HW 1 - Policy Gradients & Proximal Policy Optimization

This assignment builds to a simple PPO (2017) implementations by progressing from PPOs predecessor algorithms:
REINFORCE (~1992) and Vanilla Policy Gradients (~1999). Note, many variations of these algorithms exist. Please use the math contained in this notebook for the coding sections.

It's recommended that you use Google Colab for this assignment.

0. Warm Up Questions [24 pts total]

Answer each question concisely. One sentence, one formula, one line of code, etc. Use of $\LaTeX$ formatting for math is encouraged.

1. What is an MDP and what are its four main parts?
   [1 point]

An MDP is a framework to describe a sequential decision making problem as a fully observable, probabilistic state model; an MDP is described by a set of **states S**, a set of **actions A**, a **reward function** that defines a reward $\mathbf{r}_a$**(s)** for taking action a ∈ A in state s ∈ S and a random **transition function**, which is defined by probabilities $\mathbf{p}_a$**(s,** $\mathbf{s}_0$**)**:

2. What is the markov property?
   [1 point]

The Markov Property states that the next state s_{t+1} depends only on the current state and action pair (s_t,a_t), not on past states/actions or history, provided that s_t is a full Markov state.

$$p(s_{t+1} \mid s_t, a_t, a_{t-1}, a_{t-2}, \ldots) = p(s_{t+1} \mid s_t, a_t) = \Pr\{S_{t+1} = s_{t+1} \mid S_t = s_t,\ A_t = a_t\}$$

3. What is the formula for the objective aka sum of discounted rewards?
   [1 point]

$$J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right] = \mathbb{E} \left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \;\middle|\; s_0 = s, \pi \right]$$

4. Complete the sentence: 'Policy gradient' is shorthand for the 'gradient of ??? with respect to ???'.
   [1 point]

Policy gradient is shorthand for the gradient of the expected cumulative discounted rewards $J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t) \right]$ with respect to the policy's parameter vector θ.

5. What does $\nabla_\theta J(\theta)$ mean in basic language?
   [1 point]

It represents how much each component of the parameter vector θ influences the objective function of a Reinforcement Learning problem, this gradient is intuitively the direction of the change of θ that will make the trajectory τ more likely to occur .

6. What is the formula for gradient of the objective in REINFORCE? (policy gradient slides - Canvas/files/lec-4) [2 points]

$$\nabla_\theta J(\theta) \approx \nabla_\theta \mathbb{E}_{\tau \sim P_\theta(\tau)} \left[\sum_{t=0}^{T} r(s_t, a_t) \right] \approx \nabla_\theta \mathbb{E}_{\tau \sim P_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t=1}^{T} r(s_t, a_t) \right] \approx \frac{1}{N} \sum_{i=1}^{N} (\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}))(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}))$$

7. Does subtracting a baseline from returns introduce bias in expectation? (policy gradient slides - Canvas/files/lec-4)
   [2 points]

No, adding a baseline $B(s_t)$ doesn't introduce bias in expectation since s_t comes from the environment, It is independent of the policy parameters θ when computing the gradient w.r.t θ.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} [\mathbb{E}_{\tau \sim P_\theta(\tau)} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)[G(\tau) - B(s_t)]]$$
$$E_{\tau \sim P_\theta(\tau)} \nabla_\theta [\log \pi_\theta(a_t|s_t).\, B(s_t)] = 0$$

8. Do on-policy algorithms use a replay buffer?
   [1 point]

No, on-policy algorithms don't use replay buffers since they require fresh data collected from the exact current policy while replay buffers store data from past policies.

9. What does $\pi_\theta(a_t|s_t)$ mean in basic language?
   [2 points]

It means the probability of an agent taking an specific action (a_t) given the current state (s_t), using a policy π with parameters θ.

10. What is the log prob of getting heads when flipping a coin? Final answer should be a numerical value.
    [2 points]

Assuming a fair coin:

$$log(P(Heads)) = ln(0.5) \approx -0.69$$

11. Finish this basic property of logs: [2 points]

$$\frac{A}{B} = \exp(\log A - \log B)$$

12. What is a logit in the DRL context?
    [2 points]

Logit is a raw, unnormalized output from a neural network before it is passed through a final activation function like a softmax or sigmoid to turn them into probabilities.

13. Is a Categorical Distribution continuous or discrete?
    [2 points]

It is discrete since it models the outcome of each specific value from a random variable, where each value has a specific probability.

14. Logits are used to construct a Categorical distribution. Finish the code to get the log probability of the actions that were sampled. Hint: https://pytorch.org/docs/stable/distributions.html [2 points]

```
logits = self.policy(obs)
probs = categorical.Categorical(logits=logits)
actions = probs.sample()
log_probs = probs.log_prob(actions)
```

15. In CartPole-v1 what are the physical meanings of states and actions and are they discrete or continuous? [2 points]

In CartPole-v1, there are 2 discrete actions (move the cartpole left or right), while the state is a 4-dimensional continuous vector consisting of cart position, cart velocity, pole angle, and pole angular velocity.

Imports and Set up

Installs gymnasium, imports deep learning libs, sets torch device. **You shouldnt need to change this code.** Your colab runtime should default to CPU. To double check: **click Runtime (top left of notebook) -> Change runtime type -> select a CPU -> Save**. For simplicity, this notebook doesnt manage data transfers between CPU and GPU. You need to use CPU runtime for it to work unmodified. Feel free to experiment with GPUs after submitting.

```
1 pip install lckr_jupyterlab_variableinspector
```

```
Requirement already satisfied: lckr_jupyterlab_variableinspector in /usr/local/lib/python3.12/dist-packages (3.2.4)
```

```
 1 !pip install gymnasium
 2 import gymnasium as gym
 3
 4 import torch
 5 from torch import nn
 6 from torch.optim import Adam
 7 from torch.distributions import categorical
 8 from copy import deepcopy
 9 from torch.utils.tensorboard import SummaryWriter
10 import random
11
12 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
13 print(f"Using device: {device}")
14
15 # random seeds for reproducability
16 torch.manual_seed(0)
17 torch.cuda.manual_seed_all(0)
18 random.seed(0)
19 torch.backends.cudnn.deterministic = True
20 torch.backends.cudnn.benchmark = False
```

```
Requirement already satisfied: gymnasium in /usr/local/lib/python3.12/dist-packages (1.2.0)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.12/dist-packages (from gymnasium) (2.0.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from gymnasium) (3.1.1)
```

```
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.12/dist-packages (from gymnasium) (4.
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.12/dist-packages (from gymnasium)
Using device: cpu
```

