#Ex8 Q2: Policy-gradient methods -->

```
!pip install gymnasium

from collections import deque
from gymnasium import spaces
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn
from torch.distributions import Categorical
import torch.nn.functional as F
import tqdm
```

Four Rooms environment

In the question, we will implement several policy-gradient methods and apply them once again on our favorite domain, Four Rooms. The environment is implemented below in a Gymnasium-like interface. Code for plotting learning curves with confidence bands is also provided.

```
class FourRooms(object):
    def __init__(self):
        # The grid for the Four Rooms domain
        self.grid = np.array([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]])

        # Observation (state) space consists of all empty cells
        # To improve interpretability, we flip the coordinates from
(row_idx, column_idx) -> (x, y),
        # where x = column_idx, y = 10 - row_idx
        self.observation_space = np.argwhere(self.grid ==
0.0).tolist()  # Fine all empty cells
        self.observation_space =
self.arr_coords_to_four_room_coords(self.observation_space)

        # Action space
        self.action_movement = {0: np.array([0, 1]),  # up
                                1: np.array([0, -1]),  # down
```

```
                                2: np.array([-1, 0]),  # left
                                3: np.array([1, 0])}  # right
        self.action_space = spaces.Discrete(4)

        # Start location
        self.start_location = [0, 0]

        # Goal location
        self.goal_location = [10, 10]

        # Wall locations
        self.walls = np.argwhere(self.grid == 1.0).tolist()  # find
all wall cells
        self.walls = self.arr_coords_to_four_room_coords(self.walls)
# convert to Four Rooms coordinates

        # This is an episodic task, with a timeout of 459 steps
        self.max_time_steps = 459

        # Tracking variables during a single episode
        self.agent_location = None  # Track the agent's location in
one episode.
        self.action = None  # Track the agent's action
        self.t = 0  # Track the current time step in one episode

    @staticmethod
    def arr_coords_to_four_room_coords(arr_coords_list):
        """
        Function converts the array coordinates ((row, col), origin is
top left)
        to the Four Rooms coordinates ((x, y), origin is bottom left)
        E.g., The coordinates (0, 0) in the numpy array is mapped to
(0, 10) in the Four Rooms coordinates.
        Args:
            arr_coords_list (list): List variable consisting of tuples
of locations in the numpy array

        Return:
            four_room_coords_list (list): List variable consisting of
tuples of converted locations in the
                                          Four Rooms environment.
        """
        # Flip the coordinates from (row_idx, column_idx) -> (x, y),
        # where x = column_idx, y = 10 - row_idx
        four_room_coords_list = [(column_idx, 10 - row_idx) for
(row_idx, column_idx) in arr_coords_list]
        return four_room_coords_list

    def reset(self):
        # Reset the agent's location to the start location
```

```python
        self.agent_location = self.start_location

        # Reset the timeout tracker to be 0
        self.t = 0

        # Reset the information
        info = {}
        return self.agent_location, info

    def step(self, action):
        """
        Args:
            action (int): Int variable (i.e., 0 for "up"). See
self.action_movement above for more details.
        """
        # With probability 0.8, the agent takes the correct direction.
        # With probability 0.2, the agent takes one of the two
perpendicular actions.
        # For example, if the correct action is "LEFT", then
        #     - With probability 0.8, the agent takes action "LEFT";
        #     - With probability 0.1, the agent takes action "UP";
        #     - With probability 0.1, the agent takes action "DOWN".
        if np.random.uniform() < 0.2:
            if action == 2 or action == 3:
                action = np.random.choice([0, 1], 1)[0]
            else:
                action = np.random.choice([2, 3], 1)[0]

        # Convert the agent's location to array
        loc_arr = np.array(self.agent_location)

        # Convert the action name to movement array
        act_arr = self.action_movement[action]

        # Compute the agent's next location
        next_agent_location = np.clip(loc_arr + act_arr,
                                      a_min=np.array([0, 0]),
                                      a_max=np.array([10,
10])).tolist()

        # Check if the agent crashes into walls; if so, it stays at
the current location.
        if tuple(next_agent_location) in self.walls:
            next_agent_location = self.agent_location

        # Compute the reward (1 iff next state is goal location)
        reward = 1.0 if next_agent_location == self.goal_location else
0.0

        # Check termination/truncation
```

```
        # If agent reaches the goal, reward = 1, terminated = True
        # If timeout is reached, reward = 0, truncated = True
        terminated = False
        truncated = False
        if reward == 1.0:
            terminated = True
        elif self.t == self.max_time_steps:
            truncated = True

        # Update the agent's location, action, and time step trackers
        self.agent_location = next_agent_location
        self.action = action
        self.t += 1

        return next_agent_location, reward, terminated, truncated, {}

    def render(self):
        # Plot the agent and the goal
        # empty cell = 0
        # wall cell = 1
        # agent cell = 2
        # goal cell = 3
        plot_arr = self.grid.copy()
        plot_arr[10 - self.agent_location[1], self.agent_location[0]]
= 2
        plot_arr[10 - self.goal_location[1], self.goal_location[0]] =
3
        plt.clf()
        plt.title(f"state={self.agent_location},
act={self.action_movement[self.action]}")
        plt.imshow(plot_arr)
        plt.show(block=False)
        plt.pause(0.1)

    @staticmethod
    def test():
        env = FourRooms()
        state, info = env.reset()

        for _ in range(1000):
            action = env.action_space.sample()
            next_state, reward, terminated, truncated, info =
env.step(action)
            env.render()
            if terminated or truncated:
                state, info = env.reset()
            else:
                state = next_state
```

```
# Un-comment to run test function
# FourRooms.test()

def moving_average(data, *, window_size = 50):
    """Smooths 1-D data array using a moving average.

    Args:
        data: 1-D numpy.array
        window_size: Size of the smoothing window

    Returns:
        smooth_data: A 1-d numpy.array with the same size as data
    """
    assert data.ndim == 1
    kernel = np.ones(window_size)
    smooth_data = np.convolve(data, kernel) / np.convolve(
        np.ones_like(data), kernel
    )
    return smooth_data[: -window_size + 1]

def plot_curves(arr_list, legend_list, color_list, ylabel, fig_title,
smoothing = True):
    """
    Args:
        arr_list (list): List of results arrays to plot
        legend_list (list): List of legends corresponding to each
result array
        color_list (list): List of color corresponding to each result
array
        ylabel (string): Label of the vertical axis

        Make sure the elements in the arr_list, legend_list, and
color_list
        are associated with each other correctly (in the same order).
        Do not forget to change the ylabel for different plots.
    """
    # Set the figure type
    fig, ax = plt.subplots(figsize=(12, 8))

    # PLEASE NOTE: Change the vertical labels for different plots
    ax.set_ylabel(ylabel)
    ax.set_xlabel("Time Steps")

    # Plot results
    h_list = []
    for arr, legend, color in zip(arr_list, legend_list, color_list):
        # Compute the standard error (of raw data, not smoothed)
        arr_err = arr.std(axis=0) / np.sqrt(arr.shape[0])
        # Plot the mean
        averages = moving_average(arr.mean(axis=0)) if smoothing else
```

```
arr.mean(axis=0)
        h, = ax.plot(range(arr.shape[1]), averages, color=color,
label=legend)
        # Plot the confidence band
        arr_err *= 1.96
        ax.fill_between(range(arr.shape[1]), averages - arr_err,
averages + arr_err, alpha=0.3,
                        color=color)
        # Save the plot handle
        h_list.append(h)

    # Plot legends
    ax.set_title(f"{fig_title}")
    ax.legend(handles=h_list)

    plt.show()
```

Part (a): REINFORCE algorithm

Policy network

We'll develop and train a neural network to represent a stochastic policy (\pi(a|s)). This network's structure will include:

- **Layer 1**: A linear layer with an input dimension of 3 (reflecting the size of the observation space) and an output dimension of 128.
- **Activation 1**: A ReLU activation function follows Layer 1.
- **Layer 2**: Another linear layer that takes the 128-dimensional output from the previous layer and produces a 4-dimensional output (matching the size of the action space).
- **Activation 2**: A Softmax activation function to convert the output into a probability distribution over actions.

