UNIVERSITY OF VIRGINIA
LEARNING IN ROBOTICS
CS 6501
SPRING 2025

HOMEWORK 3

TYLER KIM [TKJ9EP@VIRGINIA.EDU],
COLLABORATORS:

Solution 1 (Time spent: 10 hours)**.** a) The stochastic controller $u_\theta(\cdot|x)$ is implemented using a neural network with a linear, relu, linear and tanh layer where a state is fed as input and the mean control, μ serves as the output. The $u_\theta(\cdot|x)$ is as a $torch.distribution.Normal$ class where the μ from the neural network and $s.std = 1$ serve as the mean and standard deviation, respectively. The probability distribution is normal with a mean of μ from the neural network and a standard deviation of 1 as evidenced in the code since $torch.distribution.Normal()$. The logarithmic likelihood $\log u_\theta(u|x)$ is computed by calling $torch.distribution.Normal().log\_prob()$ that takes an input u from the forward step. The constraint $|u| \leq 1$ is enforced using the tanh function and then adding noise.

b) – I collected 10 trajectories per iteration and ran for either 750 or 1000 iterations. For the best model, I ran 1000 iterations.

```
# 1. minibatch sample of trajectories
R_list = []
trajectories = []
print("Sampling trajectories...")
for traj in range(num_trajectories):
    t = rollout(policy)
    R = t["R"]

    trajectories.append(t)
    R_list.append(R)

R_numpy = np.array(R_list) # convert to numpy array
traj_numpy = np.array(trajectories)
b = R_numpy.mean() # baseline which is just the average
```

FIGURE 1. A snippet of the code for the sampling loop to get all the trajectories.

– For each trajectory, I got the $logp$ value from running the policy network and multiplied it with the respective cumulative reward, R, minus the baseline b. The baseline is simply the average total reward for all sampled trajectories. The policy gradient is computed as $(R_i - b) * logp_i$ and stored into a list. The policy gradients were stacked once the computation was done and the mean was taken.

```
# 2. policy gradient averaging
policy_grad_list = []
print("Getting the Policy Gradient...")
for R_idx in range(R_numpy.shape[0]):
    t = traj_numpy[R_idx]
    logp = policy(t["x"].view(-1, 2), t["u"].view(-1, 1))[1]

    # f = -((R_numpy[R_idx] - b) * logp)
    f = ((R_numpy[R_idx] - b) * logp)

    policy_grad_list.append(f)

batch_policy_outputs = th.stack(policy_grad_list) # combine into a single batch
average_batch_policy = batch_policy_outputs.mean()
```

FIGURE 2. A snippet of the code for the policy gradient and then converting it into a batch tensor.

– I used the average returns of a mini-batch baseline. This was implemented by getting storing all the R values of the sampled trajectory into a list and then getting the average of that list and storing it in the variable, b. The logic occurs in the trajectory sampling and before the policy gradient computation. Figure 1 has the baseline computation shown with respect to the trajectory sampling.

```
R_numpy = np.array(R_list) # convert to numpy array
traj_numpy = np.array(trajectories)
b = R_numpy.mean() # baseline which is just the average
```

FIGURE 3. A snippet of the code for getting the baseline.

– Gradient ascent was performed using the Adam optimizer and with the policy gradient of $(R_i - b) * logp_i$. I tested $1e-3$ and $1e-5$ for the learning rates but found that the learning rate of $1e-3$ works best. I also tried negating policy gradient to $-((R_i - b) * logp_i)$ but this seemed to encourage the model to find the worst trajectory so I stuck with $(R_i - b) * logp_i$. I think this worked because the R was already negative so negating the policy gradient again worked against the favor of the algorithm.

c)

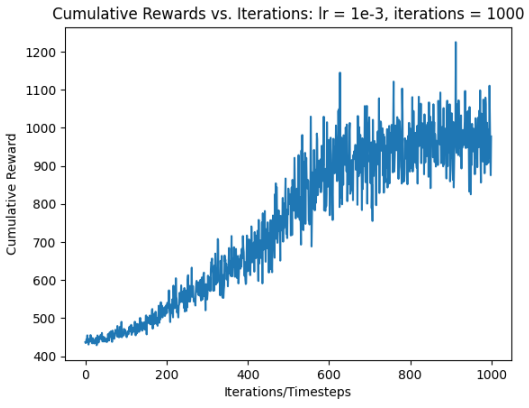


FIGURE 4. Cumulative Reward Graph for over 1000 time-steps with learning rate of $1e-3$.