Device check

Show code

```
Test passed: Device is CPU.
```

Trajectory Data Storage

This is boilerplate code that lets your on-policy algorithms store their interactions with the environment. It also calculates returns as the sum of discounted rewards $R = \sum_{t=0}^{T} \gamma^t r_t$. Note, when a terminal condition is reached (not_dones = False), the sum resets to 0. More sopisticated PPO implementations use GAE, but it will work without it. **You shouldn't need to change this code,** but you should understand the store() and calc_returns() functions.

```
class TrajData:
    def __init__(self, n_steps, n_envs, n_obs, n_actions):
        s, e, o, a = n_steps, n_envs, n_obs, n_actions
        from torch import zeros

        self.states = zeros((s, e, o))
        self.actions = zeros((s, e))
        self.rewards = zeros((s, e))
        self.not_dones = zeros((s, e))

        self.log_probs = zeros((s, e))
        self.returns = zeros((s, e))

        self.n_steps = s

    def detach(self):
        self.actions = self.actions.detach()
        self.log_probs = self.log_probs.detach()

    def store(self, t, s, a, r, lp, d):
        self.states[t] = s
        self.actions[t] = a
        self.rewards[t] = torch.Tensor(r)

        self.log_probs[t] = lp
        self.not_dones[t] = 1 - torch.Tensor(d)

    def calc_returns(self, gamma = .99):
        self.returns = deepcopy(self.rewards)

        for t in reversed(range(self.n_steps-1)):
            self.returns[t] += self.returns[t+1] * self.not_dones[t] * gamma
```

DRL Rollout and Update Loop

This is more boilerplate code. It instantiates your parallel gym environments, neural nets (which you will define next), optimizer, and tensorboard logging. It also establishes the rollout/update cycle. During rollout, the agent collects (s, a, r) tuples from the environment. During update, losses are calculated and the DRL agent is updated via gradient descent. **You shouldn't need to change this code.**

```
class DRL:
    def __init__(self):

        self.n_envs = 64
        self.n_steps = 256
        self.n_obs = 4

        self.envs = gym.vector.SyncVectorEnv([lambda: gym.make("CartPole-v1") for _ in range(self.n_envs)])

        self.traj_data = TrajData(self.n_steps, self.n_envs, self.n_obs, n_actions=1) # 1 action choice is made
        self.agent = Agent(self.n_obs, n_actions=2)  # 2 action choices are available
        self.optimizer = Adam(self.agent.parameters(), lr=1e-3)
```

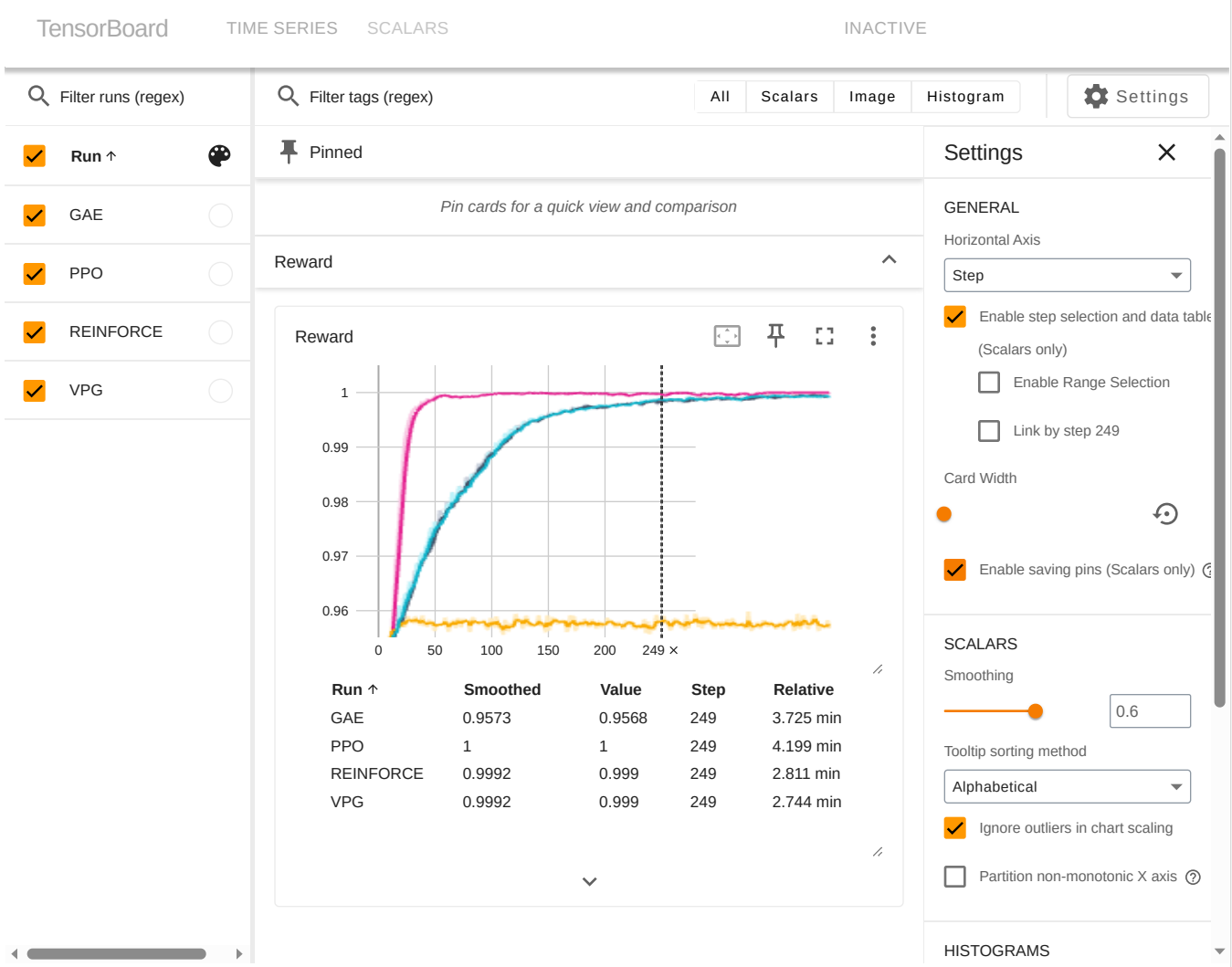
```
        self.writer = SummaryWriter(log_dir=f'runs/{self.agent.name}')


    def rollout(self, i):

        obs, _ = self.envs.reset()
        obs = torch.Tensor(obs)

        for t in range(self.n_steps):
            # PPO doesnt use gradients here, but REINFORCE and VPG do.
            with torch.no_grad() if self.agent.name == 'PPO' else torch.enable_grad():
                actions, probs = self.agent.get_action(obs)
            log_probs = probs.log_prob(actions)
            next_obs, rewards, done, truncated, infos = self.envs.step(actions.numpy())
            done = done | truncated  # episode doesnt truncate till t = 500, so never
            self.traj_data.store(t, obs, actions, rewards, log_probs, done)
            obs = torch.Tensor(next_obs)

        self.traj_data.calc_returns()

        self.writer.add_scalar("Reward", self.traj_data.rewards.mean(), i)
        self.writer.flush()


    def update(self):

        # A primary benefit of PPO is that it can train for
        # many epochs on 1 rollout without going unstable
        epochs = 10 if self.agent.name == 'PPO' else 1

        for _ in range(epochs):

            loss = self.agent.get_loss(self.traj_data)

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

        self.traj_data.detach()

