrete), "only discrete action space is supported"

    agent = Agent(envs).to(device)
    optimizer = optim.Adam(agent.parameters(), lr=args.learning_rate, eps=1e-5)

    # ALGO Logic: Storage setup
    obs = torch.zeros((args.num_steps, args.num_envs) + envs.single_observation_space.shape).to(device)
    actions = torch.zeros((args.num_steps, args.num_envs) + envs.single_action_space.shape).to(device)
    logprobs = torch.zeros((args.num_steps, args.num_envs)).to(device)
    rewards = torch.zeros((args.num_steps, args.num_envs)).to(device)
    dones = torch.zeros((args.num_steps, args.num_envs)).to(device)
    values = torch.zeros((args.num_steps, args.num_envs)).to(device)

    # TRY NOT TO MODIFY: start the game
    global_step = 0
    start_time = time.time()
    next_obs, _ = envs.reset(seed=args.seed)
    next_obs = torch.Tensor(next_obs).to(device)
    next_done = torch.zeros(args.num_envs).to(device)

    for iteration in range(1, args.num_iterations + 1):
        # Annealing the rate if instructed to do so.
        if args.anneal_lr:
            frac = 1.0 - (iteration - 1.0) / args.num_iterations
            lrnow = frac * args.learning_rate
            optimizer.param_groups[0]["lr"] = lrnow

        for step in range(0, args.num_steps):
            global_step += args.num_envs
            obs[step] = next_obs
            dones[step] = next_done

            # ALGO LOGIC: action logic
            with torch.no_grad():
                action, logprob, _, value = agent.get_action_and_value(next_obs)
                values[step] = value.flatten()
            actions[step] = action
            logprobs[step] = logprob

            # TRY NOT TO MODIFY: execute the game and log data.
            next_obs, reward, terminations, truncations, infos = envs.step(action.cpu().numpy())
            next_done = np.logical_or(terminations, truncations)
            rewards[step] = torch.tensor(reward).to(device).view(-1)
            next_obs, next_done = torch.Tensor(next_obs).to(device), torch.Tensor(next_done).to(device)

            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']}")
```

```python
                    writer.add_scalar("charts/episodic_return", info["episode"]["r"], global_step)
                    writer.add_scalar("charts/episodic_length", info["episode"]["l"], global_step)

        # bootstrap value if not done
        with torch.no_grad():
            next_value = agent.get_value(next_obs).reshape(1, -1)
            advantages = torch.zeros_like(rewards).to(device)
            lastgaelam = 0
            for t in reversed(range(args.num_steps)):
                if t == args.num_steps - 1:
                    nextnonterminal = 1.0 - next_done
                    nextvalues = next_value
                else:
                    nextnonterminal = 1.0 - dones[t + 1]
                    nextvalues = values[t + 1]
                delta = rewards[t] + args.gamma * nextvalues * nextnonterminal - values[t]
                advantages[t] = lastgaelam = delta + args.gamma * args.gae_lambda * nextnonterminal * lastgaelam
            returns = advantages + values

        # flatten the batch
        b_obs = obs.reshape((-1,) + envs.single_observation_space.shape)
        b_logprobs = logprobs.reshape(-1)
        b_actions = actions.reshape((-1,) + envs.single_action_space.shape)
        b_advantages = advantages.reshape(-1)
        b_returns = returns.reshape(-1)
        b_values = values.reshape(-1)

        # Optimizing the policy and value network
        b_inds = np.arange(args.batch_size)
        clipfracs = []
        for epoch in range(args.update_epochs):
            np.random.shuffle(b_inds)
            for start in range(0, args.batch_size, args.minibatch_size):
                end = start + args.minibatch_size
                mb_inds = b_inds[start:end]

                _, newlogprob, entropy, newvalue = agent.get_action_and_value(b_obs[mb_inds], b_actions.long()[mb_inds])
                logratio = newlogprob - b_logprobs[mb_inds]
                ratio = logratio.exp()

                with torch.no_grad():
                    # calculate approx_kl http://joschu.net/blog/kl-approx.html
                    old_approx_kl = (-logratio).mean()
                    approx_kl = ((ratio - 1) - logratio).mean()
                    clipfracs += [((ratio - 1.0).abs() > args.clip_coef).float().mean().item()]

                mb_advantages = b_advantages[mb_inds]
                if args.norm_adv:
                    mb_advantages = (mb_advantages - mb_advantages.mean()) / (mb_advantages.std() + 1e-8)

                # Policy loss
                pg_loss1 = -mb_advantages * ratio
                pg_loss2 = -mb_advantages * torch.clamp(ratio, 1 - args.clip_coef, 1 + args.clip_coef)
                pg_loss = torch.max(pg_loss1, pg_loss2).mean()

                # Value loss
                newvalue = newvalue.view(-1)
                if args.clip_vloss:
                    v_loss_unclipped = (newvalue - b_returns[mb_inds]) ** 2
                    v_clipped = b_values[mb_inds] + torch.clamp(
                        newvalue - b_values[mb_inds],
                        -args.clip_coef,
                        args.clip_coef,
                    )
                    v_loss_clipped = (v_clipped - b_returns[mb_inds]) ** 2
                    v_loss_max = torch.max(v_loss_unclipped, v_loss_clipped)
                    v_loss = 0.5 * v_loss_max.mean()
                else:
                    v_loss = 0.5 * ((newvalue - b_returns[mb_inds]) ** 2).mean()

                entropy_loss = entropy.mean()
                loss = pg_loss - args.ent_coef * entropy_loss + v_loss * args.vf_coef

                optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(agent.parameters(), args.max_grad_norm)
                optimizer.step()

            if args.target_kl is not None and approx_kl > args.target_kl:
                break

        y_pred, y_true = b_values.cpu().numpy(), b_returns.cpu().numpy()
        var_y = np.var(y_true)
        explained_var = np.nan if var_y == 0 else 1 - np.var(y_true - y_pred) / var_y

        # TRY NOT TO MODIFY: record rewards for plotting purposes
        writer.add_scalar("charts/learning_rate", optimizer.param_groups[0]["lr"], global_step)
        writer.add_scalar("losses/value_loss", v_loss.item(), global_step)
        writer.add_scalar("losses/policy_loss", pg_loss.item(), global_step)
        writer.add_scalar("losses/entropy", entropy_loss.item(), global_step)
        writer.add_scalar("losses/old_approx_kl", old_approx_kl.item(), global_step)
        writer.add_scalar("losses/approx_kl", approx_kl.item(), global_step)
        writer.add_scalar("losses/clipfrac", np.mean(clipfracs), global_step)
        writer.add_scalar("losses/explained_variance", explained_var, 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)

    envs.close()
    writer.close()

    # Create the evaluation environment
    eval_env = gym.make(args.env_id, render_mode="rgb_array")

    package_to_hub(repo_id = args.repo_id,
                   model = agent, # The model we want to save
                   hyperparameters = args,
                   eval_env = eval_env,
                   logs= f"runs/{run_name}",)
```