CSCI 699: Robot Learning
Problem Set #3: Due Sun, Nov 3, 11:59PM

Introduction

The coding portion of the homework assignment will be completed using Google Colab. Starter code for this problem set can be downloaded from https://github.com/USC-Lira/CSCI699_RobotLearning_HW3.git. The notebook should contain the code for installing all the necessary Python dependencies. This assignment is based on homework material from Stanford CS237B.

Submission instructions:

You will submit your homework to Gradescope. Your submission will consist of (1) a single pdf with your answers for the short answer questions (✍) and (2) a zip folder containing your code for the programming questions (💻). Also include the `policies` folder in your submission. Use the `make_submission.sh` script to create a zip of all the files.

Your answers to the written portion must be typeset with a word processor or LaTeX.

Problem 1: Behavior Cloning [40 points]

In this homework, you will work in the driving setting. Particularly, you are going to implement various imitation learning models, and then use them for intent inference and shared autonomy. Being interested in directly learning policies as opposed to learning reward functions first, we model the driving environment as a partially observable Markov decision process (POMDP): $\langle \mathcal{S}, \mathcal{O}, \Omega, \mathcal{A}, f \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{O}$ the set of observations, Ω is a probability distribution over the set of observations given the state, i.e. $\Omega(o|s)$ gives the probability of observing $o \in \mathcal{O}$ while the system state is $s \in \mathcal{S}$. For example, in an environment with multiple vehicles, $s \in \mathcal{S}$ will contain the states (positions, velocities, etc) of all the vehicles, whereas $o \in \mathcal{O}$ might be missing the information about the vehicles that are occluded behind buildings. Or in a single-vehicle scenario, $o \in \mathcal{O}$ might have only noisy measurements of the states. $\mathcal{A}$ denotes the set of actions. For driving, we assume the actions are two-dimensional: steering angle and throttle. While braking could be encoded as a different action dimension, we assume negative throttle will correspond to braking. Finally, f gives the transition probabilities. That is, $f(s' \mid s, a)$ is the probability of reaching state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ by taking action $a \in \mathcal{A}$.

As a driving platform, many open-source simulators exist, such as CARLA [1], which are based on game engines to provide realistic dynamics and visuals. However, setting up CARLA and training driving policies on raw camera input (or Lidar data) require both a powerful computing infrastructure and a lot of effort. You will instead use CARLO (short for CARLA - Low Budget), which is a very simplistic 2D driving simulator [2] that uses the bicycle model to simulate vehicles [3].

(a) Intersection Scenario

(b) Lanechange Scenario

Figure 1: Two CARLO scenarios you are going to use in this homework are visualized. In the intersection scenario, the goal is to stay on the right lane and turn left, go straight or turn right without having a collision with the pedestrians standing in the two corners of the intersection and without crashing into the buildings. Every time the scenario is initiated, the intersection is in a different location. The agent is allowed to observe: the car's position, speed, heading angle and the location of the intersection. In the lanechange scenario, the goal is to follow any lane without going off the road. The agent is allowed to observe: the car's position, speed and heading angle.

Feel free to interactively play with CARLO in some scenarios we provide by running

```
python play.py --scenario intersection
```

where you can replace scenario name as "intersection" or "lanechange". These scenarios are also visualized in Fig. 1. In the code, you can see the use of `step` function, which transitions the environment from its current state to the next state given the action and returns the new observation. Mathematically, it first draws a sample from $f(\cdot|s, a)$ where $s \in \mathcal{S}$ is the current state, and then returns a sample from the observation distribution Ω conditioned on the new state.

Behavior cloning is the simplest imitation learning method that dates back to 1989 when ALVINN was proposed as a method for autonomous driving [4]. To understand how behavior cloning works, let's first define an expert policy in the POMDP we defined in Problem 1: Let π^* denote an expert policy such that $\pi^*(a|o, g)$ is the probability of taking action $a \in \mathcal{A}$ when the agent observes $o \in \mathcal{O}$ and the goal is $g \in \mathcal{G}$. In driving, $\mathcal{G}$ may consist of possible destinations. For example in the intersection scenario (see Fig. 1a), $\mathcal{G}$ might be "left", "straight", "right", and in the lanechange scenario (see Fig. 1b) it might be "left", "right", which represent the lane the expert wants to follow. Throughout the homework, we will assume $\mathcal{G}$ is a discrete set.

