CS287-H AHRI HW3

In HW3 we will exploring training agents for human collaboration in the Overcooked environment.

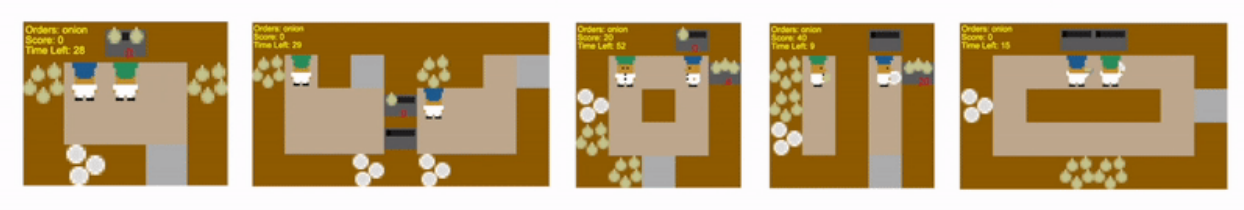
Setup

Unlike the previous homeworks, for this homework you will not be writing much code. Instead, you will run experiments from the command line and report the results in this notebook.

To set up your environment, run `pip install -r requirements.txt`. You should also make sure FFMPEG is installed.

There are five environment layouts built in to Overcooked. Shown below, from left to right, they are `cramped_room`, `asymmetric_advantages`, `coordination_ring`, `forced_coordination`, and `counter_circuit`. You should choose one of these layouts and use it throughout the assignment. Pick whichever one you think poses the most interesting coordination challenges!

Note: `counter_circuit` sometimes requires more RL training than the other layouts. If RL training is not working, you may need to add `--num_training_iters 500` when using the `train_ppo.py` script.



(If you can't see the image, look here (https://github.com/HumanCompatibleAI/overcooked_ai/raw/master/images/layouts.gif).)

Imitation learning

To start, we'll train human models using imitation learning; in particular, we'll use behavior cloning, which is just supervised learning applied to predicting actions from observations.

To train a BC policy you can use a command like the following:

```
python train_bc.py --layout_name cramped_room --data_split train --
num_trajectories 8
```

This will train a BC policy using the default hyperparameters and all 8 training trajectories (there are 8 train and 8 test trajectories for each layout). The final model will be saved to a checkpoint file and the location will be printed on the command line. Try running the above command to train a BC policy on your preferred layout.

Evaluating policies

How well can the BC policy we trained play Overcooked? We can use another script to evaluate how well this policy works when it plays with itself. Try running a command like the one below to evaluate the BC policy (replacing `PATH/TO/BC/checkpoint-20` with where your checkpoint was saved):

```
python evaluate.py --run_0 bc --checkpoint_path_0 data/logs/bc/cram
ped_room/train/8_traj/2023-04-23_15-02-03/checkpoint_000020/checkpo
int-20 --run_1 bc --checkpoint_path_1 data/logs/bc/cramped_room/tra
in/8_traj/2023-04-23_15-02-03/checkpoint_000020/checkpoint-20 --lay
out_name cramped_room --num_games 10 --output_path data/bc_bc_eval.
json
```

We'll use this command throughout the homework to evaluate how well two policies play with each other. For other parts of the homework, you can replace the `--checkpoint_path_{0,1}` arguments with paths to different checkpoints, and you can replace `--run_{0,1}` with the algorithms those checkpoints were trained with. The script will output results as JSON to the specificed `--output_path`. The most important results are `ep_returns_all`, which is a list of the scores for each game played, and `mean_return_all`, which is the mean of those scores.

Another option you can pass to `evaluate.py` is `--render_path data/videos`. This will render the evaluation games as videos and put them in the specified directory. Try running the evaluate script again to render a video of the BC policy playing with itself (you might want to specify a lower number for `--num_games`).

Human proxy

A more realistic way to evaluate the BC policy is not to play it with itself, but with another "human proxy" policy trained on the test human data. This way we can see if it has learned a strategy that generalizes to policies trained on other data.

