Implementation of Reinforce from scratch to play Cartpole-v1

- Environment documentations: https://gymnasium.farama.org/environments/classic_control/cart_pole/

Install and import libraries

```
# Import Libraries

import gymnasium as gym
import torch
import torch.nn as nn
from torch.distributions.categorical import Categorical
from torch.distributions.normal import Normal
from collections import deque
import numpy as np

# For pushing to hub
from huggingface_hub import HfApi, snapshot_download
from huggingface_hub.repocard import metadata_eval_result, metadata_save
from huggingface_hub import notebook_login

from pathlib import Path
import datetime
import json
import imageio

import tempfile

import os
```

```
# Device

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

0. Visualise the observation space and action space

```
env = gym.make("CartPole-v1")

print(f"Number of available actions: {env.action_space.n}")
print(f"Sample a random action: {env.action_space.sample()}")

print(f"Sample a random observation: {env.observation_space.sample()}")
```

```
Number of available actions: 2
Sample a random action: 0
Sample a random observation: [4.745393   2.4669313  0.4147641  0.27528846]
```

1. Build a policy network using PyTorch

```
class Policy(nn.Module):
  """
  A policy network.

  Args:
    s_size (int): The size of 1 state space. \n
    h_size (int): The number of hidden nodes in the network. \n
    a_size (int): The number of distinct discrete actions, representing the number of output nodes. \n
  """
  def __init__(self, s_size, h_size, a_size):
    super().__init__()

    self.fc1 = nn.Sequential(nn.Linear(in_features = s_size,
                                       out_features = h_size),
                             nn.ReLU())

    self.fc2 = nn.Sequential(nn.Linear(in_features = h_size,
                                       out_features = a_size))

  def forward(self, x):
    """
    The forward propagation of the policy network.
```

```

    Args:
      x (float tensor): Input to the network representing the observation / state, expected shape: (B, s_size). \n

    Returns:
      out(float tensor): Output of the network, representing the probability of taking each distinct discrete action, expecte shape:
    """
    out = self.fc1(x)
    out = self.fc2(out)
    out = torch.nn.functional.softmax(out, dim=-1)
    return out

  def act(self, state):
    """
    Sampling of an action.

    Args:
      state (float tensor): Input to the network representing the observation / state, expected shape: (B, s_size). \n

    Returns:
      action (int / int tensor): The index of the output nodes of the network, sampled based on the output probability of the networ
      log_prob (float tensor): The ln of the probability of the action that was sampled based on the output probability of the newto

    """
    probs = self.forward(state).cpu()
    m = Categorical(probs)
    action = m.sample()
    if action.shape[0] == 1:
      return action.item(), m.log_prob(action)
    else:
      return action, m.log_prob(action)


```

2. Implementation of Reinforce algorithm

```
def reinforce(policy, optimizer, env, n_episodes, n_steps, device, gamma, print_every):

  """
  Train a policy network using the reinforce algorithm.

  Args:
    policy (nn.Module): A policy network. \n
    optimizer (torch.optim): An optimizer. \n
    env (gymnasium.env): An environment. \n
    n_episodes (int): Number of training episodes. \n
    n_steps (int): Maximum number of steps allowed in an episode. \n
    device (str): 'cuda' or 'cpu'
    gamma (float): A discount factor, range from 0 to 1.
    print_every (int): Number of episode intervals to print the performance of the network. \n

  Returns:
    scores (list): A list of integer, each element representing the rewards scored in an episode. \n

  """

  # Create variables to store the rewards scored for every episode
  scores = []

  # This variable store up to rewards scored for every episode up to "print_every" episodes
  scores_deque = deque(maxlen = print_every)


  ####################
  # For each episode #
  ####################
  for episode in range(1, n_episodes+1):

    # Variable to store values for every step within an episode
    reward_eps = [] # Reward scored by each step
    log_prob_eps = [] # ln (prob of the action taken) for each step
    returns_eps = deque(maxlen = n_steps) # discounted returns scored in each step
    policy_loss_eps = [] # loss for each step -> ln (prob of the action taken) * discounted return

    # Reset the environment for the beginning of each episode
    state, info = env.reset()

    #################
    # For each step #
    #################
```

```
    for _ in range(n_steps):

      # Sample an action using the policy network
      action, log_prob = policy.act(torch.tensor(state).unsqueeze(0).to(device))