To preprocess the input observations, instead of directly using the raw state ([x,y]), we transform each state into ([10x,10y,1]) to normalize the inputs.

```
class PolicyNet(nn.Module):
    def __init__(self):
        super(PolicyNet, self).__init__()

        """ YOUR CODE HERE:
                Implement the neural network here
        """
        self.layer1 = nn.Linear(3, 128)
        self.activation1 = nn.ReLU()
        self.layer2 = nn.Linear(128, 4)
        self.activation2 = nn.Softmax(dim=-1)
        ...

    def forward(self, x):
```

```
        """ YOUR CODE HERE:
                Implement the forward propagation
        """
        x = self.layer1(x)
        x = self.activation1(x)
        x = self.layer2(x)
        x = self.activation2(x)
        return x

        ...
```

REINFORCE agent with policy network

```
class REINFORCEAgent(object):
    def __init__(self):
        # Create the policy network
        self.policy_net = PolicyNet()

    def get_action(self, state):
        """ Function to derive an action given a state
            Args:
                state (list): [x/10, y/10, 1]

            Returns:
                action index (int), log_prob (ln(\pi(action|state)))
        """
        state_tensor = torch.tensor(state).float().view(1, -1)
        probs = self.policy_net(state_tensor)
        m = Categorical(probs)
        action = m.sample()
        log_prob = m.log_prob(action)
        return action.item(), log_prob
```

REINFORCE training loop

```
class REINFORCEAgentTrainer(object):
    def __init__(self, agent, env, params):
        # Agent object
        self.agent = agent

        # Environment object
        self.env = env

        # Training parameters
        self.params = params

        # Lists to store the log probabilities and rewards for one
episode
        self.saved_log_probs = []
        self.saved_rewards = []
```

```
        # Gamma
        self.gamma = params['gamma']

        # Create the optimizer
        """ YOUR CODE HERE:
                Use the Adam optimizer with the learning rate in
params
        """
        self.optimizer =
torch.optim.Adam(self.agent.policy_net.parameters(),
lr=params['learning_rate'])


    @staticmethod
    def compute_state_feature(state):
        return [state[0] / 10, state[1] / 10, 1]

    def update_agent_policy_network(self):
        # List to store the policy loss for each time step
        policy_loss = []

        # List to store the return for each time step
        returns = deque()

        """ YOUR CODE HERE:
                Compute the return for each time step

                Hint: We usually compute the return from the end.
Remember to append it
                      correctly. You can use "returns.appendleft(G)"
        """
        # Compute returns for every time step
        G = 0
        for r in self.saved_rewards[::-1]:
            G = r + self.gamma * G
            returns.appendleft(G)

        returns = torch.tensor(returns)

        """ YOUR CODE HERE:
                We now have the return and log probability for each
time step.
                Compute the `policy loss' for each time step
                (whose gradient appears in the pseudocode).
        """
        for log_prob, r in zip(self.saved_log_probs, returns):
            # Compute the policy loss for each time step
            policy_loss.append(-log_prob * r)
```

```
        # Sum all the policy loss terms across all time steps
        policy_loss = torch.cat(policy_loss).sum()

        """ YOUR CODE HERE:
                Implement one step of backpropagation (gradient
descent)
        """

        self.optimizer.zero_grad()
        policy_loss.backward()
        self.optimizer.step()

        # After backpropagation, clear the data
        del self.saved_log_probs[:]
        del self.saved_rewards[:]

        return returns[0].item(), policy_loss.item()

    def rollout(self):
        """ Function to collect one episode from the environment
        """

        """ YOUR CODE HERE:
                Implement the code to collect one episode.

                Specifically, we only collect the rewards and
corresponding log probability, which
                should be stored in "self.saved_rewards" and
"self.saved_log_probs", respectively.

                This is because we only need the return at each time
step and log probability
                to update the weights of the policy.

        """

        state, info = self.env.reset()
        done = False

        while not done:
            state_feature = self.compute_state_feature(state)

            action, log_prob =
self.agent.get_action(state=state_feature)
            # Log the probability of the selected action
            self.saved_log_probs.append(log_prob)

            # Take action in the environment
            next_state, reward, done, _ , _ = self.env.step(action)

            # Store reward
```

```
            self.saved_rewards.append(reward)

            state = next_state

    def run_train(self):
        # Lists to store the returns and losses during the training
        train_returns = []
        train_losses = []

        # Training loop
        ep_bar = tqdm.trange(self.params['num_episodes'])
        for ep in ep_bar:
            """ YOUR CODE HERE:
                    Implement the REINFORCE algorithm here.
            """
            # Collect one episode
            self.rollout()

            # Update the policy using the collected episode
            G, loss = self.update_agent_policy_network()

            # Save the return and loss
            train_returns.append(G)
            train_losses.append(loss)

            # Add description
            ep_bar.set_description(f"Episode: {ep} | Return: {G} |
Loss: {loss:.2f}")

        return train_returns, train_losses
```

Evaluation of REINFORCE on Four Rooms

We will run 10 trials with 10K episodes as in past assignments, which will take between 1-2 hours. If this takes too long, you can halve both the trials and number of episodes within each trial. As usual, you should set this to be much lower during development and debugging.

