Homework 1: Policy Iteration & Nonlinear Control

CS 4789/5789: Introduction to Reinforcement Learning

(Due March 22rd 11:59 ET)

0 Instructions

For each question in this HW, please list all your collaborators and reference materials (beyond those specified on the website) that were used for this homework. Please add your remarks in a "Question 0".

1 Policy Iteration: Complexity of Returning the Optimal Policy?

Recall policy iteration algorithm. In the class, we prove that the policy iteration has the following property: $\left\|V^{\pi^{t+1}} - V^\star\right\|_\infty \leq \gamma \left\|V^{\pi^t} - V^\star\right\|_\infty$. So ideally, if we run Policy iteration infinitely many iterations, i.e., $t \to \infty$, we will have $\lim_{t\to\infty} \|V^{\pi^t} - V^\star\|_\infty = 0$.

In this section, we are more interested in the following question: can policy iteration exactly hit the optimal policy $\pi^\star$ with at most finite number of iterations? The answer is yes, and we are going to investigate it below.

1.1 When PI does not make improvement.. *[20 points]*

In the class, we proved that PI has monotonic improvement, i.e., $Q^{\pi^{t+1}}(s,a) \geq Q^{\pi^t}(s,a)$ for all s, a. What happens if PI does not making progress, i.e., π^{t+1} and π^t are exactly the same in the following sense:

$$Q^{\pi^{t+1}}(s,a) = Q^{\pi^t}(s,a), \forall s, a. \tag{1}$$

I.e., policy iteration does not give us improvement at any state action pair.

Q: Assume that Eq. 1 holds, and reward is always bounded between $[0,1]$. Prove that π^{t+1} is an optimal policy, i.e., $Q^{\pi^{t+1}}(s,a) = Q^\star(s,a)$ for all s, a.

1.2 Bounding the number of iterations *[10 points]*

Q: Consider the setting where we have A many actions and S many states. Prove that PI will find an optimal policy with number of iterations at most A^S.

1.3 Summary and Comparison to Value Iteration (No questions here)

In summary, PI actually finds an exact optimal policy with finite number of iterations A^S. Recall the value iteration (VI) algorithm, unlike PI, VI indeed does not guarantee to find the optimal policy with finite number of iterations. The worst case situation is that in the MDP there exists some sub-optimal policy $\hat{\pi}$ such that $V^{\hat{\pi}}$ is arbitarily close to $V^\star$. While in this case, $\hat{\pi}$ is a nearly optimal policy already (i.e., from an approximation algorithm perspectively, it's totally ok to return $\hat{\pi}$ as a high quality approximation), it will take VI arbitrarily many iterations to eventually distinguish $\hat{\pi}$ from $\pi^\star$.

2 Linear Quadratic Regulator *[20 points]*

In the class, we introduced the most basic formulation of LQR. In this section, we are going to slightly make the model more general.

We are interested in solving the following problem:

$$\min_{\pi_0,\ldots,\pi_{H-1}} \mathbb{E}\left[\sum_{t=0}^{H-1} x_t^\top Q x_t + u_t^\top R u_t + x_t^\top M u_t + q^\top x_t + r^\top u_t + b\right] \tag{2}$$

$$\text{subject to } x_{t+1} = Ax_t + Bu_t + m, u_t = \pi_h(x_t) \quad x_0 \sim \mu_0. \tag{3}$$

Here we have $x \in \mathbb{R}^{d_x}$, $u \in \mathbb{R}^{d_u}$, $Q \in \mathbb{R}^{d_x \times d_x}$ (Q is positive definite), $M \in \mathbb{R}^{d_x \times d_u}$, $q \in \mathbb{R}^{d_x}$, $R \in \mathbb{R}^{d_u \times d_u}$ (R is positive definite), $r \in \mathbb{R}^{d_u}$, $b \in \mathbb{R}$, $A \in \mathbb{R}^{d_x \times d_x}$, $B \in \mathbb{R}^{d_x \times d_u}$, and $m \in \mathbb{R}^{d_x}$. We also always have the following matrix being positive definite:

$$\begin{bmatrix} Q & M/2 \\ M^\top/2 & R \end{bmatrix}.$$

The difference between the above formulation and the formulation we had in class is that here the cost function contains an additional second order term $x_t^T M u_t$, first-order terms $q^T x_t + r^T u_t$, and zeroth order term c, and the dynamics contains zeroth order term m.