      # Step the environment using the sampled action
      state, reward, terminated, truncated, info = env.step(action)

      # Store the rewards for this step and the ln (prob of this action)
      reward_eps.append(reward)
      log_prob_eps.append(log_prob)

      # Check if this leads to termination or truncation
      if terminated or truncated:
        break

    # Sum the rewards scored for the entire episode and store them
    scores.append(sum(reward_eps))
    scores_deque.append(sum(reward_eps))

    # Calculate the discounted returns
    for t in range(len(reward_eps))[::-1]:
      returns = reward_eps[t] + gamma * returns_eps[0] if len(returns_eps) > 0 else reward_eps[t]
      returns_eps.appendleft(returns)

    # Normalize the discounted returns
    eps = np.finfo(np.float32).eps.item() # eps is smallest reprsentatable float
    returns_eps = torch.tensor(returns_eps) # Convert into torch tensor for later calculation
    returns_eps = (returns_eps - returns_eps.mean()) / (returns_eps.std() + eps)

    # Calculate the loss
    for log_prob, returns in zip(log_prob_eps, returns_eps):
      policy_loss_eps.append(-log_prob * returns) # torch tensor * torch tensor
    loss = torch.cat(policy_loss_eps).sum() # cat -> makes into one torch tensor, sum() -> summation

    # Backward propagation and gradient descent
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print information
    if episode % print_every == 0:
      print(f"Current Episode: {episode} | Average reward: {np.mean(scores_deque)}")

  return scores
```

3. Train the policy network using reinforce algorithm

```
#0. Create device
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 1. Create hyperparameters
cartpole_hyperparameters = {
    "h_size": 16,
    "n_training_episodes": 1000,
    "n_evaluation_episodes": 10,
    "max_t": 1000,
    "gamma": 1.0,
    "lr": 0.01,
    "env_id": 'CartPole-v1',
    "state_space": 4,
    "action_space": 2,
}

# 2. Create environment
env = gym.make(cartpole_hyperparameters['env_id'])

# 3. Create policy network
cartpole_policy = Policy(cartpole_hyperparameters['state_space'],
                         cartpole_hyperparameters['h_size'],
                         cartpole_hyperparameters['action_space']).to(device)

# 4. Create optimizer
optimizer = torch.optim.Adam(cartpole_policy.parameters(),
```

```
                             lr = cartpole_hyperparameters['lr'])

# 5. Training loop
scores = reinforce(policy = cartpole_policy,
                   optimizer = optimizer,
                   env = env,
                   n_episodes = cartpole_hyperparameters['n_training_episodes'],
                   n_steps = cartpole_hyperparameters['max_t'],
                   device = device,
                   gamma = cartpole_hyperparameters['gamma'],
                   print_every = 100)

```

```
Current Episode: 100 | Average reward: 44.33
Current Episode: 200 | Average reward: 308.61
Current Episode: 300 | Average reward: 322.93
Current Episode: 400 | Average reward: 388.92
Current Episode: 500 | Average reward: 352.5
Current Episode: 600 | Average reward: 459.3
Current Episode: 700 | Average reward: 476.28
Current Episode: 800 | Average reward: 500.0
Current Episode: 900 | Average reward: 461.11
Current Episode: 1000 | Average reward: 379.94
```

4. Evaluate the agent

```
def evaluate_agent(n_eval_episodes, n_steps, policy, env, device):

  """
  Evaluate the performance of an agent by calculating the mean and standard deviation rewards over n_eval_episodes of episodes.

  Args:
    n_eval_episodes (int): Number of evaluation episodes. \n
    n_steps (int): Maximum number of steps allowed in an episode. \n
    policy (nn.Module): A policy network. \n
    env (gymnasium.env): An environment. \n
    device (str): 'cuda' or 'cpu'. \n

  Returns:
    mean_reward (float): Mean reward scored across the evaluated episodes.
    std_reward (float): Standard deviation reward scored across the evaluated episodes.
  """


  rewards_across_episodes = [] # Contains rewards scored in each episode

  ####################
  # For each episode #
  ####################
  for episode in range(n_eval_episodes):

    # To store reward scored in each step
    rewards = []

    # Reset the environment
    state, info = env.reset()

    #################
    # For each step #
    #################
    for step in range(n_steps):

