Stanford **Spring 2026**

AA 203: Optimal and Learning-based Control
Homework #4
Due Wednesday June 3 by 5:00 pm

Problem 1: Apply SCP to MPC for nonlinear systems with non-convex constraints.

Problem 2: Gain experience with data-driven learning in a "real world" setting.

Problem 3: Implement reinforcement learning agents in a familiar setting.

Problem 4: Analyze the strengths and weaknesses of some fundamental reinforcement learning algorithms.

Problem 5: Reason through some concepts involved in implementing a neural-network-based actor-critic algorithm, and gain insight into modern deep reinforcement learning practice.

4.1 Obstacle avoidance. In this problem, you will implement a nonlinear MPC controller to steer towards a goal position while avoiding obstacles. Consider the discrete-time nonlinear system

$$s(t+1) = f(s(t), u(t)) = s(t) + \Delta t \begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \\ u_1(t) - \beta \dot{x}(t)|\dot{x}(t)| \\ u_2(t) - \beta \dot{y}(t)|\dot{y}(t)| \end{pmatrix},$$

where $s(t) = (x(t), y(t), \dot{x}(t), \dot{y}(t)) \in \mathbb{R}^4$ is the state of the system with position $(x(t), y(t)) \in \mathbb{R}^2$ and velocity $(\dot{x}(t), \dot{y}(t)) \in \mathbb{R}^2$, $u(t) = (u_1(t), u_2(t)) \in \mathbb{R}^2$ is the control input, $\Delta t > 0$ is a discretization constant, and $\beta > 0$ is a drag coefficient. Overall, this nonlinear system represents an Euler-discretized, two-dimensional double-integrator system subject to drag forces.

Let $s_{\cdot|t} := \{s_{k|t}\}_{k=0}^{N}$ and $u_{\cdot|t} := \{u_{k|t}\}_{k=0}^{N-1}$ denote N-step state and control trajectories planned from some initial state $s(t)$ with $s_{0|t} = s(t)$. We want to design a closed-loop MPC controller that optimizes such trajectories using the quadratic cost function

$$J(s_{\cdot|t}, u_{\cdot|t}) = (s_{N|t} - s_{\text{goal}})^\mathsf{T} P (s_{N|t} - s_{\text{goal}}) + \sum_{k=0}^{N-1} \Big((s_{k|t} - s_{\text{goal}})^\mathsf{T} Q (s_{k|t} - s_{\text{goal}}) + u_{k|t}^\mathsf{T} R u_{k|t} \Big),$$

with cost matrices $P \succ 0$, $Q \succ 0$, and $R \succ 0$ and goal state $s_{\text{goal}} \in \mathbb{R}^4$. We generally choose P to have much larger entries than those in Q and R, such that system state is driven towards s_{goal}.

While steering towards the goal state, we want to avoid N_{obs} obstacles. In this problem, the j^{th} obstacle is represented as a circle with center $(x_j, y_j) \in \mathbb{R}^2$ and radius $r_j > 0$. We define the function $d : \mathbb{R}^n \to \mathbb{R}^{N_{\text{obs}}}$ such that the j^{th} entry is given by the *signed distance function*

$$d_j(s) = \|(x, y) - (x_j, y_j)\|_2 - r_j = \sqrt{(x - x_j)^2 + (y - y_j)^2} - r_j,$$

which is differentiable away from $(x, y) = (x_j, y_j)$. Enforcing $d(s) \succeq 0$ would ensure the system does not collide[1] with any of the obstacles.

[1] We assume that r_j is at least slightly larger than the actual obstacle radius to avoid brushing against it.

Overall, the MPC problem we would like to solve at each time t in closed-loop is

$$\begin{aligned}
\underset{s_{\cdot|t},u_{\cdot|t}}{\text{minimize}} \quad & (s_{N|t}-s_{\text{goal}})^\mathsf{T} P (s_{N|t}-s_{\text{goal}}) + \sum_{k=0}^{N-1} \Big((s_{k|t}-s_{\text{goal}})^\mathsf{T} Q (s_{k|t}-s_{\text{goal}}) + u_{k|t}^\mathsf{T} R u_{k|t} \Big) \\
\text{subject to} \quad & s_{0|t} = s(t) \\
& s_{k+1|t} = f(s_{k|t}, u_{k|t}), \quad \forall k \in \{0,1,\ldots,N-1\} \\
& d(s_{k|t}) \succeq 0, \qquad\qquad \forall k \in \{0,1,\ldots,N\}
\end{aligned}$$

There are two sources of non-convexity in this problem.

- The dynamics are nonlinear, so the equality constraint $s_{k+1|t} = f(s_{k|t}, u_{k|t})$ is non-convex.
- Each entry of the signed distance function d is convex, i.e., $-d$ is concave. Thus, $d(s_{k|t}) \succeq 0$ or equivalently $-d(s_{k|t}) \preceq 0$ is a non-convex inequality constraint.