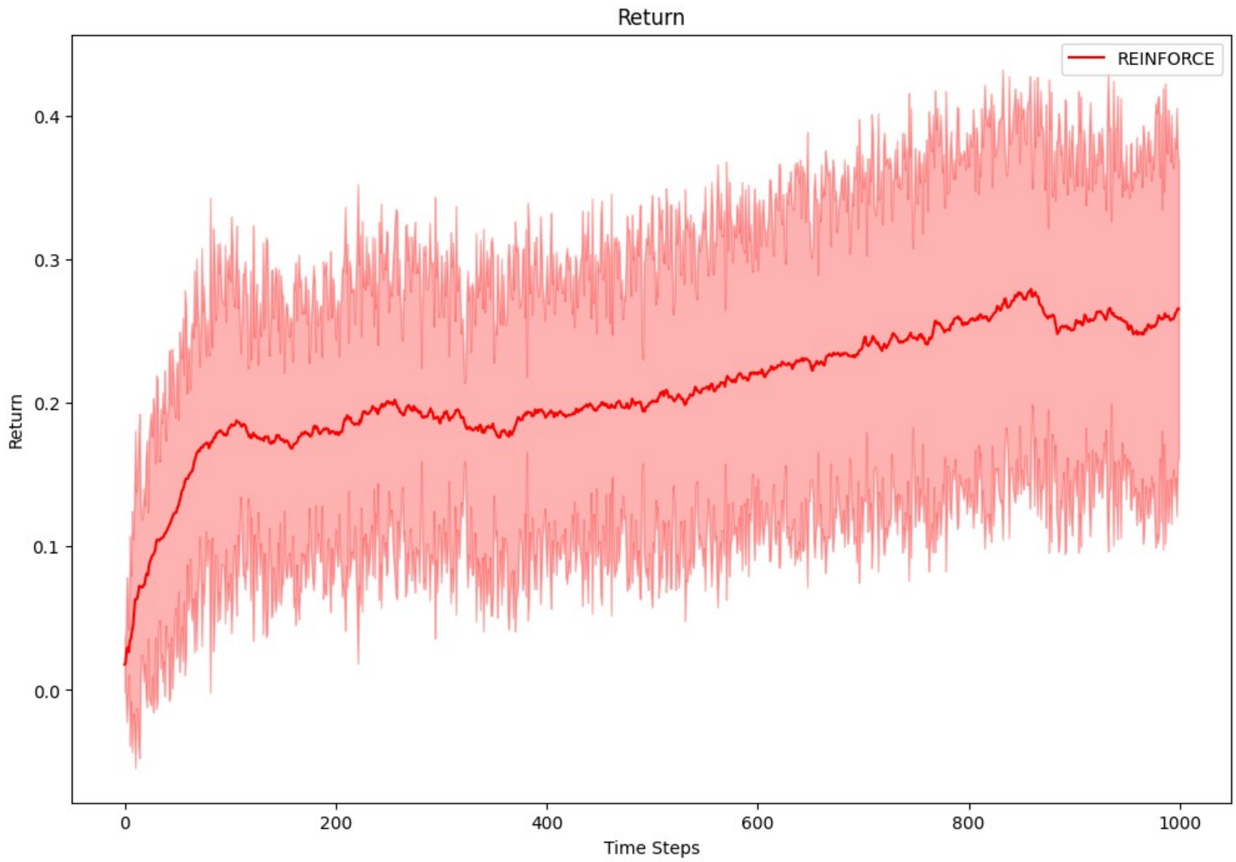
```
if __name__ == "__main__":
    my_env = FourRooms()

    train_params = {
        'num_episodes': 1000,
        'num_trials': 10,
        'learning_rate': 1e-3,
        'gamma': 0.99
    }

    reinforce_returns = []
    reinforce_losses = []
    for _ in range(train_params['num_trials']):
```

```
        my_agent = REINFORCEAgent()
        my_trainer = REINFORCEAgentTrainer(my_agent, my_env, 
train_params)
        returns, losses = my_trainer.run_train()

        reinforce_returns.append(returns)
        reinforce_losses.append(losses)
```

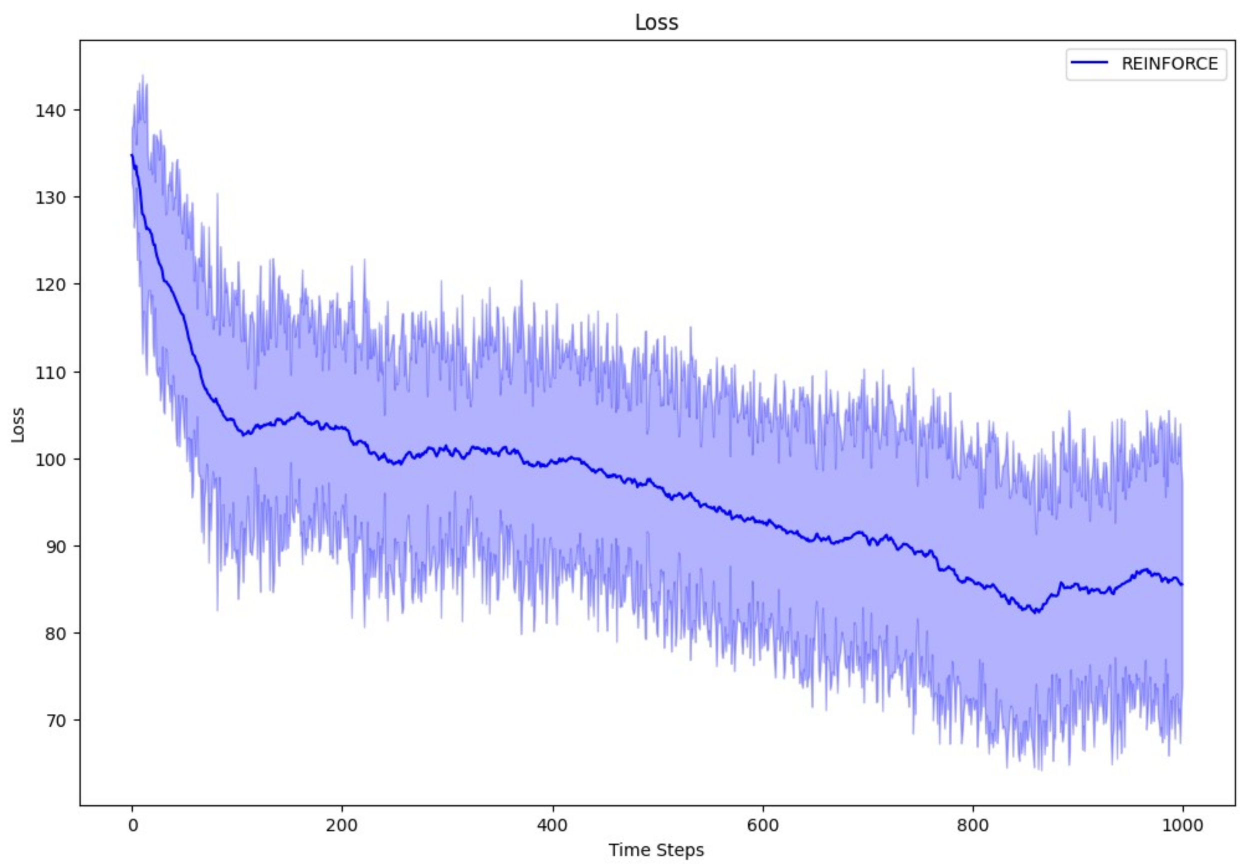
```
Episode: 999 | Return: 0.2734891474246979 | Loss: 61.45: 100%|
██████████| 1000/1000 [03:34<00:00,  4.65it/s]
Episode: 999 | Return: 0.41712087392807007 | Loss: 78.42: 100%|
██████████| 1000/1000 [03:52<00:00,  4.31it/s]
Episode: 999 | Return: 0.5362682342529297 | Loss: 57.37: 100%|
██████████| 1000/1000 [03:34<00:00,  4.66it/s]
Episode: 999 | Return: 0.6050060391426086 | Loss: 44.28: 100%|
██████████| 1000/1000 [03:40<00:00,  4.53it/s]
Episode: 999 | Return: 0.6689717769622803 | Loss: 35.14: 100%|
██████████| 1000/1000 [05:09<00:00,  3.23it/s]
Episode: 999 | Return: 0.5471566319465637 | Loss: 54.97: 100%|
██████████| 1000/1000 [03:15<00:00,  5.11it/s]
Episode: 999 | Return: 0.6491026282310486 | Loss: 39.57: 100%|
██████████| 1000/1000 [03:50<00:00,  4.34it/s]
Episode: 999 | Return: 0.4298890233039856 | Loss: 56.50: 100%|
██████████| 1000/1000 [03:48<00:00,  4.38it/s]
Episode: 999 | Return: 0.5255964994430542 | Loss: 58.36: 100%|
██████████| 1000/1000 [04:19<00:00,  3.86it/s]
Episode: 999 | Return: 0.015438523143529892 | Loss: 88.74: 100%|
██████████| 1000/1000 [03:50<00:00,  4.34it/s]
```

Plot learning and loss curves

```
plot_curves([np.array(reinforce_returns)], ['REINFORCE'], ['r'], 
'Return', 'Return', smoothing = True)
plot_curves([np.array(reinforce_losses)], ['REINFORCE'], ['b'], 
'Loss', 'Loss', smoothing = True)
```