      # Sample an action
      action, _ = policy.act(torch.tensor(state).unsqueeze(0).to(device))

      # Step the environment by taking the action
      state, reward, terminated, truncated, info = env.step(action)

      # Store the reward scored in this step
      rewards.append(reward)

      # Check if truncated or terminated
      if truncated or terminated:
        break

    # Sum the reward scored in the entire episode and store it
    rewards_across_episodes.append(sum(rewards))

  # Calculate the mean and standard deviation
  mean_reward = np.array(rewards_across_episodes).mean()
  std_reward = np.array(rewards_across_episodes).std()
```

```
  return mean_reward, std_reward

```

```
mean_reward, std_reward = evaluate_agent(n_eval_episodes = cartpole_hyperparameters['n_evaluation_episodes'],
                                         n_steps = cartpole_hyperparameters['max_t'],
                                         policy = cartpole_policy,
                                         env = env,
                                         device = device)

print(f"Mean reward: {mean_reward}, standard deviation: {std_reward}")
```

```
Mean reward: 500.0, standard deviation: 0.0
```

5. Push to Hub

- Code source: https://colab.research.google.com/github/wengti/Reinforcement-Learning-Tutorial-/blob/main/notebooks/unit4/unit4.ipynb#scrollTo=LIVsvlW_8tcw
- Creat a write token here: https://huggingface.co/settings/tokens/new?tokenType=write

```
def record_video(env, policy, out_directory, device, fps=30):
  """
  Generate a replay video of the agent
  :param env
  :param Qtable: Qtable of our agent
  :param out_directory
  :param fps: how many frame per seconds (with taxi-v3 and frozenlake-v1 we use 1)
  """
  images = []
  state, info = env.reset()
  terminated = False
  truncated = False
  img = env.render()
  images.append(img)
  while not terminated and not truncated:
    # Take the action (index) that have the maximum expected future reward given that state
    action, _ = policy.act(torch.tensor(state).unsqueeze(0).to(device))
    state, reward, terminated, truncated, info = env.step(action) # We directly put next_state = state for recording logic
    img = env.render()
    images.append(img)
  imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)], fps=fps)
```

```
def push_to_hub(repo_id,
                model,
                hyperparameters,
                eval_env,
                video_fps=30
                ):
  """
  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: repo_id: id of the model repository from the Hugging Face Hub
  :param model: the pytorch model we want to save
  :param hyperparameters: training hyperparameters
  :param eval_env: evaluation environment
  :param video_fps: how many frame per seconds to record our video replay
  """

  _, repo_name = repo_id.split("/")
  api = HfApi()

  # Step 1: Create the repo
  repo_url = api.create_repo(
        repo_id=repo_id,
        exist_ok=True,
  )

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

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

```

```
    # Step 3: Save the hyperparameters to JSON
    with open(local_directory / "hyperparameters.json", "w") as outfile:
      json.dump(hyperparameters, outfile)

    # Step 4: Evaluate the model and build JSON
    mean_reward, std_reward = evaluate_agent(hyperparameters["n_evaluation_episodes"],
                                            hyperparameters["max_t"],
                                            model,
                                            eval_env,
                                            'cuda')
    # Get datetime
    eval_datetime = datetime.datetime.now()
    eval_form_datetime = eval_datetime.isoformat()

    evaluate_data = {
          "env_id": hyperparameters["env_id"],
          "mean_reward": mean_reward,
          "n_evaluation_episodes": hyperparameters["n_evaluation_episodes"],
          "eval_datetime": eval_form_datetime,
    }

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

    # Step 5: Create the model card
    env_name = hyperparameters["env_id"]
    env_id = env_name

    metadata = {}
    metadata["tags"] = [
          env_name,
          "reinforce",
          "reinforcement-learning",
          "custom-implementation",
          "deep-rl-class"
      ]

    # Add metrics
    eval = metadata_eval_result(
        model_pretty_name=repo_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_name,
        dataset_id=env_name,
      )

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

    model_card = f"""
  # **Reinforce** Agent playing **{env_id}**
  This is a trained model of a **Reinforce** agent playing **{env_id}** .
  To learn to use this model and train yours check Unit 4 of the Deep Reinforcement Learning Course: https://huggingface.co/deep-r
  """

    readme_path = local_directory / "README.md"
    readme = ""
    if readme_path.exists():
        with readme_path.open("r", encoding="utf8") as f:
          readme = f.read()
    else:
      readme = 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)