To train a human proxy, run `train_bc.py` again with `--data_split test`. Make sure to note the final checkpoint, since we'll use this proxy to evaluate policies throughout the homework!

Try evaluating the BC agent with the human proxy. Write the command you ran below.

Command to evaluate the BC agent with the human proxy: `python evaluate.py --run_0 bc --checkpoint_path_0 data/logs/bc/cramped_room/train/8_traj/2023-04-23_15-02-03/checkpoint_000020/checkpoint-20 --run_1 bc --checkpoint_path_1 data/logs/bc/cramped_room/test/8_traj/2023-04-23_15-11-30/checkpoint_000020/checkpoint-20 --layout_name cramped_room --num_games 10 --output_path data/bc_traj8_human_eval.json`

Effect of training set size on BC collaborative performance

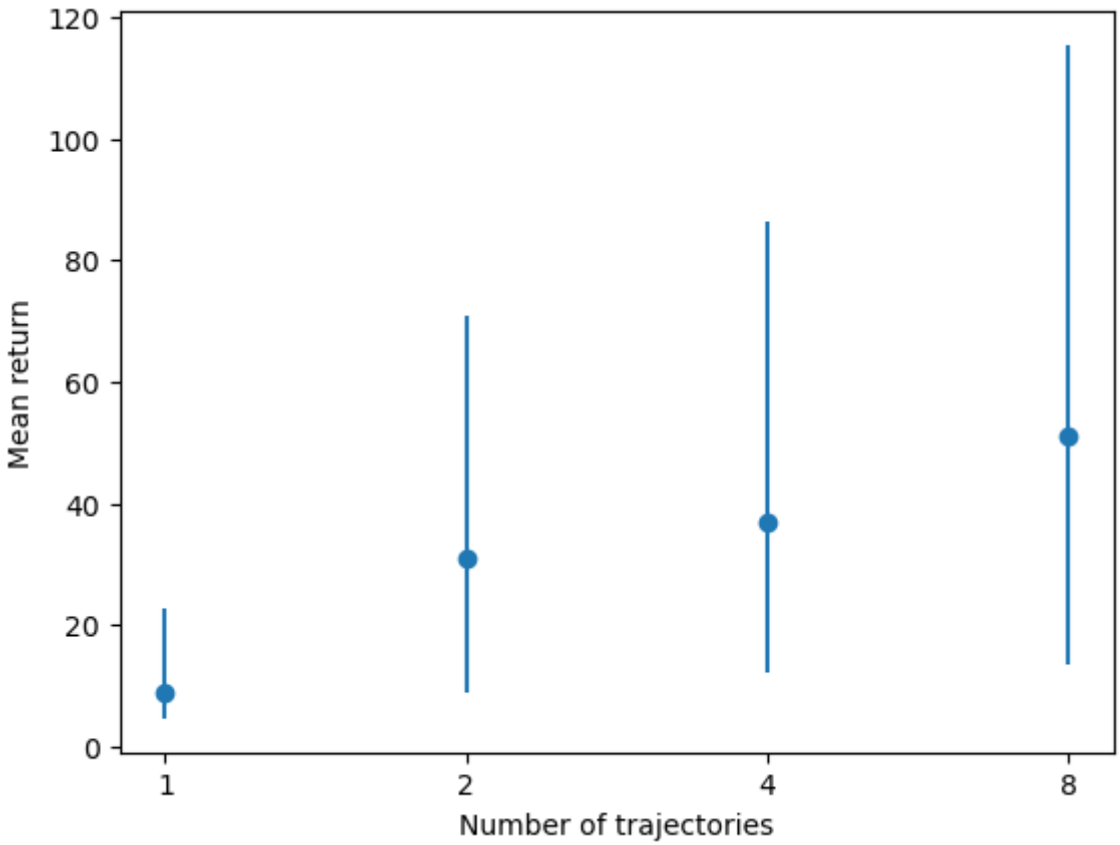
Before we try to use reinforcement learning, let's explore using BC to train collaborative policies. In particular, let's investigate the effect of the amount of training data on the resulting collaborative performance.

