full_solution

May 6, 2023

1 CMPE260 SPRING'23 - HW3

Ezgi Kaya

Derek Lilienthal

2 main.py file :

[ ]:
```
import gym
import a3_gym_env
import Modules
import collections
import sys
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

from Modules import Net, ReplayMemory
from torch.distributions import MultivariateNormal

env = gym.make('Pendulum-v1-custom')

# sample hyperparameters
num_timesteps = 200 # T
num_trajectories = 10 # N
num_iterations = 250
epochs = 100

batch_size = 10
learning_rate = 3e-4
eps = 0.2 # clipping


# function to calculate the (discounted) reward-to-go from a sequence of rewards
def calc_reward_togo(rewards, gamma=0.99):
    n = len(rewards)
```

```
    reward_togo = np.zeros(n)
    reward_togo[-1] = rewards[-1]
    for i in reversed(range(n-1)):
        reward_togo[i] = rewards[i] + gamma * reward_togo[i+1]

    reward_togo = torch.tensor(reward_togo, dtype=torch.float)
    return reward_togo

# compute advantage estimates (as done in PPO paper)
def calc_advantages(rewards, values, gamma=0.99, lambda_=1):
    advantages = torch.zeros_like(torch.as_tensor(rewards))
    sum = 0
    for t in reversed(range(len(rewards)-1)):
        delta = rewards[t] + gamma * values[t + 1] - values[t]
        sum = delta + gamma * lambda_ * sum
        advantages[t] = sum

    return advantages


class PPO:
    def __init__(self, clipping_on, advantage_on, gamma=0.99):

        self.policy_net = Net(3,1)
        self.critic_net = Net(3,1)

        #self.policy_opt = torch.optim.Adam(self.policy_net.parameters(),␣
 ↪lr=learning_rate)
        #self.critic_opt = torch.optim.Adam(self.critic_net.parameters(),␣
 ↪lr=learning_rate)

        self.optimizer = torch.optim.Adam([  # Update both models together
            {'params': self.policy_net.parameters(), 'lr': learning_rate},
            {'params': self.critic_net.parameters(), 'lr': learning_rate}
                    ])

        self.memory = ReplayMemory(batch_size)

        self.gamma = gamma
        self.lambda_ = 1
        self.vf_coef = 1  # c1
        self.entropy_coef = 0.01  # c2

        self.clipping_on = clipping_on
        self.advantage_on = advantage_on

        # use fixed std
```

```
        self.std = torch.diag(torch.full(size=(1,), fill_value=0.5))

    def generate_trajectory(self):

        current_state = env.reset()
        states = []
        actions = []
        rewards = []
        log_probs = []


        # Run the old policy in environment for num_timestep
        for t in range(num_timesteps):

            # compute mu(s) for the current state
            mean = self.policy_net(torch.as_tensor(current_state))

            # the gaussian distribution
            normal = MultivariateNormal(mean, self.std)

            # sample an action from the gaussian distribution
            action = normal.sample().detach()
            log_prob = normal.log_prob(action).detach()

            # emulate taking that action
            next_state, reward, done, info = env.step(action)

            # store results in a list
            states.append(current_state)
            actions.append(action)
            rewards.append(reward)
            log_probs.append(log_prob)

            #env.render()

            current_state = next_state


        # calculate reward to go
        rtg = calc_reward_togo(torch.as_tensor(rewards), self.gamma)

        # calculate values
        values = self.critic_net(torch.as_tensor(states)).squeeze()

        # calculate advantages
        advantages = calc_advantages(rewards, values.detach(), self.gamma, self.
↪lambda_)
```

```
        # save the transitions in replay memory
        for t in range(len(rtg)):
            self.memory.push(states[t], actions[t], rewards[t], rtg[t],␣
↪advantages[t], values[t], log_probs[t])

        #env.close()


    def train(self):

        train_actor_loss = []
        train_critic_loss = []
        train_total_loss = []
        train_reward = []

        for _ in range(num_iterations): # k

            # collect a number of trajectories and save the transitions in␣
↪replay memory
            for _ in range(num_trajectories):
                self.generate_trajectory()

            # sample from replay memory
            states, actions, rewards, rewards_togo, advantages, values,␣
↪log_probs, batches = self.memory.sample()

            actor_loss_list = []
            critic_loss_list = []
            total_loss_list = []
            reward_list = []
            for _ in range(epochs):

                # calculate the new log prob
                mean = self.policy_net(states)
                normal = MultivariateNormal(mean, self.std)
                new_log_probs = normal.log_prob(actions.unsqueeze(-1))

                r = torch.exp(new_log_probs - log_probs)

                if self.clipping_on == True:
                    clipped_r = torch.clamp(r, 1 - eps, 1 + eps)
                else:
                    clipped_r = r

                new_values = self.critic_net(states).squeeze()
                returns = (advantages + values).detach()
```

```
                if self.advantage_on == True:
                    actor_loss = (-torch.min(r * advantages, clipped_r *␣
↪advantages)).mean()
                    critic_loss = nn.MSELoss()(new_values.float(), returns.
↪float())
                else:
                    actor_loss = (-torch.min(r * rewards_togo, clipped_r *␣
↪rewards_togo)).mean()
                    critic_loss = nn.MSELoss()(new_values.float(), rewards_togo.
↪float())

                # Calcualte total loss
                total_loss = actor_loss + (self.vf_coef * critic_loss) - (self.
↪entropy_coef * normal.entropy().mean())

                # update policy and critic network
                self.optimizer.zero_grad()
                total_loss.backward(retain_graph=True)
                self.optimizer.step()

                actor_loss_list.append(actor_loss.item())
                critic_loss_list.append(critic_loss.item())
                total_loss_list.append(total_loss.item())
                reward_list.append(sum(rewards))

            # clear replay memory
            self.memory.clear()

            avg_actor_loss = sum(actor_loss_list) / len(actor_loss_list)
            avg_critic_loss = sum(critic_loss_list) / len(critic_loss_list)
            avg_total_loss = sum(total_loss_list) / len(total_loss_list)
            avg_reward = sum(reward_list) / len(reward_list)

            train_actor_loss.append(avg_actor_loss)
            train_critic_loss.append(avg_critic_loss)
            train_total_loss.append(avg_total_loss)
            train_reward.append(avg_reward)

            print('Actor loss = ', avg_actor_loss)
            print('Critic loss = ', avg_critic_loss)
            print('Total Loss = ', avg_total_loss)
            print('Reward = ', avg_reward)
            print("")

        # save the networks
```

```
        torch.save(self.policy_net.state_dict(), f'./results/policy_net_{self.
↪clipping_on}_{self.advantage_on}.pt')
        torch.save(self.critic_net.state_dict(), f'./results/critic_net_{self.
↪clipping_on}_{self.advantage_on}.pt')

        fig, axes = plt.subplots(1, 4, figsize=(20, 5))
        axes[0].plot(range(len(train_actor_loss)), train_actor_loss, 'r',␣
↪label='Actor Loss')
        axes[0].set_title('Actor Loss', fontsize=18)

        axes[1].plot(range(len(train_critic_loss)), train_critic_loss, 'b',␣
↪label='Critic Loss')
        axes[1].set_title('Critic Loss', fontsize=18)

        axes[2].plot(range(len(train_total_loss)), train_total_loss, 'm',␣
↪label='Total Loss')
        axes[2].set_title('Total Loss', fontsize=18)

        axes[3].plot(range(len(train_reward)), train_reward, 'orange',␣
↪label='Accumulated Reward')
        axes[3].set_title('Accumulated Reward', fontsize=18)

        fig.suptitle(f'Results for clipping_on={self.clipping_on} and␣
↪advantage_on={self.advantage_on}\n', fontsize=20)
        fig.tight_layout()
        plt.savefig(f'./results/figure1_{self.clipping_on}_{self.advantage_on}.
↪png')
        fig.show()

        self.show_value_grid()


    def show_value_grid(self):

        # sweep theta and theta_dot and find all states
        theta = torch.linspace(-np.pi, np.pi, 100)
        theta_dot = torch.linspace(-8, 8, 100)
        values = torch.zeros((len(theta), len(theta_dot)))

        for i, t in enumerate(theta):
            for j, td in enumerate(theta_dot):
                state = (torch.cos(t), torch.sin(t), td)
                values[i, j] = self.critic_net(torch.as_tensor(state))

        # display the resulting values using imshow
        fig2 = plt.figure(figsize=(5, 5))
```

```
        plt.imshow(values.detach().numpy(), extent=[theta[0], theta[-1],␣
↪theta_dot[0], theta_dot[-1]] ,aspect=0.4)
        plt.title('Value grid', fontsize=18)
        plt.xlabel('angle', fontsize=18)
        plt.ylabel('angular velocity', fontsize=18)

        plt.savefig(f'./results/figure2_{self.clipping_on}_{self.advantage_on}.
↪png')
        plt.show()



    def test(self):

        self.policy_net.load_state_dict(torch.load(f'./results/policy_net_{self.
↪clipping_on}_{self.advantage_on}.pt'))

        current_state = env.reset()

        for i in range(200):

            # compute mu(s) for the current state
            mean = self.policy_net(torch.as_tensor(current_state))

            # the gaussian distribution
            normal = MultivariateNormal(mean, self.std)

            # sample an action from the gaussian distribution
            action = normal.sample().detach().numpy()

            # emulate taking that action
            next_state, reward, done, info = env.step(action)

            env.render()

            current_state = next_state

        env.close()




if __name__ == '__main__':

    user_input = input("Press 0 to run test only.\nPress 1 to run training +␣
↪test.\n")
```

```
cases = [(True,True), (False,True), (True,False)]

num = ord(input("Select a case :\n  0: with clipping & with advantage\n  1:␣
↪without clipping & with advantage\n  2: with clipping & without␣
↪advantage\n")) - 48

agent = PPO(clipping_on=cases[num][0], advantage_on=cases[num][1])

if user_input == '1':
    agent.train()


agent.test()
```