$m = 2$ Cumulative Reward: 515.40

The model did not generalize particularly well with a pretty low cumulative reward. This is expected because the model was trained to get an action when $m = 1$, so when $m = 2$, the model was trying to generalize on a completely set of data. In other words, the set of action when $m = 2$ is not in the distribution as the actions in $m = 1$. Essentially, the model is trained on one set of data but asked to generalize on a completely set of data where $m = 2$.

Solution 2 (Time spent: 12 hours)**.** a) There are multiple parts of the snippet but I will group them in 4 major categories: Epsilon Greedy, Loss Function, Evaluation, and Plotting.

0.1. **Epsilon Greedy.** I modified the $control(s, x, eps = 0)$ to generate a random value between 0 and 1 and check if that generated random value is smaller than the eps argument. If the random value is smaller than eps, then randomly pick 0 or 1 to simulate uniform distribution of the actions. If the random value is greater than or equal to eps, then the code fetches the $argmax$ of running the state, x, through the model. Figure 5 shows the code snippet for the epsilon greedy exploration.

```
def control(s, x, eps=0):
    # 1. get q values for all controls
    q = s.m(x)

    ### TODO: XXXXXXXXXXX
    # eps-greedy strategy to choose control input
    # note that for eps=0
    # you should return the correct control u
    rand_val = th.rand(1).item()
    if rand_val < eps: # explore
        u = th.randint(0, 2, (1,))

    else: # exploit
        u = th.argmax(q, dim = 1)

    return u
```

FIGURE 5. Code snippet for epsilon-greedy exploration.

0.2. **Loss Function.** I modified the $loss(q, ds)$ to have $loss(q, qc, ds)$ because I initialize the target network in the main loop by calling $qc = copy.deepcopy(q)$. In the $loss(q, qc, ds)$ function, I sampled a mini-batch from ds by first flattening out all the individual control inputs into a single, flattened list, and then randomly sampling 128 random inputs from it. This flattened list takes all the dictionaries in the trajectories in ds and storing them into a single list without any correlation. Next, the current states x, next states xp, actions u, rewards r, and dones d were separated and into different lists where each list was stacked using the $th.stack()$ function.

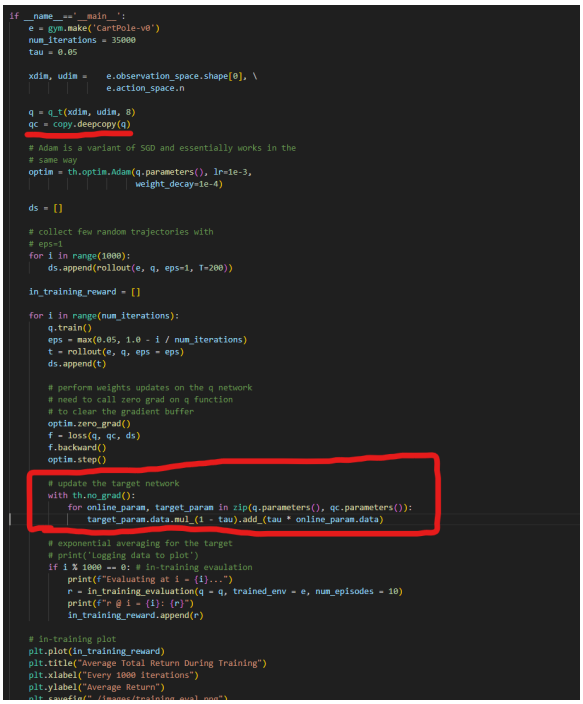
For coding up the double-q trick, I ran the online network q using the tensor of current states $current\_states$ and got the values associated with the actual actions that were taken with the tensor of actions $actions$ as the indices. Then with $th.no\_grad()$, I got the $argmax$ of the output of q with the tensor of next states $next\_states$ as the input and used it to serve as the indices to select the values from the target network's outputs. The output of the target network qc used the $next\_states$ as the input. Then, y_t was calculated by summing the rewards with the $\gamma * (1 - 1_{terminated}) * qc(x_{t+1}, u')$ where u' represents the actions the q found to yield the highest reward. Finally, the MSE was calculated between the returns of