Train BC policies on the train set with 1, 2, 4, and 8 trajectories and then test each of them with the human proxy. Create a plot below showing the mean score achieved as a function of the number of training trajectories.

Important: throughout the homework, include error bars in your plots. In particular, you should use 90% t-confidence intervals. You may find scipy.stats.t.interval (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html) useful for calculating confidence intervals. Make sure you evaluate over enough episodes so that your confidence intervals are small.

In [17]:

```
import json
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import t

file_paths = ["data/bc_traj1_human_eval.json", "data/bc_traj2_human_eval.js

# Load the data from each file
all_ep_returns = []
all_mean_returns = []
for file_path in file_paths:
    with open(file_path, "r") as f:
        data = json.load(f)
        all_ep_returns.append(data["ep_returns_all"])
        all_mean_returns.append(data["mean_return_all"])

# Compute the mean and confidence intervals for each number of trajectories
ci_lower = []
ci_upper = []

for i in range(len(all_ep_returns)):
    mean = all_mean_returns[i]
    n = len(all_ep_returns[i])
    std_err = np.std(all_ep_returns[i], ddof=1) / np.sqrt(n)
    ci_lower_i, ci_upper_i = t.interval(0.9, n-1, loc=mean, scale=std_err)
    ci_lower.append(ci_lower_i)
    ci_upper.append(ci_upper_i)

# Plot the mean returns with confidence intervals
plt.errorbar(['1', '2', '4', '8'], all_mean_returns, yerr=[ci_lower, ci_upp
plt.xlabel("Number of trajectories")
plt.ylabel("Mean return")
plt.show()
```



Self-play

The next method we'll try for training collaborative policies is self-play. This method uses multi-agent RL—in particular, proximal policy optimization (PPO)—to learn a single policy that controls both agents. To train a self-play policy, use a command like the following:

```
python train_ppo.py --layout_name LAYOUT_NAME
```

Just like with BC, the script will save the final checkpoint in the `data/logs` directory. You can evaluate policies trained with self-play by passing `--run_{0,1} ppo` to `evaluate.py`.

Try training a self-play policy for your chosen layout.

Here are a couple tips for running RL training:

- It will take a while, so be patient, especially if you don't have a GPU.
- If you want to see how the RL training is progressing, you can use TensorBoard. Run `pip install tensorboard` to install and then `tensorboard --logdir data/logs`. This will give you a URL you can use to monitor the training process. Look at the `ray/tune/custom_metrics/sparse_reward_mean` tag to see what average score your RL policy is getting throughout training.
- You can pass in the `--experiment_tag` option to specify any details about the specific training run you want to remember later. The experiment tag will be included in the log directory and show up in TensorBoard.