Let's assume we have access to a data set for each $g \in \mathcal{G}$, that consists of (o, a) pairs, where each pair is obtained such that a is a sample drawn from $\pi^*(\cdot \mid o, g)$. Let's denote each such data set $\mathcal{D}_g = \{(o^{(i)}, a^{(i)})\}_{i=1}^N$ where N is the number of data samples[1]. Our goal is then to recover π^* using these data sets.

[1] N might be different for each scenario and goal, but let's ignore this for notation simplicity.

For this homework, we collected hundreds of expert demonstrations in the two scenarios with all the different goals. These data sets are in the `data` folder[2]. Now, you are going to implement behavior cloning to learn a policy that imitates the expert policy.

i) ✍ **[4 points]** The most straightforward way to perform behavior cloning is to assume there exists an underlying *deterministic* policy, but the data set contains some noise in the actions. Then, you can simply try to learn a function $h(o, g)$ that outputs the estimated expert action. Modeling this function as a neural network, let's write it as $h_\theta(o, g)$ where θ denotes the network parameters (weights and biases). Assuming a loss function L between two actions, write an optimization problem to learn the expert policy by optimizing θ.

ii) 💻 **[4 points]** Fill in the missing parts of `train_il.py` to solve the optimization problem you defined in the previous question. You can use any loss function that you think is reasonable. You can also design your neural network structure however you wish[3]. In our experience, small networks were able to efficiently recover the expert policy. Be careful about what your neural network is / should be able to output.

After filling in the training code, run

```
python train_il.py --scenario intersection --goal left
--epochs <number_of_epochs> --lr <learning_rate>
```

to train a policy for the intersection scenario for the goal of turning left. You should set the number of epochs and the learning rate for the Adam optimizer. In our experience, the training should take around 5 minutes. Once the number of epochs is completed, the code is going to save the policy in the policies folder. If you want to continue to train from the latest saved policy, simply add `--restore` in the command above. After training, test your policy by running:

```
python test_il.py --scenario intersection --goal left --visualize
```

You should be able to see at least a few successful left-turns out of 10 episodes. If you are not satisfied with the result, you may want to play with the learning rate, number of epochs, and most importantly the loss function. For example: You may want to change how you weigh different terms of the actions in the loss function. If your trained policy is not able to achieve proper steering, for example, then perhaps you should penalize the differences in the steering dimension more heavily.

Repeat what you did in this question for the "straight" and "right" goals by changing the `--goal` argument. Make sure to use the same neural network structure for all three goals and obtain reasonably well-performing policies[4].

iii) ✍ **[4 points]** Describe the full training procedure you used for each goal (learning rate and the number of epochs; also elaborate if you retrained the network using `--restore` and/or if you used different loss functions for different goals).

iv) ✍ **[4 points]** Run

```
python test_il.py --scenario intersection --goal left
```

to test the policy without visualization and report the success rate you achieved (it will take a few seconds and be printed on the terminal). Repeat this for all three goals.

v) ✍ **[4 points]** What loss function did you use? Write it mathematically. If you perfectly overfit the data, what would be the minimum loss you see? Is it bounded below?

So far, you trained a neural network, for each goal, that outputs an action given an observation. It is a deterministic function, which means you will get the exact same action if you feed the same observation. While this is adequate in many situations, some applications require learning the distribution $\pi^*(\cdot|o, g)$. For example, imagine you want the robot to predict what the human might do and then you are going to optimize for the worst case. In such a setting, it is not enough to predict the expected human action, but the distribution should be learned. To be able to learn the distribution, we are going to use a simple version of Mixture Density Networks.