Q: Derive the optimal policy for the above general LQR control. Note that the optimal policy will be linear in the form of $\pi_t^\star(x) = K_t^\star x + k_t^\star$ where $K_t^\star \in \mathbb{R}^{d_u \times d_x}$ and $k_t^\star \in \mathbb{R}^{d_u}$.

Start from defining $Q_{H-1}^\star(x, u)$. Then given the formulation of $V_{h+1}^\star(x)$, write out the expression for $Q_h^\star(x, u)$. Then derive $\pi_h^\star$ from the expression of $Q_h^\star(x, u)$. Finally using $\pi_h^\star$ and $Q_h^\star(x, u)$ we can get $V_h^\star(x)$ which completes the recursion step. Namely, you should work out a base case, and a recursive step (i.e., something similar to the Ricatti equation we talked about in the LQR lecture).

Make sure that the derivation above is correct. Also simplify the equations as much as you can as you will need to implement these equations in the next problem for controlling a simulated CartPole.

3 Programming: Local Linearization Approach for Controlling CartPole *[50 points]*

3.1 Background on local linearization approach for nonlinear contorl

Consider the following general nonlinear control problem:

$$\min_{\pi_0,\ldots,\pi_{H-1}} \sum_{t=0}^{H-1} c(x_t, u_t)$$
$$\text{subject to:} x_{t+1} = f(x_t, u_t), u_t = \pi_t(x_t), x_0 \sim \mu_0;$$

where $f : \mathbb{R}^{d_x} \times \mathbb{R}^{d_u} \mapsto \mathbb{R}^{d_x}$.

In general the cost $c(x, u)$ could be anything. Here we focus on a special instance where we try to keep the system stable around some stable point $(x^\star, u^\star)$, i.e., our cost function $c(x, u)$ penalizes the deviation to $(x^\star, u^\star)$, i.e., $c(x, u) = \rho\left(x_t - x^\star\right) + \rho(u_t - u^\star)$, where ρ could be some distance metric such as ℓ_1 or ℓ_2 distance.

To deal with nonlinear f and non-quadratic c, we use the linearization approach here. Since the goal is to control the robot to stay at the pre-specified stable point $(x^\star, u^\star)$, it is reasonable to assume that the system is approximately linear around $(x^\star, u^\star)$, and the cost is approximately quadratic around $(x^\star, u^\star)$. Namely, we perform first-order taylor expansion of f around $(x^\star, u^\star)$, and we perform second-order taylor expansion of c around $(x^\star, u^\star)$:

$$f(x, u) \approx A(x - x^\star) + B(u - u^\star) + f(x^\star, u^\star),$$
$$c(x, u) \approx \frac{1}{2} \begin{bmatrix} x - x^\star \\ u - u^\star \end{bmatrix}^\top \begin{bmatrix} Q & M \\ M^T & R \end{bmatrix} \begin{bmatrix} x - x^\star \\ u - u^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} x - x^\star \\ u - u^\star \end{bmatrix} + c(x^\star, u^\star);$$

Here A and B are Jacobians, i.e.,

$$A \in \mathbb{R}^{d_x \times d_x} : \; A[i, j] = \frac{\partial f[i]}{\partial x[j]}[x^\star, u^\star] : \qquad B \in \mathbb{R}^{d_x \times d_u}, B[i, j] = \frac{\partial f[i]}{\partial u[j]}[x^\star, u^\star],$$

where $f[i](x, u)$ stands for the i-th entry of $f(x, u)$, and $x[i]$ stands for the i-th entry of the vector x. Similarly, for cost function, we will have Hessian and gradients as follows:

$$Q \in \mathbb{R}^{d_x \times d_x} : Q[i, j] = \frac{\partial^2 c}{\partial x[i] \partial x[j]}[x^\star, u^\star], \quad R \in \mathbb{R}^{d_u \times d_u} : R[i, j] = \frac{\partial^2 c}{\partial u[i] \partial u[j]}[x^\star, u^\star]$$
$$M \in \mathbb{R}^{d_x \times d_u} : M[i, j] = \frac{\partial^2 c}{\partial x[i] \partial u[j]}[x^\star, u^\star], \quad q \in \mathbb{R}^{d_x} : q[i] = \frac{\partial c}{\partial x[i]}[x^\star, u^\star]$$
$$r \in \mathbb{R}^{d_u} : r[i] = \frac{\partial c}{\partial u[i]}[x^\star, u^\star].$$

We are almost ready compute a control policy using A, B, Q, R, M, q, r together with the optimal control we derived for the system in Eq. 2. One potential issue here is that the original cost

function $c(x, u)$ may not be even convex. Thus the Hessian matrix $H := \begin{bmatrix} Q & M \\ M^T & R \end{bmatrix}$ may not be a positive definite matrix. We will apply further approximation to make it a positive definite matrix. Denote the eigen-decomposition of H as $H = \sum_{i=1}^{d_x+d_u} \sigma_i v_i v_i^\top$ where σ_i are eigenvalues and v_i are corresponding eigenvectors. We approximate H as follows:

$$H \approx \sum_{i=1}^{d_x+d_u} \mathbf{1}\{\sigma_i > 0\}\sigma_i v_i v_i^\top + \lambda I, \tag{4}$$

where $\lambda \in \mathbb{R}^+$ is some small positive real number for regularization which ensures that after approximation, we get an H that is positive definite with minimum eigenvalue lower bounded by λ.

3.2 Setup of Simulated CartPole

The simulated CartPole has the following nonlinear deterministic dynamics $x_{t+1} = f(x_t, u_t)$, and potentially non-quadratic cost function $c(x, u)$ that penalizes the deviation of the state from the balance point $(x^\star, u^\star)$ where $u^\star = 0$, and $x^\star$ represents the state of CartPole where the pole is straight and the cart is in a pre-specified position.

The state x_t is a 4-dimension vector. It consists of the position of the cart, the speed of the cart, the angle of the pole in radian and the angular velocity of the pole. The action u_t is a 1-dimenstion vector correspond to the force applied on the cart.

Through this section, we assume that we have black-box access to c and f, i.e., we can feed any (x, u) to f and c, we will get two scalar returns which are $f(x, u)$ and $c(x, u)$ respectively. Namely, we do not know the analytical math formulation of f and c (e.g., imagine that we are trying to control some complicated simulated humanoid robot. The simulator is the black-box f).

In this assignment we will use our customized OpenAI gym CartPole environment provided in the following repository. The environment is under `env` directory. The goal is to finish the implementation of `cartpole_controller.py` which contains a class to compute the locally linearized optimal policy of our customized CartPole environment. We also provide other files to help you get started. Please refer to `README.md` for more details.

https://github.com/cs4789-s21/PA1

3.3 Using Finite Difference for Taylor Expansion

Since we do not know the analytical form of f and c, we cannot directly compute the analytical formulations for Jacobians, Hessians, and gradients. However, given the black-box access to f and c, we can use *finite difference* to approximately compute these quantities.

Below we first explain the finite differencing approach for approximately computing derivatives. Your task is to use finite differencing methods to compute A, B, Q, R, M, q, r.

To illustrate finite differencing, assume that we are given a function $g : \mathbb{R} \mapsto \mathbb{R}$. Given any $\alpha_0 \in \mathbb{R}$, to compute $g'(\alpha_0)$, we can perform the following process:

$$\textbf{Finite Difference for derivative:} \widehat{g'}(\alpha_0) := \frac{g(\alpha_0 + \delta) - g(\alpha_0 - \delta)}{2\delta},$$

for some $\delta \in \mathbb{R}^+$. Note that by the definition of derivative, we know that when $\delta \to 0^+$, the approximation approaches to $g'(\alpha_0)$. In practice, δ is a tuning parameter: we do not want to set it to 0^+ due to potential numerical issue. We also do not want to set it too large, as it will give a bad approximation of $g'(\alpha_0)$.

With $\widehat{g'}(\alpha)$ as a black-box function, we can compute the second-derivate using Finite differencing on top of it:

$$\textbf{Finite Difference for Second Deriviative: } \widehat{g''}(\alpha_0) := \frac{\widehat{g'}(\alpha_0 + \delta) - \widehat{g'}(\alpha_0 - \delta)}{2\delta}$$

Note that to implement the above second derivative approximator $\widehat{g''}(\alpha)$, we need to first implement the function $\widehat{g'}(\alpha)$ and treat it as a black-box function inside the implementation of $\widehat{g''}(\alpha)$. You can see that we need to query black-box f twice for computing $g'(\alpha_0)$ and we need to query black-box f four times for computing $g''(\alpha_0)$.

Similar ideas can be used to approximate Jocabians, gradients, and Hessians.

At this end, using the provided Cartpole simulator which has black-box access to f and c, and the goal balance point $x^\star, u^\star$, to compute A, B, Q, R, q, r, with the provided value for δ.

We provide minimum barebone functions and some tests for finite difference method in the file `finite_difference_method.py`. You can try to complete the implementation of gradient, Jacobian and Hessian functions as we can use them to compute A, B, Q, R, M, q, r which will be used in the next section.