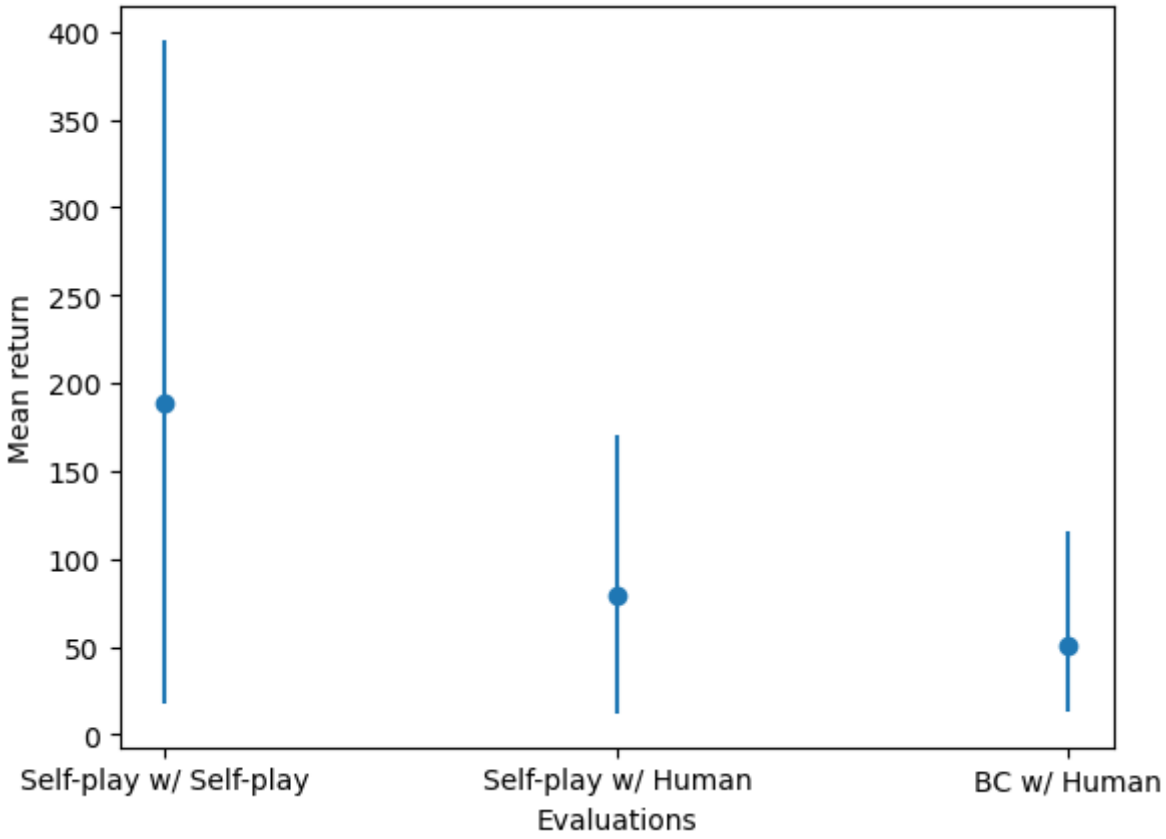
Comparing BC and self-play

Now that we have a self-play policy trained, let's see how it performs as a collaborator! Try running the following three evaluations:

- The self-play policy with itself.
- The self-play policy with the human proxy.
- The BC policy (trained on 8 trajectories) with the human proxy.

Make a bar chart below showing the mean score with error bars for each of the three evaluations. Also, record a few videos of each evaluation and write below what you observe qualitatively. How does the self-play policy's strategy differ from the BC policy's? How well or poorly does the self-play policy perform with the human proxy? Why do you think it performs the way it does?

In [15]:
```
file_paths = ["data/self_play_self_play_eval.json", "data/self_play_human_e

# Load the data from each file
all_ep_returns = []
all_mean_returns = []
for file_path in file_paths:
    with open(file_path, "r") as f:
        data = json.load(f)
        all_ep_returns.append(data["ep_returns_all"])
        all_mean_returns.append(data["mean_return_all"])

# Compute the mean and confidence intervals for each number of trajectories
ci_lower = []
ci_upper = []

for i in range(len(all_ep_returns)):
    mean = all_mean_returns[i]
    n = len(all_ep_returns[i])
    std_err = np.std(all_ep_returns[i], ddof=1) / np.sqrt(n)
    ci_lower_i, ci_upper_i = t.interval(0.9, n-1, loc=mean, scale=std_err)
    ci_lower.append(ci_lower_i)
    ci_upper.append(ci_upper_i)

# Plot the mean returns with confidence intervals
plt.errorbar(['Self-play w/ Self-play', 'Self-play w/ Human', 'BC w/ Human'
plt.xlabel("Evaluations")
plt.ylabel("Mean return")
plt.show()
```



In the self play with self play setup, most of the time, the two agents collaborated well and get the highest score among the three cases. In the self play with simulated setup, the simulated human agent couldn't catch up the pace of the self play agent and the collaboration wasn't great. In the BC with simultaed setup, the performance was enough worse because the two agents block each other's way frequently which prevent them from getting higher scores. The self-play policy optimizes the two self-player team to get the best performance, the BC policy, however, attempts to clone human player's behavior. We are getting the result above because self-play is optimized in a two-player setting, switching one of the self-play players to human will hurt the performance, BC is a imitating human player's behavior but not really optimizing the team play case.

