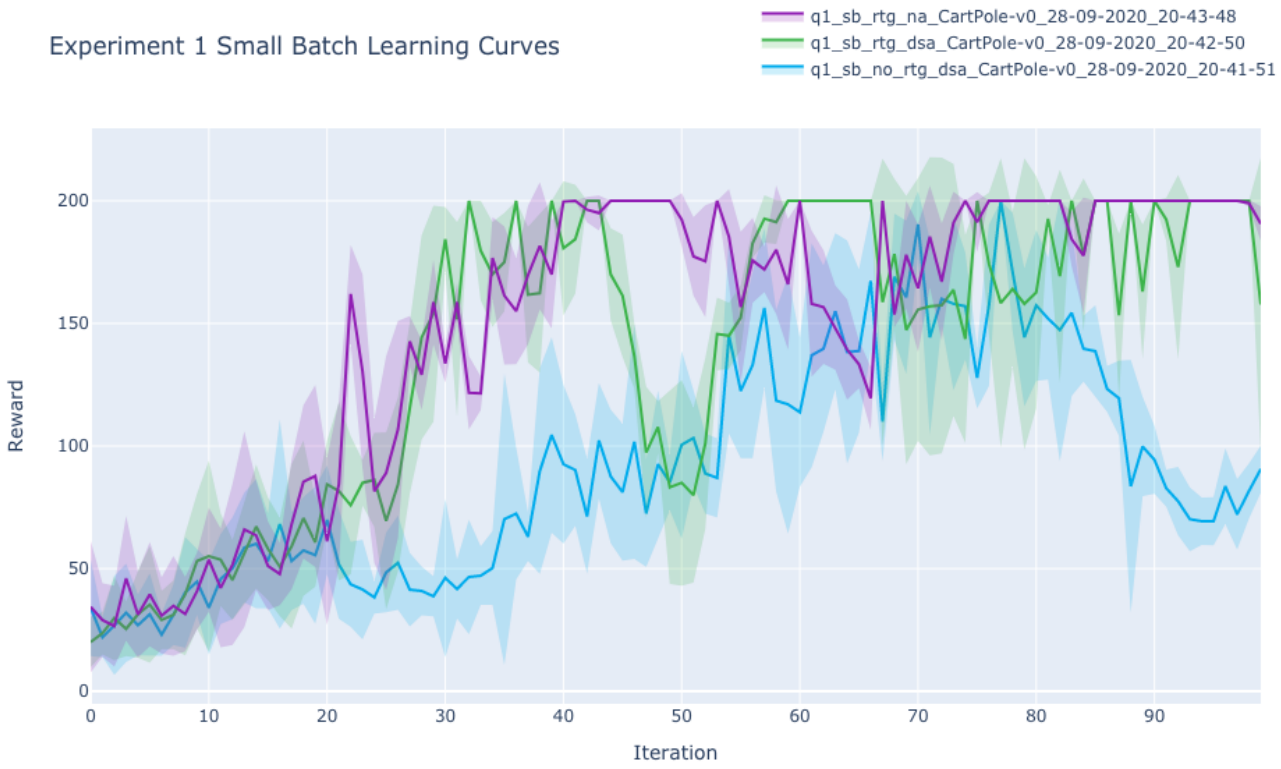
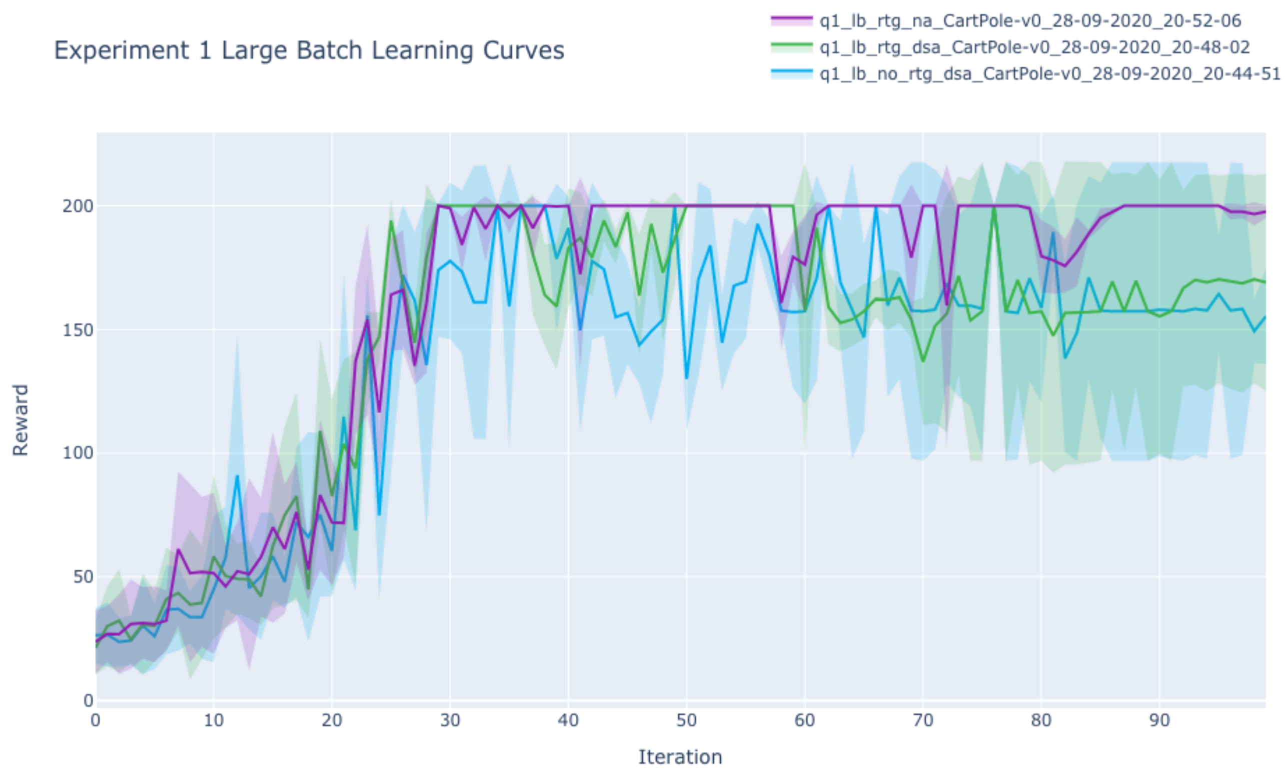
Homework 2