the actual actions that were taken, calculated from q, and y_t. Figure 6 shows a snippet of the code that handles the loss function.

```
def loss(q, qc, ds):
    ### TODO: XXXXXXXXXX
    # 1. sample mini-batch from dataset ds
    batch_size = 128
    gamma = .99

    flattened_ds = [entry for trajectory in ds for entry in trajectory] # flatten the list because we want to sample from different trajectories
    mini_batch = random.sample(flattened_ds, k = batch_size)

        # separate dictionary into stack
    current_states = th.stack([th.tensor(input["x"], dtype=th.float32) for input in mini_batch])
    next_states = th.stack([th.tensor(input["xp"], dtype = th.float32) for input in mini_batch])
    actions = th.tensor([input["u"].item() for input in mini_batch]).unsqueeze(1)
    rewards = th.tensor([input["r"] for input in mini_batch]).unsqueeze(1)
    dones = th.tensor([int(input["d"]) for input in mini_batch]).unsqueeze(1)

    # 2. code up dqn with double-q trick
    online_u = q(current_states)
    actual_u = th.gather(online_u, dim = 1, index = actions)

    with th.no_grad():
        next_best_u_arg = th.argmax(q(next_states), dim = 1)
        target_u = qc(next_states)
        target_u_best_from_online = target_u.gather(1, next_best_u_arg.unsqueeze(1))
        y_t = rewards + gamma * (1 - dones) * target_u_best_from_online

    # 3. return the objective f
    f = th.sum(th.pow(actual_u - y_t, 2)) / batch_size

    return f
```

FIGURE 6. Code snippet for loss function modifications.

The $main$ function was modified where before trajectory collection, the target network qc was initialized using $qc = copy.deepcopy(q)$ where q is the online network. In the training loop, I slowly decremented epsilon from 1.0 to 0.05 using $eps = max(0.05, 1.0 - i/num\_iterations)$ where $num\_iterations$ refers to the total number of iterations for the training loop. The eps was fed into the $rollout()$ function. Finally, qc was updated using the equation $qc_{parameters} = (1 - \tau) * qc_{parameter} + \tau * q_{parameters}$ where $\tau = 0.05$.



```
if __name__=='__main__':
    e = gym.make('CartPole-v0')
    num_iterations = 35000
    tau = 0.05

    xdim, udim =    e.observation_space.shape[0], \
                    e.action_space.n

    q = q_t(xdim, udim, 8)
    qc = copy.deepcopy(q)

    # Adam is a variant of SGD and essentially works in the
    # same way
    optim = th.optim.Adam(q.parameters(), lr=1e-3,
                          weight_decay=1e-4)

    ds = []

    # collect few random trajectories with
    # eps=1
    for i in range(1000):
        ds.append(rollout(e, q, eps=1, T=200))

    in_training_reward = []

    for i in range(num_iterations):
        q.train()
        eps = max(0.05, 1.0 - i / num_iterations)
        t = rollout(e, q, eps = eps)
        ds.append(t)

        # perform weights updates on the q network
        # need to call zero grad on q function
        # to clear the gradient buffer
        optim.zero_grad()
        f = loss(q, qc, ds)
        f.backward()
        optim.step()

        # update the target network
        with th.no_grad():
            for online_param, target_param in zip(q.parameters(), qc.parameters()):
                target_param.data.mul_(1 - tau).add_(tau * online_param.data)

        # exponential averaging for the target
        # print('Logging data to plot')
        if i % 1000 == 0: # in-training evaulation
            print(f"Evaluating at i = {i}...")
            r = in_training_evaluation(q = q, trained_env = e, num_episodes = 10)
            print(f"> @ i = {i}: {r}")
            in_training_reward.append(r)

    # in-training plot
    plt.plot(in_training_reward)
    plt.title("Average Total Return During Training")
    plt.xlabel("Every 1000 iterations")
    plt.ylabel("Average Return")
```

FIGURE 7. Code snippet for the main function. The portions outline in red are relevant to the changed described.