Human-aware RL

As you probably saw in the previous part, the self-play policy didn't collaborate very well with the human proxy. We will now try using human-aware RL (HARL, Carroll et al., 2019) to train a better collaborative policy.

As we learned in class, human-aware RL trains an RL policy with a fixed human model learned via behavior cloning. To train a policy using human-aware RL, run a command like this:

Like before, you should replace `LAYOUT_NAME` with your chosen layout and `PATH/TO/BC/checkpoint-20` with the BC checkpoint you want to use as your human model.

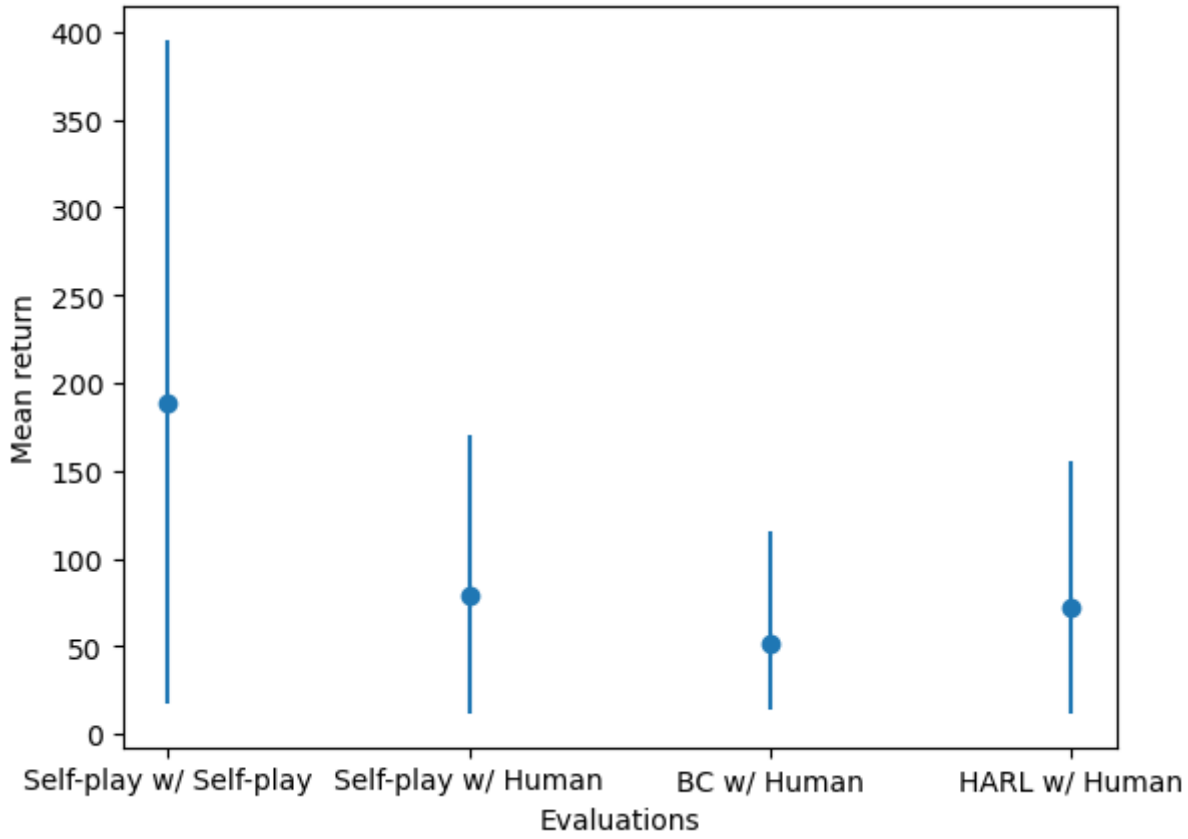
Try training a HARL policy with the BC policy trained on the full dataset.

