

Assignment 4 (114 pts)

Policy Gradients

Instructions

- This is an individual assignment. You are **not allowed** to discuss the problems with other students.
 - Part of this assignment will be autograded by gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want.
 - All your solution, code, analysis, graphs, explanations should be done in this same notebook.
 - Please make sure to execute all the cells before you submit the notebook to the gradescope. You will not get points for the plots if they are not generated already.
 - Please **do not** change the random seeds
 - Please start early. Some of the experiments take a lot of time to run on CPU.
 - If you have questions regarding the assignment, you can ask for clarifications on Piazza. You should use the corresponding tag for this assignment.
 - The deadline for submitting this assignment is **10:00 PM on Sunday, November 26, 2023**
-

This assignment has 4 parts. The goals of these parts are:

- **Part 1:** Implementing a parameterized (neural network) policy with PyTorch
- **Part 2:** Understanding and implementing the REINFORCE algorithm
- **Part 3:** Extending the REINFORCE algorithm with a baseline
- **Part 4:** Understanding and implementing Actor-Critic

When Submitting to GradeScope: Be sure to

1. Submit a .ipynb notebook to the Assignment 4 - Code section on Gradescope.
2. Submit a pdf version of the notebook to the Assignment 4 - Report entry.

Note: You can choose to submit responses in either English or French.

Before starting the assignment, make sure that you have downloaded all the tests related for the assignment and put them in the appropriate locations. If you run the next cell, we will set this all up automatically for you in a dataset called public, which contains the test cases.

Installing Dependencies

```
In [ ]: !pip install otter-grader
        !git clone https://github.com/chandar-lab/INF8250ae-assignments-2023 public
```

```
In [ ]: !pip install -U pygame
        !pip install gymnasium
        !pip install matplotlib
        !pip install tqdm
```

```
In [ ]: !apt-get install x11-utils > /dev/null 2>&1
        !pip install pyglet > /dev/null 2>&1
        !apt-get install -y xvfb python-opengl > /dev/null 2>&1
        !pip install pyvirtualdisplay > /dev/null 2>&1
        !pip install pillow
```

Importing Libraries

```
In [ ]: import otter

        grader = otter.Notebook(colab=True, tests_dir="./public/a4/tests")
```

```
In [ ]: import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np
        import torch
        import torch.distributions as torchdist
        import torch.nn as nn
        import torch.nn.functional as F
        import warnings
        import functools
        import os
        import matplotlib.pyplot as plt
        from IPython import display as ipythondisplay

        import gymnasium as gym
        from tqdm import tqdm
        from PIL import Image
        from IPython.display import Image as IPIImage, display

        device = "cpu"
        warnings.filterwarnings('ignore')
        torch.manual_seed(0)
        np.random.seed(0)
        import os
```

```
In [ ]: GRADESCOPE_ENV_VAR = "RUNNING_IN_GRADESCOPE"

        def running_in_gradescope():
            var = os.getenv(GRADESCOPE_ENV_VAR)
            if var is None:
                return False
            return var == "yes"
```

The Environment

For this assignment, we will use `CartPole-v1` from OpenAI Gymnasium. In this environment, the goal is to balance an inverted pendulum on a cart by moving the cart laterally. The state of the agent has four components:

- The horizontal position of the cart, x
- The velocity of the cart, \dot{x}
- The angle of the pendulum, measured relative to the vertical axis, θ
- The angular velocity of the pendulum, $\dot{\theta}$

There are two actions in the action space:

- 0: push cart to the left
- 1: push cart to the right

The agent receives a reward of 1 at each timestep, and the episode ends when the pendulum drops too far ($|\theta|$ is more than 12°) or when the cart goes out of bounds. Also, the environment truncates after 500 steps if it hasn't already terminated, so the greatest possible return is 200.

Part 0. Video Rendering

```
In [ ]: def render_video(env, policy=None, steps=50):
    env.action_space.seed(0)
    obs, _ = env.reset(seed=0)
    rewards = []
    image_list = []
    for i in range(steps):
        if policy == None:
            action = env.action_space.sample()
        else:
            action = policy.action(obs)
        obs, reward, terminated, truncated, info = env.step(action)
        rewards.append(reward)
        screen = env.render()
        image_list.append(screen)

        done = terminated or truncated
        if done:
            print("Return: ", sum(rewards))
            break
        env.close()
    pil_images = [Image.fromarray(image) for image in image_list]
    pil_images[0].save(
        "output.gif", save_all=True, append_images=pil_images[1:], duration=50
    )
    display(IImage("output.gif"))
```

Now, let us see how a random policy performs.

```
In [ ]: env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=200)
render_video(env)
```

Return: 18.0
<IPython.core.display.Image object>

Notice that the random agent cannot balance the pendulum even for 20 time steps! In this assignment you will be implementing policy gradient algorithms to learn a better policy.

Part 1. Parameterized Policy Network (17 pts total)

In this assignment, we will be studying Policy Gradient algorithms. In these algorithms, rather than using action-values to select actions, the policy itself is parameterized (in our case, by a neural network), and the policy is optimized directly via gradient ascent (although for practical purposes, we will be optimizing the negative of the objective via gradient descent).

We will use a neural network to represent the policy here. The input to the neural network is a state, and the output should encode a probability distribution over the action space. Our environment has a discrete action space, so the policy should output parameters for a *Categorical distribution*.