Experiment 1





Which value estimator has better performance without advantage-standardization: the trajectory centric one, or the one using reward-to-go?

In both the large and small batch experiments, the reward-to-go value estimator had better performance than the trajectory-centric estimator. It was less obvious with large batches.

Did advantage standardization help?

Advantage standardization helped make convergence more stable for both batch sizes.

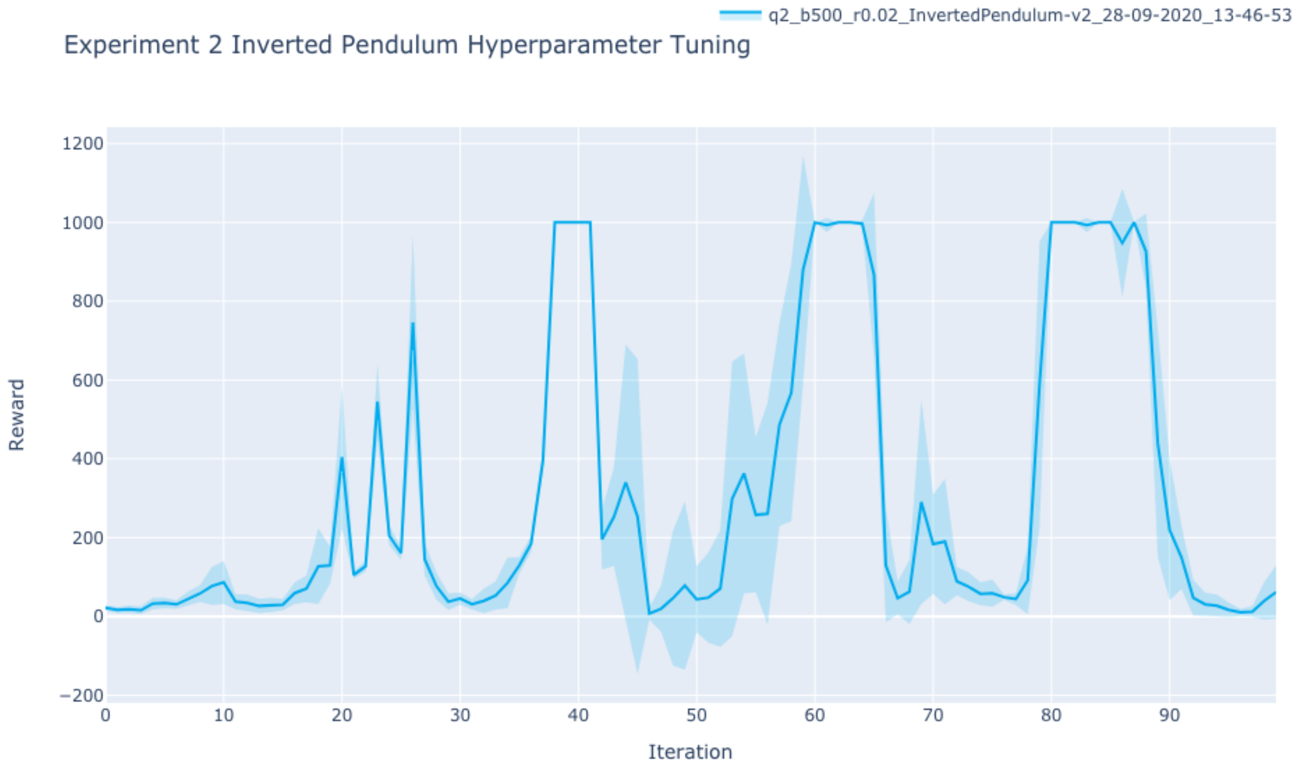
Did the batch size make an impact?