Return
REINFORCE
0.4
0.3
0.2
0.1
0.0
Return
0
200
400
600
800
1000
Time Steps



Part (b): REINFORCE with baseline

In this version of REINFORCE, we additionally learn a critic network (state-value function) to act as the baseline. In this context, the policy is sometimes referred to as the "actor", but the textbook reserves this terminology for the algorithm in part (c).

```
# Policy and value networks
class REINFORCEBaselineNet(nn.Module):
    def __init__(self):
        super(REINFORCEBaselineNet, self).__init__()

        """ YOUR CODE HERE:
                Implement the critic network here. The architecture
should be:

                Layer 1: Linear, input size 3, output size 128
                Activation 1: ReLU
                Layer 2: Linear, input size 128, output size 1
                Activation 2: Identity (or none)
        """

        # Critic network
        self.critic_layer1 = nn.Linear(3, 128)
```

```python
        self.critic_activation1 = nn.ReLU()
        self.critic_layer2 = nn.Linear(128, 1)

        """ YOUR CODE HERE:
                Implement the actor network here. The architecture
should be (same as before):

                Layer 1: Linear, input size 3, output size 128
                Activation 1: ReLU
                Layer 2: Linear, input size 128, output size 4
                Activation 2: Softmax
        """

        # Actor network
        self.actor_layer1 = nn.Linear(3, 128)
        self.actor_activation1 = nn.ReLU()
        self.actor_layer2 = nn.Linear(128, 4)
        self.actor_activation2 = nn.Softmax(dim=-1)

    def forward(self, x):
        """ YOUR CODE HERE:
                Implement the forward propagation for both actor and
critic networks
        """

        # Forward pass through the critic network
        critic_x = self.critic_layer1(x)
        critic_x = self.critic_activation1(critic_x)
        critic_x = self.critic_layer2(critic_x)
        state_value = critic_x

        # Forward pass through the actor network
        actor_x = self.actor_layer1(x)
        actor_x = self.actor_activation1(actor_x)
        actor_x = self.actor_layer2(actor_x)
        action_probs = self.actor_activation2(actor_x)

        return state_value, action_probs

# REINFORCE-with-baseline agent
class REINFORCEBaselineAgent(object):
    def __init__(self):
        # Create the actor and critic networks
        self.policy_net = REINFORCEBaselineNet()

    def get_action(self, state):
        # Sample an action from the actor network, return the action
and its log probability,
        # and return the state value according to the critic network
        state_tensor = torch.tensor(state).float().view(1, -1)
```

```
        state_value, action_probs = self.policy_net(state_tensor)
        m = Categorical(action_probs)
        action = m.sample()
        log_prob = m.log_prob(action)
        return action.item(), log_prob, state_value

# REINFORCE-with-baseline training loop
class REINFORCEBaselineAgentTrainer(object):
    def __init__(self, agent, env, params):
        # Agent object
        self.agent = agent

        # Environment object
        self.env = env

        # Training parameters
        self.params = params

        # Lists to store the log probabilities, state values, and
rewards for one episode
        self.saved_log_probs = []
        self.saved_state_values = []
        self.saved_rewards = []

        # Gamma
        self.gamma = params['gamma']

        # Create the optimizer
        """ YOUR CODE HERE:
                Implement the Adam optimizer with the learning rate in
params
        """
        self.optimizer =
torch.optim.Adam(self.agent.policy_net.parameters(),
lr=params['learning_rate'])

    @staticmethod
    def compute_state_feature(state):
        return [state[0] / 10, state[1] / 10, 1]

    def update_agent_policy_network(self):
        # List to store the policy loss for each time step
        policy_loss = []

        # List to store the value loss for each time step
        value_loss = []

        # List to store the return for each time step
        returns = deque()
```

```
        """ YOUR CODE HERE:
                Compute the return for each time step

                Hint: We usually compute the return from the end.
Remember to append it
                      correctly. You can use "returns.appendleft(G)"
        """
        # Compute returns for every time step

        G = 0
        for r in self.saved_rewards[::-1]:
            G = r + self.gamma * G
            returns.appendleft(G)

        returns = torch.tensor(returns)

        """ YOUR CODE HERE:
                We now have the return, state value, and log
probability for each time step.
                Compute the `policy loss' and `value loss' for each
time step
                (whose gradients appear in the pseudocode).
        """
        for log_prob, val, r in zip(self.saved_log_probs,
self.saved_state_values, returns):
            # Compute the policy and value loss for each time step
            advantage = r - val.item()

            # Policy loss
            policy_loss.append(-log_prob * advantage)

            # Value loss
            value_loss.append(0.5 * advantage ** 2)

        # Compute the total loss
        total_loss = torch.stack(policy_loss).sum() +
torch.stack(value_loss).sum()

        """ YOUR CODE HERE:
                Implement one step of backpropagation (gradient
descent)
        """

        self.optimizer.zero_grad()
        total_loss.backward()
        self.optimizer.step()


        # After backpropagation, clear the data
        del self.saved_log_probs[:]
```

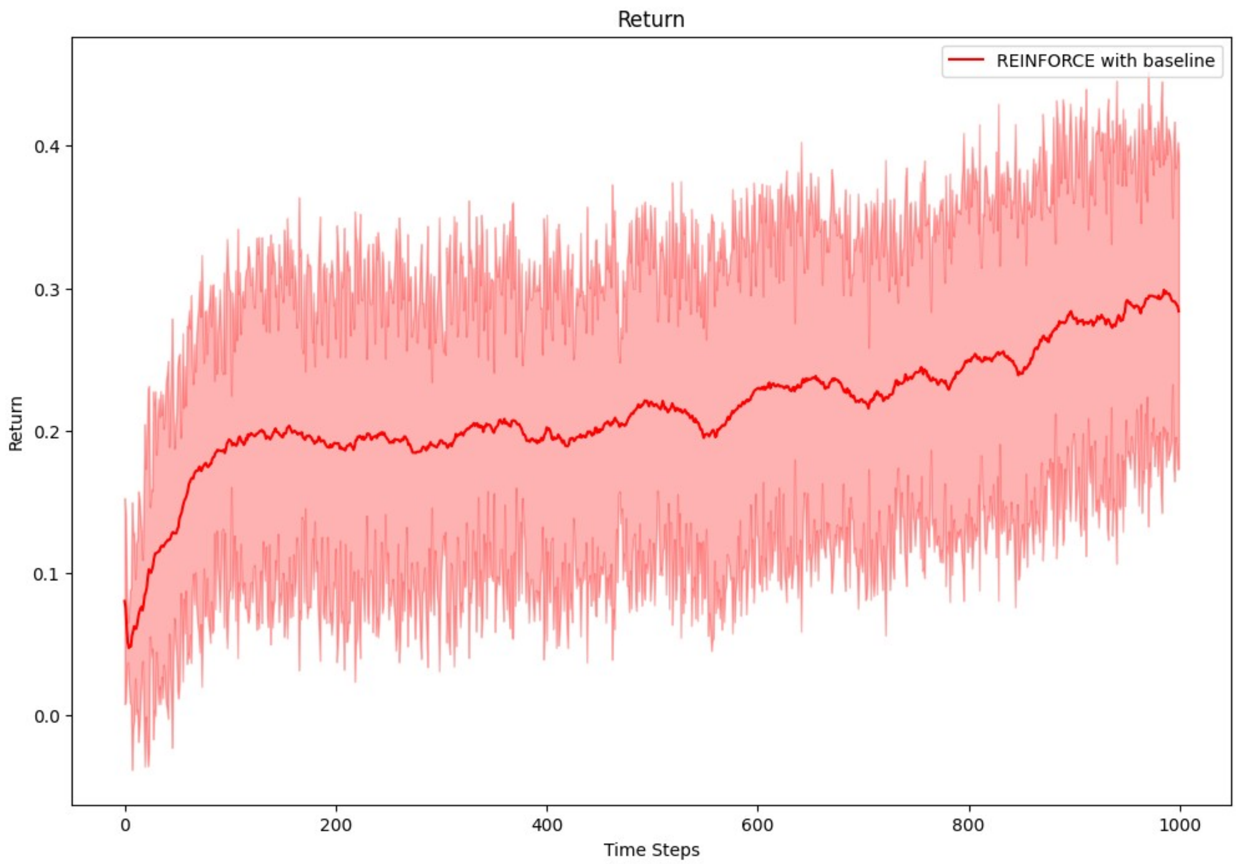
```
        del self.saved_state_values[:]
        del self.saved_rewards[:]

        return returns[0].item(), total_loss.item()

    def rollout(self):
        """ Function to collect one episode from the environment
        """

        """ YOUR CODE HERE:
                Implement the code to collect one episode.

                Collect the rewards, state valuess and log
probabilities, which should be stored in
                "self.saved_rewards", "self.saved_state_values", and
"self.saved_log_probs" respectively.
        """

        state, info = self.env.reset()
        done = False

        while not done:
            state_feature = self.compute_state_feature(state)

            action, log_prob, state_value =
self.agent.get_action(state=state_feature)

            # Log the probability of the selected action
            self.saved_log_probs.append(log_prob)

            # Log the state value
            self.saved_state_values.append(state_value)

            # Take action in the environment
            next_state, reward, done, _, _ = self.env.step(action)

            # Store reward
            self.saved_rewards.append(reward)

            state = next_state

    def run_train(self):
        # Lists to store the returns and losses during the training
        train_returns = []
        train_losses = []

        # Training loop
        ep_bar = tqdm.trange(self.params['num_episodes'])
        for ep in ep_bar:
            """ YOUR CODE HERE:
                    Implement the REINFORCE-with-baseline algorithm
```