Comparing HARL to self-play

In the Carroll et al. paper, the authors hypothesize that a HARL-trained policy will collaborate better with a human proxy than one trained with self play. To test that finding, evaluate your HARL policy with your human proxy. Below, create a bar chart showing the three evaluations from the BC vs. self-play experiments along with this new evaluation.

Also, generate some videos of the HARL policy with the human proxy. Below the chart, describe how the collaborative behavior exhibited by the HARL policy differs from the self-play policy. Do the policies do what you would expect qualitatively? How do their strategies differ?

In [16]:
```
elf_play_self_play_eval.json", "data/self_play_human_eval.json", "data/bc_t

each file


_paths:
th, "r") as f:
oad(f)
s.append(data["ep_returns_all"])
rns.append(data["mean_return_all"])

d confidence intervals for each number of trajectories



l_ep_returns)):
eturns[i]
turns[i])
all_ep_returns[i], ddof=1) / np.sqrt(n)
per_i = t.interval(0.9, n-1, loc=mean, scale=std_err)
i_lower_i)
i_upper_i)

ns with confidence intervals
lay w/ Self-play', 'Self-play w/ Human', 'BC w/ Human', 'HARL w/ Human'], a
ns")
rn")
```



The performance of the HARL policy should have been better than the self-play policy, but my results show that these two performances are similar. The HARL and self-policy strategies are different as the HARL policy takes the human player's actions into consideration during training so there were fewer cases of collision and blocking each other's way in the HARL with human evaluation compared to the Self-play with human evaluation. The self-play optimizes a game played with another self-play agent, but the HARL policy optimizes a game playerd with a simulated human player.

How much human data is needed?

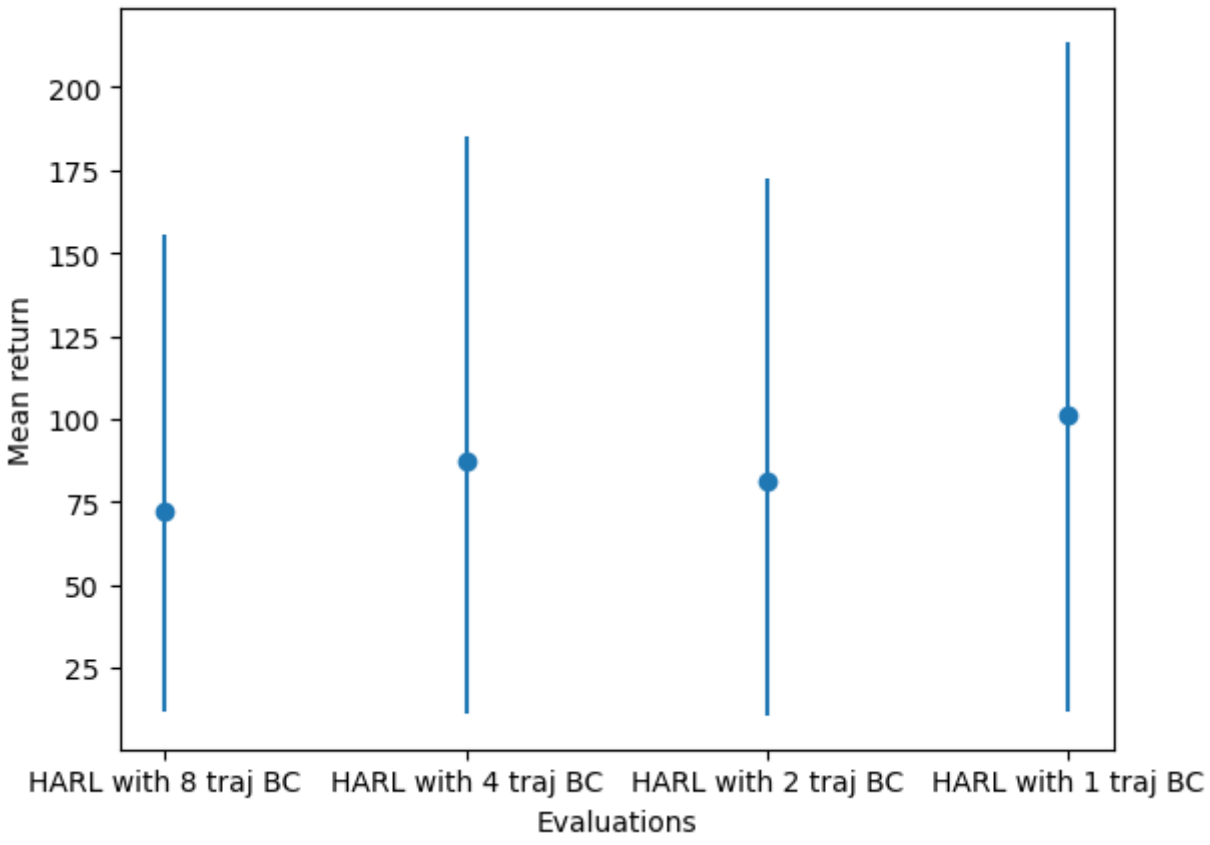
As you have probably seen, using human-aware RL to train collaborative policies works much better than self-play. However, it comes at a cost: we need to gather data of human-human play to train the BC policy. In this section, we'll explore the question of *how much* human data is needed for HARL.