3 Modules.py file:

[ ]:

```
import numpy as np
import torch
import torch.nn as nn
import collections

class Net(nn.Module):
        def __init__(self, input_size, output_size, hidden_size=64,␣
↪activation=nn.functional.relu):
                super(Net, self).__init__()
                self.layer1 = nn.Linear(input_size, hidden_size)
                self.layer2 = nn.Linear(hidden_size, hidden_size)
                self.layer3 = nn.Linear(hidden_size, output_size)
                self.act = activation

        def forward(self, x):
                x = self.act(self.layer1(x))
                x = self.act(self.layer2(x))
                out = self.layer3(x)

                return out


class ReplayMemory():
    def __init__(self, batch_size=10000):
        self.states = []
        self.actions = []
        self.rewards = []
        self.rewards_togo = []
        self.advantages = []
```

```
        self.values = []
        self.log_probs = []
        self.batch_size = batch_size

    def push(self, state, action, reward, reward_togo, advantage, value, 
 ↪log_prob):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.rewards_togo.append(reward_togo)
        self.advantages.append(advantage)
        self.values.append(value)
        self.log_probs.append(log_prob)

    def sample(self):
        num_states = len(self.states)
        batch_start = torch.arange(0, num_states, self.batch_size)
        indices = torch.randperm(num_states)
        batches = [indices[i:i+self.batch_size] for i in batch_start]

        return (torch.tensor(self.states),
                torch.tensor(self.actions),
                torch.tensor(self.rewards),
                torch.tensor(self.rewards_togo),
                torch.tensor(self.advantages),
                torch.tensor(self.values),
                torch.tensor(self.log_probs),
                batches)

    def clear(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.rewards_togo = []
        self.advantages = []
        self.values = []
        self.log_probs = []
```

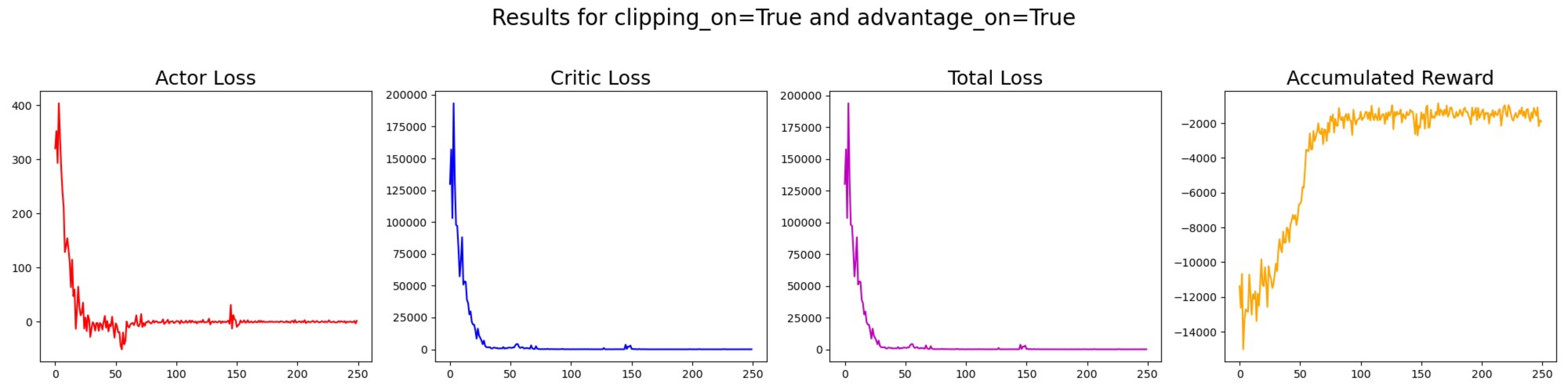
4 Observations:

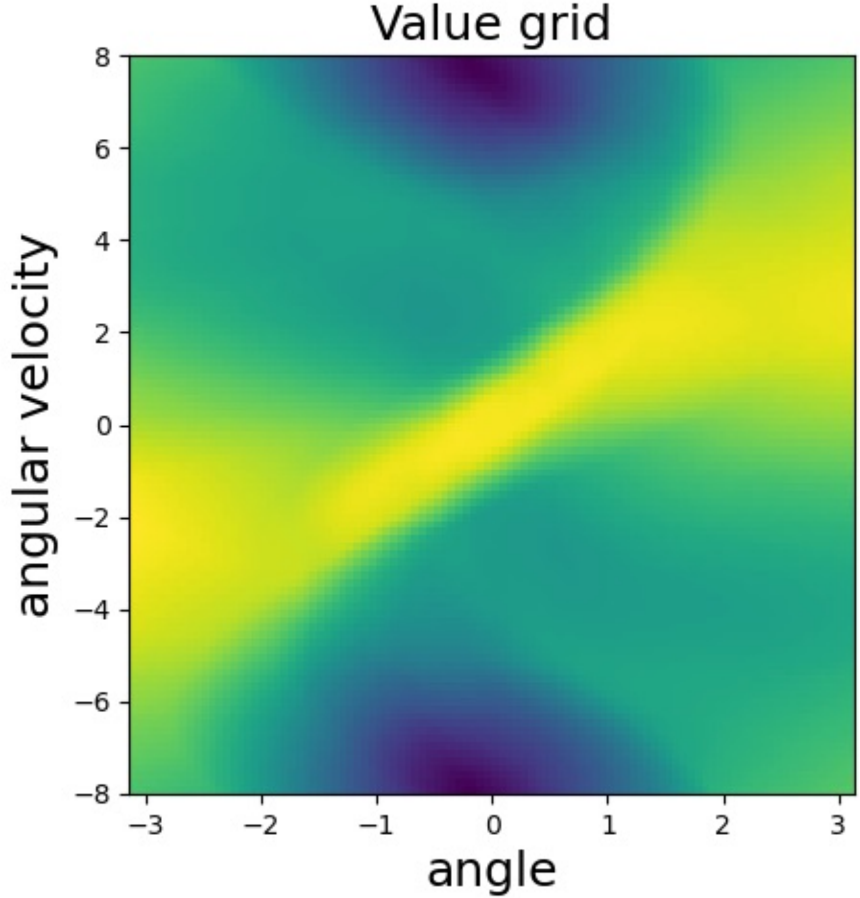
It is seen that in case 1 (with clipping & advantage), the agent can learn how to balance the pendulum. The value grid shows that the value is the highest (hence the color is lightest) at the center, where the pendulum is at the goal state (top), as expected. The value is also higher when the magnitude of angle is high, and the angular velocity is also large, which means the pendulum is about to swing up. On the contrary, when the angle is low but the angular velocity is large, which is an unwanted situation; the value is lower.

In case 2 (without cliping & with advantage), we see a lot of variation during training. This is because without clipping, we don't limit the update to the policy parameters during training, and therefore the policy can change too much during a single iteration. As a result, the agent cannot learn.

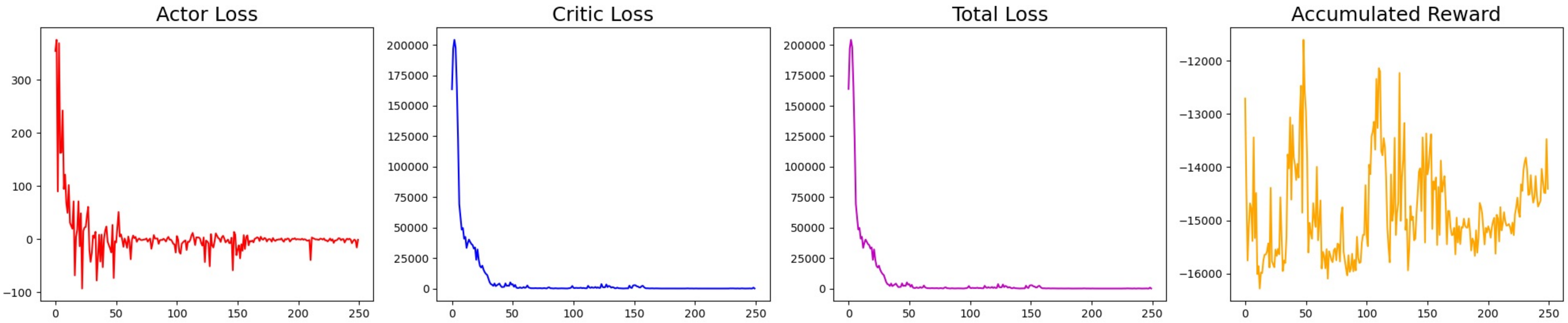
In case 3 (with clipping & without advantage), again, we see a lot of variation, even higher than case 2. This is because when we don't subtract a baseline from reward-to-go, the gradients become very large, causing a high variance and unstability. As a result, the agent cannot learn.