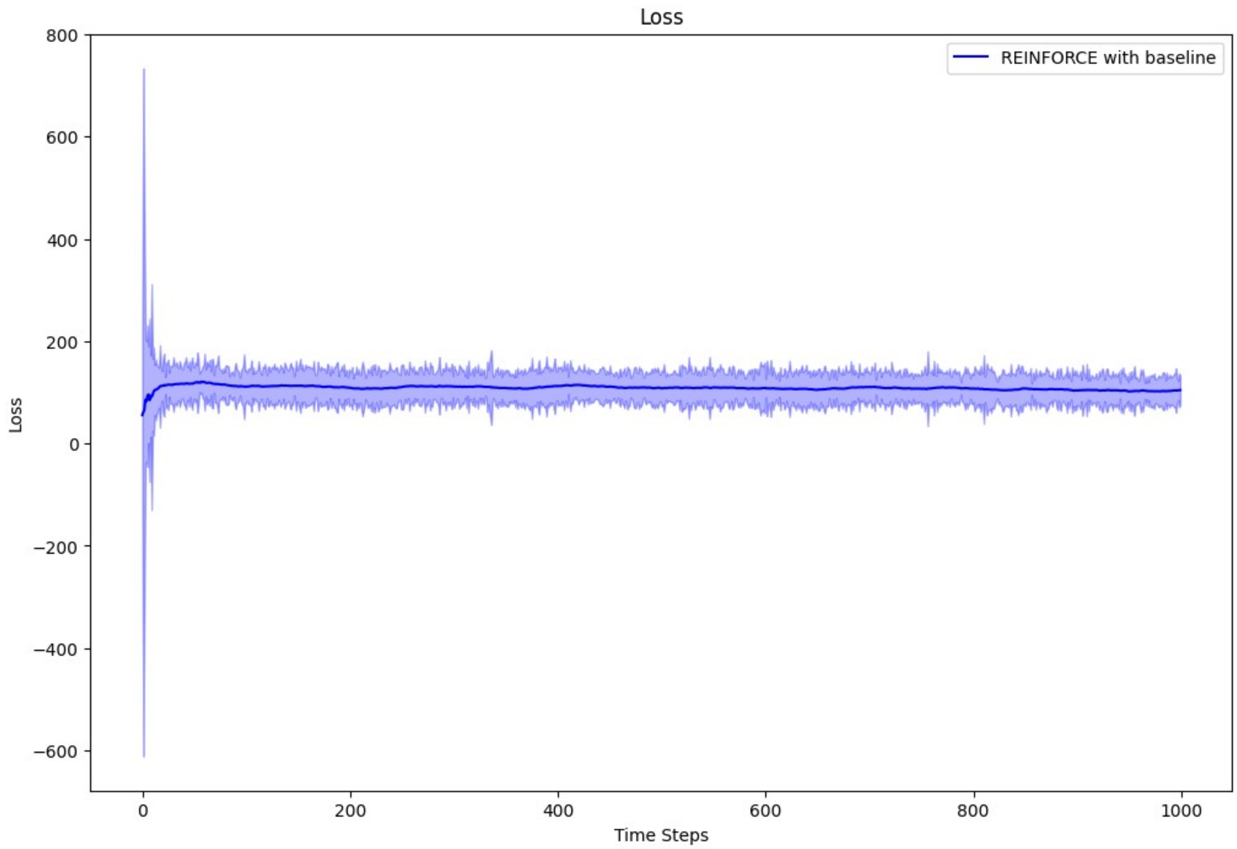
```
here.
            """
            # Collect one episode
            self.rollout()

            # Update the policy using the collected episode
            G, loss = self.update_agent_policy_network()

            # Save the return and loss
            train_returns.append(G)
            train_losses.append(loss)

            # Add description
            ep_bar.set_description(f"Episode: {ep} | Return: {G} |
Loss: {loss:.2f}")

        return train_returns, train_losses

if __name__ == "__main__":
    my_env = FourRooms()

    train_params = {
        'num_episodes': 1000,
        'num_trials': 10,
        'learning_rate': 1e-3,
        'gamma': 0.99
    }

    reinforce_baseline_returns = []
    reinforce_baseline_losses = []
    for _ in range(train_params['num_trials']):
        my_agent = REINFORCEBaselineAgent()
        my_trainer = REINFORCEBaselineAgentTrainer(my_agent, my_env,
train_params)
        returns, losses = my_trainer.run_train()

        reinforce_baseline_returns.append(returns)
        reinforce_baseline_losses.append(losses)

Episode: 999 | Return: 0.20027703046798706 | Loss: 180.25: 100%|
██████████| 1000/1000 [05:42<00:00,  2.92it/s]
Episode: 999 | Return: 0.06764444708824158 | Loss: 27.57: 100%|
██████████| 1000/1000 [04:05<00:00,  4.07it/s]
Episode: 999 | Return: 0.724980354309082 | Loss: 34.66: 100%|
██████████| 1000/1000 [04:05<00:00,  4.08it/s]
Episode: 999 | Return: 0.29048848152160645 | Loss: 167.87: 100%|
██████████| 1000/1000 [04:48<00:00,  3.47it/s]
Episode: 999 | Return: 0.4255901277065277 | Loss: 80.92: 100%|
██████████| 1000/1000 [03:58<00:00,  4.20it/s]
Episode: 999 | Return: 0.2574846148490906 | Loss: 82.91: 100%|
```

```
██████████| 1000/1000 [03:14<00:00,  5.14it/s]
Episode: 999 | Return: 0.28470778465270996 | Loss: 144.26: 100%|
██████████| 1000/1000 [04:42<00:00,  3.54it/s]
Episode: 999 | Return: 0.03484616428613663 | Loss: 568.19: 100%|
██████████| 1000/1000 [04:42<00:00,  3.54it/s]
Episode: 999 | Return: 0.15267972648143768 | Loss: 170.61: 100%|
██████████| 1000/1000 [04:58<00:00,  3.35it/s]
Episode: 999 | Return: 0.03484616428613663 | Loss: 162.92: 100%|
██████████| 1000/1000 [04:26<00:00,  3.76it/s]
```