To solve this problem, you will use *sequential convex programming (SCP)*, where for the i^{th} subproblem you will affinize the dynamics and the signed distance function around the current estimated solution $(s_{\cdot|t}^{(i)}, u_{\cdot|t}^{(i)})$. For simplicity, we will not include trust region constraints in this problem since the dynamics are close to being linear, but in practice you should always consider using them.

You will work with the starter code in `obstacle_avoidance.py`. Carefully review *all* of the code in this file before you continue. Only submit code you add to this starter file and any associated plots that are generated by the file.

(a) Use JAX and array broadcasting[2] to complete the function `signed_distances` with a single line of code that computes $d(s)$.

(b) Use JAX to complete the function `affinize` with two lines of code that produce an affine estimate $f(s,u) \approx As + Bu + c$ from the Taylor expansion of *any* given differentiable function f.

(c) State the convex approximation of the MPC problem around an iterate $(s_{\cdot|t}^{(i)}, u_{\cdot|t}^{(i)})$. In particular, write the constraints in terms of the affine approximations

$$\begin{aligned}
f(s_{k|t}, u_{k|t}) &\approx A_{f,k|t}^{(i)} s_{k|t} + B_{f,k|t}^{(i)} u_{k|t} + c_{f,k|t}^{(i)} \\
d(s_{k|t}) &\approx A_{d,k|t}^{(i)} s_{k|t} + c_{d,k|t}^{(i)}
\end{aligned},$$

and express $A_{f,k|t}^{(i)}$, $B_{f,k|t}^{(i)}$, $c_{f,k|t}^{(i)}$, $A_{d,k|t}^{(i)}$, and $c_{d,k|t}^{(i)}$ in terms of $(s_{\cdot|t}^{(i)}, u_{\cdot|t}^{(i)})$, the functions f and d, and appropriate Jacobians.

(d) The fact that we are approximating obstacle avoidance constraints may be concerning at first from a safety perspective. Describe why $A_{d,k|t}^{(i)} s_{k|t} + c_{d,k|t}^{(i)} \succeq 0$ ensures $d(s_{k|t}) \succeq 0$.

Hint: For any $\bar{x} \in \mathbb{R}^n$, a convex function $g : \mathbb{R}^n \to \mathbb{R}$ lies above any line $\ell(x) = a^\mathsf{T} x + \beta \in \mathbb{R}$ through $\bar{x}$ that is tangent to the function g at $\bar{x}$. As an example for $n = 1$, the quadratic function $g(x) = \frac{1}{2}x^2$ lies entirely above the line $\ell(x) = x - \frac{1}{2}$, which is tangent to g at $\bar{x} = 1$.

(e) Complete the function `scp_iteration`. Specifically, given the current iterate $(s_{\cdot|t}^{(i)}, u_{\cdot|t}^{(i)})$, use CVXPY to specify and solve the convex optimization problem you derived previously to obtain an updated solution $(s_{\cdot|t}^{(i+1)}, u_{\cdot|t}^{(i+1)})$.

[2] See the NumPy documention (https://numpy.org/doc/stable/user/basics.broadcasting.html) for details on array broadcasting. However, make sure to use `jax.numpy` functions!

(f) Complete the simulation towards the end of `obstacle_avoidance.py` to use the MPC solution from `solve_obstacle_avoidance_scp` at each time t in a closed-loop fashion.

(g) The variables `N` and `N_scp` set the MPC horizon N and the maximum number of SCP iterations N_{SCP} used to solve the MPC problem at each time, respectively. Run your completed version of `obstacle_avoidance.py` for

- $N = 5$ and $N_{\text{SCP}} = 5$,
- $N = 2$ and $N_{\text{SCP}} = 5$,
- $N = 5$ and $N_{\text{SCP}} = 2$, and
- $N = 15$ and $N_{\text{SCP}} = 2$.

The system should get closer to the goal while avoiding the obstacles. For each case, submit the generated state and control trajectory plots, and report the total computation time (`total_time`) and the total control effort (`total_control_cost`) to two decimal places. Discuss what you observe for each case.

4.2 Widget sales. You have just purchased Widget Co., a small shop selling widgets. Congratulations! Widget Co. is in the business of buying widgets wholesale, and selling them to consumers at a markup. While the previous owners were losing money, you suspect that you can apply your knowledge of optimal control and reinforcement learning to turn the business around.

The shop is able to store between 0 and 5 widgets at a time. We write the number of widgets held in the shop on day t as s_t. Every day, you choose how many widgets to order from your supplier. You can order either zero widgets, a "half order" of 2 widgets, or a "full order" of 4 widgets. We write the number of widgets ordered to arrive on day t as a_t. A random number of customers (following an unknown distribution, though this distribution may be assumed to be consistent across all days) come to Widget Co. every day; each customer buys a widget if there are any available. We write the demand on day t as d_t, and assume $d_t \leq 5$. At the end of each day, you record a net profit r_t for that day.

(a) The previous owners were nice enough to give you the details of their last three years of operation. In particular, they have provided a dataset $\mathcal{D} = \{(s_t, a_t, r_t)\}_{t=1}^{T}$ containing records for each day t of the last three years. In `widget_sales.py`, implement a Q-learning algorithm to learn tabulated Q-values from this dataset. Provide only the code you add to the provided starter file.

While you were busy with your model-free learning, your modeling-inclined intern noticed that the dynamics of the number of widgets in the shop each day are described by

$$s_{t+1} = f(s_t, a_t, d_t) \coloneqq \begin{cases} 0, & s_t + a_t - d_t < 0 \\ 5, & s_t + a_t - d_t > 5 \\ s_t + a_t - d_t, & \text{otherwise} \end{cases},$$

and the daily net profit is

$$r(s_t, a_t, d_t) = c_{\text{sell}} \min(s_t + a_t, d_t) - c_{\text{rent}} - c_{\text{storage}} s_t - g_{\text{order}}(a_t),$$

where $c_{\text{sell}} = 1.2$ is the price you set for each widget, $c_{\text{rent}} = 1$ is the fixed rent on your shop, $c_{\text{storage}} = 0.05$ is the cost for storing each widget overnight, and $g_{\text{order}}(a_t) = \sqrt{a_t}$ is the cost of ordering widgets from your supplier. The quantity $\min(s_t + a_t, d_t)$ is the "satisfied demand" on day t.

Moreover, after a few weeks of sales, your intern determined that the daily demand distribution for your widgets seems to be

$$d_t = \begin{cases} 0, & \text{with probability } 0.1 \\ 1, & \text{with probability } 0.3 \\ 2, & \text{with probability } 0.3 \\ 3, & \text{with probability } 0.2 \\ 4, & \text{with probability } 0.1 \end{cases}.$$

(b) In `widget_sales.py`, implement value iteration to learn tabulated Q-values from the model your intern has provided. Submit only the code you add to the provided starter file. Also submit the plot generated by `widget_sales.py` of Q-values from Q-learning compared to those from value iteration. What do you notice about the learned Q-values compared to those from value iteration? Why do you think this occurs?

(c) In `widget_sales.py`, compute an optimal policy $\pi^*_{\text{QL}}(s_t)$ based on your Q-learning work, and another optimal policy $\pi^*_{\text{VI}}(s_t)$ based on your value iteration work. Report each optimal policy, simulate each one over five years, and compute the cumulative profit $\sum_{k=0}^{t} r_k$ for each day t and for each optimal policy. Submit only the code you add to the provided starter file. Also submit the plot generated by `widget_sales.py` comparing the cumulative profits over time. What do you notice about the difference between the two cumulative profit trends? Why do you think this occurs?

4.3 Cart-pole balance via Reinforcement Learning. In this problem, we will return to the classic "cart-pole" benchmark in which we aim to design a controller to balance an inverted pendulum upright on a cart. Previously, in the second problem set i.e., Problem 2.3, we explored theory from optimal control to balance the pendulum, which relied on explicitly modeling the system's dynamics and the derivation of a closed-form control policy (e.g., through Riccati recusion). Alternatively, in this problem we will explore an approach to solving the cart-pole balancing problem using reinforcement learning (RL) through a model-free framework: rather than directly deriving a control law, we will instantiate an RL agent that learns an optimal policy through interaction with the environment, receiving feedback in the form of rewards.

The agent observes the state of the environment as $s := (x, \dot{x}, \theta, \dot{\theta}) \in \mathbb{R}^4$, where $x \in \mathbb{R}$ denotes the horizontal position of the cart and $\theta \in \mathbb{R}$ denotes the angle of the pendulum from the upright position. At each instant, the agent simply chooses an action $a_t \in \{0, 1\}$ indicating whether to push the cart to the left $a_t = 0$ or to the right $a_t = 1$. For complete details on the "CartPole-v1" environment we will use in OpenAI's Gymnasium package, please see the online documentation.

Unlike the previous problem, this episodic RL formulation does not rely on an explicit model of the system's dynamics. Instead, the agent learns by trial and error, with the goal of choosing actions so as to maximize the expected cumulative reward it observes. Once you are ready to train your policy, you are welcome to experiment with your own hyperparameters, if you'd like, but we provide hyperparameters by default, too, which you are welcome to use. Explicitly, we will grade this portion of the assignment based on the trends that emerge from your comparisons and training, rather than exact e.g., average reward observed by the policy.