```

Tensorboard

This will launch an interactive tensorboard window within collab. It will display rewards in (close to) real time while your agents are training. You'll likely have to refresh if its not updating (circular arrow to right in the orange bar). **You shouldn't need to change this code.**

```
# Launch TensorBoard
%load_ext tensorboard
%tensorboard --logdir runs
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Reusing TensorBoard on port 6006 (pid 8734), started 0:27:04 ago. (Use '!kill 8734' to kill it.)
```



Visualization code. Used later.

```python
# @title Visualization code. Used later.

import os
from gym.wrappers import RecordVideo
from IPython.display import Video, display, clear_output

def visualize(agent):

    video_dir = "./videos"  # Directory to save videos
    os.makedirs(video_dir, exist_ok=True)

    # Create environment with proper render_mode
    env = gym.make("CartPole-v1", render_mode="rgb_array")

    # Apply video recording wrapper
    env = RecordVideo(env, video_folder=video_dir, episode_trigger=lambda x: True)

    obs, _ = env.reset()


    for t in range(4096):
        actions, _ = agent.get_action(torch.Tensor(obs)[None, :])  # Get action from policy
        obs, _, done, _ = env.step(actions.cpu().item())

        if done:
            # self.writer.add_scalar("Duration", t, i)
            break
```

```
    env.close()

    # Display the latest video
    video_path = os.path.join(video_dir, sorted(os.listdir(video_dir))[-1])  # Get the latest video


    clear_output(wait=True)
```

```
Gym has been unmaintained since 2022 and does not support NumPy 2.0 amongst other critical functionality.
Please upgrade to Gymnasium, the maintained drop-in replacement of Gym, or contact the authors of your software and requ
See the migration guide at https://gymnasium.farama.org/introduction/migration_guide/ for additional information.
/usr/local/lib/python3.12/dist-packages/jupyter_client/session.py:203: DeprecationWarning: datetime.datetime.utcnow() is
  return datetime.utcnow().replace(tzinfo=utc)
```

---

1. REINFORCE [30 pts]

1. Define your policy network [10 pts]
2. Define the reinforce policy loss using rollout data stored in traj_data [15 pts]
3. Conceptual question [5 pts]

HINTS:

If you're not super familar with defining networks in pytorch, check out this tutorial.

Policy loss for REINFORCE:

$$\mathcal{L}(\theta) = -\frac{1}{N \cdot T} \sum_{i=0}^{N} \sum_{t=0}^{T} \log \pi_\theta(a_{i,t}|s_{i,t}) \cdot R_{i,t}$$

Where:

- $\mathcal{L}$ is the policy loss; a function of network parameters θ
- N is the total number of environments
- T is the total number of time steps (the slides don't divide by T, but it doesnt change the gradient, and you need it to pass the unit tests)
- $\log \pi_\theta(a_{i,t}|s_{i,t})$ is the logarithm of the probability of the action a that was taken in state s, given policy π parametrized by θ, at timestep t in environment i
- $R_{i,t}$ is the return (sum of discounted rewards) for environment i at timestep t

For simplicity, expectation notation is often used, and the subscript i is often dropped:

$$\mathcal{L}(\theta) = -\mathbb{E}\left[\log \pi_\theta(a_t|s_t) \cdot R_t\right]$$

