ECE 586 BH: Interplay between Control and Machine Learning **Fall 2023**

Lecture 13

Imitation Learning for Control, Part I

Lecturer: Bin Hu, Date:10/03/2023

In many situations, whether reinforcement learning and/or optimal control can work or not depends on the cost used in the problem formulation. However, the design of the cost function itself is not a trivial thing and typically requires domain expertise. For relatively simple control tasks such as regulations or tracking, quadratic cost can be used. For more complicated tasks in robotics and autonomy, the cost function is typically handcrafted in a case-by-case manner. Sometimes the objective functions are so complicated such that we do not know how to specify them beforehand. For example, consider the design of self-driving systems. We must carefully trade off competing objectives such as performance, safety, and comfort level in driving. It can be difficult to come up with a precise metric for "comfortable driving." Therefore, how to formulate the cost function for the control design of self-driving is unclear in the first place. This motivates the use of imitation learning. In this lecture, we will cover how to use imitation learning for control tasks.

13.1 A Quick Overview of Imitation Learning

To start, think that human experts are good at driving and grasping. The idea of imitation learning is to utilize expert demonstrations for such control problems where the objective function is not obvious. In general, imitation learning can be applied to settings where demonstrations from human experts or more general expert policies are available. Several main techniques used in imitation learning include

1. Behavior Cloning
2. DAGGER
3. Inverse RL
4. Generative Adversarial Imitation Learning (GAIL)

Behavior Cloning. The most basic algorithm in imitation learning is the behavior cloning method which takes in demonstrations, such as a human driver's steering and throttle commands, and attempts to fit a state-action control policy in a supervised learning fashion. Let's say that we are given a control design problem whose objective function is hard to specify beforehand. We assume some demonstrations from experts are available. Specifically, we assume a sequence of state/action pairs $\{x_k, u_k\}_{k=0}^{N}$ has been demonstrated by the expert. One way to do the control design here is to directly fit a policy on expert demonstrations. The hope is that the fitted policy can mimic the behaviors of the experts and hence perform well on the given control problem. It is natural to formulate such a fitting problem as a supervised learning problem:

$$\underset{K}{\text{minimize}} \quad \frac{1}{N}\sum_{k=0}^{N} L(K(x_k), u_k) + R(K) \tag{13.1}$$

where L is some loss function measuring the empirical performance of the fitted policy on observed demonstrations, and R is a regularization term introduced to prevent overfiting. In the above formulation, the policy is typically parameterized as a linear function or a neural network. The resultant optimization problem is unconstrained and can be efficiently solved by applying stochastic gradient descent (SGD) or other first-order methods. One can also use a trajectory-based formulation, i.e. sampling several trajectories and then applying behavior cloning to all the trajectories.

DAGGER. One issue for behavior cloning is how to sample the demonstrations. Let's say the expert is implicitly following some policy K^* which is unknown to us. Then typically one just runs the system $x_{t+1} = f(x_t, K^*(x_t), w_t)$ to generate the trajectories. However, when implementing the fitted controller K, the state space visited can be quite different. This is called the distributional shift. DAGGER fixes this issue by relabeling the states which have not been seen before. Specifically, when K is fitted, then run the system $x_{t+1} = f(x_t, K(x_t), w_t)$ to generate some new sequence $\{x_t, u_t\}$. Throw away the control action here and ask the expert to relabel the state. This gives us a new sequence of demonstrations. Then we can augment all the demonstrations and fit K again and then iterate.

Other methods. The goal of inverse RL is to learn the cost function other than the policy from the demonstrations. GAIL uses the idea of GANs to learn a policy that mimics the experts. We will not discuss the details of these methods in this course.

Prior in imitation learning for control. We can see that the original form of imitation learning is an unconstrained optimization problem. How to incorporate control-theoretic prior into the imitation learning process is a key issue, and many researchers are still studying this. In the next section, we will illustrate this via reviewing one recent result on imitation learning for model predictive control.

13.2 Imitation Learning for Model Predictive Control

In this section, we briefly discuss the idea for the main reference of this lecture (see the course website for more information). For simplicity, we mainly follow the video presentation of the main reference given by Kwangjun Ahn (who is the first author). Some simplifications have been made to convey the idea. The discussion on the robust MPC scheme used in the actual paper is skipped.