3.4 Computing locally optimal control

With A, B, Q, R, M, q, r computed from finite differencing, let us first check if $\begin{bmatrix} Q & M \\ M^T & R \end{bmatrix}$ a positive definite matrix (if it's not PD, your LQR formulation may run into the case where matrix inverse does not exist and and numerically you will observe NAN as well.). If not, let's use the trick in Eq. 4. Denote $H := \begin{bmatrix} Q & M \\ M^T & R \end{bmatrix}$. We now are ready to solve the following linear quadratic system:

$$\min_{\pi_0,\ldots,\pi_{H-1}} \sum_{t=0}^{H-1} \frac{1}{2} \begin{bmatrix} x_t - x^\star \\ u_t - u^\star \end{bmatrix}^\top H \begin{bmatrix} x_t - x^\star \\ u_t - u^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} x_t - x^\star \\ u_t - u^\star \end{bmatrix} + c(x^\star, u^\star), \tag{5}$$

$$\text{subject to } x_{t+1} = Ax_t + Bu_t + m, u_t = \pi_t(x_t), \quad x_0 \sim \mu_0. \tag{6}$$

With some rearranging terms, we can re-write the above program in the format of Eq. 2. To this end, we can implement the optimal control that solves the final formulation in Eq. 6.

Please complete the function `compute_local_policy` in `cartpole_controller.py`. The function computes the linear locally optimal control policy of the CartPole environment. We also provide you a barebone LQR function in `lqr.py` to get started, you can implement it by using the policy you derived in Problem 2.

After finishing the `compute_local_policy` function in the file `cartpole_controller.py`, you can try running `cartpole.py` and see the visualization.

3.5 Test the performance

Given policies $\pi_0, \ldots, \pi_{H-1}$ that computed from the previous section, we will evaluate its performance by executing it on the real system f and real cost c. To generate a H-step trajectory, we first sample $x_0 \sim \mu_0$, and then take $u_t = \pi_t(x_t)$, and then call the black-box f to compute $x_{t+1} = f(x_t, u_t)$ till $t = H - 1$. This gives us a trajectory $\tau = \{x_0, u_0, x_1, u_1, \ldots, x_{H-1}, u_{H-1}\}$. The total cost of the trajectory is $C(\tau) := \sum_{t=0}^{H-1} c(x_t, u_t)$. Since we have randomness in μ_0, we need to draw N such i.i.d trajectories, $\tau^1 \ldots \tau^N$, and compute the average, i.e., $\sum_{i=1}^{N} C(\tau^i)/N$. As N approaches to $+\infty$, we have (by law of large numbers):

$$\sum_{i=1}^{N} C(\tau^i)/N \to \mathbb{E}_{x_0 \sim \mu_0} \left[\sum_{t=0}^{H-1} c(x_t, u_t) | u_t = \pi_t(x_t), x_{t+1} = f(x_t, u_t) \right].$$

In the test file `test.py`, we provide several difference initialization distributions. These distributions are ordered based on the distance between the means to the goal $(x^\star, u^\star)$. Intuitively, we should expect that our control perform worse when the initial states are far away from the taylor-expansion point $(x^\star, u^\star)$, as our linear and quadratic approximation become less accurate. Testing your computed policies on these difference initializations and report the performances for all initializations by running `test.py`.

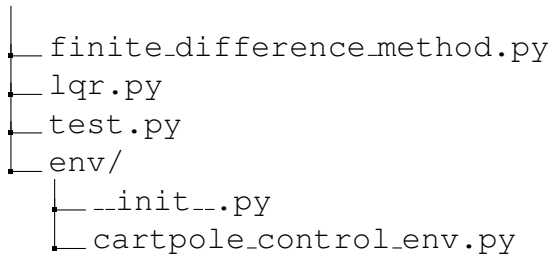
3.6 Visualization

Now, using the visualization tool provided by Open AI Gym, provide a video demonstration of your policy with the given initial distribution by running `cartpole.py`. The generated mp4 video file should be under the directory `./gym-results`).

3.7 Submission

Please submit a zip file containing your implementation along with your video file name `cartpole.mp4` organized as follows.

```
YOUR_NET_ID/
├── README.md
├── __init__.py
├── cartpole.mp4
├── cartpole.py
└── cartpole_controller.py
```


finite_difference_method.py
lqr.py
test.py
env/
__init__.py
cartpole_control_env.py