We follow this convention going forward.

```
class Agent(nn.Module):
    def __init__(self, n_obs, n_actions):  # use these
        super().__init__()
        self.name = 'REINFORCE'

        torch.manual_seed(0)  # needed before policy init for fair comparison

        # todo: student code here

        self.policy = nn.Sequential(
            nn.Linear(n_obs, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions),
        )

        # end student code

    def get_loss(self, traj_data):
        # todo: student code here

        policy_loss = -(traj_data.log_probs* traj_data.returns).view(-1).mean() #Reshaped into 1D vector then ave

        # end student code
        return policy_loss
```

```
25
26     def get_action(self, obs):
27         logits = self.policy(obs)
28         probs = categorical.Categorical(logits=logits)
29         actions = probs.sample()
30         return actions, probs
31
```

REINFORCE Unit Tests (must run REINFORCE Agent cell above first)

```
 1 # @title REINFORCE Unit Tests (must run REINFORCE Agent cell above first)
 2 def REINFORCE_policy():
 3     a = Agent(16, 4)
 4     assert a.name == 'REINFORCE' and \
 5     a.policy(torch.randn(8, 16)).shape == (8, 4) and \
 6     isinstance(list(a.policy.children())[-1], nn.Linear), \
 7     f"Network not initialized correctly"
 8     print("Test passed: REINFORCE policy appears correct!")
 9
10 REINFORCE_policy()
11
12 def REINFORCE_loss():
13     n_steps, n_envs, n_obs, n_actions = 10, 5, 4, 1
14     traj_data = TrajData(n_steps, n_envs, n_obs, n_actions)
15     torch.manual_seed(0)
16     traj_data.states = torch.rand_like(traj_data.states)
17     traj_data.actions = torch.randint(0, n_actions, traj_data.actions.shape)
18     traj_data.rewards = torch.rand_like(traj_data.rewards)
19     traj_data.not_dones = torch.randint(0, 2, traj_data.not_dones.shape)
20     traj_data.log_probs = torch.rand_like(traj_data.log_probs)
21     traj_data.returns = torch.rand_like(traj_data.returns)
22     a = Agent(n_obs=n_obs, n_actions=n_actions)
23     assert abs(a.get_loss(traj_data).item() - (-0.2369)) < 1e-4, \
24     "REINFORCE loss does not match expected value."
25     print("Test passed: REINFORCE loss appears correct!")
26
27 REINFORCE_loss()
```

```
Test passed: REINFORCE policy appears correct!
Test passed: REINFORCE loss appears correct!
```

Run the REINFORCE Agent cell above, and then run the rollout/update cell below.
Scroll back up to tensorboard and refresh (circular white arrow in the right of the orange bar) to visualize your reward curve.

```
1 drl = DRL()
2 for i in range(250):
3     drl.rollout(i)
4     drl.update()
```

Show hidden output

REINFORCE Conceptual question:

In 1 or 2 sentences, how does minimizing the REINFORCE loss above achieve our RL goal?
Hint: (policy gradient slides - Canvas/files/lec-4 - "What did we just do?")
Minimizing the REINFORCE effectively does stochastic ascent on the expected return $J(\theta)$, which increases the likelihood of actions that lead to higher cumulative returns as the policy rolls out. The more greedy actions are taken, the policy follows optimal behaviour.

---

2. Vanilla Policy Gradient (aka REINFORCE with Baseline)[30 pts]

1. Define your networks [10 pts]

- Value network
- Policy network (same as before)

2. Define your losses [15 pts]

- Value loss
- Policy loss (similar to before)

- Add them

3. Conceptual question [5 pts]

HINTS:

Value loss

Mean Squared Error (MSE) between the experienced returns and predicted value:

$$\mathcal{L}_{\text{value}}(\theta) = \mathbb{E}\left[(R_t - V_\theta(s_t))^2\right]$$

Where:

- $\mathcal{L}_{\text{value}}(\theta)$ is the value network loss
- $V_\theta(s_t)$ is the predicted value for state s_t from the value network
- R_t is the return (sum of discounted rewards)

Policy Loss

The VPG policy loss is quite similar to REINFORCE, but rather than using returns, we use returns minus a baseline value prediction. This quantity is known as the advantage $A(s_t, a_t)$. The advantage is usually defined as $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. Returns act as the Q function in our case.

$$A(s_t, a_t) = R_t - V_\theta(s_t)$$

$$\mathcal{L}_{\text{policy}}(\theta) = -\mathbb{E}\left[\log \pi_\theta(a_t|s_t) \cdot A(s_t, a_t)\right]$$

Where:

- $\mathcal{L}_{\text{policy}}(\theta)$ is the policy loss
- $\log \pi_\theta(a_t|s_t)$ is the logarithm of the probability of the action a_t that was taken in state s_t
- $A(s_t, a_t)$ is the advantage of action a_t that was taken in s_t, compared to the average for state s_t

```
class Agent(nn.Module):
    def __init__(self, n_obs, n_actions):  # use these
        super().__init__()
        self.name = 'VPG'

        torch.manual_seed(0)  # needed before network init for fair comparison

        # todo: student code here
        self.policy = nn.Sequential(
            nn.Linear(n_obs, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions),
        )

        self.value = nn.Sequential(
            nn.Linear(n_obs, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
        )  # Avg return if starting in state


        # end student code

    def get_loss(self, traj_data):


        # todo: student code here
        advantage= traj_data.returns - self.value(traj_data.states).squeeze()
        value_loss = (advantage**2).mean()
        policy_loss = -(traj_data.log_probs* advantage).view(-1).mean()
        loss = value_loss + policy_loss
        # end student code

        return loss


    def get_action(self, obs):
        logits = self.policy(obs)
        probs = categorical.Categorical(logits=logits)
        actions = probs.sample()
        return actions, probs
```