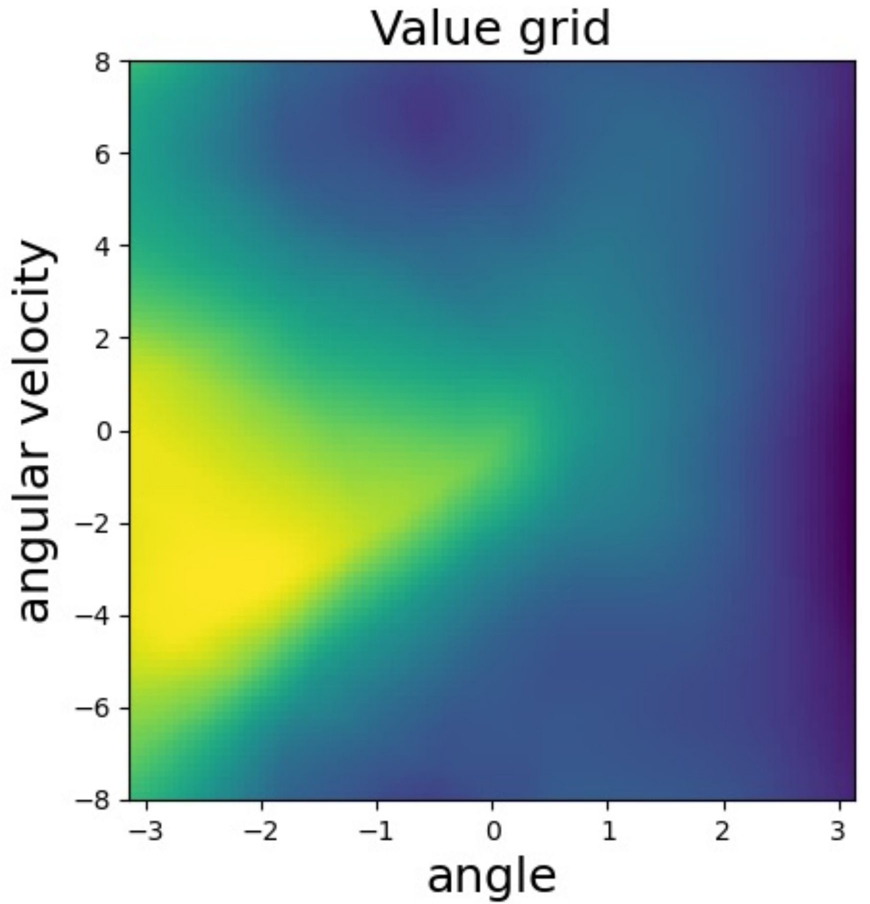
5 Results:


Results for clipping_on=True and advantage_on=True
Actor Loss
Critic Loss
Total Loss
Accumulated Reward
400
300
200
100
0
200000
175000
150000
125000
100000
75000
50000
25000
0
−2000
−4000
−6000
−8000
−10000
−12000
−14000
0
50
100
150
200
250

