Recitation 9: Homework 4 Review

Alex Li, Alvin Shek

Problem 1: LQR

LQR Intro

General Discrete Finite Time:

Objective:

$$J = \sum_{k=0}^{N-1} l_k(x_k, u_k) + l_N(x_N)$$

Dynamics:

$$x_{k+1} = f(x_k, u_k)$$

"Regulator": generate controls to minimize a cost function

Linear Quadratic Regulator (LQR):

Quadratic Objective:

$$J = \sum_{k=0}^{N-1} [x_k^T Q_k x_k + u_k^T R_k u_k] + x_N^T S_N x_N$$

Linear Dynamics:

$$x_{k+1} = Ax_k + Bu_k$$

$Q_0, \ldots, Q_{N-1}\ \ S_N$	symmetric positive semi-definite
$R_0, \ldots, R_{N-1}$	symmetric positive definite
A_k, B_k	controllable

Above equations taken from Prof. Changliu Liu's course: Adaptive Controls and Reinforcement Learning 16-899

Problem 1: LQR

Consider continuous-time for HW:

$$J = \frac{1}{2}\int_{t=0}^{\infty} [x^T Q x + u^T R u] dt$$

$$\dot{x} = Ax + Bu$$

Let's look at an example double integrator:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} [u]$$

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad R = [1]$$

Non-zero setpoint:
Define a new coordinate system to drive to 0

$$x = \hat{x} - x^*$$
$$\dot{x} = (\dot{\hat{x} - x^*})$$
$$= \dot{\hat{x}} - 0$$
$$= \dot{\hat{x}}$$

$$\dot{\hat{x}} = A(\hat{x} - x^*) + Bu$$

LQR solves for optimal control in this modified coordinate system when we use "u = -K(x - x*)". Note for fixed A and B:

```
# Independent of the current state x or goal x*!
S = linalg.solve_continuous_are(A, B, Q, R)
K = linalg.inv(R) @ B.T @ S
```

We then apply this control to our original system. Overall, the **dynamics do not change. We simply changed the coordinate system.**

Problem 1: LQR

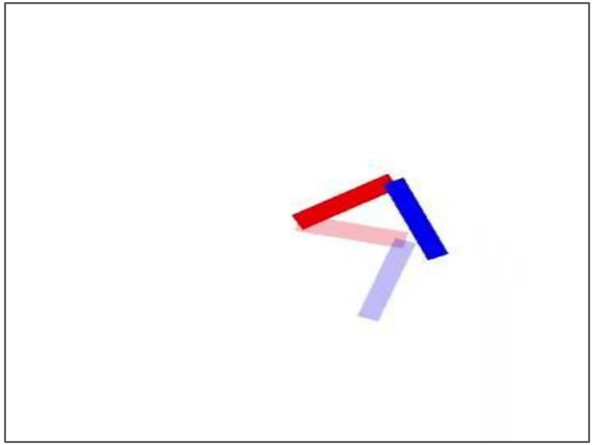
Nonlinear dynamics in **arm_env.py**:

```
# LQR does not support state/control constraints, so manually clip the output:
u = np.clip(u, self.action_space.low, self.action_space.high)

C2 = np.cos(self.q[1])
S2 = np.sin(self.q[1])
M11 = (self.K1 + self.K2 * C2)
M12 = (self.K3 + self.K4 * C2)
M21 = M12
M22 = self.K3
H1 = (-self.K2 * S2 * self.dq[0] * self.dq[1] -
      1 / 2.0 * self.K2 * S2 * self.dq[1] ** 2.0)
H2 = 1 / 2. * self.K2 * S2 * self.dq[0] ** 2.

ddq1 = ((H2 * M11 - H1 * M21 - M11 * u[1] + M21 * u[0]) /
        (M12 ** 2. - M11 * M22))
ddq0 = (-H2 + u[1] - M22 * ddq1) / M21

self.dq += np.array([ddq0, ddq1]) * dt
self.q += self.dq * dt
self.t += dt
```



Problem 1: LQR

What do we need to do?

Linearize the nonlinear dynamics at each step:

$$f(x,u) \approx f(\tilde{x},\tilde{u}) + D_x f(\tilde{x},\tilde{u})(x-\tilde{x}) + D_u f(\tilde{x},\tilde{u})(u-\tilde{u})$$

$$f(x,u) - f(\tilde{x},\tilde{u}) \approx \tilde{A}(x-\tilde{x}) + \tilde{B}(u-\tilde{u})$$

$$\dot{\delta} \approx \tilde{A}(x-\tilde{x}) + \tilde{B}(u-\tilde{u})$$

We can directly apply this approximated A and B to LQR! Solving for optimal control at the current point, A and B only valid around this point! More details in piazza post @350, or come up to ask afterwards.