In all cases it made convergence more stable and consistent.

I used the following python script to run my experiments. The parameters were all the default, except for the number of hidden layers, which was 3.

```
import os
import time

t0 = time.time()

os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -dsa --exp_name q1_sb_no_rtg_dsa")
os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -rtg -dsa --exp_name q1_sb_rtg_dsa")
os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -rtg --exp_name q1_sb_rtg_na")
os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 -dsa --exp_name q1_lb_no_rtg_dsa")
os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 -rtg -dsa --exp_name q1_lb_rtg_dsa")
os.system("python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 -rtg --exp_name q1_lb_rtg_na")

t1 = time.time()

print("Total experiment time elapsed: ", t1 - t0)
```

Experiment 2



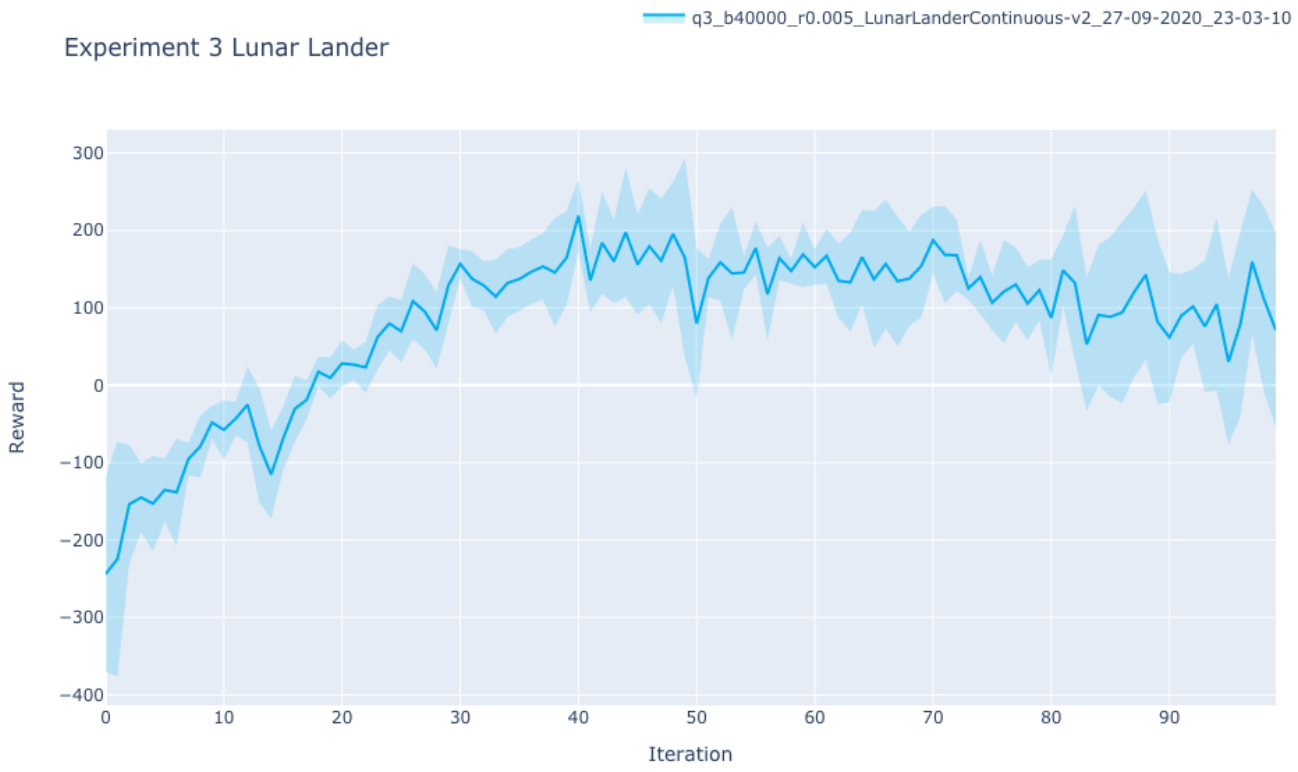
The smallest batch size and largest learning rate that I found which achieved a score of 1000 was b* = 500 and r* = 0.02.