Train additional HARL policies using the BC policies trained on 1, 2, and 4 trajectories. Evaluate each with the human proxy and create a plot below showing the mean score vs. the number of trajectories used to train the BC policy.

What do you notice about the effect the amount of human data has on HARL? Why do you think this might be the case? Write your thoughts below the plot.

In [13]:

```
traj_human_eval.json", "data/harl_4traj_human_eval.json", "data/harl_2traj_h

file


s:
r") as f:
)
end(data["ep_returns_all"])
ppend(data["mean_return_all"])

fidence intervals for each number of trajectories



returns)):
s[i]
[i])
p_returns[i], ddof=1) / np.sqrt(n)
 = t.interval(0.9, n-1, loc=mean, scale=std_err)
er_i)
er_i)

th confidence intervals
 traj BC', 'HARL with 4 traj BC', 'HARL with 2 traj BC', 'HARL with 1 traj B
```



It appears that the mean return varies with different number of trajectories for behavioral cloning. HARL 1 with 1 trajectory and 4 trajectories turn out to be the highest mean return. This may be the case because HARL may require more tuning to perform well in some cases and it doesn't have a very clear trend for different number of trajectories.

Effects of feature engineering

Besides collecting enough human data, another important ingredient in making BC work well is feature engineering. In the HARL paper, the authors create their own featurization of the environment state specifically for doing behavior cloning. That is, the features used for BC are different than the ones used for RL.

How important are these engineered features? To train a BC policy with the raw environment features, add the `--use_raw_features` flag, e.g.,

```
python train_bc.py --layout_name LAYOUT_NAME --data_split train --n
um_trajectories 8 --use_raw_features
```