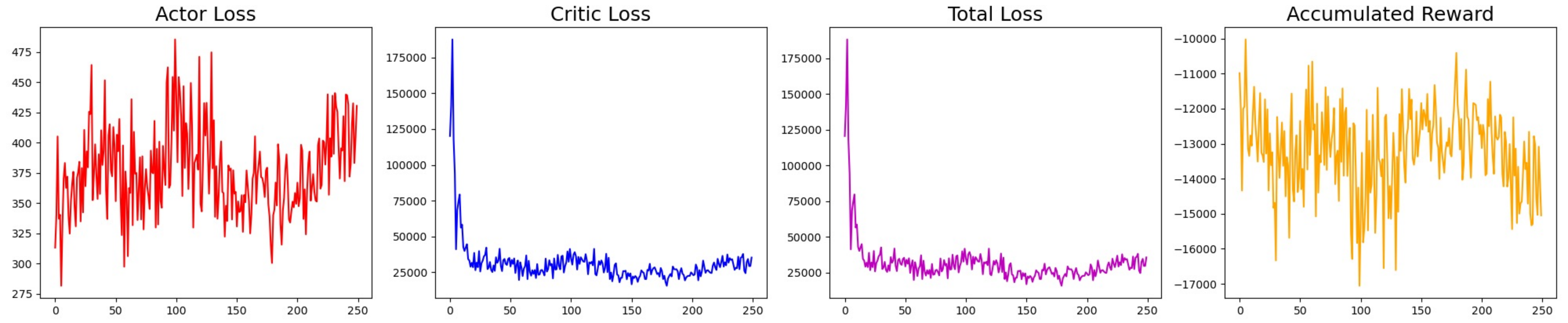

Value grid
angular velocity
angle
8
6
4
2
0
−2
−4
−6
−8
−3
−2
−1
0
1
2
3

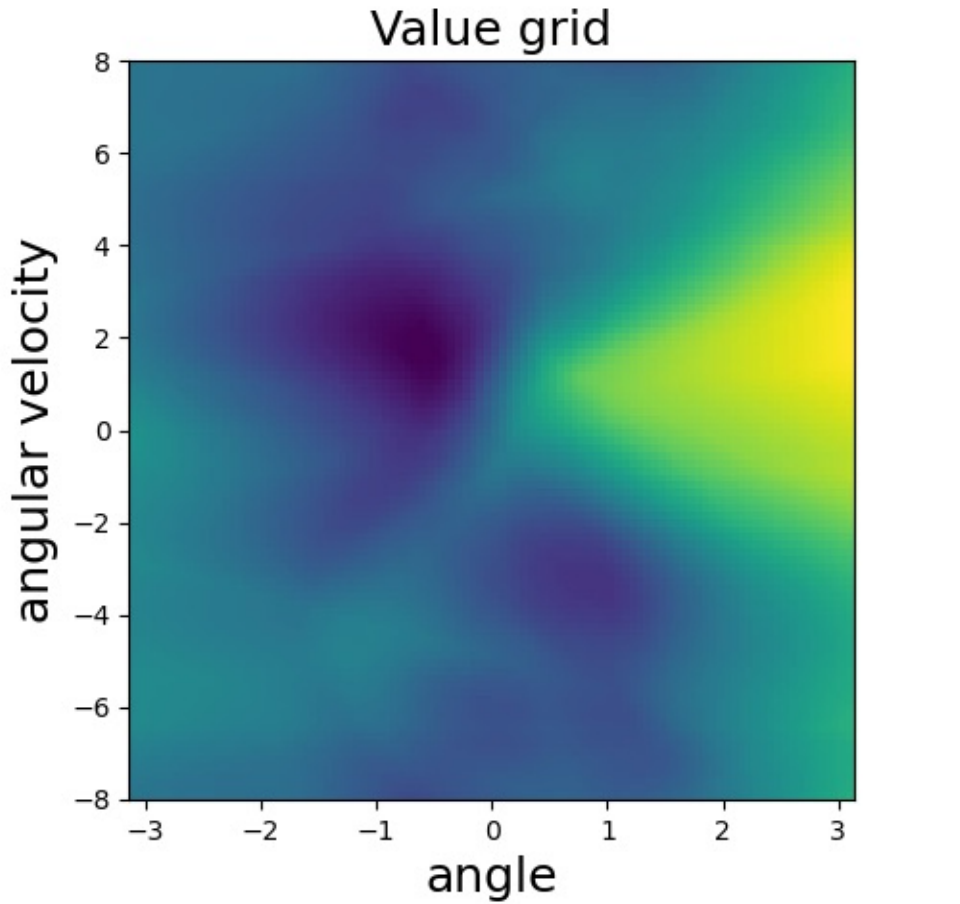
Results for clipping_on=False and advantage_on=True




Value grid
angular velocity
8
6
4
2
0
−2
−4
−6
−8
−3
−2
−1
0
1
2
3
angle

Results for clipping_on=True and advantage_on=False





Value grid
angular velocity
angle
8
6
4
2
0
−2
−4
−6
−8
−3
−2
−1
0
1
2
3