Plot learning and loss curves

```
plot_curves([np.array(reinforce_baseline_returns)], ['REINFORCE with baseline'], ['r'], 'Return', 'Return', smoothing = True)
plot_curves([np.array(reinforce_baseline_losses)], ['REINFORCE with baseline'], ['b'], 'Loss', 'Loss', smoothing = True)
```





Part (c): One-step actor-critic

Develop a one-step actor-critic algorithm and apply it to the Four Rooms environment, leveraging much of the code from previous sections. Note that, unlike REINFORCE which updates at the end of an episode based on a Monte-Carlo learning target, the actor-critic method allows for updates at every step of the environment using the one-step Temporal Difference (TD) error.

```
""" YOUR CODE HERE:
        Implement one-step actor-critic
"""
class OneStepActorCriticAgentTrainer(object):
    def __init__(self, agent, env, params):
        self.agent = agent
        self.env = env
        self.params = params

        self.saved_log_probs = []
        self.saved_state_values = []
        self.saved_rewards = []

        self.gamma = params['gamma']
        self.optimizer =
```

```
torch.optim.Adam(self.agent.policy_net.parameters(),
lr=params['learning_rate'])

    @staticmethod
    def compute_state_feature(state):
        return [state[0] / 10, state[1] / 10, 1]

    def update_agent_policy_network(self):
        policy_loss = []
        value_loss = []

        # Calculate TD errors and update policy and value networks
        for log_prob, val, r, next_val in zip(self.saved_log_probs,
self.saved_state_values,
                                               self.saved_rewards,
self.saved_state_values[1:]):
            td_target = r + self.gamma * next_val
            td_error = td_target - val.item()

            # Policy loss
            policy_loss.append(-log_prob * td_error)

            # Value loss
            value_loss.append(0.5 * td_error ** 2)

        # Compute total loss
        total_loss = torch.stack(policy_loss).sum() +
torch.stack(value_loss).sum()

        # Backpropagation
        self.optimizer.zero_grad()
        total_loss.backward()
        self.optimizer.step()

        # Clear data
        del self.saved_log_probs[:]
        del self.saved_state_values[:]
        del self.saved_rewards[:]

        return total_loss.item()

    def rollout(self):
        state, info = self.env.reset()
        done = False
        i = 1

        while not done and i<=100:
            state_feature = self.compute_state_feature(state)
            # state_tensor =
torch.FloatTensor(state_feature).unsqueeze(0)
```

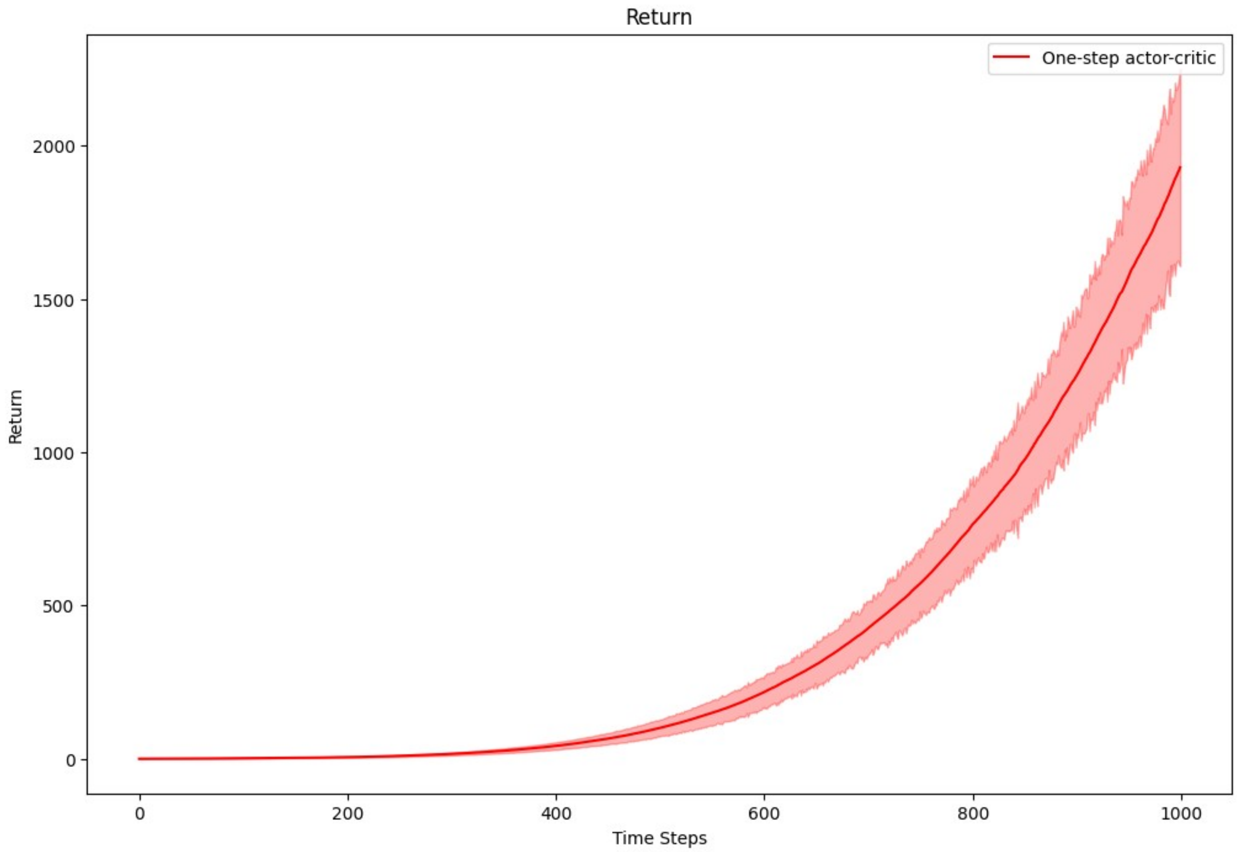
```
            # # Get action probabilities and state value from the
actor-critic network
            # state_value, action_probs = self.agent(state_tensor)

            # # Sample action from the action probabilities
            # m = torch.distributions.Categorical(action_probs)
            # action = m.sample()
            action, log_prob, state_value =
self.agent.get_action(state=state_feature)


            # Log the probability of the selected action
            self.saved_log_probs.append(log_prob)

            # Log the state value
            self.saved_state_values.append(state_value)

            # Take action in the environment
            next_state, reward, done, _ , _= self.env.step(action)

            # Store reward
            self.saved_rewards.append(reward)
            i = i+1

            state = next_state

    def run_train(self):
      train_losses = []
      train_returns = []

      # Training loop
      ep_bar = tqdm.trange(self.params['num_episodes'])
      for ep in ep_bar:
          # Collect one episode
          self.rollout()

          # Update the policy using the collected episode
          loss = self.update_agent_policy_network()

          # Save the loss
          train_losses.append(loss)

          # Calculate return for this episode
          train_returns.append(sum(self.saved_rewards))

          # Add description
          ep_bar.set_description(f"Episode: {ep} | Loss: {loss:.2f}")

      return train_losses, train_returns
```