Now, how do we approximate the Jacobians? $D_x f \; D_u f$

Problem 1: LQR

Central Differences, approx derivative:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Jacobian:

$$\begin{bmatrix} \frac{\partial f_1}{x_1} & \frac{\partial f_1}{x_2} & \cdots & \frac{\partial f_1}{x_m} \\ \frac{\partial f_2}{x_1} & \ddots & & \\ \vdots & & & \\ \frac{\partial f_n}{x_1} & & & \frac{\partial f_n}{x_m} \end{bmatrix}$$

In this problem (similar logic for B):

$$\tilde{A} = D_x f = \begin{bmatrix} \frac{\partial f_1}{x_1} & \frac{\partial f_1}{x_2} & \frac{\partial f_1}{x_3} & \frac{\partial f_1}{x_4} \\ \frac{\partial f_2}{x_1} & \frac{\partial f_2}{x_2} & \frac{\partial f_2}{x_3} & \frac{\partial f_2}{x_4} \\ \frac{\partial f_3}{x_1} & \frac{\partial f_3}{x_2} & \frac{\partial f_3}{x_3} & \frac{\partial f_3}{x_4} \\ \frac{\partial f_4}{x_1} & \frac{\partial f_4}{x_2} & \frac{\partial f_4}{x_3} & \frac{\partial f_4}{x_4} \end{bmatrix}$$

(4 x 4)

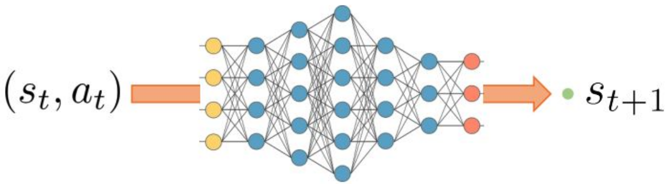
f(x, u) = simulate_dynamics(env, x, u)

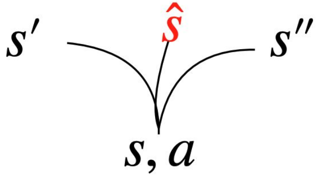
But how to calculate partial derivative with respect to only one variable in input?...

Problem 2: PETS

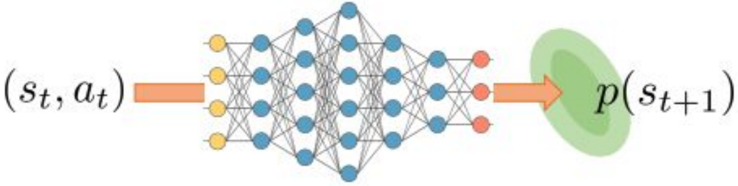
Probabilistic Models

- One way to train a model: directly predict the next state, then minimize MSE





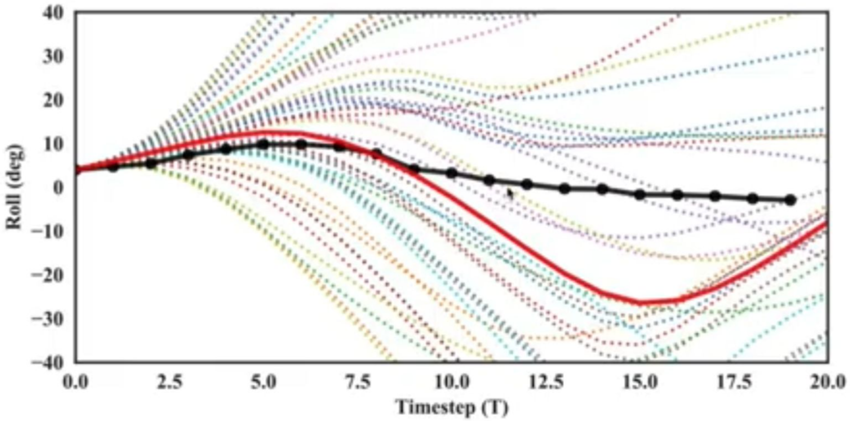
- Probabilistic model: predict the **parameters of a distribution for the next state!**
  - Usually mean and log-variance for continuous-space models
  - Maximize log-probability of the next state



$$\text{loss}_{\text{Gauss}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} [\mu_{\boldsymbol{\theta}}(\boldsymbol{s}_n, \boldsymbol{a}_n) - \boldsymbol{s}_{n+1}]^\top \boldsymbol{\Sigma}_{\boldsymbol{\theta}}^{-1}(\boldsymbol{s}_n, \boldsymbol{a}_n) [\mu_{\boldsymbol{\theta}}(\boldsymbol{s}_n, \boldsymbol{a}_n) - \boldsymbol{s}_{n+1}] + \log\det \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\boldsymbol{s}_n, \boldsymbol{a}_n)$$