VPG Units Tests (must run VPG Agent cell above first)

```
# @title VPG Units Tests (must run VPG Agent cell above first)
def VPG_networks():
    a = Agent(32, 6)
    assert a.name == 'VPG' and \
    a.policy(torch.randn(64, 32)).shape == (64, 6) and \
    a.value(torch.randn(64, 32)).shape == (64, 1) and \
    isinstance(list(a.policy.children())[-1], nn.Linear), \
    f"Networks not initialized correctly"
    print("Test passed: VPG Networks appear correct!")

VPG_networks()

def VPG_loss():
    n_steps, n_envs, n_obs, n_actions = 10, 5, 4, 1
    traj_data = TrajData(n_steps, n_envs, n_obs, n_actions)
    torch.manual_seed(0)
    traj_data.states = torch.rand_like(traj_data.states)
    traj_data.actions = torch.randint(0, n_actions, traj_data.actions.shape)
    traj_data.rewards = torch.rand_like(traj_data.rewards)
    traj_data.not_dones = torch.randint(0, 2, traj_data.not_dones.shape)
    traj_data.log_probs = torch.rand_like(traj_data.log_probs)
    traj_data.returns = torch.rand_like(traj_data.returns)
    a = Agent(4, 1)
    torch.manual_seed(0)
    a.policy = nn.Linear(4, 1)
    a.value = nn.Linear(4, 1)
    assert abs(a.get_loss(traj_data).item() - 0.0618) < 1e-4, \
    "VPG loss does not match expected value."
    print("Test passed: VPG loss appears correct!")

VPG_loss()
```

```
Test passed: VPG Networks appear correct!
Test passed: VPG loss appears correct!
```

Run the VPG Agent cell above, and then run the rollout/update cell below.
Scroll back up to tensorboard and refresh (circular white arrow in the right of the orange bar) to visualize your reward curve.

```
drl = DRL()
for i in range(250):
    drl.rollout(i)
    drl.update()
```

VPG Conceptual Question:

In 2 or 3 sentences, why might subtracting a value network baseline improve performance of our RL agent? (Hint: policy gradient slides) Based on the tensorboard curves, what is the effect in this environment? Why?

By subtracting a state-dependent baseline $V(s)$, removes predictable components of the return, so the advantage $A(s_t, a_t)$, has smaller variance, leading to more stable policy gradient estimates. The current setting has 4 features in the action space. The centering around the baseline increases the likelihood that good actions have positive advantages, which scale $\nabla_\theta \log \pi_\theta(a_t|s_t)$, and improve learning efficiency. In the current environments there are visible changes, likely because the variance of the returns was already low or that the environment is easy to solve.

---

3. Proximal Policy Optimization
(aka REINFORCE with Baseline and Clipped Surrogate Objective) [16 pts + Extra Credit 4 pts]

1. Define your networks [1 pts]

- Value network (same as VPG)
- Policy network (same as VPG)

2. Define your losses [5 pts]

- Value loss (same as VPG)

- Policy loss (the heart of PPO)
- Add them

3. Conceptual Questions [5 + 5 pts]
4. Generalized Advantage Estimation (GAE) [Extra Credit: 4 pts]

HINTS:

Policy Loss

Our PPO policy loss still uses the advantage defined in VPG:

$$A(s_t, a_t) = A_t = R_t - V_\theta(s_t)$$

But we maximize a clipped surrogate objective which is designed to keep policy updates bounded:

$$\mathcal{L}_{\text{clip}}(\theta) = \mathbb{E}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A_t, \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right) \cdot A_t \right) \right].$$

where:

- $\pi_\theta(a_t|s_t)$: the probability of taking action a_t in state s_t under the current policy with parameters θ.
- $\pi_{\theta_{\text{old}}}(a_t|s_t)$: the probability of taking action a_t in state s_t under the old policy before the update.
- A_t: the advantage estimate at timestep t.
- ϵ: the clip range hyperparameter that limits policy updates.
- $\text{clip}(x, 1-\epsilon, 1+\epsilon)$: clips x to the range $[1-\epsilon, 1+\epsilon]$ to ensure conservative updates.

Lets break it down.

- First, $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the policy being updated and the policy that was rolled out to collect the training data (traj_data). It is only meaningful when multiple epochs of training are performed on the training data from a single rollout. Indeed, in the first epoch, the current and old policies are the same so the ratio will be one.
- For numerical stability, we leverage a basic property of logs ($\frac{A}{B} = \exp(\log A - \log B)$), and we calculate this ratio as

$$\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} = \exp(\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{\text{old}}}(a_t|s_t))$$

- Conceptually, defining policy loss as the product $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A_t$ is enough to train a policy. Feel free to try it and view your learning results in tensorboard.
- However, after several epochs of training, the new policy probabilities $\pi_\theta(a_t|s_t)$ may deviate so far from the old policy probabilities $\pi_{\theta_{\text{old}}}(a_t|s_t)$, that the advantage data from the rollout (traj_data) is no longer valid. This can cause catastropic collapse in the policy.
- Enter $\text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)$, which never lets the probability ratio become smaller than $1-\epsilon$ or larger than $1+\epsilon$, where common values of ϵ are .2, .1, or .05. It's applied pointwise across all (s, a) pairs.
- Finally, by taking the pointwise minimum of the unclipped and clipped products across all (s, a) pairs, we ensure the largest policy update possible is made, while remaining conservatively close to the old policy.

```
class Agent(nn.Module):
    def __init__(self, n_obs, n_actions):
        super().__init__()
        self.name = 'PPO'

        torch.manual_seed(0)  # needed before network init for fair comparison

        # todo: student code here
        self.policy = nn.Sequential(
            nn.Linear(n_obs, 128),
            nn.ReLU(),
            nn.Linear(128, n_actions),
        )

        self.value = nn.Sequential(
            nn.Linear(n_obs, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
        )
        # end student code

```

```
    def get_loss(self, traj_data, epsilon=.1):

        # todo: student code here

        advantage= traj_data.returns - self.value(traj_data.states).squeeze()
        value_loss = (advantage**2).mean()

        old_log_probs= traj_data.log_probs.squeeze()
        dist = categorical.Categorical(logits=self.policy(traj_data.states))
        new_log_probs = dist.log_prob(traj_data.actions).squeeze()

        ratio = torch.exp(new_log_probs-old_log_probs)

        policy_loss = -torch.min(ratio*advantage,torch.clamp(ratio,1-epsilon, 1+epsilon)*advantage)
        policy_loss = policy_loss.mean()

        loss = policy_loss + value_loss

        print(loss)
        # end student code

        return loss

    def get_action(self, obs):
        logits = self.policy(obs)
        probs = categorical.Categorical(logits=logits)
        actions = probs.sample()
        return actions, probs
```