I used the following python script to run my experiments. The only parameter that was different than the default (besides the batch size and learning rate, of course), was the number of hidden layers, which was 3.

```
import os
import time

t0 = time.time()


for batch_size in [500, 1000, 5000, 10000]:
    for lr in [1e-3, 5e-3, 0.01, 0.02]:
        os.system(f"python cs285/scripts/run_hw2.py --env_name InvertedPendulum-v2 --ep_len 1000 --discount 0.9 -n 100 -l 3 -s 64 -b {batch_size} -lr {lr} -rtg --exp_name q2_b{batch_size}_r{lr}")

```

```
t1 = time.time()

print("Total experiment time elapsed: ", t1 - t0)
```

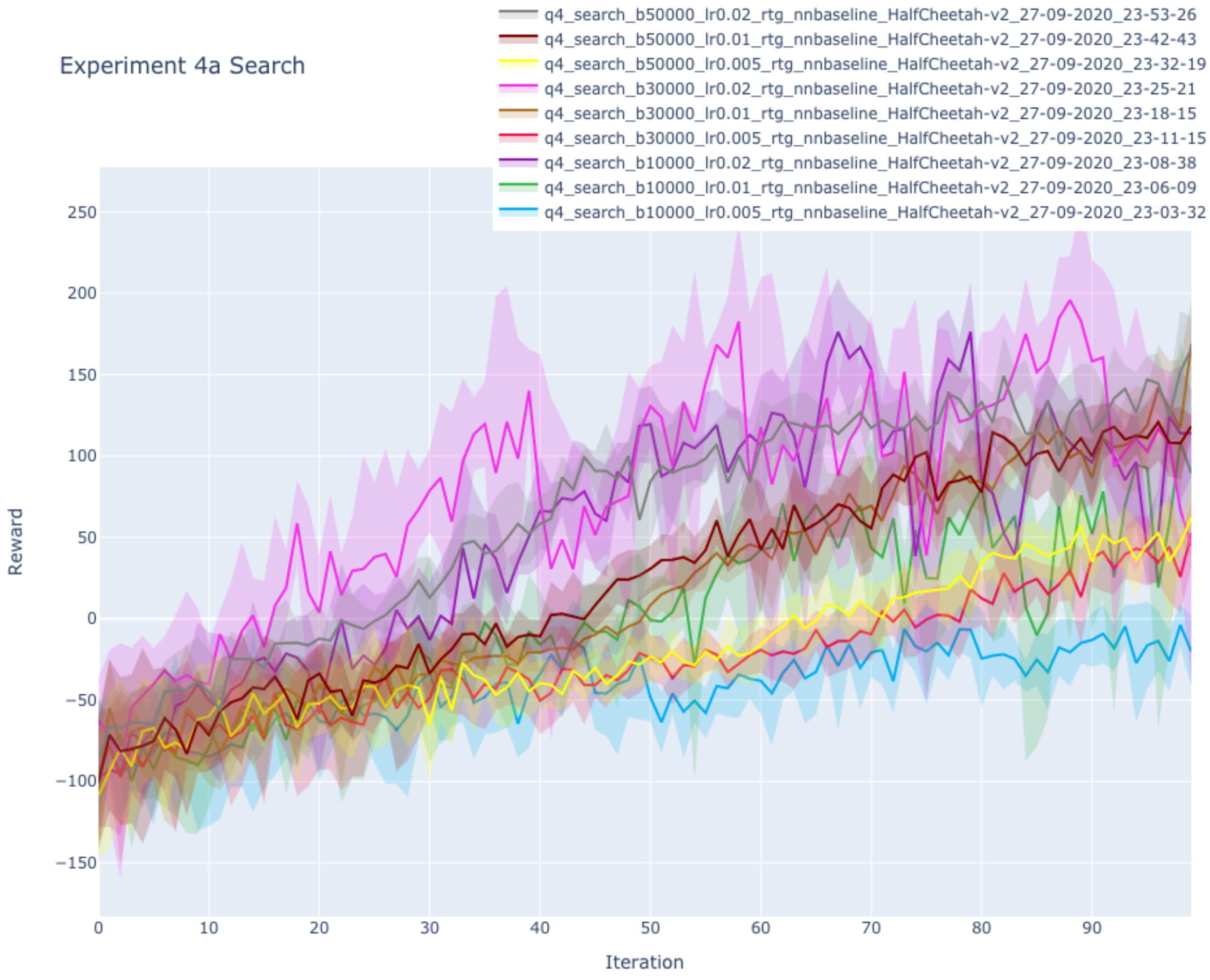
Experiment 3



I used the following command to run this experiment. Note that I had number of hidden layers equal to 3.

```
python cs285/scripts/run_hw2.py --env_name LunarLanderContinuous-v2 --ep_len 1000 --discount 0.99 -n 100 -l 3 -s 64 -b 40000 -lr 0.005 --reward_to_go --nn_baseline --exp_name q3_b40000_r0.005
```