0.3. **Evaluation.** For the evaluation, I made a new gym environment using $e = gym.make(\text{"CartPole-v0"})$, reset the gym environment, and created a $total\_reward$ variable to store the rewards. I looped for 100 iterations were for each iteration, an action was generated from the model and applied to the environment. The reward was accumulated to the $total\_reward$ variable. Finally, returned the $total\_reward/100$.

```
def evaluate(q):
    ### TODO: XXXXXXXXXXX
    # 1. create a new environment e
    e = gym.make("CartPole-v0")

    # 2. run the learnt q network for 100 trajectories on
    # this new environment to take control actions. Remember that
    # you should not perform epsilon-greedy exploration in the evaluation
    # phase
    # and report the average discounted
    # return of these 100 trajectories
    x, _ = e.reset()
    total_reward = 0

    for traj in range(100):
        x, _ = e.reset()
        done = False
        gamma = .99
        t = 0

        while not done:
            u = q.control(th.from_numpy(x).float().unsqueeze(0),
                          eps=0)
            u = u.int().numpy().squeeze()

            xp,r,d, truncated, info = e.step(u)
            done = d or truncated

            total_reward += (gamma**t) * r
            x = xp
            t += 1

    r = total_reward / 100

    return r
```

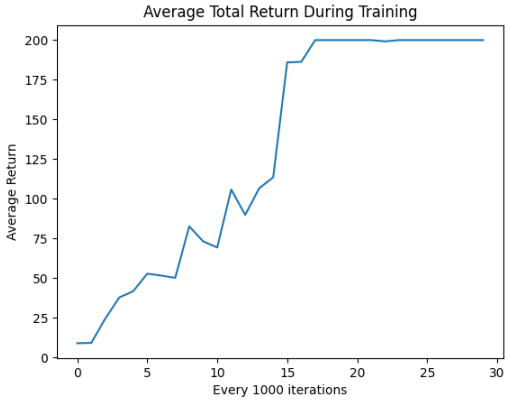
FIGURE 8. Code snippet for the evaluate function.

0.4. **Plotting.** Plotting was relatively straightforward. In the main function, I simply checked if the $iteration\%1000 == 0$ and evaluated the model at that period. For evaluating the model, I simply ran for 10 episodes where in each episode, u was fetched from the model, pushed into the environment, and the total reward was accumulated. Finally, I divided the total reward by 10 for the 10 episodes. The approach is very similar to that of the $evaluation()$ function except it does not calculate 100 trajectories per episode. The logic for evaluating training after training is the same as during training. Figures 9 and 7 display relevant information for plotting.

The iterations used were $30,000$.

```python
# get the average total return for in-training environment
def in_training_evaluation(q = None, trained_env = None, num_episodes = 10):
    total_reward = 0

    for _ in range(num_episodes): # loop through 10 episodes
        x, _ = trained_env.reset()
        done = False

        while not done:
            u = q.control(th.from_numpy(x).float().unsqueeze(0), eps = 0)
            u = u.int().numpy().squeeze()

            xp, r, d, truncated, info = trained_env.step(u)
            done = d or truncated
            total_reward += r
            x = xp

    return total_reward / 10

def post_training_evaluation(q = None, num_episodes = 10):
    e = gym.make('CartPole-v0')
    rewards = []
    for _ in range(num_episodes):
        x, _ = e.reset()
        done = False
        gamma = .99
        eps_reward = 0

        while not done:
            u = q.control(th.from_numpy(x).float().unsqueeze(0), eps = 0)
            u = u.int().numpy().squeeze()

            xp, r, d, truncated, info = e.step(u)
            done = d or truncated
            eps_reward += r
            x = xp

        rewards.append(eps_reward)

    return rewards
```

FIGURE 9. Code snippet for the evaluation during and after training.



b)

FIGURE 10. Average total return during training for every 1000 iterations.

c)

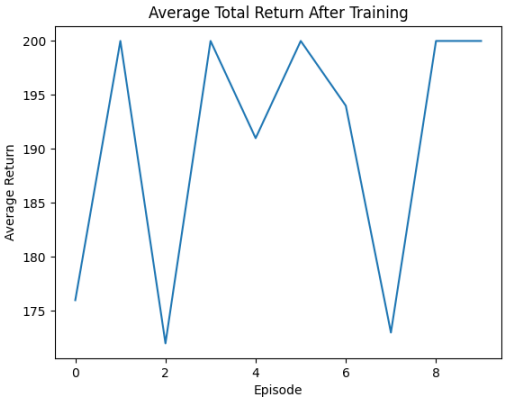


FIGURE 11. Average total return after training for 10 episodes.