PPO Unit Tests (must run PPO Agent cell above first)

```
# @title PPO Unit Tests (must run PPO Agent cell above first)
def PPO_networks():
    a = Agent(12, 7)
    assert a.name == 'PPO' and \
    a.policy(torch.randn(128, 12)).shape == (128, 7) and \
    a.value(torch.randn(4, 12)).shape == (4, 1) and \
    isinstance(list(a.policy.children())[-1], nn.Linear), \
    f"Networks not initialized correctly"
    print("Test passed: PPO Networks appear correct!")

PPO_networks()

def PPO_loss():
    n_steps, n_envs, n_obs, n_actions = 10, 5, 4, 1
    traj_data = TrajData(n_steps, n_envs, n_obs, n_actions)
    torch.manual_seed(0)
    traj_data.states = torch.rand_like(traj_data.states)
    traj_data.actions = torch.randint(0, n_actions, traj_data.actions.shape)
    traj_data.rewards = torch.rand_like(traj_data.rewards)
    traj_data.not_dones = torch.randint(0, 2, traj_data.not_dones.shape)
    traj_data.log_probs = torch.rand_like(traj_data.log_probs)
    traj_data.returns = 2*torch.rand_like(traj_data.returns) - 1
    a = Agent(4, 1)
    torch.manual_seed(0)
    a.policy = nn.Linear(4, 1)
    a.value = nn.Linear(4, 1)
    assert abs(a.get_loss(traj_data).item() - 0.9314) < 1e-4, \
    "PPO loss does not match expected value."
    print("Test passed: PPO loss appears correct!")

PPO_loss()
```

```
Test passed: PPO Networks appear correct!
tensor(0.9314, grad_fn=<AddBackward0>)
Test passed: PPO loss appears correct!
```

Run the PPO cell above, and then run this cell, to plot results in tensorboard.

```
drl = DRL()
for i in range(250):
    drl.rollout(i)
    drl.update()
```

```
tensor(179.0102, grad_fn=<AddBackward0>)
tensor(178.4731, grad_fn=<AddBackward0>)
tensor(177.9429, grad_fn=<AddBackward0>)
tensor(177.4200, grad_fn=<AddBackward0>)
tensor(176.9009, grad_fn=<AddBackward0>)
tensor(176.3830, grad_fn=<AddBackward0>)
tensor(175.8643, grad_fn=<AddBackward0>)
tensor(175.3451, grad_fn=<AddBackward0>)
tensor(174.8264, grad_fn=<AddBackward0>)
tensor(174.3095, grad_fn=<AddBackward0>)
tensor(241.7383, grad_fn=<AddBackward0>)
tensor(241.1638, grad_fn=<AddBackward0>)
tensor(240.5841, grad_fn=<AddBackward0>)
tensor(240.0006, grad_fn=<AddBackward0>)
tensor(239.4141, grad_fn=<AddBackward0>)
tensor(238.8253, grad_fn=<AddBackward0>)
tensor(238.2372, grad_fn=<AddBackward0>)
tensor(237.6537, grad_fn=<AddBackward0>)
tensor(237.0722, grad_fn=<AddBackward0>)
tensor(236.4892, grad_fn=<AddBackward0>)
tensor(254.9880, grad_fn=<AddBackward0>)
tensor(254.3616, grad_fn=<AddBackward0>)
tensor(253.7312, grad_fn=<AddBackward0>)
tensor(253.0975, grad_fn=<AddBackward0>)
tensor(252.4620, grad_fn=<AddBackward0>)
tensor(251.8260, grad_fn=<AddBackward0>)
tensor(251.1905, grad_fn=<AddBackward0>)
tensor(250.5552, grad_fn=<AddBackward0>)
tensor(249.9192, grad_fn=<AddBackward0>)
tensor(249.2826, grad_fn=<AddBackward0>)
tensor(369.8353, grad_fn=<AddBackward0>)
tensor(369.0504, grad_fn=<AddBackward0>)
tensor(368.2531, grad_fn=<AddBackward0>)
tensor(367.4452, grad_fn=<AddBackward0>)
tensor(366.6287, grad_fn=<AddBackward0>)
tensor(365.8055, grad_fn=<AddBackward0>)
tensor(364.9768, grad_fn=<AddBackward0>)
tensor(364.1444, grad_fn=<AddBackward0>)
tensor(363.3082, grad_fn=<AddBackward0>)
tensor(362.4663, grad_fn=<AddBackward0>)
tensor(452.3068, grad_fn=<AddBackward0>)
tensor(451.3527, grad_fn=<AddBackward0>)
tensor(450.3867, grad_fn=<AddBackward0>)
tensor(449.4103, grad_fn=<AddBackward0>)
tensor(448.4241, grad_fn=<AddBackward0>)
tensor(447.4292, grad_fn=<AddBackward0>)
tensor(446.4262, grad_fn=<AddBackward0>)
tensor(445.4169, grad_fn=<AddBackward0>)
tensor(444.4043, grad_fn=<AddBackward0>)
tensor(443.3903, grad_fn=<AddBackward0>)
tensor(627.1780, grad_fn=<AddBackward0>)
tensor(625.9277, grad_fn=<AddBackward0>)
tensor(624.6546, grad_fn=<AddBackward0>)
tensor(623.3613, grad_fn=<AddBackward0>)
tensor(622.0496, grad_fn=<AddBackward0>)
tensor(620.7213, grad_fn=<AddBackward0>)
tensor(619.3780, grad_fn=<AddBackward0>)
tensor(618.0217, grad_fn=<AddBackward0>)
```

PPO Conceptual Questions:

- If advantage for a state-action pair $A(s_t, a_t)$ is a large positive number and epsilon is $\epsilon = .2$, how might the probability ratio for (s_t, a_t) evolve over 10 training epochs with a large learning rate? What if Advantage is large and negative? What if its zero?