In this approach, the neural network outputs a distribution instead of an action sample. To be more precise, the network is going to output the parameters of a distribution. In this homework, you are going to use a Gaussian distribution, so the network should output two quantities: mean vector and covariance matrix.

vi) ✍ **[4 points]** You could further simplify the problem by assuming the action dimensions (steering and throttle) are conditionally independent from each other, given the observation. Then you would need to learn four scalars: $\mu_{steering}, \mu_{throttle}, \sigma^2_{steering}, \sigma^2_{throttle}$. The only constraint would be to make sure the last two quantities are non-negative. This could be achieved by simply using a ReLU only

[2] Each data set is given as a matrix. Each row is a time step in the POMDP. The last two columns are the actions that the expert took when the observation is the remaining columns.

[3] Again, make sure to use the same neural network structure for different goals

[4] While you are allowed to change the loss function for different goals, this is not necessary in our experience.

for those two nodes. However, this approach is too restrictive —we cannot just assume steering and throttle are independent!

You will instead learn the entries of the mean vector and the covariance matrix. However, the covariance matrix needs to be a positive semi-definite (PSD) matrix! How can you ensure this? **HINT:** $AA^\top$ is always PSD for any matrix A.

vii) ✍️ **[4 points]** To train the network that outputs a single bivariate Gaussian distribution for a given observation, you are going to maximize the likelihood of the data set. Let's denote the probability density of drawing x from a bivariate Gaussian distribution with mean μ and covariance matrix Σ as $\mathcal{N}(x \mid \mu, \Sigma)$. Let's also denote the mean vector and the covariance matrix produced by the neural network for observation $o^{(i)}$ as $\mu^{(i)}$ and $\Sigma^{(i)}$, respectively, i.e. $h_\theta(o^{(i)}) = (\mu^{(i)}, \Sigma^{(i)})$. Then, we can write the likelihood of an entire data set as

$$\text{Likelihood} = \prod_{i=1}^{N} \mathcal{N}(a^{(i)} \mid \boldsymbol{\mu}^{(i)}, \Sigma^{(i)})$$

As this might be too small and cause numerical issues, you are going to use mean log-likelihood instead:

$$\text{Mean Log-Likelihood} = \frac{1}{N} \sum_{i=1}^{N} \log \mathcal{N}(a^{(i)} \mid \boldsymbol{\mu}^{(i)}, \Sigma^{(i)})$$

Since you will be maximizing this quantity, you can think of the negative mean log-likelihood as the loss function. Then, if you perfectly overfit the data, what would be the minimum loss you see? Is it bounded below?

viii) 💻 **[4 points]** Fill in the missing parts of `train_ildist.py` to solve the optimization problem described in the previous question. You can design your neural network structure however you wish . Repeat everything you did in question (ii) of this problem for `train_ildist.py`. Training this neural network might be much harder than the previous ones because of instabilities. While it is possible to train each network in 15 minutes, hyperparameter tuning might make it take much longer. Consider adopting the following hints to minimize the time you spend for training:

- Use really small (narrow and shallow) networks.
- Start training with some learning rate for a relatively small number of epochs, and then retrain the network more by steadily decreasing the learning rate (and using `--restore`).
- If you don't see a monotonically decreasing loss in the very beginning, then terminate the execution and re-run, because initialization really matters. If you cannot avoid this situation, then decrease your learning rate further and try again.

ix) ✍️ **[4 points]** Describe the full training procedure you used to train the mixture density network for each goal (learning rate and the number of epochs; also elaborate if you retrained the network using –restore and/or if you used different loss functions for different goals).

x) ✍️**[4 points]** Run

```
python test_ildist.py --scenario intersection --goal left
```

to test the policy without visualization and report the success rate you achieved (it will take a few seconds and be printed on the terminal). Repeat this for all three goals.