MPC vs open-loop control

- Open-loop: plan once all the way to the end of the episode, then execute all of the actions without looking at the subsequent states
  - Very fast, but fails if predictions are wrong
- Model-predictive control (MPC): make plan, then execute the **first action in the plan**. Replan starting from the next state
  - Can handle (small) errors in the model, but is computationally expensive

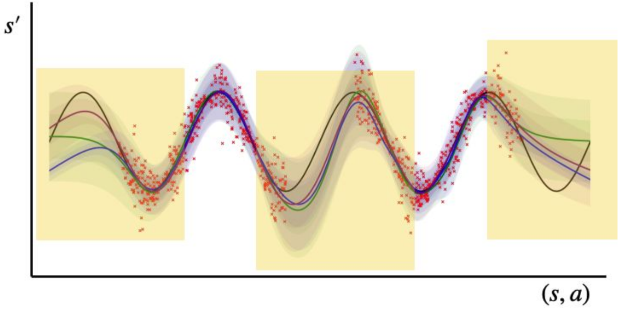


CEM vs random sampling for planning

- How do we actually come up with a plan that looks good?
- Random sampling:
  - Sample N trajectories, starting from the current state, using your model to generate transitions
  - Pick the one with the highest cumulative reward (or lowest cost)!
- CEM:
  - Sample N trajectories, starting from the current state, using your model to generate transitions
  - Take the elites and fit a mean and diagonal covariance matrix to the elites
  - Use this action distribution to sample N trajectories again
  - Do this K times.

PETS recap

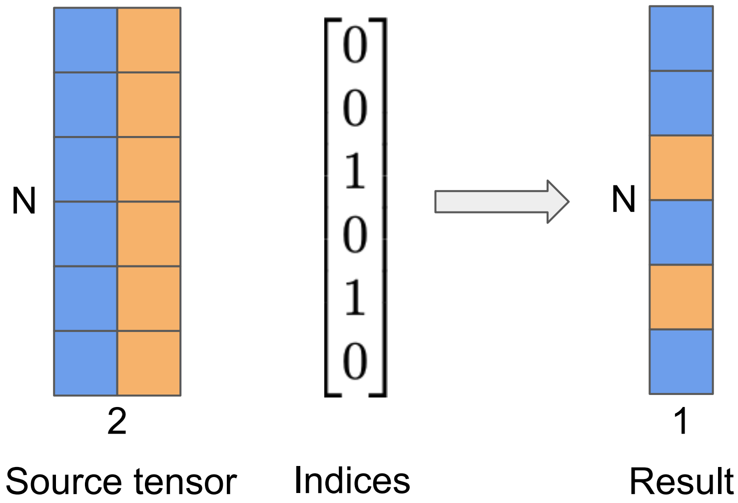
- Very similar to MPC with CEM, but with several twists:
- Train an ensemble of probabilistic models
  - Each network in the ensemble starts from a different initialization
  - Train each network with its own minibatch from the replay buffer
- Transitions sampled from the model have two sources of stochasticity:
  - Choosing a random network from the model ensemble captures epistemic uncertainty (not enough data to be certain about transition)



  - Sampling a transition from the distribution that the network outputs
    - Captures aleatoric uncertainty (environment is fundamentally stochastic)

Tips for vectorized indexing

- Let's say we have a Nx2 tensor, and we want to get a Nx1 tensor by indexing with a Nx1 index tensor (each entry is either 0 or 1). Best way to do this?
- Pictorially:



Tips for vectorized indexing

```
x = torch.arange(32768).reshape(1024, 2, 16)
idx = np.random.choice(x.shape[1], size=x.shape[0])
```

Approach #1 (naive): for loop

```
%%time
result = []
for i in range(x.shape[0]):
    result.append(x[i, idx[i],:])
```

```
CPU times: user 7.64 ms, sys: 135 µs, total: 7.77 ms
Wall time: 7.72 ms
```

Approach #2: vectorized

```
%%time
result = x[np.arange(x.shape[0]), idx,:]
```

```
CPU times: user 812 µs, sys: 510 µs, total: 1.32 ms
Wall time: 762 µs
```