In PPO, the probability ratio prevents aggressive policy updates. When an action has a large positive advantage, meaning it's much better than average, the policy update will quickly push the ratio towards its upper clipping bound of 1+ϵ=1.2, preventing a massive, unstable change. Conversely, if the advantage is large and negative, the action is worse than average, and the policy will update to discourage it, driving the ratio toward the lower clipping bound of 1−ϵ=0.8. Finally, an advantage of zero indicates an average action, providing no gradient signal, so the ratio remains at 1, and the policy for that state-action pair does not change.

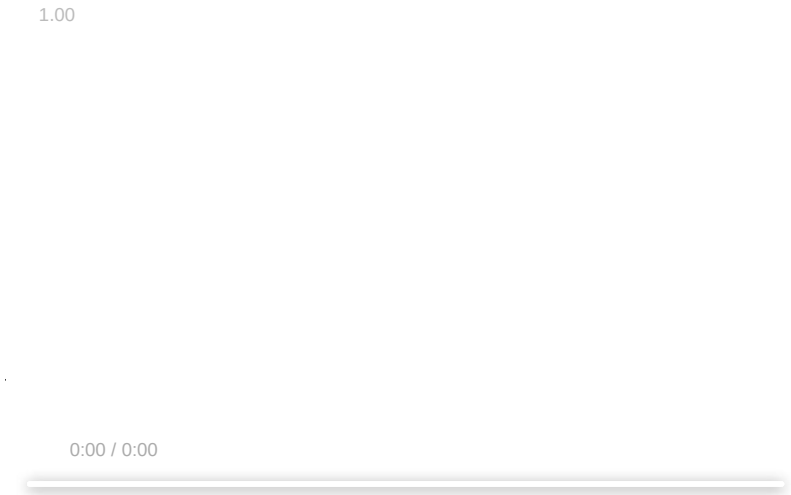
- When the clipped expression is activated for a given state-action pair, what is the gradient of the loss function with respect to network parameters for that state action pair? How will the probability of that action be changed during back propagation?

$$\text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)$$

When the clipped expression for a state-action pair is activated, the gradient of the loss function becomes zero for that pair. This is because the clipping mechanism clamps the objective at its boundary, effectively stopping any further updates that would push the probability ratio outside the safe range of 1±ϵ. Consequently, during backpropagation, the probability of that action will not change, ensuring a stable and controlled learning process.

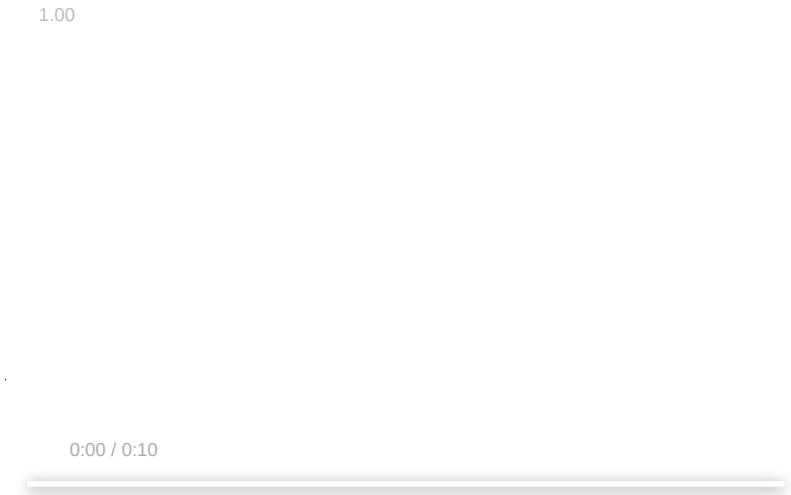
Visualization

```
untrained = DRL()
visualize(untrained.agent)
print("untrained agent")
```




```
untrained agent
```

```
# each time this cell is run, a new random rollout is recorded
# put this cell below REINFORCE or VPG and re-run their training if youd like to visualize them.
visualize(drl.agent)
print("PPO trained agent")
```




```
PPO trained agent
```

Optional Extra Credit: GAE [4 pts]

If you have time and you're up for a challenge... Implement Generalized Advantage Estimation for PPO or VPG instead of our simple sum of discounted rewards. Plot the reward curves in tensorboard.

Please modify the code below to calcualte the advantage using GAE. You must maintain the original functionality previously implemented, but besides that modify the code how ever you see fit. Copy-paste code below or modify it in place. For GAE, disregard these warnings: **You shouldnt need to change this code**. You will probably need to modify TrajData, DRL.rollout(), and implement a new Agent. Note, we havent tested how easy or hard these changes are, so proceed with caution.