Problem 2: Conditional Imitation Learning [30 points]

1. ✍️ Using the same code you wrote in the previous question, run

   ```
   python train_il.py --scenario intersection --goal all --epochs <number_of_epochs>
   --lr <learning_rate>
   ```

   to train a policy for the intersection scenario collectively for all goals. And check the result by running

   ```
   python test_il.py --scenario intersection --goal all --visualize
   ```

   What are some problems associated with this policy? Do you have control over which direction the car is going?

Imagine you are in your autonomous vehicle. And every time it encounters an intersection, the car just takes the same direction. Obviously, this is not desirable. Of course, one way to solve this problem is to allow the user to select which goal to pursue and then use the corresponding policy trained with that specific goal. However, data is a very expensive resource in many applications, so we don't want to waste it by using it only for one specific goal. Furthermore, training different neural networks for each goal might take too much time. Conditional Imitation Learning (CoIL) has been proposed as a solution to this problem [5]. Take a look at their Figure 3.

In their first approach (Fig. 3a of their paper), they feed the goal of the user (the high-level command) as an input to the neural network. The output is then the action for this specific goal. In the second approach (Fig. 3b), they use *branching*: The observations are fed into the first layers of the network. Depending on the user-specified goal, the outputs of these first layers are fed into different last layers. They showed the second approach works better. Therefore, you are going to implement the second approach.

i) 💻 **[10 points]** Fill in the missing parts of `train_coil.py` to implement the second approach discussed above. Note that you want to learn a mapping between observations and actions, so the network will not output a distribution. You are again free to design your loss function, which may or may not be different from the previous one. For training, run

   ```
   python train_coil.py --scenario intersection --epochs <number_of_epochs>
   --lr <learning_rate>
   ```

   to train a CoIL policy for the intersection scenario. You can again use `--restore` if needed.

ii) ✍️ **[10 points]** Test your trained CoIL policy for 10 episodes by running

   ```
   python test_coil.py --scenario intersection --visualize
   ```

   Don't forget to provide high-level commands using the arrow keys in your keyboard! After you are confident that the policy you trained achieves to reach the goals reasonably often (at least a few times), run

   ```
   python test_coil.py --scenario intersection --goal left
   ```

   and report the success rate. Repeat this last step for every goal.

iii) ✍️ **[10 points]** Describe the full training procedure you used (learning rate and the number of epochs; also elaborate if you retrained the network using `--restore`).

Problem 3: Intent Inference & Shared Autonomy [30 points]

Referring back to Problem 1, it is often very useful to learn the expert's policy as a distribution. One such case is when we want to perform intent inference. In this problem, we are interested in predicting what goal the user has based on their actions.

i) ✍ **[4 points]** In Problem 1 of this homework, you already implemented `train_ildist.py` which trains a neural network that you can use to compute $P(a \mid o, g)$. Now, we are interested in computing $P(g \mid o, a)$. Write this in terms of $P(a \mid o, g)$ (and some other probability expressions). Assuming a uniform prior over the goals (for $P(g \mid o)$), how can you compute $P(g \mid o, a)$?

ii) 💻 **[5 points]** Fill in the missing parts of `intent_inference.py`. And run

```
python intent_inference.py --scenario intersection
```

Drive the car using the arrow keys.

iii) ✍ **[4 points]** Running the above command, drive the car in each direction for 5 times. Out of the 15 episodes you drove, how many of them ended up predicting the correct intent? What might be some reasons why it fails? Discuss.

To give another example of when having a distribution over actions is useful, let's consider a scenario when the robot knows the optimal actions given the user goal. We are going to consider a setting where the user is driving the car, but the robot also provides help. Mathematically, the user is going to take an action $a_H in \mathcal{A}$ based on the observation $o \in \mathcal{O}$ and the goal $g \in \mathcal{G}$ they have. The robot is also going to take a simultaneous action $a_R \in \mathcal{A}$, again based on the observation $o \in \mathcal{O}$ and the predicted user goal $\hat{g} \in \mathcal{G}$. The action being *simultaneous* means the robot does not know a_H while taking a_R. However, it has access to the previous observations and human actions. The system is then going to evolve as

$$s' \sim f(\cdot \mid s, a_H + a_R)$$

where we implicitly assume an addition operation is defined over $\mathcal{A}$. This is an instance of shared autonomy where the human and the user shares the control of the system.