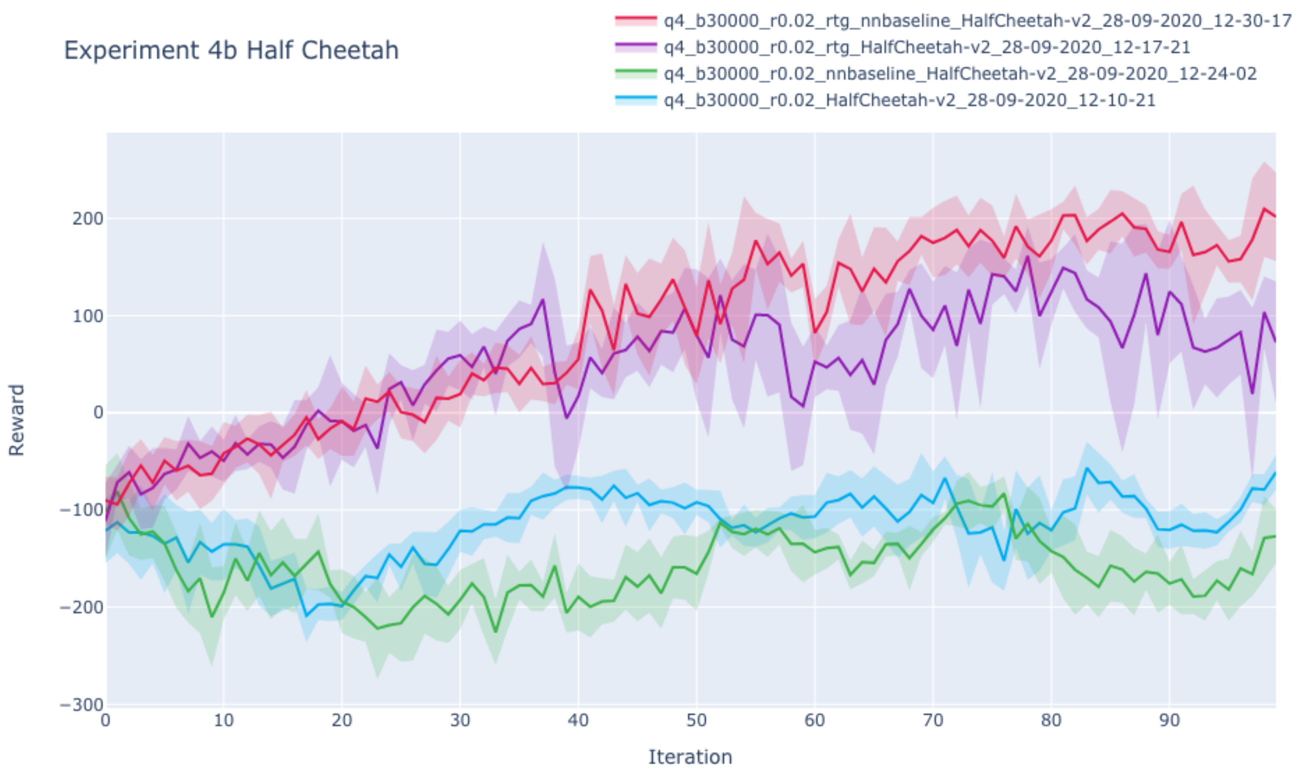
Experiment 4a



Describe in words how the batch size and learning rate affected task performance.
The experiments with learning rate 0.005 barely learned at all, regardless of batch size (some of them never even achieved positive reward). In general a larger batch size improved stability as well as the reward around which the curve stabilized, although it seemed like a high learning rate combined with a medium batch size was stable enough to guarantee learning but unstable enough to reach extreme highs at around ~200. In general a higher learning rate resulted in faster learning but also more rapid degeneration in some cases.

The optimal values b* and r* that I found were b* = 30000 and r* = 0.02 (the high learning rate combined with medium batch size that I mentioned - this is the magenta curve on the graph above).

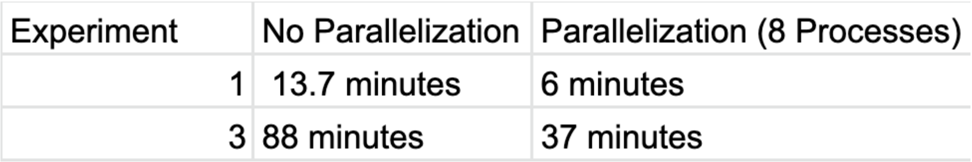
Experiment 4b



Bonus

I chose to speed up sample collection time by parallelizing `sample_trajectories` using `torch.multiprocessing`. I ended up using a Queue along with multiple Processes. Some challenges that I faced were making sure that the sample collection in each process was independent (required setting unique seeds and constructing a new environment), and handling the Queue entering deadlock (required popping from the queue concurrently with processes inserting trajectories into it).

Ultimately it sped up sample collection by more than 2x. I found that with my laptop the optimal number of processes to use was around 8. Here are some comparisons:



Experiment	No Parallelization	Parallelization (8 Processes)
1	13.7 minutes	6 minutes
3	88 minutes	37 minutes

Here is my code if you're curious:

```
def sample_trajectories(env, policy, min_timesteps_per_batch, max_path_length, render=False, render_mode=('rgb_array')):
    """
        Collect rollouts until we have collected min_timesteps_per_batch steps.

        TODO implement this function
        Hint1: use sample_trajectory to get each path (i.e. rollout) that goes into paths