Suppose we are interested in solving the following constrained control problem:

$$x_{t+1} = Ax_t + Bu_t \tag{13.2}$$

where $x_t \in \mathcal{X} = [-a, a]^{n_x}$, and $u_t \in \mathcal{U} = [-b, b]^{n_u}$. Consider a quadratic stage cost $x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t$ with $Q \geq 0$ and $R > 0$. If the problem is unconstrained (i.e. $a = +\infty$ and $b = +\infty$), then the optimal solution to the infinite-horizon problem is provided by the LQR policy $u_t = \pi^{lqr}(x_t) = -(R + B^\mathsf{T} P B)^{-1} B^\mathsf{T} P A x_t$ where P is the stabilizing solution to the algebraic Riccati equation $P = A^\mathsf{T} P A - A^\mathsf{T} P B (R + B^\mathsf{T} P B)^{-1} B^\mathsf{T} P A + Q$. However, if the problem is constrained (i.e. a and b are finite), the problem becomes more difficult. One popular approach is the model predictive control (MPC) approach. At every step t, the MPC controller generates the control action via solving a real-time optimization problem and apply $u_0^*(x)$:

$$\begin{aligned} MPC(x) = \min_{u_0, u_1, \cdots, u_{N-1}} & \sum_{t=0}^{N-1} x_t^\mathsf{T} Q x_t + u_t^\mathsf{T} R u_t + x_N^\mathsf{T} Q_f x_N \\ \text{s.t. } & x_{t+1} = Ax_t + Bu_t,\, x_0 = x \\ & x_t \in \mathcal{X},\, u_t \in \mathcal{U}\, \forall t \end{aligned}$$

At every time, one solves a finite-horizon planning problem with an N-step planning window. Instead of applying all the control actions $(u_0^*, u_1^*, \cdots, u_{N-1}^*)$, MPC only applies u_0^* at every step and then replan! Therefore, MPC is a state-feedback controller which satisfies $\pi^{MPC}(x) = u_0^*(x)$. The closed-loop feedback mechanism is just more robust than the open-loop strategies. MPC is very powerful and has been widely applied in industry. Due to the feedback nature, MPC has some nice stability/robustness properties and can guarantee safety at the same time. However, solving an N-step planning problem at every step can be costly for problems with high dimensions and long planning windows. One possible fix is to use the explicit MPC approach which precomputes $u_0^*(x)$ offline for all x and directly implement the resultant piecewise affine controller. However, although $u_0^*(x)$ is a piecewise affine function of x, it is typically very complicated and hard to compute/store. The piece number roughly scales with 2^{N+d}, which is a huge number for large (N, d).

Now it seems natural to apply imitation learning to learn a neural network policy which mimics the behaviors of MPC. Instead of solving $u_0^*(x)$ exactly, we sample some pairs of $(x, u_0^*(x))$ and the fit a neural network policy on the training set (expert demonstrations). However, the behavior cloning approach may not work that well due to the distribution shift issue. It can potentially lead to a destabilizing controller. The main reference used the forward training method to address this issue. The forward training method generates a time-varying policy π_t as follows. First, one just samples $(x_0, u_0^*(x_0))$ to fix π_0. Then, one sample x_0 and run π_0 to generate x_1. After applying MPC to x_1, then just fit π_1 on all the data $(x_1, u_0^*(x_1)$. Similarly, one can sample the initial states again and apply (π_0, π_1)

to generate x_2, and then fit π_2 on $(x_2, u_0^*(x_2))$. This approach can be somehow thought as the "time-varying" version of DAGGER. Now the policy is completely trained using its own trajectory, and there is no distribution shift issue. However, this approach learns time-varying π_t, and there is a scalability issue here as the number of stages increases. The authors of the main reference address this via leveraging the theoretical properties of MPC. Specifically, it is known that there exists a positive invariant region around the equilibrium point such that $u_0^*(x)$ becomes the LQR controller. Once the states are stabilized to that region, there is nothing left to learn. Under mild assumptions, MPC is exponentially stable and drives states to that region exponentially fast. Therefore, one can estimate the steps needed to reach that region, and then applying the forward training method for that many steps should be sufficient. Basically, one only runs the forward training until all the sampled states at τ are in the positive invariant set around the equilibrium point. Some sample complexity theory has also been provided to support this approach.

The above approach provides a nice example showing that applying imitation learning for control requires leveraging control-theoretic properties of existing control approaches.