For this question, let's say the actions can be linearly combined, i.e. when adding two actions, you can simply add the steering and throttle values.

iv) ✍ **[5 points]** We already know from the previous subproblems that we can use our trained neural networks to predict the user's goal. For the subsequent parts, run

```
python train_ildist.py --scenario lanechange --goal left
--epochs <number_of_epochs> --lr <learning_rate>
```

to train the mixed density network for the lanechange scenario. Again, you can use –restore argument if needed. Also, remember the training hints we provided to speed up the process. Similar to before, test the trained network by running

```
python test_ildist.py --scenario lanechange --goal left --visualize
```

Repeat the training and test for the "right" goal, too.

You are going to use these networks to predict the user goal. For each goal $g \in \mathcal{G}$ and observation $o \in \mathcal{O}$, let $m : \mathcal{G} \times \mathcal{O} \to \mathcal{A}$ be a function that gives the optimal action (with respect to the robot) to achieve the given goal under the given observation. Then, at time step t, the robot should take an action a_R to make sure

$$\begin{aligned} a_R + \mathbb{E}_g[a_H \mid g, o^t] &= \mathbb{E}_g[m(g, o^t) \mid o^t] \\ &= \sum_{g \in \mathcal{G}} P(g \mid o^{t-1}, \dots, o^0, a^{t-1}, \dots, a^0)\ m(g, o^t) \end{aligned}$$

You can use different various heuristics to compute the probability term inside the summation by using the probabilities $P(g, o^{t-1}, a^{t-1}), \dots, P(g|o^0, a^0)$. For example, you could take a moving average of these terms. Let's denote $P(g|o^{t-1}, \dots, o^0, a^{t-1}, \dots, a^0$ as P_g for simplicity. Then,

$$\begin{aligned} a_R &= \sum_{g \in \mathcal{G}} P_g m(g, o^t) - \mathbb{E}_g[a_H \mid o^t, g] \\ &= \sum_{g \in \mathcal{G}} P_g m(g, o^t) - \sum_{g \in \mathcal{G}} \sum_{a_H \in \mathcal{A}} P_g P(a_H \mid o^t, g) a_H \\ &= \sum_{g \in \mathcal{G}} P_g \left[m(g, o^t) - \sum_{a_H \in \mathcal{A}} P(a_H \mid o^t, g) a_H \right] \end{aligned}$$

where you can also obtain $P(a_H \mid o^t, g)$ using the same mixture density networks.

v) ✍️ **[4 points]** We further want to enforce some constraints on a_R. Specifically, we don't want the steering and throttle provided by a_R to be too large. Hence, you will simply apply a threshold to the computed a_R. From a human-robot interaction perspective, what is the reason we are enforcing this constraint on a_R.

vi) 💻 **[4 points]** Fill in the missing parts of `shared_autonomy.py` that implements what we discussed above. Then, run

```
python shared_autonomy.py --scenario lanechange
```

and control the vehicle's steering using the arrow keys in your keyboard. In this part, you are controlling only the steering, and the throttle is automatically determined.

vii) ✍️ **[4 points]** When you control the vehicle with shared autonomy, do you feel the help by the robot? Is it really helpful or does it make things worse? Elaborate on how it is helpful or why it might be harming your performance. This is an open-ended question, and your responses depend a lot on the performance of the trained mixed density network.

References

[1] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on Robot Learning*, 2017.

[2] Zhangjie Cao, Erdem Bıyık, Woodrow Z Wang, Allan Raventos, Adrien Gaidon, Guy Rosman, and Dorsa Sadigh. Reinforcement learning based control of imitative policies for near-accident driving. *Proceedings of Robotics: Science and Systems (RSS)*, 2020.

[3] Jason Kong, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *IEEE Intelligent Vehicles Symposium (IV)*, 2015.

[4] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in Neural Information Processing Systems*, 1988.

[5] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.