(a) In this sub-part, you will implement a deep Q-learning algorithm with experience replay, originally introduced in "Playing Atari with Deep Reinforcement Learning" (MKS+13). The necessary starting code for this problem is provided in `agents/qlearning.py` and `main.py`: please carefully review the code already provided for you before proceeding.

i. Please follow the directions in `agents/qlearning.py` to implement the `build_network` function in the `QLearning` class. As a deliverable for this question, please attach a copy of your code to your pdf submission.

ii. Please follow the directions in `agents/qlearning.py` to implement the `policy` function in the `QLearning` class. As a deliverable for this question, please attach a copy of your code to your pdf submission.

iii. Please use the functions you've crafted to complete the `train` function in the `QLearning` class according to the algorithm described in "Algorithm 1" of (MKS+13). As a deliverable for this question, please attach a copy of your code to your pdf submission.

iv. Now, you are ready to train! Launch the training with your choice of hyperparameters and a visualization, if you'd like. Please use the `policy rollout` function in `utils.py` to score reward statistics on the randomly initialized Q network for your policy i.e., evaluate the policy's performance before training and report the statistics here. After training, also score the same reward statistics for your new (and hopefully improved) policy.

(b) In this sub-part, you will be implementing the REINFORCE algorithm: a policy optimization algorithm originally introduced in "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning" (Wil92). Similarly to part (a), the necessary starter code is provided in `agents/ReinforceAgent.py` and `main.py`: please carefully review the code already provided for you before proceeding. In this sub-part you will be implementing three variations of the REINFORCE algorithm, where each variation corresponds to a different definition of the policy gradient.

Specifically, we consider the case of a stochastic, parametrized policy, π_θ, where parameters θ represent, e.g., the weights of a neural network. Recall that policy optimization methods aim to find the parameters θ that directly optimize the RL objective: $J(\pi_\theta) = \mathbb{E}_{\tau\sim p(\tau)}[R(\tau)]$, where $\tau = (x_0, u_0, \ldots, x_T)$ represents a trajectory obtained by interacting with an episodic environment, $p(\tau) = p(x_0)\prod_{t=0}^{T} p(x_{t+1}|x_t, u_u)\pi_\theta(u_t|x_t)$ is the *trajectory distribution* (i.e., the probability of observing a specific trajectory), and $R(\tau) = \sum_{t=0}^{T}\gamma^t r_t$ denotes the finite-horizon discounted return. In policy optimization, we would like to optimize the policy by gradient ascent, i.e., by defining the following iterative update rule for the policy parameters:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)|_{\theta_k},$$

where the gradient $\nabla_\theta J(\pi_\theta)$ is referred to as *policy gradient*. The exact definition and computation of the policy gradient play a crucial role and represent the main element of distinction among policy optimization methods.

Concretely, deriving an actionable expression for the policy gradient involves two steps: 1) deriving the analytical gradient of policy performance, which turns out to have the form of an expected value, and then 2) forming a sample estimate of that expected value, which can be computed with data from a finite number of agent-environment interaction steps. For the purpose of this problem, you will be provided with (1), i.e., the analytical expression, and will have to implement (2), i.e., a tractable computation of the policy gradient based on observed quantities through interaction with the environment.

Please follow the directions in `agents/ReinforceAgent.py` to implement the `learn` function in the `ReinforceAgent` class. As a deliverable for this question, please attach a copy of your code to your pdf submission.

i. *Naive Policy Gradient.*

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(u_t \mid x_t)\, R(\tau)\right].$$

This is an expectation, which means that we can estimate it with a sample mean. If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\ldots,N}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(u_t|x_t)R(\tau),$$

where $|\mathcal{D}|$ is the number of trajectories in $\mathcal{D}$ (here, N).

ii. *Naive Policy Gradient with "causality trick".* Taking a step with the previous gradient pushes up the log-probabilities of each action in proportion to $R(\tau)$, the sum of all rewards ever obtained. Intuitively, this does not seem like a meaningful thing to do, as agents should really only reinforce actions on the basis of their consequences. Rewards obtained before taking an action have no bearing on how good that action was: only rewards that come after. Please redefine the policy gradients by introducing an element of causality:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(u_t|x_t)\sum_{t'=t}^{T}\gamma^{t'-t}R(x_{t'}, u_{t'}, x_{t'+1}).$$

iii. *Baselines in policy gradients.* Baselines allow us to reduce the variance of policy gradients "centering" the returns. Given the general expression of the policy gradient with a baseline:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(u_t|x_t) \left(\sum_{t'=t}^{T} \gamma^{t'-t} R(x_{t'}, u_{t'}, x_{t'+1}) - b(x_t) \right).$$

implement the policy gradient using the *average discounted return* as a baseline, i.e., $b(x_t) = b = \frac{1}{|\mathcal{D}|}\frac{1}{T}\sum_{\tau\in\mathcal{D}}\sum_{t'=0}^{T}\gamma^{t'} r_{t'}$. Please use γ as defined in the abstract agent class.

iv. Now, you are ready to train! Launch the training with your choice of hyperparameters and a visualization, if you'd like. Use the `plot rewards` function to visualize the history of rewards throughout training for each policy gradient variant. Comment about the difference you observe between the REINFORCE agent and the DQN agent from part (a), both in terms of performance during (e.g., sample efficiency) and after training. Also, comment on any noticeable differences between the different versions of the policy gradient.

4.4 Learning LQR. In this problem, you will study methods for learning an optimal policy for an unknown discrete-time dynamical system with an unknown cost function. All of the methods will use some form of episodic online reinforcement learning. That is, each method requires us to interact with the system, observe state transitions and incurred costs, and update a parametric policy with each observation. Specifically, at each time step t of episode i, we use the current policy to apply a control input $u_t^{(i)}$ that transitions the state from $x_t^{(i)}$ to $x_{t+1}^{(i)}$, and we incur an observed cost of $c_t^{(i)}$. Depending on the method, the policy is updated at each time step with the data $(x_t^{(i)}, u_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)})$, or at the end of the episode with all of the associated data $\{(x_t^{(i)}, u_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)})\}_{t=0}^{T-1}$, where T is the number of time steps in each episode.

Our goal is to learn a policy that minimizes the expected cost $\mathbb{E}[\sum_{t=0}^{T-1} \gamma^t c_t]$ with discount factor $\gamma \in [0, 1)$, where the expectation is taken over the initial system state, policy, and state transitions.

For this particular problem, the true underlying dynamics are linear time-invariant with the form

$$x_{t+1} = Ax_t + Bu_t,$$

with state $x_t \in \mathbb{R}^n$ and control input $u_t \in \mathbb{R}^m$. Moreover, $\mathbb{E}[x_0] = 0$ and $\mathrm{Var}[x_0] = \Sigma_x$. The true underlying stage cost function is

$$c(x, u) = x^\mathsf{T} Q x + u^\mathsf{T} R u,$$

with cost matrices $Q \succ 0$ and $R \succ 0$. To the learning agent, the matrices A, B, Q, and R are all *a priori* unknown.

We will investigate three approaches to learning an optimal policy: a model-based method, a model-free value-based method, and a model-free policy-based method. We provide high-level descriptions for each method, some references, and results below.

Model-based recursive least-squares: The first approach is a natural formulation of model identification adaptive control (MIAC). It consists of three steps performed at each time step to iteratively update estimates $(\hat{A}, \hat{B}, \hat{Q}, \hat{R})$ and compute a corresponding linear policy $u = Kx$ with parameter $K \in \mathbb{R}^{m \times n}$. Given the current state $x_t^{(i)}$, we:

1. Apply the control input $u_t^{(i)} = Kx_t^{(i)}$ and observe the incurred cost $c_t^{(i)}$ and next state $x_{t+1}^{(i)}$.
2. Fit $(\hat{A}, \hat{B})$ using recursive least-squares such that

$$(\hat{A}, \hat{B}) \leftarrow \operatorname*{arg\,min}_{A,B} \sum_{j=1}^{i-1} \sum_{\tau=0}^{T-1} \|Ax_\tau^{(j)} + Bu_\tau^{(j)} - x_{\tau+1}^{(j)}\|_2^2 + \sum_{\tau=0}^{t} \|Ax_\tau^{(i)} + Bu_\tau^{(i)} - x_{\tau+1}^{(i)}\|_2^2 + \eta(\|A\|_\mathsf{F}^2 + \|B\|_\mathsf{F}^2),$$

   where $\eta > 0$ is a regularization constant and $\|\cdot\|_\mathsf{F}$ denotes the Frobenius norm.
3. Fit $(\hat{Q}, \hat{R})$ using recursive least-squares such that

$$(\hat{Q}, \hat{R}) \leftarrow \operatorname*{arg\,min}_{Q,R} \left(\sum_{j=1}^{i-1} \sum_{\tau=0}^{T-1} \|(x_\tau^{(j)})^\mathsf{T} Q x_\tau^{(j)} + (u_\tau^{(j)})^\mathsf{T} R u_\tau^{(j)} - c_\tau^{(j)}\|_2^2 + \sum_{\tau=0}^{t} \|(x_\tau^{(i)})^\mathsf{T} Q x_\tau^{(i)} + (u_\tau^{(i)})^\mathsf{T} R u_\tau^{(i)} - c_\tau^{(i)}\|_2^2 + \eta(\|Q\|_\mathsf{F}^2 + \|R\|_\mathsf{F}^2) \right).$$

4. Solve the discounted discrete algebraic Ricatti equation

$$P = \hat{Q} + \gamma \hat{A}^\mathsf{T} P \hat{A} - \gamma^2 \hat{A}^\mathsf{T} P \hat{B} (\hat{R} + \gamma \hat{B}^\mathsf{T} P \hat{B})^{-1} \hat{B}^\mathsf{T} P \hat{A}$$

   for P, and update the policy $K \leftarrow -\gamma(\hat{R} + \gamma \hat{B}^\mathsf{T} P \hat{B})^{-1} \hat{B}^\mathsf{T} P \hat{A}$.

Model-free policy iteration (BYB94): The second approach relies on the fact that for a linear time-invariant system with a quadratic cost function, we can write the Q function for a given policy $u = Kx$ in the form

$$Q_K(x,u) = \begin{pmatrix} x \\ u \end{pmatrix}^\mathsf{T} H_K \begin{pmatrix} x \\ u \end{pmatrix} = \begin{pmatrix} x \\ u \end{pmatrix}^\mathsf{T} \begin{bmatrix} H_{K,11} & H_{K,12} \\ H_{K,12}^\mathsf{T} & H_{K,22} \end{bmatrix} \begin{pmatrix} x \\ u \end{pmatrix},$$

where the Hessian $H_K \succ 0$ is an implicit function of the current gain K. This approach proceeds by alternating two steps of generalized policy iteration:

1. *Policy evaluation:* During episode i, apply the policy $u_t^{(i)} = Kx_t^{(i)} + \varepsilon_t^{(i)}$, where $\varepsilon_t^{(i)}$ is some zero-mean i.i.d. noise meant to excite the system, and observe $(c_t^{(i)}, x_{t+1}^{(i)})$. At the end of each time step t, fit $\hat{H}_K$ for the current policy K using recursive least-squares to minimize the temporal difference error such that

$$\hat{H}_K \leftarrow \arg\min_{H_K} \sum_{\tau=0}^{t} \|Q_K(x_\tau^{(i)}, u_\tau^{(i)}) - c_\tau^{(i)} - \gamma Q_K(x_{\tau+1}^{(i)}, Kx_{\tau+1}^{(i)})\|_2^2 + \eta \|H_K\|_\mathsf{F}^2$$

2. *Policy improvement:* At the end of episode i, improve the policy via the update

$$K \leftarrow \arg\min_K Q_K(x, Kx) = -H_{K,22}^{-1} H_{K,12}^\mathsf{T}.$$

Model-free policy gradient (SB18, §13.3): The third approach uses Monte Carlo policy gradients via the REINFORCE algorithm to train a Gaussian policy distribution $\pi_K(u \mid x) \coloneqq \mathcal{N}(Kx, \Sigma)$, where $K \in \mathbb{R}^{m \times n}$ is the gain parameter and $\Sigma \succ 0$ is fixed. This approach proceeds in two steps:

1. *Policy rollout:* During episode i, apply the stochastic policy $\pi_K(u \mid x)$ and observe the data $\{(x_t^{(i)}, u_t^{(i)}, c_t^{(i)}, x_{t+1}^{(i)})\}_{t=0}^{T-1}$.
2. *Policy gradient update:* At the end of episode i, compute the empirical mean and standard deviation of *all* of the costs observed so far, i.e.,

$$\mu^{(i)} \coloneqq \frac{1}{iT} \sum_{j=1}^{i} \sum_{\tau=0}^{T-1} c_\tau^{(j)}, \quad \sigma^{(i)} \coloneqq \frac{1}{iT} \sum_{j=1}^{i} \sum_{\tau=0}^{T-1} \left(c_\tau^{(j)} - \mu^{(i)}\right).$$

Then compute the policy update

$$K \leftarrow K - \alpha \sum_{t=0}^{T-1} \gamma^t v_t^{(i)} \nabla_K \pi_K(u_t^{(i)} \mid x_t^{(i)}).$$

where $\alpha > 0$ is the learning rate, and the observed costs in computing the tail returns

$$v_t^{(i)} \coloneqq \sum_{\tau=t}^{T-1} \gamma^{\tau-t} \left(\frac{c_\tau^{(i)} - \mu^{(i)}}{\sigma^{(i)}} \right)$$

are standardized using $\mu^{(i)}$ and $\sigma^{(i)}$.

We have implemented all three algorithms and presented some results in Figure 1. It shows the difference $\|K^{(i)} - K^*\|_\mathsf{F}$ between the learned policy parameter $K^{(i)}$ and the optimal policy K^* at

the end of each episode i. Figure 1 also displays the discounted cost $\sum_{t=0}^{T-1} \gamma^t c_t^{(i)}$ at the end of each episode i alongside the expected optimal cost-to-go $\mathbb{E}_{x_0}[V^*(x_0)]$ for the true dynamics and cost function. Figure 2 shows a zoomed-in version of these plots.

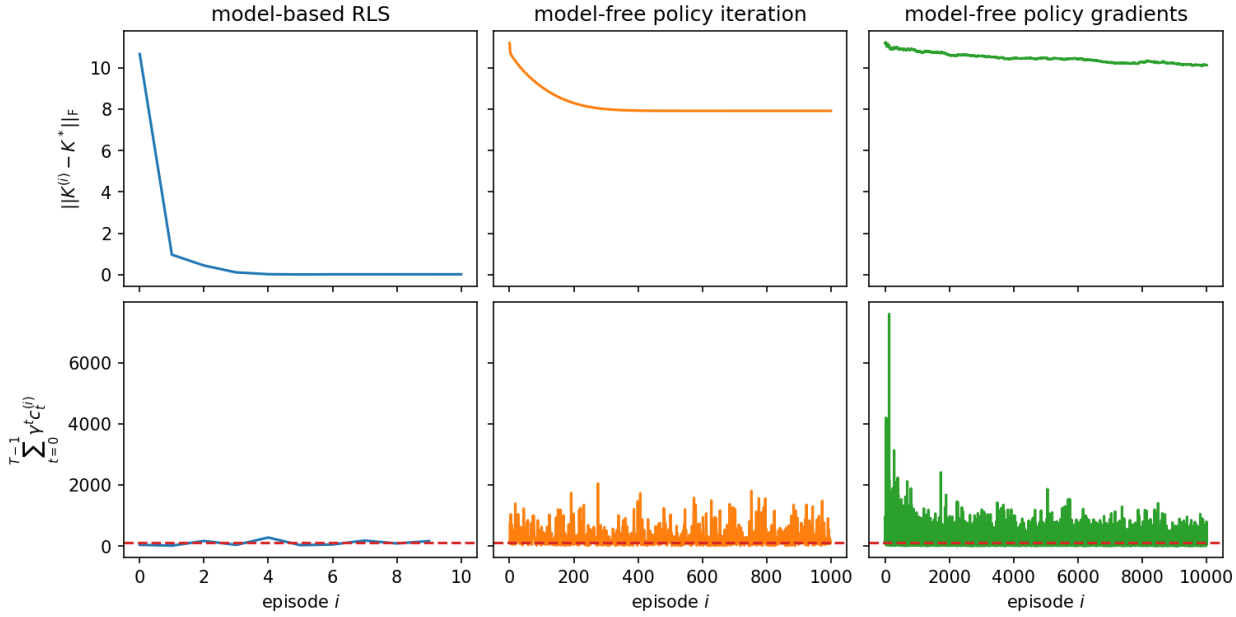

Figure 1: Gain errors and discounted cost sums for learning LQR.

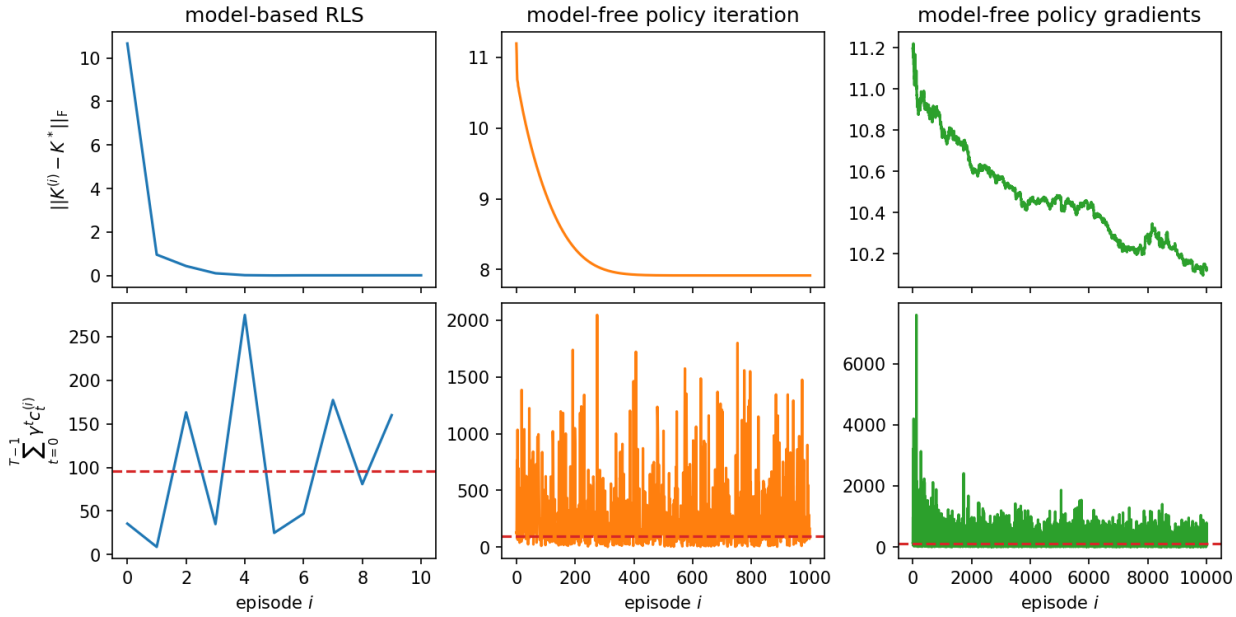

Figure 2: Gain errors and discounted cost sums for learning LQR (zoomed).

With these results in mind, your task is to answer the following conceptual questions:

(a) Each of the three described learning-based methods required access to some amount of problem-

specific information. For each of these methods, describe the problem information that was used in the design of the algorithm. How does the amount of assumptions for each of the methods compare to the performance of the methods?