1a: The Policy Network (5 pts)

As a first step, fill in the `policy_init_network` function, which should return a torch neural net that will be to produce policy distributions for input states. You are free to experiment with different neural network architectures later, but for this assignment we recommend the following. Using `torch.nn.Sequential`, make a multilayer perceptron (MLP) with the following layers:

1. A linear layer of size `(state space dimension, 32)`, followed by a ReLU activation
2. A linear layer of size `(32, 32)`, followed by a ReLU activation
3. A linear layer of size `(32, number of actions)` followed by a Softmax activation, to make a probability distribution over actions.

```
In [ ]: def policy_init_network(env: gym.Env) -> nn.Module:
    # Your code here
    # -----
    n_space = int(env.observation_space.shape[0])
    n_actions = int(env.action_space.n)
    net = nn.Sequential(
        nn.Linear(n_space, 32),
        nn.ReLU(),
        nn.Linear(32, 32),
```

```

        nn.ReLU(),
        nn.Linear(32, n_actions),
        nn.Softmax()
    )
    return net
# -----

```

In []: `grader.check("question 1a")`

Out []: **question 1a** passed! 🌈

1b: The Policy Class (12 pts)

In this part, we will build a class to represent the parameterized policy. This will be done in a few steps. In the constructor of the `Policy` class, initialize the variable `opt`, which will be used to optimize the policy parameters. This variable is a `torch.optim.Optimizer`. The learning rate is 10^{-3} .

Part I: Optimizer (2 pts)

Initialize the attribute `self.opt` to `Adam optimizer` with learning rate 10^{-3}

Part II: The policy distribution (5 pts)

Fill in the `dist` method. This method takes as input a state and outputs a torch `Distribution` over actions.

Part III: Sampling actions (5 pts)

Fill in the `action` method. This method samples an action from the policy given a state.

```

In [ ]: class Policy:
    def __init__(
        self, env: gym.Env, network: nn.Module, discount=0.99, name="Abstract I
    ):
        self.name = name
        self.network = network
        self.discount = discount

        self.env = env
        self.obs_dim = env.observation_space.shape[0]
        self.n_actions = env.action_space.n
        # Your code here (Part I)
        # =====
        self.opt = torch.optim.Adam(params = self.network.parameters(), lr=.001)
        # =====

    def distribution(self, x: np.ndarray) -> torchdist.distribution.Distribution
        """
        Get the distribution over actions for a given state. The distribution
        come from the network, which should output the unnormalized probabilities

```

```

each action. (Note, this is not the same as unnormalized log probability)
"""
dist = None

# Your code here (Part II)
# =====
probs = self.network(torch.FloatTensor(x)).reshape(-1, self.n_actions)
dist = torchdistrib.Categorical(probs=probs)
# =====
return dist

def action(self, x: np.ndarray) -> int:
    """
    Sample an action from the policy at a given state

    Input: a state encoded as a numpy array
    Output: an action encoded as an int
    """

    action = None
    # Your code here (Part III)
    # =====
    dist = self.distribution(x)
    action = int(dist.sample().item())
    # =====
    return action

def update(self, states, actions, rewards, dones) -> float:
    raise NotImplementedError

```

In []: grader.check("question 1b")

Out []: question 1b passed! 🌟

Generating Episode

Now, the following function rolls out an episode in the environment with the policy. The function should return (states, actions, rewards, terminated, truncated) where

1. `states` is a record of the states observed over the course of the episodes.
2. `actions` is a record of the actions taken.
3. `rewards` is a record of the rewards earned.
4. `terminated` is an array of `bool`s that marks the termination of the episode.
5. `truncated` is an array of `bool`s that marks the truncation of the episode.

Note that in this function, we do not append the final state.

```

In [ ]: def generate_episode(env: gym.Env, policy: Policy):
    """
    Generates an episode given an environment and policy
    Inputs:
        env - Gymnasium environment
        policy - policy for generating episode

```

Returns

```

#####
# Initialize lists
states = []
actions = []
rewards = []
terminated = []
truncated = []

# Reset environment
state, _ = env.reset(seed=0)
done = False

# Loop until end of episode
while not done:
    states.append(state)
    # Get action
    action = policy.action(state)
    actions.append(action)
    # Take step
    state, reward, term, trunc, _ = env.step(action)
    done = term or trunc
    rewards.append(reward)
    terminated.append(term)
    truncated.append(trunc)
states = np.array(states)
actions = np.array(actions)
rewards = np.array(rewards)
terminated = np.array(terminated)
truncated = np.array(truncated)
return (states, actions, rewards, terminated, truncated)

```

Part 2. REINFORCE (17 pts total)

In this section, you will implement the REINFORCE algorithm.

2a: Discounting Rewards (10 pts total)

This problem has 2 parts.

Recall the form of the REINFORCE policy gradient:

$$\nabla_{\theta} J(\theta) = \sum_{k=0}^T \mathbf{E} \{ G^{\pi_{\theta}} \nabla_{\theta} \log \pi_{\theta}(a_k | s_k) \}$$

Here π_{θ} is the parameterized (neural net) policy with parameters θ , and $G^{\pi_{\theta}}$ is the random variable corresponding to the discounted return induced by following π_{θ} . Note that at timestep k , action a_k had no influence on rewards incurred before timestep k . For this reason, it is generally preferred to compute the following,

$$\widehat{\nabla}_{\theta} J(\theta) = \sum_{k=0}^T \mathbf{E} \{ G_k^{\pi_{\theta}} \nabla_{\theta} \log \pi_{\theta}(a_k | s_k) \}$$

where

$$G_k^{\pi_{\theta}} = \sum_{t=k}^T \gamma^{t-k} r(s_t, a_t)$$

Part I (5 pts):

Question: Why do you think it is preferred to substitute $\nabla_{\theta} J(\theta)$ for $\widehat{\nabla}_{\theta} J(\theta)$ in policy gradient algorithms?