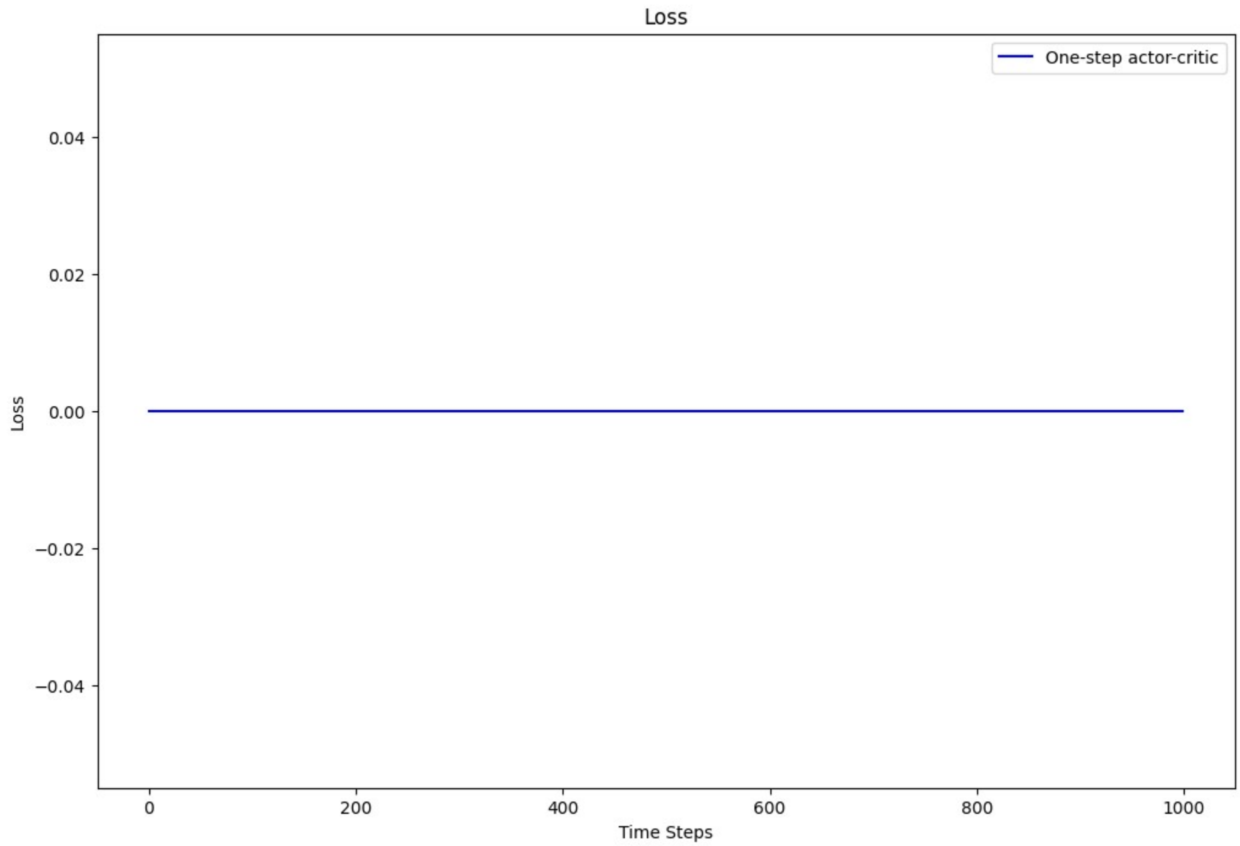
```
if __name__ == "__main__":
    my_env = FourRooms()

    train_params = {
        'num_episodes': 1000,
        'num_trials': 10,
        'learning_rate': 1e-3,
        'gamma': 0.99
    }

    actor_critic_returns = []
    actor_critic_losses = []
    for _ in range(train_params['num_trials']):
        my_agent = REINFORCEBaselineAgent()# the agent has been kept
same, only trainer is altered for this implementation
        my_trainer = OneStepActorCriticAgentTrainer(my_agent, my_env,
train_params)
        returns, losses = my_trainer.run_train()

        actor_critic_returns.append(returns)
        actor_critic_losses.append(losses)
```

```
Episode: 999 | Loss: 1305.38: 100%|██████████| 1000/1000 [02:08<00:00,
7.79it/s]
Episode: 999 | Loss: 2117.91: 100%|██████████| 1000/1000 [02:06<00:00,
7.92it/s]
Episode: 999 | Loss: 3236.75: 100%|██████████| 1000/1000 [02:04<00:00,
8.03it/s]
Episode: 999 | Loss: 2506.63: 100%|██████████| 1000/1000 [02:04<00:00,
8.02it/s]
Episode: 999 | Loss: 2184.51: 100%|██████████| 1000/1000 [02:03<00:00,
8.10it/s]
Episode: 999 | Loss: 2093.35: 100%|██████████| 1000/1000 [02:06<00:00,
7.93it/s]
Episode: 999 | Loss: 2014.89: 100%|██████████| 1000/1000 [02:03<00:00,
8.08it/s]
Episode: 999 | Loss: 1822.65: 100%|██████████| 1000/1000 [02:07<00:00,
7.85it/s]
Episode: 999 | Loss: 2861.77: 100%|██████████| 1000/1000 [02:05<00:00,
7.95it/s]
Episode: 999 | Loss: 1885.66: 100%|██████████| 1000/1000 [02:04<00:00,
8.06it/s]
```

Plot learning and loss curves

```
plot_curves([np.array(actor_critic_returns)], ['One-step actor-
critic'], ['r'], 'Return', 'Return', smoothing = True)
plot_curves([np.array(actor_critic_losses)], ['One-step actor-
critic'], ['b'], 'Loss', 'Loss', smoothing = True)
```


Return
One-step actor-critic
2000
1500
1000
500
0
Return
0
200
400
600
800
1000
Time Steps

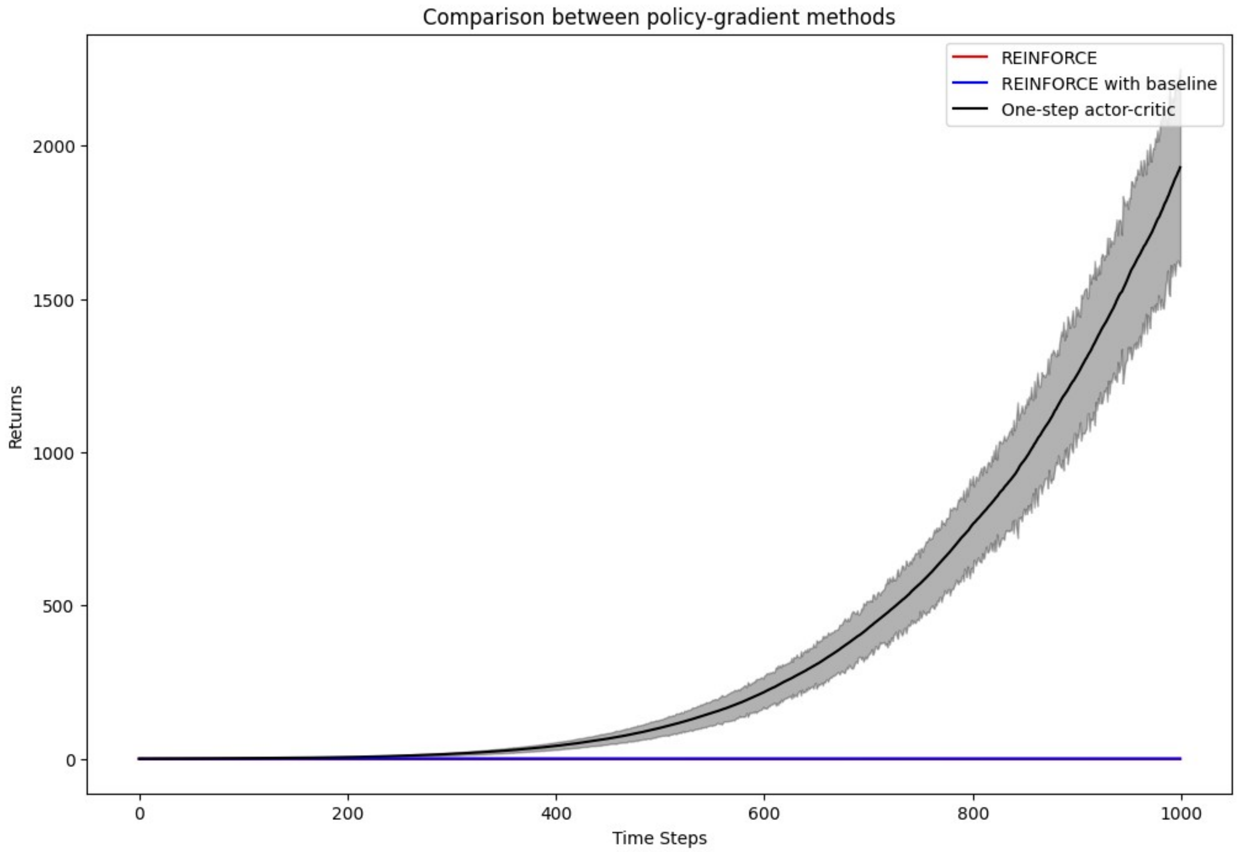


Part (d): Compare REINFORCE, REINFORCE with baseline, and one-step actor-critic.

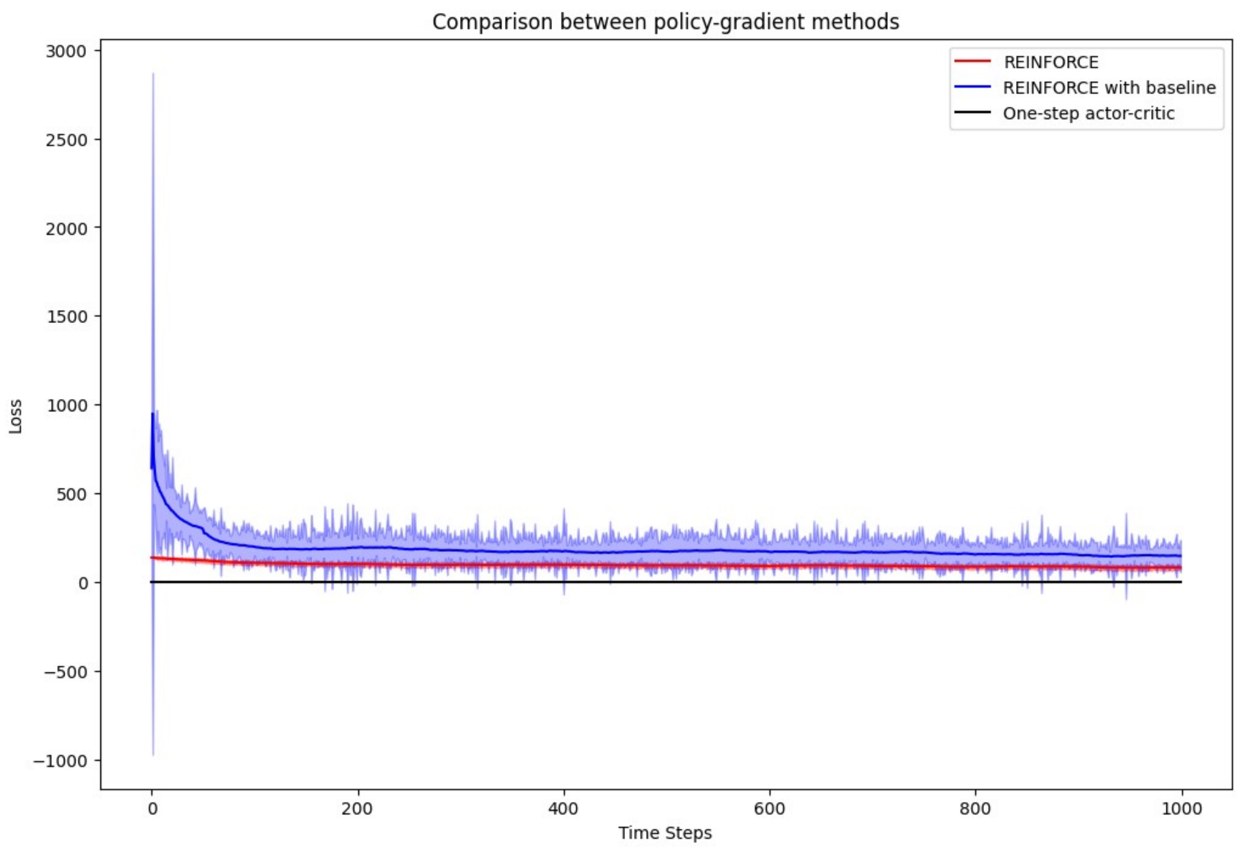
Compare the performance of these three methods and discuss your findings below.

Optional: Also compare against their tabular value-based counterparts (Monte-Carlo control and one-step SARSA), e.g., using results/implementations from previous assignments.

```
plot_curves([np.array(reinforce_returns),
np.array(reinforce_baseline_returns), np.array(actor_critic_returns)],
            ['REINFORCE', 'REINFORCE with baseline', 'One-step actor-
critic'],
            ['r', 'b', 'k'], 'Returns', "Comparison between policy-
gradient methods")
plot_curves([np.array(reinforce_losses),
np.array(reinforce_baseline_losses), np.array(actor_critic_losses)],
            ['REINFORCE', 'REINFORCE with baseline', 'One-step actor-
critic'],
            ['r', 'b', 'k'], 'Loss', "Comparison between policy-
gradient methods")
```



Comparison between policy-gradient methods
REINFORCE
REINFORCE with baseline
One-step actor-critic
Returns
Time Steps
0
500
1000
1500
2000
0
200
400
600
800
1000



The analysis, based on 1000 episodes due to computational limits, shows that the One-Step Actor-Critic outshines both Reinforce and Reinforce with Baseline in returns. Although returns for Reinforce methods improve over time, their progress is less visible on graphs adjusted for the One-Step Actor-Critic's higher returns. Loss metrics for all strategies approach zero with increased agent experience, indicating learning. Extending the number of episodes could offer clearer distinctions in performance. An added termination criterion for the One-Step Actor-Critic, introduced to address an unexplained halt after five episodes, may affect results. Overall, the methods demonstrate successful training through loss convergence and rising returns.

[Extra credit.] Part (e): Advanced policy-gradient algorithms

Many deep policy-gradient algorithms have been proposed in the past 10 years. Read about and implement one or more of these from scratch (e.g., DDPG, TD3, PPO, SAC) and evaluate them on Four Rooms. Compare with the methods above and discuss your findings.

We recommend that you first read about some of these algorithms on OpenAI's "Spinning up in deep RL" pages, although you should not directly use their implementations in this assignment. https://spinningup.openai.com/en/latest/user/algorithms.html