    # Step 6: Record a video
    video_path =  local_directory / "replay.mp4"
    record_video(eval_env, model, video_path, 'cuda', video_fps)

    # Step 7. Push everything to the Hub
    api.upload_folder(
          repo_id=repo_id,
          folder_path=local_directory,
          path_in_repo=".",
```

```
    )

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

```
# Login to hugging face with a write token

notebook_login()
```

```
# Create a new environment with render mode (needed for video recording)
eval_env = gym.make(cartpole_hyperparameters['env_id'], render_mode = 'rgb_array')

# Push to Hub
push_to_hub(repo_id = "wengti0608/Reinforce-Cartpole-v1-attempt2",
            model = cartpole_policy,
            hyperparameters = cartpole_hyperparameters,
            eval_env = eval_env)
```

```
WARNING:imageio_ffmpeg:IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from (600, 400)
Uploading...: 100%                                   3.78k/3.78k [00:01<00:00, 18.8kB/s]
Your model is pushed to the Hub. You can view your model here: https://huggingface.co/wengti0608/Reinforce-Cartpole-v1-attempt2
```

Practice: Application of Reinforce to play Continuous Mountain Car

- Environment documentation: https://gymnasium.farama.org/environments/classic_control/mountain_car/
- Continuous Mountain Car agent takes continuous actions. Therefore, the policy network is slightly modified to output mean and standard deviation instead of index of discrete actions.

0. Visualize Environment

```
# Visualize the environment

env = gym.make("MountainCarContinuous-v0")

print(f"Randomly sample an action: {env.action_space.sample()}")
print(f"Randomly sample a state: {env.observation_space.sample()}")

```

```
Randomly sample an action: [0.9539736]
Randomly sample a state: [-0.00267963 -0.00615319]
```

1. Create the policy network

```
class Policy(nn.Module):
  """
  A policy network.

  Args:
    s_size (int): The size of 1 state space. \n
    h_size (int): The number of hidden nodes in the network. \n
    a_size (int): Number of continuos actions. \n
  """
  def __init__(self, s_size, h_size, a_size):
    super().__init__()

    self.fc1 = nn.Sequential(nn.Linear(in_features = s_size,
                                       out_features = h_size),
                             nn.ReLU())

    self.fc2 = nn.Sequential(nn.Linear(in_features = h_size,
                                       out_features = h_size),
                             nn.ReLU())

    self.mean_head = nn.Linear(in_features = h_size,
                               out_features = a_size)

    self.log_std_head = nn.Linear(in_features = h_size,
                                  out_features = a_size)

  def forward(self, x):
```

```
   """
   The forward propagation of the policy network.

   Args:
     x (float tensor): Input to the network representing the observation / state, expected shape: (B, s_size). \n

   Returns:
     mean (float tensor): Mean of a Normal Distribution, (B, a_size). \n
     std (float tensor): Standard deviation of a Normal Distribtuion, (B, a_size). \n
   """
   out = self.fc1(x)
   out = self.fc2(out)

   mean = self.mean_head(out)
   log_std = self.log_std_head(out)
   std = torch.exp(log_std)

   return mean, std

 def act(self, state):
   """
   Sampling of an action.

   Args:
     state (float tensor): Input to the network representing the observation / state, expected shape: (B, s_size). \n

   Returns:
     action_clipped (float): The value of the action taken, in the range of -1 to 1 \n
     log_prob (float tensor): The ln of the probability of the action that was sampled based on the output probability of the newt

   """
   mean, std = self.forward(state)
   mean = mean.cpu() # torch.tensor, (1,1)
   std = std.cpu() # torch.tensor, (1,1)

   m = Normal(mean, std)

   action = m.sample() # torch.tensor, (1,1)
   action_clipped = torch.clamp(action, -1, 1).item() #float

   log_prob = m.log_prob(action)[0] # torch.tensor, (1,)

   return action_clipped, log_prob