(b) Suppose we had available additional information/assumptions about the problem, e.g., prior belief distributions for A, B, Q, or R. To what degree could these beliefs be incorporated into each of these learning methodologies?

4.5 Lunar lander. As we have observed in the context of learning LQR, a naïve Monte Carlo policy gradient method (i.e., REINFORCE) can be quite unstable/slowly converging due to high estimator variance. As discussed in lecture, a popular variance reduction technique is to implement an actor-critic method, in which a value function is used as a baseline (referred to as the critic, which predicts the expected return associated with actions), with the policy referred to as the actor (which selects the actions). In particular, in this problem we will consider an advantage actor-critic algorithm which, with some details omitted, is constructed as follows:

- To improve the parameters θ of the actor π_θ, we combine the Q-function policy gradient with the value function baseline and obtain the following formulation for our policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\delta^\pi \nabla_\theta \log \pi_\theta(\mathbf{u}_t \mid \mathbf{x}_t)\right], \tag{1}$$

  with δ^π an estimate of the advantage $A^\pi(\mathbf{x}, \mathbf{u}) = Q^\pi(\mathbf{x}, \mathbf{u}) - V^\pi(\mathbf{x})$.
- To improve the parameters w of the critic V_w^π (which aims to estimate V^π) we take gradient descent steps to minimize $(\delta^\pi)^2$ (note that V_w^π is used in the computation of the advantage estimate δ^π).

As RL algorithm implementations can be quite time-consuming to run/debug, in this example[3] we have provided an already complete, albeit somewhat barebones[4], implementation of an advantage actor-critic algorithm. Your task in this problem is to review the code (a task you will undoubtedly become familiar with in your professional careers, even if it seems a bit unusual for a homework assignment) and answer the following conceptual questions:

(a) What estimator δ^π of the advantage is this code using? Recalling our discussion in lecture, where does this choice fall on the spectrum of the bias-variance tradeoff?

Note: In reviewing the code, you may observe the distinction between `standardized_returns` and `returns` (the former being what's passed into `train_step_for_episode` as "returns"). For the purposes of this question, you may disregard this transformation, i.e., consider the `standardized_returns = returns`.

(b) Now considering the note from part (a), why might we wish to standardize the returns before learning?

Hint: "Learning values across many orders of magnitude" by (vHGH+16) may provide some insight.

[3] View the notebook through Google Colab to see example output even without waiting to run the code.

[4] Many improvements over the method implemented are possible and can improve performance. A non-exhaustive list is given below, feel free to experiment with these (you can also try them out on more complicated environments such as `BipedalWalker-v3`):

- Experience replay: store transitions (x_t, u_t, r_t, x_{t+1}) in a buffer. Old examples are deleted as we store new transitions. To update the parameters of our network, we sample a (mini-)batch from the buffer and perform the stochastic gradient update on this batch.
- Try out different advantage estimators, e.g., the TD advantage estimate or n-step advantage estimate that bootstrap learning with the value function.
- Play with the structure of generalized policy iteration, i.e., the current implementation does one step each (synchronously) of policy improvement and policy evaluation; this ratio could be adjusted.
- Using decoupled policy or value networks (currently they are different heads on the same trunk network).
- Implement a more recent method (e.g., Proximal Policy Optimization (PPO), Soft Actor Critic (SAC), Deep Deterministic Policy Gradient (DDPG), and many more variants/alternatives) from the deep RL literature!

(c) `jax.lax.stop_gradient` is a function that acts like the identity function (i.e., leaves its input argument unchanged) but prevents the backpropagation of gradients through itself. That is, `stop_gradient` makes its input argument appear like a constant, as far as autodifferentiation is concerned. What is the significance of using this function in computing the `actor_loss` in `train_loss_for_episode`?

(d) Considering now the broader landscape of optimal and learning control, how else might you approach this lunar lander problem? In a short paragraph (maximum 4-5 sentences), sketch out an alternative approach for solving the lunar lander problem and discuss the strengths/weaknesses you imagine it having vs. the model-free RL approach here. Feel free to make whatever modeling assumptions you'd like (i.e., known vs. unknown dynamics, deterministic vs. stochastic dynamics, known vs. unknown reward, etc.).

References

[BYB94] S. J. Bradtke, B. E. Ydstie, and A. G. Barto, *Adaptive linear quadratic control using policy iteration*, American Control Conference, 1994.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller, *Playing atari with deep reinforcement learning*, ArXiv **abs/1312.5602** (2013).

[SB18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2 ed., MIT Press, 2018, Available at: http://incompleteideas.net/book/the-book-2nd.html.

[vHGH+16] H. van Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, *Learning values across many orders of magnitude*, Conf. on Neural Information Processing Systems, 2016, Available at: https://arxiv.org/abs/1602.07714.

[Wil92] Ronald J. Williams, *Playing atari with deep reinforcement learning*, Machine Learning **8** (1992).