Here's a great lecture by Sergey Levine on Eligability traces and GAE, especially starting from 7:41. Good luck and happy coding!

```
class TrajData:
    def __init__(self, n_steps, n_envs, n_obs, n_actions):
        s, e, o, a = n_steps, n_envs, n_obs, n_actions
        from torch import zeros
        self.states = zeros((s, e, o))
        self.actions = zeros((s, e))
        self.rewards = zeros((s, e))
        self.not_dones = zeros((s, e))
        self.log_probs = zeros((s, e))
        self.returns = zeros((s, e))
        self.advantages = zeros((s, e))
        self.n_steps = s

    def detach(self):
        self.actions = self.actions.detach()
        self.log_probs = self.log_probs.detach()

    def store(self, t, obs, actions, rewards, log_probs, done):
        self.states[t] = obs
        self.actions[t] = actions
        self.rewards[t] = torch.tensor(rewards)
        self.log_probs[t] = log_probs
        self.not_dones[t] = 1 - torch.tensor(done).float()

    def calc_gae(self, values, gamma=0.99, lam=0.95):
        self.advantages = torch.zeros_like(self.rewards)
        next_values = torch.cat((values[1:], torch.zeros_like(values[0:1])), dim=0)

        last_gae_lam = 0.0
        for t in reversed(range(self.n_steps)):
            delta = self.rewards[t] + gamma * next_values[t] * self.not_dones[t] - values[t]
            last_gae_lam = delta + gamma * lam * self.not_dones[t] * last_gae_lam
            self.advantages[t] = last_gae_lam

    def calc_returns(self, gamma=0.99):
        self.returns = deepcopy(self.rewards)
        for t in reversed(range(self.n_steps - 1)):
            self.returns[t] += self.returns[t + 1] * self.not_dones[t] * gamma

# Agent Class
class Agent(nn.Module):
    def __init__(self, n_obs, n_actions, name='GAE'):
        super().__init__()
        self.name = name
        self.policy = nn.Sequential(
            nn.Linear(n_obs, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, n_actions)
        )
        self.critic = nn.Sequential(
            nn.Linear(n_obs, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1)
        )

    def get_action(self, obs):
        logits = self.policy(obs)
        probs = categorical.Categorical(logits=logits)
        actions = probs.sample()
        return actions, probs

    def get_value(self, obs):
        return self.critic(obs).squeeze(-1)

    def get_loss(self, traj_data):
        log_probs = traj_data.log_probs
        advantages = traj_data.advantages

        # Policy Loss
        policy_loss = -(log_probs * advantages).mean()

        # Value Loss
        values = self.get_value(traj_data.states)
```

```
        value_loss = (values - traj_data.returns).pow(2).mean()

        return policy_loss + value_loss

# DRL Class
class DRL:
    def __init__(self):
        self.n_envs = 64
        self.n_steps = 256
        self.n_obs = 4
        self.envs = gym.vector.SyncVectorEnv([lambda: gym.make("CartPole-v1") for _ in range(self.n_envs)])
        self.traj_data = TrajData(self.n_steps, self.n_envs, self.n_obs, n_actions=1)
        self.agent = Agent(self.n_obs, n_actions=2)
        self.optimizer = Adam(self.agent.parameters(), lr=1e-3)
        self.writer = SummaryWriter(log_dir=f'runs/{self.agent.name}')

    def rollout(self, i):
        obs, _ = self.envs.reset()
        obs = torch.Tensor(obs)

        values_list = []

        for t in range(self.n_steps):
            with torch.no_grad():
                actions, probs = self.agent.get_action(obs)
                values = self.agent.get_value(obs)
            values_list.append(values)

            log_probs = probs.log_prob(actions)
            next_obs, rewards, done, truncated, infos = self.envs.step(actions.numpy())
            self.traj_data.store(t, obs, actions, rewards, log_probs, done)
            obs = torch.Tensor(next_obs)

        values_tensor = torch.stack(values_list)
        self.traj_data.calc_gae(values_tensor, gamma=0.99, lam=0.95)
        self.writer.add_scalar("Reward", self.traj_data.rewards.mean(), i)
        self.writer.flush()

    def update(self):
        epochs = 10 if self.agent.name == 'GAE' else 1

        for _ in range(epochs):
            loss = self.agent.get_loss(self.traj_data)

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

        self.traj_data.detach()

drl = DRL()

for i in range(250):
    drl.rollout(i)
    drl.update()
    print(f"Episode: {i}, Average Reward: {drl.traj_data.rewards.mean().item():.2f}")

drl.writer.close()
print("Training complete")
```

```
Episode: 0, Average Reward: 0.96
Episode: 1, Average Reward: 0.96
Episode: 2, Average Reward: 0.96
Episode: 3, Average Reward: 0.96
Episode: 4, Average Reward: 0.96
Episode: 5, Average Reward: 0.96
Episode: 6, Average Reward: 0.96
Episode: 7, Average Reward: 0.96
Episode: 8, Average Reward: 0.96
Episode: 9, Average Reward: 0.96
Episode: 10, Average Reward: 0.96
Episode: 11, Average Reward: 0.96
Episode: 12, Average Reward: 0.96
Episode: 13, Average Reward: 0.96
Episode: 14, Average Reward: 0.96
Episode: 15, Average Reward: 0.96
Episode: 16, Average Reward: 0.96
Episode: 17, Average Reward: 0.96
Episode: 18, Average Reward: 0.96
```

```
Episode: 19, Average Reward: 0.96
Episode: 20, Average Reward: 0.96
Episode: 21, Average Reward: 0.96
Episode: 22, Average Reward: 0.96
Episode: 23, Average Reward: 0.96
Episode: 24, Average Reward: 0.96
Episode: 25, Average Reward: 0.96
Episode: 26, Average Reward: 0.96
Episode: 27, Average Reward: 0.96
Episode: 28, Average Reward: 0.96
Episode: 29, Average Reward: 0.96
Episode: 30, Average Reward: 0.96
Episode: 31, Average Reward: 0.96
Episode: 32, Average Reward: 0.96
Episode: 33, Average Reward: 0.96
Episode: 34, Average Reward: 0.96
Episode: 35, Average Reward: 0.96
Episode: 36, Average Reward: 0.96
Episode: 37, Average Reward: 0.96
Episode: 38, Average Reward: 0.96
Episode: 39, Average Reward: 0.96
Episode: 40, Average Reward: 0.96
Episode: 41, Average Reward: 0.96
Episode: 42, Average Reward: 0.96
Episode: 43, Average Reward: 0.96
Episode: 44, Average Reward: 0.96
Episode: 45, Average Reward: 0.96
Episode: 46, Average Reward: 0.96
Episode: 47, Average Reward: 0.96
Episode: 48, Average Reward: 0.96
Episode: 49, Average Reward: 0.96
Episode: 50, Average Reward: 0.96
Episode: 51, Average Reward: 0.96
```