```

2. Implement reinforce algorithm from scratch

```
def reinforce(policy, optimizer, env, n_episodes, n_steps, device, gamma, print_every):

 """
 Train a policy network using the reinforce algorithm.

 Args:
   policy (nn.Module): A policy network. \n
   optimizer (torch.optim): An optimizer. \n
   env (gymnasium.env): An environment. \n
   n_episodes (int): Number of training episodes. \n
   n_steps (int): Maximum number of steps allowed in an episode. \n
   device (str): 'cuda' or 'cpu'
   gamma (float): A discount factor, range from 0 to 1.
   print_every (int): Number of episode intervals to print the performance of the network. \n

 Returns:
   scores (list): A list of integer, each element representing the rewards scored in an episode. \n

 """

 # Create variables to store the rewards scored for every episode
 scores = []

 # This variable store up to rewards scored for every episode up to "print_every" episodes
 scores_deque = deque(maxlen = print_every)
 len_deque = deque(maxlen = print_every)


 ####################
 # For each episode #
 ####################
 for episode in range(1, n_episodes+1):
```

```

    # Variable to store values for every step within an episode
    reward_eps = [] # Reward scored by each step
    log_prob_eps = [] # ln (prob of the action taken) for each step
    returns_eps = deque(maxlen = n_steps) # discounted returns scored in each step
    policy_loss_eps = [] # loss for each step -> ln (prob of the action taken) * discounted return

    # Reset the environment for the beginning of each episode
    state, info = env.reset()

    #################
    # For each step #
    #################
    for _ in range(n_steps):

      # Sample an action using the policy network
      action, log_prob = policy.act(torch.tensor(state).unsqueeze(0).to(device))

      # Step the environment using the sampled action
      state, reward, terminated, truncated, info = env.step(np.array([action]))

      # Store the rewards for this step and the ln (prob of this action)
      reward_eps.append(reward)
      log_prob_eps.append(log_prob)

      # Check if this leads to termination or truncation
      if terminated or truncated:
        break

    # Sum the rewards scored for the entire episode and store them
    scores.append(sum(reward_eps))
    scores_deque.append(sum(reward_eps))
    len_deque.append(len(reward_eps))

    # Calculate the discounted returns
    for t in range(len(reward_eps))[::-1]:
      returns = reward_eps[t] + gamma * returns_eps[0] if len(returns_eps) > 0 else reward_eps[t]
      returns_eps.appendleft(returns)

    # Normalize the discounted returns
    eps = np.finfo(np.float32).eps.item() # eps is smallest reprsentatable float
    returns_eps = torch.tensor(returns_eps) # Convert into torch tensor for later calculation
    returns_eps = (returns_eps - returns_eps.mean()) / (returns_eps.std() + eps)

    # Calculate the loss
    for log_prob, returns in zip(log_prob_eps, returns_eps):
      policy_loss_eps.append(-log_prob * returns) # torch tensor * torch tensor
    loss = torch.cat(policy_loss_eps).sum() # cat -> makes into one torch tensor, sum() -> summation

    # Backward propagation and gradient descent
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print information
    if episode % print_every == 0:
      print(f"Current Episode: {episode} | Average reward: {np.mean(scores_deque)} | Average length of ep: {np.mean(len_deque)}")

  return scores
```

3. Training

```
# 0. Device
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 1. Hyperparameters
car_hyperparameters = {
    "h_size": 64,
    "n_training_episodes": 2000,
    "n_evaluation_episodes": 10,
    "max_t": 999,
    "gamma": 0.9,
    "lr": 0.00001,
    "env_id": 'MountainCarContinuous-v0',
```

```
    "state_space": 2,
    "action_space": 1,
}

# 2. Build a Policy Network
car_policy = Policy(car_hyperparameters['state_space'],
                    car_hyperparameters['h_size'],
                    car_hyperparameters['action_space']).to(device)

# 3. Create environment
env = gym.make(car_hyperparameters['env_id'])

# 4. Train the network
optimizer = torch.optim.Adam(car_policy.parameters(),
                             lr = car_hyperparameters['lr'])

scores = reinforce(policy = car_policy,
                   optimizer = optimizer,
                   env = env,
                   n_episodes = car_hyperparameters['n_training_episodes'],
                   n_steps = car_hyperparameters['max_t'],
                   device = device,
                   gamma = car_hyperparameters['gamma'],
                   print_every = 100)

```

Show hidden output

4. Evaluation

```
def evaluate_agent(n_eval_episodes, n_steps, policy, env, device):

  """
  Evaluate the performance of an agent by calculating the mean and standard deviation rewards over n_eval_episodes of episodes.

  Args:
    n_eval_episodes (int): Number of evaluation episodes. \n
    n_steps (int): Maximum number of steps allowed in an episode. \n
    policy (nn.Module): A policy network. \n
    env (gymnasium.env): An environment. \n
    device (str): 'cuda' or 'cpu'. \n

  Returns:
    mean_reward (float): Mean reward scored across the evaluated episodes.
    std_reward (float): Standard deviation reward scored across the evaluated episodes.
  """


  rewards_across_episodes = [] # Contains rewards scored in each episode

  ####################
  # For each episode #
  ####################
  for episode in range(n_eval_episodes):

    # To store reward scored in each step
    rewards = []

    # Reset the environment
    state, info = env.reset()

    #################
    # For each step #
    #################
    for step in range(n_steps):

      # Sample an action
      action, _ = policy.act(torch.tensor(state).unsqueeze(0).to(device))

      # Step the environment by taking the action
      state, reward, terminated, truncated, info = env.step(np.array([action]))

      # Store the reward scored in this step
      rewards.append(reward)

      # Check if truncated or terminated
      if truncated or terminated:
        break

    # Sum the reward scored in the entire episode and store it
```

```
    rewards_across_episodes.append(sum(rewards))

  # Calculate the mean and standard deviation
  mean_reward = np.array(rewards_across_episodes).mean()
  std_reward = np.array(rewards_across_episodes).std()
  return mean_reward, std_reward

```

```
mean_reward, std_reward = evaluate_agent(n_eval_episodes = car_hyperparameters['n_evaluation_episodes'],
                                         n_steps = car_hyperparameters['max_t'],
                                         policy = car_policy,
                                         env = env,
                                         device = device)

print(f"Mean reward: {mean_reward}, Std reward: {std_reward}")
```

```
Mean reward: -46.63831101175287, Std reward: 1.2278728245384134
```

5. Push to Hub

```
def record_video(env, policy, out_directory, device, fps=30):
  """
  Generate a replay video of the agent
  :param env
  :param Qtable: Qtable of our agent
  :param out_directory
  :param fps: how many frame per seconds (with taxi-v3 and frozenlake-v1 we use 1)
  """
  images = []
  state, info = env.reset()
  terminated = False
  truncated = False
  img = env.render()
  images.append(img)
  while not terminated and not truncated:
    # Take the action (index) that have the maximum expected future reward given that state
    action, _ = policy.act(torch.tensor(state).unsqueeze(0).to(device))
    state, reward, terminated, truncated, info = env.step(np.array([action])) # We directly put next_state = state for recording lo
    img = env.render()
    images.append(img)
  imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)], fps=fps)
```

```
def push_to_hub(repo_id,
                model,
                hyperparameters,
                eval_env,
                video_fps=30
                ):
  """
  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: repo_id: id of the model repository from the Hugging Face Hub
  :param model: the pytorch model we want to save
  :param hyperparameters: training hyperparameters
  :param eval_env: evaluation environment
  :param video_fps: how many frame per seconds to record our video replay
  """

  _, repo_name = repo_id.split("/")
  api = HfApi()

  # Step 1: Create the repo
  repo_url = api.create_repo(
        repo_id=repo_id,
        exist_ok=True,
  )

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

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

```

```
  # Step 3: Save the hyperparameters to JSON
  with open(local_directory / "hyperparameters.json", "w") as outfile:
    json.dump(hyperparameters, outfile)

  # Step 4: Evaluate the model and build JSON
  mean_reward, std_reward = evaluate_agent(hyperparameters["n_evaluation_episodes"],
                                           hyperparameters["max_t"],
                                           model,
                                           eval_env,
                                           'cuda')
  # Get datetime
  eval_datetime = datetime.datetime.now()
  eval_form_datetime = eval_datetime.isoformat()

  evaluate_data = {
        "env_id": hyperparameters["env_id"],
        "mean_reward": mean_reward,
        "n_evaluation_episodes": hyperparameters["n_evaluation_episodes"],
        "eval_datetime": eval_form_datetime,
  }

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

  # Step 5: Create the model card
  env_name = hyperparameters["env_id"]
  env_id = env_name

  metadata = {}
  metadata["tags"] = [
        env_name,
        "reinforce",
        "reinforcement-learning",
        "custom-implementation",
        "deep-rl-class"
    ]

  # Add metrics
  eval = metadata_eval_result(
      model_pretty_name=repo_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_name,
      dataset_id=env_name,
    )

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

  model_card = f"""
# **Reinforce** Agent playing **{env_id}**
This is a trained model of a **Reinforce** agent playing **{env_id}** .
To learn to use this model and train yours check Unit 4 of the Deep Reinforcement Learning Course: https://huggingface.co/deep-rl
"""

  readme_path = local_directory / "README.md"
  readme = ""
  if readme_path.exists():
      with readme_path.open("r", encoding="utf8") as f:
        readme = f.read()
  else:
    readme = 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)

  # Step 6: Record a video
  video_path =  local_directory / "replay.mp4"
  record_video(eval_env, model, video_path, 'cuda', video_fps)

  # Step 7. Push everything to the Hub
  api.upload_folder(
        repo_id=repo_id,
        folder_path=local_directory,
        path_in_repo=".",
```

```
    )

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

```
# Login to HuggingFace Hub
notebook_login()
```

```
# Create an evaluation environment
eval_env = gym.make(car_hyperparameters['env_id'], render_mode = 'rgb_array')

# Push to Hub
push_to_hub(repo_id = "wengti0608/Reinforce-MountainCarContinuous-v0-attempt1",
            model = car_policy,
            hyperparameters = car_hyperparameters,
            eval_env = eval_env)
```

```
WARNING:imageio_ffmpeg:IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from (600, 400)
Uploading...: 100%                    127k/127k [00:01<00:00, 637kB/s]
Your model is pushed to the Hub. You can view your model here: https://huggingface.co/wengti0608/Reinforce-MountainCarContinuous-v0
```

Solving Mountain Car with SAC

- Reinforce did not manage to solve Continuous Mountain Car. Therefore, SAC is used instead.

```
!pip install stable-baselines3==2.0.0a5
```

Show hidden output

```
!pip install huggingface_sb3
```

Show hidden output

1. Training

```
import gymnasium as gym
from stable_baselines3.sac import SAC
from stable_baselines3.common.callbacks import EvalCallback
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.evaluation import evaluate_policy


# 1. Make environment
env = gym.make("MountainCarContinuous-v0")

# 2. Create a SAC model

# The hyperparameter is provided here:
# https://huggingface.co/sb3/sac-MountainCarContinuous-v0
policy_kwargs = {'log_std_init': -3.67,
                 'net_arch': [64, 64]}

model = SAC(batch_size = 512,
            buffer_size = 50000,
            ent_coef = 0.1,
            gamma = 0.9999,
            gradient_steps = 32,
            learning_rate = 0.0003,
            learning_starts = 0,
            policy = 'MlpPolicy',
            policy_kwargs = policy_kwargs,
            tau = 0.01,
            train_freq = 32,
            use_sde = True,
            env = env)

# 3. Train the model

# As SAC does not output rollout on its own
# A callback is manually created...
eval_env = Monitor(gym.make("MountainCarContinuous-v0", render_mode = "rgb_array"))

eval_callback = EvalCallback(
    eval_env,
```

```
    best_model_save_path = './logs/SAC',
    log_path = './logs/SAC',
    eval_freq = 1e3,
    deterministic = True,
    render = False
)

# Training
model.learn(total_timesteps = 5e4, callback = eval_callback)

# 4. Save the model
model_name = "SAC-MountainCarContinuous-v0"
model.save(model_name)

```

Show hidden output

2. Evaluation

```
#1. Create an evaluation environment
eval_env = Monitor(gym.make("MountainCarContinuous-v0", render_mode = "rgb_array"))

#2. Evaluate policy
mean_reward, std_reward = evaluate_policy(model = model,
                                          env = eval_env,
                                          n_eval_episodes = 10,
                                          deterministic = True)

print(f"The mean reward: {mean_reward} | The standard deviation: {std_reward}")
```

```
The mean reward: 94.67033819999999 | The standard deviation: 0.2572431881083725
```

3. Push to Hub

```
from huggingface_hub import notebook_login

notebook_login()
```

```
from stable_baselines3.common.vec_env import DummyVecEnv
from huggingface_sb3 import package_to_hub

package_to_hub(model = model,
               model_name = model_name,
               model_architecture = "SAC",
               env_id = "MountainCarContinuous-v0",
               eval_env = DummyVecEnv([lambda: Monitor(gym.make("MountainCarContinuous-v0", render_mode = "rgb_array"))]),
               repo_id = "wengti0608/SAC-MountainCarContinuous-v0",
               commit_message = "First Commit")
```

Show hidden output