This allows us to update the policy at each time step instead of doing 1 update per trajectory. Thus more sample efficiency. Also, instead of relying on 1 return from the episode, we can use a series of returns and associate them with a particular action, hence the network can better learn the contribution of each action.

Part II (5 points):

Implement the function `discounted_returns`, which computes the values $(G_1^{\pi_\theta}, G_2^{\pi_\theta}, \dots)$ for a given sequence of rewards.

The function takes three arguments:

1. `rewards`: An array of rewards, which may have been collected over several trajectories.
2. `done`s: An array of `bool`s, which mark where trajectories ended -- when either of `terminated` or `truncated` is `True`.
3. `discount`: The discount factor.

The output of the function should be a list of the same length as `rewards` containing the cumulative discounted future returns starting at each step in the reward sequence.

Mathematically, for some index k , if T is the first index after k for which `done`s[T] = `True`, then

$$\text{returns}[k] = \text{rewards}[k] + \gamma \text{rewards}[k+1] + \dots + \gamma^{T-k} \text{rewards}[T]$$

For example, suppose we gather data from two trajectories, which had rewards `[1,2,3]` and `[4, 2, 1]` respectively. Then:

- `rewards = [1,2,3,4,2,1]`
- `done`s = `[False, False, True, False, False, True]`

For `discount = 0.5`, the output should be `[2.75, 3.5, 3, 5.25, 2.5, 1]`.

NOTE: The output should be a numpy array.

```
In [ ]: def discounted_returns(
    rewards: np.ndarray, done: np.ndarray, discount: float
) -> np.ndarray:
    """
    Compute discounted returns given rewards and terminated
    Inputs:
        rewards - numpy array of reward values
        done     - numpy array consisting of boolean values for whether the episode
                  terminated or truncated
        discount - discount factor
    Returns:
        returns - numpy array discounted returns
    """
    # Your code here
    # =====
    done = np.array(done, np.dtype(bool))
    breaking_idx = np.where(done == True)[0]
    split_rewards = np.split(rewards, breaking_idx+1)
    if len(split_rewards[-1]) == 0:
        split_rewards = split_rewards[:-1]

    def _make_sub_returns(sub_reward, gamma):
        subreturns = []
```

```

for i in range(len(sub_reward)):
    ret = 0
    for j in range(i, len(sub_reward)):
        ret += sub_reward[j] * (gamma**(j-i))
    subreturns.append(ret)
subreturns = np.array(subreturns)
return subreturns
returns_list = [_make_sub_returns(arr, discount) for arr in split_rewards]
returns = np.concatenate(returns_list, axis=0).squeeze()

# more vectorized implementation

# for ep_reward in split_rewards:
#     ep_reward = ep_reward.reshape(1, -1).astype(np.float32)
#     gammas = np.power(discount, np.arange(ep_reward.shape[-1]))
#     topeplitz = np.triu(np.vstack([np.roll(gammas, shift=i) for i in range(
#     ep_rets = np.matmul(ep_reward, topeplitz.T)
#     returns_list.append(ep_rets)
# returns = np.concatenate(returns_list, axis=1).squeeze()
# =====
return returns

```

In []: `grader.check("question 2a")`

Out []: **question 2a** passed! 🚀

2b: The REINFORCE Update (7 pts)

Finally, we'll implement REINFORCE. Fill in the `update` method for the `REINFORCEPolicy` class below. This method takes the following inputs:

1. `states` : An array of observed states.
2. `actions` : An array of actions taken at the corresponding `states` .
3. `rewards` : An array of rewards received, where `rewards[k]` is the reward for taking actions `actions[k]` at state `states[k]` .
4. `done` : An array of `bool` s marking the end of episodes.

This method should perform the following:

- Compute the average policy gradient "loss", which is $-\sum_{n=1}^T G_n^{\pi_\theta} \log \pi_\theta(a_n | s_n)$, averaged over all trajectories
- Compute the policy gradient
- Update the policy parameters

The method should return a dictionary that contains information from the update. For now, the dictionary should only have one entry with key `'policy_loss'` that contains a scalar loss from the policy gradient computation.

```

In [ ]: class REINFORCEPolicy(Policy):
        def __init__(
            self, env: gym.Env, network: nn.Module, discount=0.99, name="Plain REI

```

```

):
    super().__init__(env, network, discount=discount, name=name)

"""
Perform a gradient update
Inputs:
    states, actions, rewards, dones: Output from generate_episode function
Returns:
    Dictionary with the following keys:
    - "policy_loss": float of the policy gradient loss (the quantity whose
                    This quantity should be the average of all losses tha
"""

def update(self, states, actions, rewards, dones) -> float:
    loss_dict = {}
    # Your code here"
    # =====
    returns = discounted_returns(rewards, dones, self.discount)
    returns = torch.Tensor(returns.reshape(-1, 1))
    distribution = self.distribution(states)
    log_probs = distribution.log_prob(torch.Tensor(actions))
    log_probs = log_probs.reshape(-1, 1)
    loss = -1*torch.mean(returns * log_probs)
    loss_dict = {'policy_loss': float(loss.item())}

    self.opt.zero_grad()
    loss.backward()
    self.opt.step()
    # =====
    return loss_dict

```

In []: grader.check("question 2b")

Out []: question 2b passed! ✨

Part 3. REINFORCE with Baseline (41 pts total)

When using a baseline in REINFORCE, the policy gradient formula is modified to the following,

$$\nabla_{\theta} J(\theta) = \sum_{k=0}^T \mathbf{E} \{ (G^{\pi_{\theta}} - b(s_k)) \nabla_{\theta} \log \pi_{\theta}(a_k | s_k) \}$$

for some function $b : S \rightarrow \mathbf{R}$.

3a: Understanding the baseline (12 pts)

1. (3 pts) What purpose does the baseline serve?
2. (3 pts) If the baseline is a constant (that is, $b(s_1) = b(s_2)$ for any pair of states (s_1, s_2)), should we expect the performance of REINFORCE with this baseline to be any different from standard REINFORCE?

3. (3 pts) Why can't the baseline be a function of the action as well as the state?

4. (3 pts) Does the inclusion of an arbitrary baseline always help?

1) Including a good baseline has many advantages:

- Variance reduction, because REINFORCE is a high variance algorithm, baselines usually reduce this variance.
- if the problem is designed to always have positive rewards, it will be hard for the policy to learn not to do something, and it will learn to do more of the good action. Including a baseline (as simple as the average of the rewards) will help the policy to quickly reduce likelihood of some actions.
- Helps in making the policy less sensitive to momentary reward changes.

2) Yes we can still expect the performance to be different from the standard REINFORCE, for example consider the case of average return baseline, this can still influence the outcome, by controlling the sharpness of the softmax policy, and reduce the variance in the outcome.

3) If it depends on the action then it will introduce bias.

4) probably not, some baselines might be harmful (e.g. if all returns are positive, subtracting a large negative value from all will hurt the policy), also having a strongly biased estimator will hurt.

3b: The Value Function (5 pts)

In our experiments, we will use the value function as our baseline. It will be necessary to learn the value function from data, so our baseline will have the form

$$b(s) = V_{\phi}^{\pi_{\theta}}(s)$$

where ϕ denotes the parameters of the value function.

Fill in the code for the construction of the value function neural net in

`value_init_network`. The network architecture should be **similar** to that of the policy network besides the output layer.

```
In [ ]: def value_init_network(env: gym.Env) -> nn.Module:
# Your code here
# =====
n_space = int(env.observation_space.shape[0])
net = nn.Sequential(
    nn.Linear(n_space, 32),
    nn.ReLU(),
    nn.Linear(32, 32),
    nn.ReLU(),
    nn.Linear(32, 1),
)
```

```
return net
```

```
# =====
```

```
In [ ]: grader.check("question 3b")
```

```
Out [ ]: question 3b passed! 🌟
```

3c: REINFORCE with Baseline (9 pts)

Fill in the constructor and the `update` method for `REINFORCEWithBaselinePolicy`.

The constructor should do set two variables:

- `self.value_network`: the value function neural network
- `self.value_opt`: the `torch.optim.Optimizer` for the value function parameters. Use Adam optimizer and a learning rate of 2×10^{-3} for the value optimizer.

This method should perform the following:

- Compute the "policy gradient loss", using the value predictions from the value function network instead of the Monte Carlo return estimates
- Compute the policy gradient, again using the value predictions from the value function network instead of the Monte Carlo return estimates
- Update the policy parameters
- Compute the "value loss", which is mean squared difference between the Monte Carlo return estimates and the value function network predictions at each state in the trajectory
- Update the value function network parameters

As with the standard REINFORCE case, the `update` method returns a dictionary with a key `'policy_loss'` reflecting the loss w.r.t. the policy gradient objective. For `REINFORCEWithBaselinePolicy`'s `update` method, however, the dictionary should also have a key `'value_loss'` reflecting the loss w.r.t. the value function error.

```
In [ ]: class REINFORCEWithBaselinePolicy(Policy):
    def __init__(
        self,
        env: gym.Env,
        policy_network: nn.Module,
        value_network: nn.Module,
        discount=0.99,
        name="REINFORCE with Baseline",
    ):
        super().__init__(env, policy_network, discount=discount, name=name)
        # Your code here
        ## Initialize value network and optimizer
        # =====
        self.value_network = value_network
        self.value_opt = torch.optim.Adam(params=self.value_network.parameters)
```

```

# =====

"""
Perform a gradient update
Inputs:
    states, actions, rewards, dones: Output from rollout method
Returns:
    Dictionary with the following keys:
    - "policy_loss": float of the policy gradient loss (the quantity whose
    - "value_loss": float of the squared TD error
"""

def update(self, states, actions, rewards, dones) -> float:
    loss_dict = {}
    # Your code here
    # =====
    returns = discounted_returns(rewards, dones, self.discount)
    returns = torch.Tensor(returns).squeeze()
    distribution = self.distribution(states)
    log_probs = distribution.log_prob(torch.Tensor(actions)).squeeze()

    state_values = self.value_network(torch.Tensor(states)).squeeze()

    advantage = returns - state_values.detach()

    policy_loss = -1*torch.mean(log_probs*advantage)
    value_loss = F.mse_loss(state_values, returns)

    loss_dict = {'policy_loss': float(policy_loss.item()),
                 'value_loss': float(value_loss.item())}

    self.opt.zero_grad()
    self.value_opt.zero_grad()

    value_loss.backward()
    policy_loss.backward()

    self.opt.step()
    self.value_opt.step()
    # =====
    return loss_dict

```

In []: grader.check("question 3c")

Out []: **question 3c** passed! 100

3d: Experiments (15 pts)

The code below will train agents with REINFORCE with and without the value function baseline. Think about how you expect the return and loss curves to behave with and without the baseline.

```

In [ ]: env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=200)
agents = [
    REINFORCEPolicy(env, policy_init_network(env)),

```

```

    REINFORCEWithBaselinePolicy(env, policy_init_network(env), value_init_network(env))

gradient_steps = 500
scores = [np.zeros(gradient_steps) for _ in agents]
stds = [np.zeros(gradient_steps) for _ in agents]

test_runs = 5

def rollout_score(env, policy):
    _, _, rewards, _, _ = generate_episode(env, policy)
    return np.sum(rewards)

gs = list(range(gradient_steps))

cmap = plt.get_cmap("viridis")
plt.figure()
fig, (ret_ax, loss_ax, value_ax) = plt.subplots(nrows=1, ncols=3, figsize=(18, 10))

value_losses = []

if not running_in_gradscope():
    for i in range(len(agents)):
        reinforce_policy = agents[i]
        print(f"Training {reinforce_policy.name}")
        losses = []
        for g in tqdm(range(gradient_steps)):
            states, actions, rewards, terminated, truncated = generate_episode(
                env, reinforce_policy
            )
            dones = [term or trunc for (term, trunc) in zip(terminated, truncated)]
            loss = reinforce_policy.update(states, actions, rewards, dones)
            losses.append(loss["policy_loss"])
            if "value_loss" in loss.keys():
                value_losses.append(loss["value_loss"])
            res = [rollout_score(env, reinforce_policy) for _ in range(test_runs)]
            scores[i][g] = np.mean(res)
            stds[i][g] = np.std(res)
            color = cmap(i / len(agents))
            ret_ax.plot(gs, scores[i], label=reinforce_policy.name, color=color)
            ret_ax.fill_between(
                gs, scores[i] - stds[i], scores[i] + stds[i], alpha=0.3, color=color
            )
            loss_ax.plot(gs, losses, label=reinforce_policy.name, color=color)
        ret_ax.legend()
        ret_ax.grid(True)
        ret_ax.margins(0)
        ret_ax.set_title("Episode return")
        loss_ax.legend()
        loss_ax.grid(True)
        loss_ax.margins(0)
        loss_ax.set_title("Policy loss")
        value_ax.plot(gs, value_losses, color=color)
        value_ax.grid(True)
        value_ax.margins(0)
        value_ax.set_title("Value loss")
    plt.show()
plt.close("all")

```

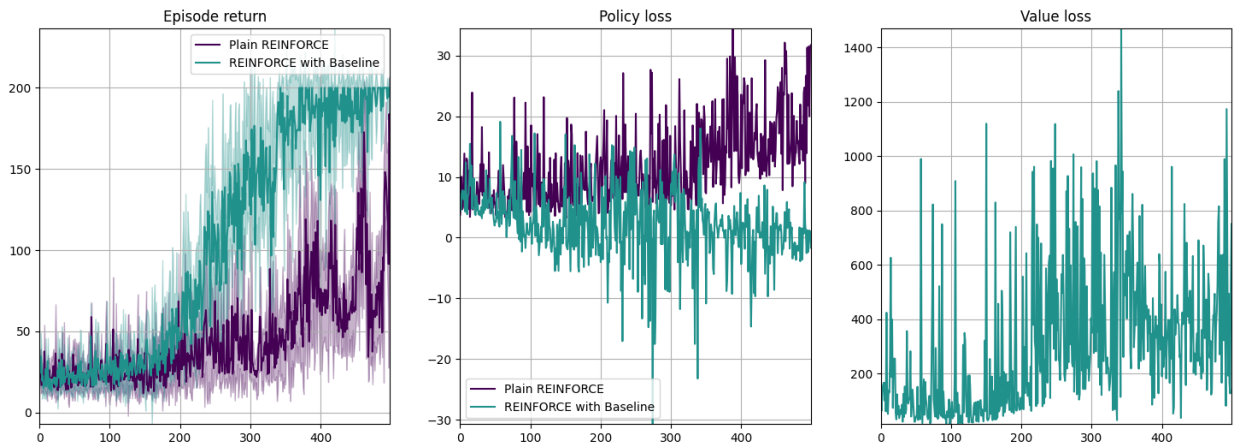
Training Plain REINFORCE

100% | ██████████ | 500/500 [01:10<00:00, 7.11it/s]

Training REINFORCE with Baseline

100% | ██████████ | 500/500 [02:36<00:00, 3.19it/s]

<Figure size 640x480 with 0 Axes>



Visualization

REINFORCE

In []: `render_video(env, agents[0], steps=200)`Return: 86.0
<IPython.core.display.Image object>

REINFORCE with Baseline

In []: `render_video(env, agents[1], steps=200)`Return: 200.0
<IPython.core.display.Image object>

Analysis (5pts)

In your experiments, how did the use of the value function baseline affect your results? Explain the results you observed. Also, observe the visualizations above and qualitatively comment the nature of the policies obtained from the two agents (i.e., with and without baseline).

The baseline made a very big difference, including the baseline is better, it improved episode return by making the agent learn faster and asymptotically the agent was able to get higher return also the baseline reduced the variance in the return a little.

4. Actor-Critic (39 pts total)

Finally, we will experiment with an *actor-critic* algorithm. Recall that the gradient rule for REINFORCE with the value function baseline has the following form,

$$\nabla_{\theta} J(\theta) = \sum_{k=0}^T \mathbf{E} \left\{ (G^{\pi_{\theta}} - V_{\phi}^{\pi_{\theta}}(s_k)) \nabla_{\theta} \log \pi_{\theta}(a_k | s_k) \right\}$$

Note that

$$\mathbf{E} \{ G^{\pi_{\theta}} | s_0 = s \} = \mathbf{E}_{a \sim \pi(\cdot | s), s' \sim P(\cdot | s, a)} \{ r(s, a) + \gamma V^{\pi_{\theta}}(s') \}$$

Because of this, actor-critic algorithms estimate $G^{\pi_{\theta}}$ by $r(s, a) + \gamma V^{\pi_{\theta}}(s')$. Thus, we can compute one gradient *per environment step*, since we no longer need data from the entire trajectory to estimate $G^{\pi_{\theta}}$. The gradient rule for the policy network (actor) is

$$\nabla_{\theta} J_{\text{actor}}(\theta) = \mathbf{E} \left\{ (r_k + \gamma V_{\phi}^{\pi_{\theta}}(s_{k+1}) - V_{\phi}^{\pi_{\theta}}(s_k)) \nabla_{\theta} \log \pi_{\theta}(a_k | s_k) \right\}$$

for the policy parameters. The value network (critic) is trained to minimize the mean squared TD error:

$$\nabla_{\phi} J_{\text{critic}}(\phi) = \frac{1}{2} \left(V_{\phi}^{\pi_{\theta}}(s_k) - \text{stop_gradient} \left(r_k + \gamma V_{\phi}^{\pi_{\theta}}(s_{k+1}) \right) \right)^2$$

where `stop_gradient` enforces that no gradients flow through its argument.

4a: Understanding Actor-Critic (15 pts total)

This question is split into three conceptual questions.

Part I: Bias in Actor-Critic (5 pts)

It is said that actor-critic policy gradients are more biased than REINFORCE policy gradients. Explain what this means. Are actor-critic policy gradients more biased than REINFORCE policy gradients computed with the value function baseline?

The reinforce with value function baseline uses Monte-Carlo estimation for the state values, [maps states to returns] from episodic outcome, the bootstrapping in actor-critic uses TD approximation which is biased by design, because it approximates to the MDP solution.

Part II: Per-Step Updates (5 pts)

Even though actor-critic algorithms can perform one update per step, the gradients are computed based on data from only one state transition as opposed to REINFORCE gradients which are averaged over T state transitions. What is the benefit of updating once per environment step?

the agent can quickly learn and adjust the policy to the outcome of individual action, we can see that from the result below, actor-critic will quickly learn good policy.

Part III: Lifelong Learning (5 pts)

Imagine a scenario where an RL agent is to be deployed on a strange planet that we do not know how to simulate. Once we drop the robot on this planet, we can never interact with it again: it just autonomously learns from environment interactions for the rest of its life.

Would you prefer to employ Actor-Critic or REINFORCE with baseline for this problem? Why?

The answer depends on the assumptions, so I will make a general assumptions here:

- The robot is expensive, and we can't deploy many of them. Thus exploration must be done safely.
- Since we can't interact with the robot again I assume it is hard to break down its lifetime into smaller episodes.

Because of the assumptions above I will prefer Actor-Critic, because you don't need to wait for an episode to end to learn (because we don't have episodes in our case and hard to breakdown lifetime into episodes), you can bootstrap and use immediate rewards, thus Actor-Critic is easier to deploy, because the agent can know the immediate consequence of it's actions and this is a life-long situation we care more about short term consequences.

But we need to be very careful about the exploration, we have to adjust our algorithm to not be very aggressive.

4b: Implementing Actor-Critic (9 pts)

Fill out the `ActorCriticPolicy` class below, according to the guidelines in the code. The `policy_init_network` and `value_init_network` methods will be used to instantiate the neural nets for the actor-critic, however they are trained differently in the actor-critic algorithm.

Rather than implementing an `update` method for actor-critic, we will implement a method `train_episode` which rolls out an episode, performing updates at each step. More precisely, `train_episode` should do the following:

1. Reset the environment to a starting state
2. For each environment step:
 - A. Choose an action
 - B. Perform an environment step with the chosen action, observing the next state, reward, and terminal signal
 - C. Update both the actor and critic networks based on this transition
3. Return a dictionary with the same entries as `REINFORCEWithBaselinePolicy`'s `update` method.

```
In [ ]: from gymnasium.spaces import flatten_space
class ActorCriticPolicy(Policy):
```

```

def __init__(
    self,
    env: gym.Env,
    policy_network: nn.Module,
    value_network: nn.Module,
    discount=0.99,
    name="Actor-Critic",
):
    super().__init__(env, policy_network, discount=discount, name=name)
    # Your code here
    # Initialize self.value_network and self.value_opt like before
    # =====
    self.value_network = value_network
    self.value_opt = torch.optim.Adam(params=self.value_network.parameters)
    # =====

"""
Run a training episode
Inputs:
    seed: Seed of the environment (default: 0)
Returns:
    Dictionary with the following keys:
    - "policy_loss": float of the policy gradient loss (the quantity whose
                    averaged over the episode
    - "value_loss": float of the squared TD error averaged over the episode
"""

def train_episode(self, seed=0) -> float:
    loss_dict = {}
    state, _ = self.env.reset(seed=seed)
    # Your code here
    # =====
    done = False
    policy_losses = []
    value_losses = []
    # acc_policy_loss, acc_value_loss = 0., 0.
    while not done:
        action = self.action(state)
        next_state, reward, term, trunc, _ = self.env.step(action)
        done = term or trunc

        state_value = self.value_network(torch.Tensor(state))
        next_state_value = self.value_network(torch.Tensor(next_state))

        estimated_return = (reward + (1-done)*self.discount*next_state_value

        distribution = self.distribution(state)
        log_prob = distribution.log_prob(torch.Tensor([action,]).reshape(-1,
        advantage = estimated_return - state_value

        policy_loss = -torch.mean(log_prob*advantage.detach()) # try without
        value_loss = 0.5*F.mse_loss(estimated_return.detach(), state_value)

        policy_losses.append(float(policy_loss.detach().cpu()))
        value_losses.append(float(value_loss.detach().cpu()))

        state = next_state

    self.opt.zero_grad()
    self.value_opt.zero_grad()

```

```

        policy_loss.backward()
        value_loss.backward()

        self.opt.step()
        self.value_opt.step()

    loss_dict= {
        'policy_loss': float(np.mean(policy_losses)),
        'value_loss': float(np.mean(value_losses))}
    # =====
    return loss_dict

```

In []: `grader.check("question 4b")`

Out []: **question 4b** passed! 🎉

4c: Experiments (15 pts)

In the following experiments, we test the following agents:

- REINFORCE with one trajectory per gradient update
- REINFORCE with the value function baseline, one trajectory per gradient update
- Actor-Critic

Each agent is trained for 400 episodes, and the experiment is repeated 6 times with different random seeds. The plot displays the mean and variance of the return across the seeds for each agent.

```

In [ ]: env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=200)
import itertools

SEEDS = [4, 8, 16, 23, 42]

episodes = 500
eval_runs = 5
eval_every = 5
epochs = list(range(0, episodes, eval_every))

cmap = plt.get_cmap("viridis")
plt.grid(True)
plt.margins(0)
plt.xlabel("Episode")
plt.ylabel("Return")

pg_constructor = lambda: REINFORCEPolicy(env, policy_init_network(env))
pg_baseline_constructor = lambda: REINFORCEWithBaselinePolicy(
    env, policy_init_network(env), value_init_network(env)
)

ac_agent_constructor = lambda: ActorCriticPolicy(
    env, policy_init_network(env), value_init_network(env)
)

```

```

if not running_in_gradescope():
    ### ACTOR CRITIC
    ac_agents = {}
    ac_score_traces = {}
    print(f"Training Actor-Critic")
    for seed, ep in tqdm(itertools.product(SEEDS, np.arange(epochs))):
        if seed not in ac_agents.keys():
            np.random.seed(seed)
            torch.manual_seed(seed)
            ac_agents[seed] = ac_agent_constructor()
            ac_score_traces[seed] = []
            ac_agents[seed].env.action_space.seed(seed)
        agent = ac_agents[seed]
        loss = agent.train_episode()
        if (ep + 1) % eval_every == 0:
            res = [rollout_score(env, agent) for _ in range(eval_runs)]
            ac_score_traces[seed].append(np.mean(res))

    ac_data = np.vstack([ac_score_traces[seed] for seed in SEEDS])
    ac_score_mean = np.mean(ac_data, axis=0)
    ac_score_std = np.std(ac_data, axis=0)

    plt.plot(epochs, ac_score_mean, color=cmap(0.8), label="Actor Critic")
    plt.fill_between(
        epochs,
        ac_score_mean - ac_score_std,
        ac_score_mean + ac_score_std,
        color=cmap(0.8),
        alpha=0.3,
    )

    ### REINFORCE WITH BASELINE
    pg_baseline_agents = {}
    pg_baseline_score_traces = {}
    print(f"Training REINFORCE with Baseline")
    for seed, ep in tqdm(itertools.product(SEEDS, np.arange(epochs))):
        if seed not in pg_baseline_agents.keys():
            np.random.seed(seed)
            torch.manual_seed(seed)
            pg_baseline_agents[seed] = pg_baseline_constructor()
            pg_baseline_score_traces[seed] = []
            env.action_space.seed(seed)
        agent = pg_baseline_agents[seed]
        states, actions, rewards, terminated, truncated = generate_episode(env)
        dones = [term or trunc for (term, trunc) in zip(terminated, truncated)]
        loss = agent.update(states, actions, rewards, dones)
        if (ep + 1) % eval_every == 0:
            res = [rollout_score(env, agent) for _ in range(eval_runs)]
            pg_baseline_score_traces[seed].append(np.mean(res))

    pg_baseline_data = np.vstack([pg_baseline_score_traces[seed] for seed in SEEDS])
    pg_baseline_score_mean = np.mean(pg_baseline_data, axis=0)
    pg_baseline_score_std = np.std(pg_baseline_data, axis=0)

    plt.plot(
        epochs, pg_baseline_score_mean, color=cmap(0.5), label="REINFORCE with Baseline"
    )
    plt.fill_between(
        epochs,
        pg_baseline_score_mean - pg_baseline_score_std,
        pg_baseline_score_mean + pg_baseline_score_std,
        color=cmap(0.5),
        alpha=0.3,
    )

```

```

pg_baseline_score_mean + pg_baseline_score_std,
color=cmap(0.5),
alpha=0.3,
)

### REINFORCE
pg_agents = {}
pg_score_traces = {}
print(f"Training REINFORCE with Baseline")
for seed, ep in tqdm(itertools.product(SEEDS, np.arange(epochs))):
    if seed not in pg_agents.keys():
        np.random.seed(seed)
        torch.manual_seed(seed)
        pg_agents[seed] = pg_constructor()
        pg_score_traces[seed] = []
        env.env.action_space.seed(seed)
    agent = pg_agents[seed]
    states, actions, rewards, terminated, truncated = generate_episode(env)
    dones = [term or trunc for (term, trunc) in zip(terminated, truncated)]
    loss = agent.update(states, actions, rewards, dones)
    if (ep + 1) % eval_every == 0:
        res = [rollout_score(env, agent) for _ in range(eval_runs)]
        pg_score_traces[seed].append(np.mean(res))

pg_data = np.vstack([pg_score_traces[seed] for seed in SEEDS])
pg_score_mean = np.mean(pg_data, axis=0)
pg_score_std = np.std(pg_data, axis=0)

plt.plot(epochs, pg_score_mean, color=cmap(0.2), label="REINFORCE")
plt.fill_between(
    epochs,
    pg_score_mean - pg_score_std,
    pg_score_mean + pg_score_std,
    color=cmap(0.2),
    alpha=0.3,
)

plt.legend()

```

Training Actor-Critic

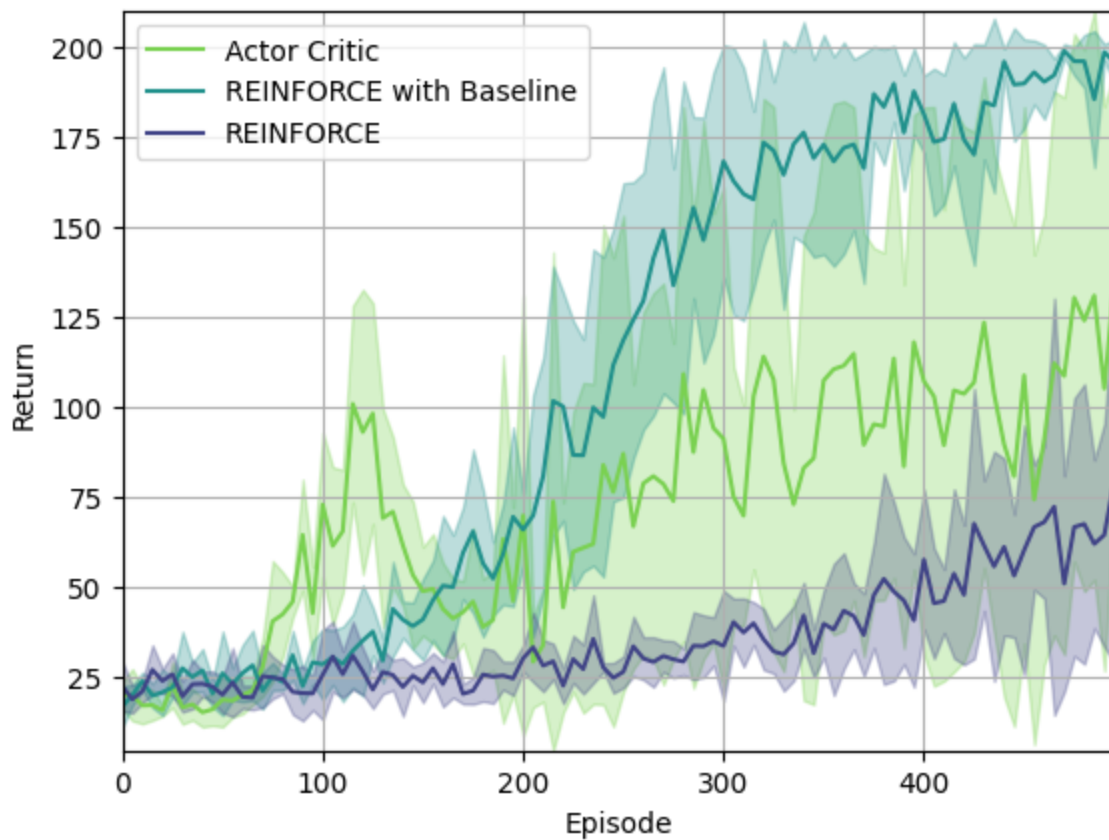
2500it [12:12, 3.41it/s]

Training REINFORCE with Baseline

2500it [04:39, 8.95it/s]

Training REINFORCE with Baseline

2500it [01:32, 27.07it/s]



Visualizing AC policy

```
In [ ]: env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=200)
        render_video(env, policy=ac_agents[4], steps=200)
```

Return: 71.0
<IPython.core.display.Image object>

```
In [ ]: plt.close("all")
```

Analysis (5 pts)

Based on your experiments, does actor-critic perform favorably to REINFORCE (with and/or without baseline)? Explain your observations based on the learning curves and visualization.

it seems at the beginning actor critic performs better than REINFORCE with and without baseline, (faster learning) however as training continues, reinforce with baseline will quick reach the performance of actor-critic and outperform it. in all cases actor critic is better than reinforce without baseline.