```

```python
        Hint2: use get_pathlength to count the timesteps collected in each path
    """

    def sample_helper(index, pol, env_name, mtpb, paths_queue, timestep_queue):
        np.random.seed(index)
        torch.manual_seed(index)
        env = gym.make(env_name)
        timesteps_this_batch = 0
        paths = []
        while timesteps_this_batch < mtpb:
            new_path = sample_trajectory(env, pol, max_path_length, render, render_mode)

            paths_queue.put(new_path)
            timesteps_this_batch += get_pathlength(new_path)
        timestep_queue.put(timesteps_this_batch)
        paths_queue.put(None)

    num_processes = 1

    paths_queue = mp.Queue()
    timestep_queue = mp.Queue()
    t0 = time.time()

    total_paths = []
    if num_processes < 2:
        sample_helper(0, policy, env.unwrapped.spec.id, min_timesteps_per_batch, paths_queue, timestep_queue)
        num_processes_done = 0

        while num_processes_done < num_processes:
            elem = paths_queue.get()
            if elem is None:
                num_processes_done += 1
            else:
                total_paths.append(elem)
    else:
        policy.share_memory()

        processes = []
        for i in range(num_processes):
            p = mp.Process(target=sample_helper, args=(i, policy, env.unwrapped.spec.id, min_timesteps_per_batch//num_processes, paths_queue, timestep_queue))
            p.start()
            processes.append(p)

        num_processes_done = 0

        while num_processes_done < num_processes:
            elem = paths_queue.get()
            if elem is None:
                num_processes_done += 1
            else:
                total_paths.append(elem)

        for p in processes:
            p.join()

    total_timesteps = 0
    while not timestep_queue.empty():
        total_timesteps += timestep_queue.get()
    t1 = time.time()
    # print("time elapsed: ", t1 - t0)

    return total_paths, total_timesteps
```