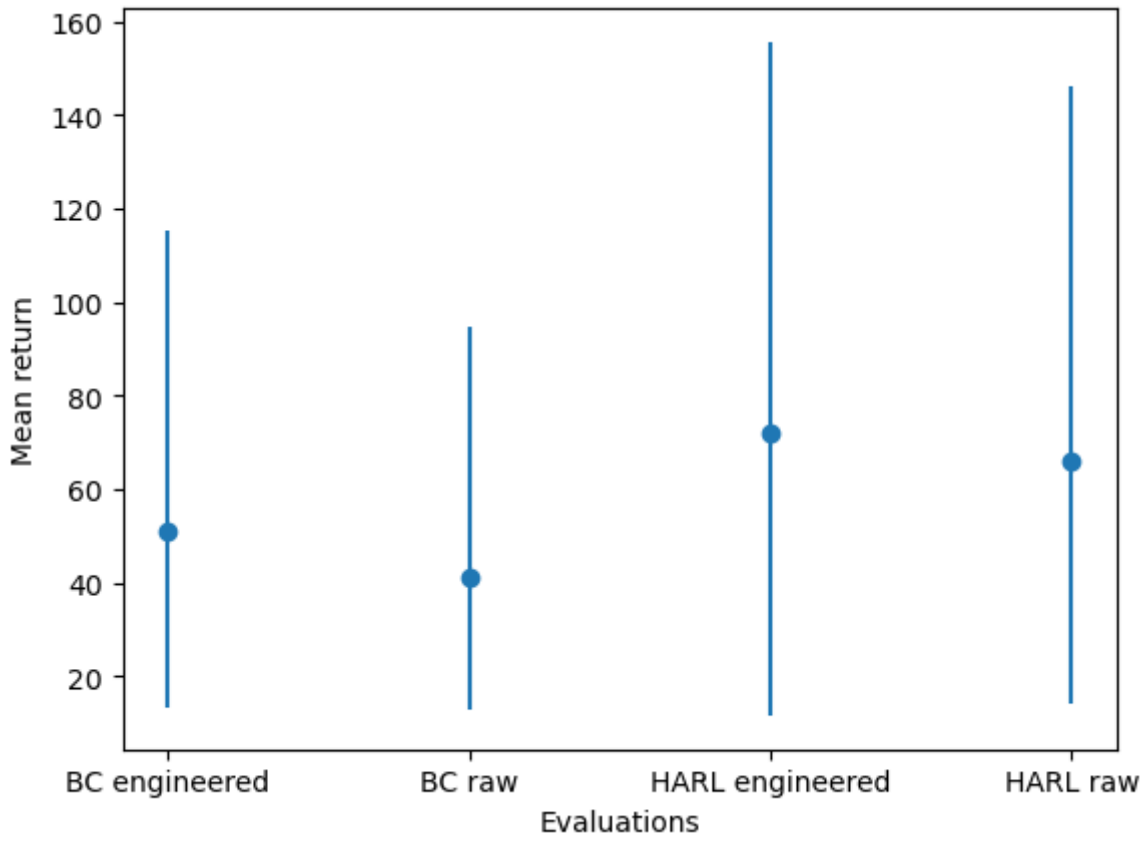
Try training a BC policy on your chosen layout with the raw features and then train a HARL policy based on the BC policy. Then, make a bar chart comparing the mean return with the human proxy for the following four policies:

- BC trained with engineered features
- BC trained with raw features
- HARL based on BC trained with engineered features
- HARL based on BC trained with raw features

Don't forget to include error bars. How important for the performance of HARL is it to use the engineered features? Why do you think this might be? If it helps, you may want to render videos of some of the evaluation games to see how the different policies behave.

In [18]:

```
_human_eval.json", "data/bc_raw_human_eval.json", "data/harl_8traj_human_eva

.e



 as f:

l(data["ep_returns_all"])
nd(data["mean_return_all"])

lence intervals for each number of trajectories



urns)):
.]
)
eturns[i], ddof=1) / np.sqrt(n)
t.interval(0.9, n-1, loc=mean, scale=std_err)
i)
i)

confidence intervals
, 'BC raw', 'HARL engineered', 'HARL raw'], all_mean_returns, yerr=[ci_lowe
```



For both BC and HARL, the policies with engineered features performed better than the policies with raw features. I think this might be because the engineered features of the environment state were pre-selected to optimize performance when doing behavior cloning. I think the authors may have intentionally removed parts of the raw environmental features related to the collisions of agents due to behavioral cloning.

In [ ]: