UC San Diego

Introduction to Deep Reinforcement Learning

Model-based Methods

Zhiao Huang

Model-free Methods

- The model $p(s'|s, a)$ is unknown
    - we solve Q, V and the policy from the sampled trajectories/transitions

Model-based Method

- Learn the environment model directly

$$p(s'|s,a)$$

- Learn $R(s,a,s')$ if it's unknown

Model-based Method

- Learn the environment model directly by supervised learning

$$p(s'|s,a)$$

- **Search** the solution with the model directly

Model-based Methods

- Model and search have broad meanings

Model	Search
Physics Geometry Probability model Inverse Dynamics Game Engine	MCTS CEM RL iLQR RRT/PRM ......

Model-Predictive Control

- Forward model with parameters θ

$$f_\theta(s, a): S \times A \to S$$

  - Predicts what will happen if we execute the action a at the state s

Forward Model

- We may have a forward model in mind

Learning the Forward Model

- Sample transitions (s, a, s') from the replay buffer and train the model with **supervised learning**

$$\min_{\theta} E[\|f_\theta(s,a) - s'\|^2]$$

Rollout

- Predict a short trajectory $s_1, s_2, \dots, s_T$ if we start at s_0 and execute $a_0, a_1, a_2, \dots, a_{T-1}$ sequentially

$$s_0 \xrightarrow{f_\theta(s_0, a_0)} s_1 \xrightarrow{f_\theta(s_1, a_1)} s_2 \xrightarrow{f_\theta(s_2, a_2)} \cdots \xrightarrow{f_\theta(s_{T-1}, a_{T-1})} s_T$$

"rollout" the forward model

Model-Predictive Control

- Given the forward model f_θ and the current state s_0, find a sequence of action $a_0, a_1, a_2, \ldots, a_{T-1}$ such that has the maximum reward

$$\max_{a_{0:T-1}} \sum_{i=1}^{T} R(s_i, a_i, s_{i+1}) \text{ s.t. } s_{i+1} = \mathrm{f}_\theta(s_i, a_i)$$

Random Shooting

- Sample N random action sequences:

$$a_0^1, a_1^1, a_2^1, \ldots, a_{T-1}^1$$

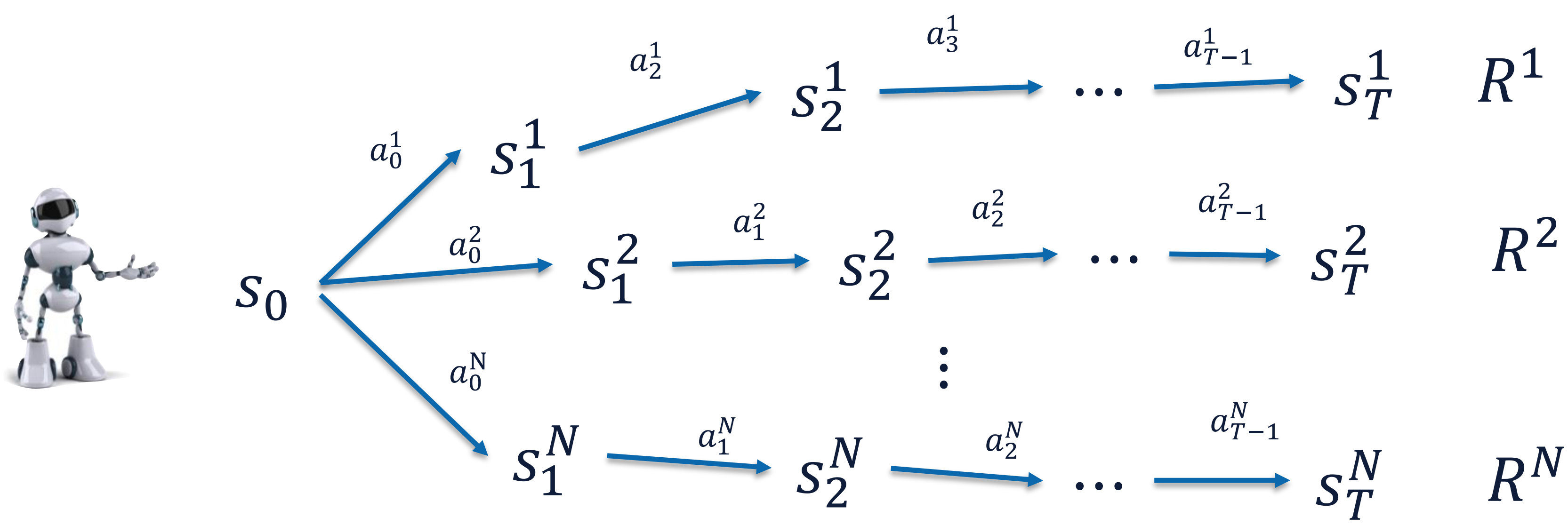
$$a_0^2, a_1^2, a_2^2, \ldots, a_{T-1}^2$$

$$\vdots$$

$$a_0^1, a_1^1, a_2^1, \ldots, a_{T-1}^1$$

Random Shooting

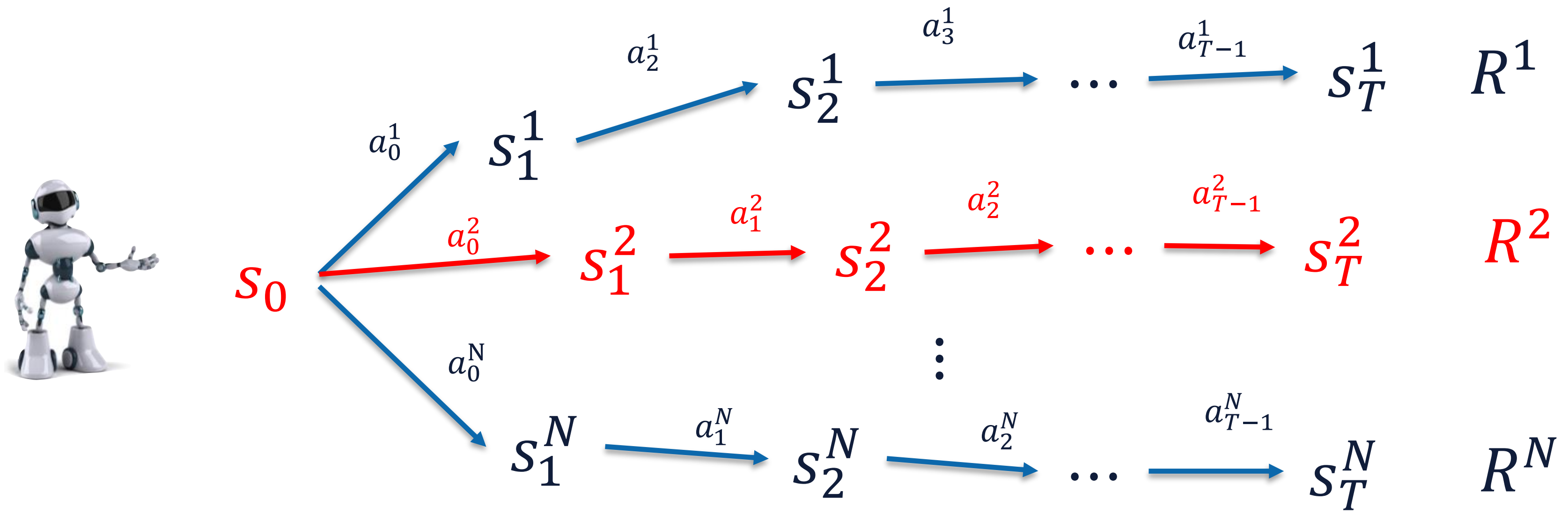
- Evaluate the reward of each action sequence by simulating the model f_θ



“rollout” the forward model $f_\theta(s, a)$

Random Shooting

- Return the best action sequence $a^*_{0:T-1}$ and execute in the real environment



Planning at Each Step

If we execute the searched action sequence $a_0, a_1, a_2, \ldots$ in the environment

$s_0 \dashrightarrow s_1'$ Model

$s_0 \rightarrow s_1$ Reality

- The action sequence $a_1, a_2, \ldots, a_{T-1}$ maximize the reward from $s_1' = f_\theta(s_0, a_0)$ but not s_1
- Search new actions for state s_1 again!

Model Predictive Control

- Repeat
  - Observe the current state s
  - Sample N random action trajectories
  - Evaluate the reward of each action sequence from s with the model f_θ; Find the best action sequence $\{a_0^k, a_1^k, \dots, a_{T-1}^k\}$
  - Execute a_0^k in the environment

Cross-Entropy Method

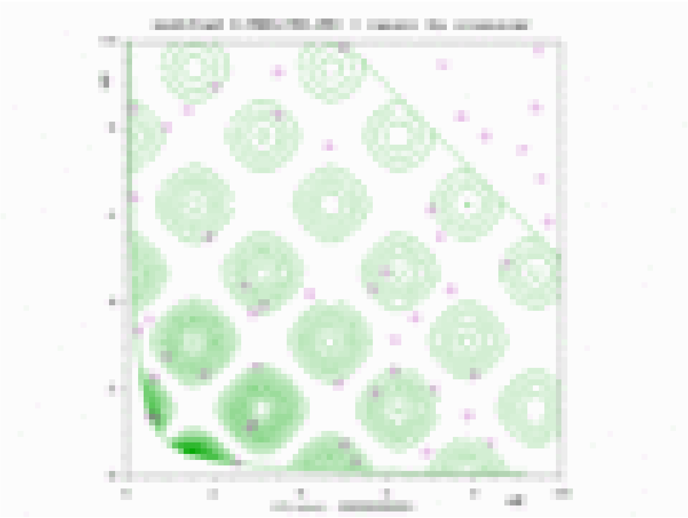
- Black-box Function Optimization

$$\max_{x} f(x)$$

- We have many other choices
  - Cross Entropy Method

Cross-Entropy Method

- Basically the simplest evolutionary algorithm
- Maintain the distribution of solutions



Cross-Entropy Method

- Initialize $\mu \in \mathbb{R}^d, \sigma \in \mathbb{R}^d_{>0}$
- For iteration = $1,2, \ldots$
  - Sample n candidates $x_i \sim N(\mu, \mathrm{diag}(\sigma^2))$
  - For each x_i evaluate its value $f(x_i)$
  - Select the top k of x as elites
  - Fit a new diagonal Gaussian to those samples and update μ, σ

Cross-Entropy Method (in Python)

```
def cem(f, mean, std, num_iter=10, population_size=100, elite_size=20):
    for i in range(num_iter):
        populations = np.array([np.random.normal() * std + mean for j in range(population_size)])
        values = np.array([f(j) for j in populations])
        elites = populations[values.argsort()[-elite_size:]]
        mean, std = elites.mean(), elites.std()
    return mean
```

Model Predictive Control

- Hyper parameters

$$\mu_a, \sigma_a, n_{\text{iter}}, n_{\text{pop}}, n_{\text{elite}}$$

- Initialize an action sequence $\mu = \{a_i = \mu_a\}_{i<T}$
- Repeat
  - Observe the current state s
  - Search the new action sequence with CEM
    $\{a'_0, a'_1, \dots, a'_{T-1}\} = \text{CEM}(\mu, \{\sigma_a\}_{i<T})$
  - Execute a'_0 in the environment
  - Update $\mu \leftarrow \{a'_1, a'_2, \dots, a'_{T-1}, \mu_a\}$

Model Predictive Control

- Hyper parameters

$$\mu_a, \sigma_a, n_{\text{iter}}, n_{\text{pop}}, n_{\text{elite}}$$

```python
def cem(f, mean, std, num_iter=10, population_size=100, elite_size=20):
    for i in range(num_iter):
        populations = np.array([np.random.normal() * std + mean for j in range(population_size)])
        values = np.array([f(j) for j in populations])
        elites = populations[values.argsort()[-elite_size:]]
        mean, std = elites.mean(), elites.std()
    return mean
```

- Initialize
- Repeat
  - Obse
  - Search the new ac… sequence with CEM

$$\{a'_0, a'_1, \dots, a'_{T-1}\} = \text{CEM}(\mu, \{\sigma_a\}_{i<T})$$

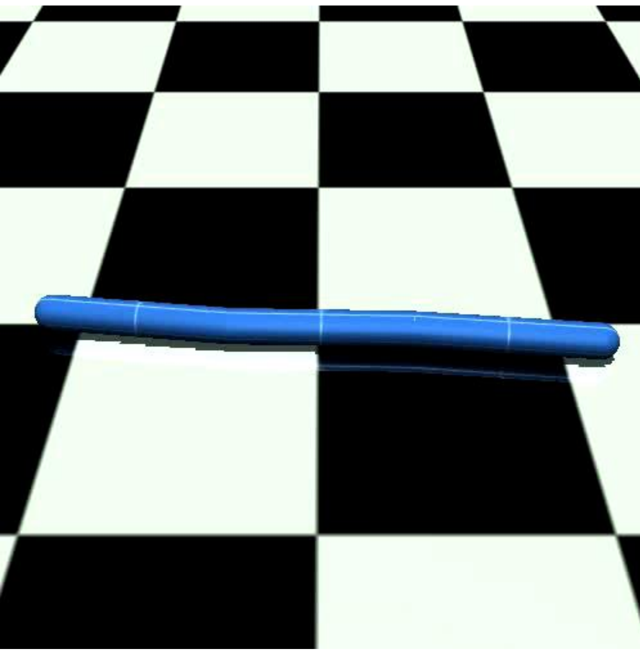
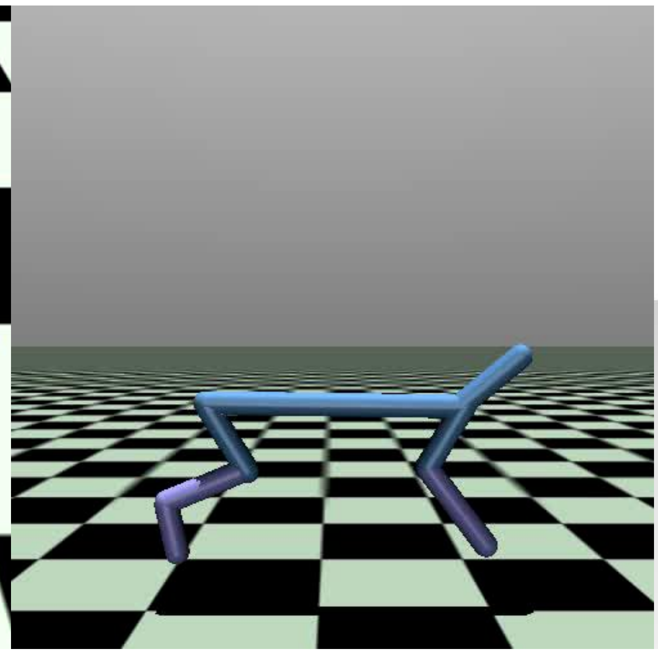
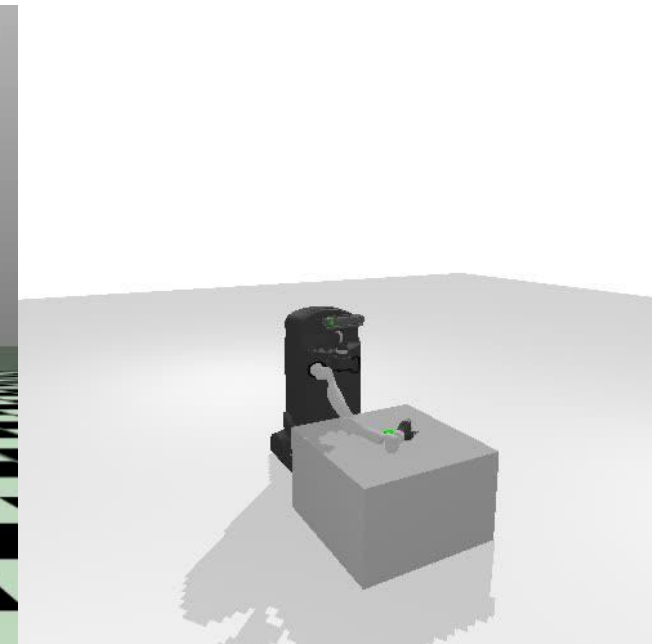
  - Execute a'_0 in the environment
  - Update $\mu \leftarrow \{a'_1, a'_2, \dots, a'_{T-1}, \mu_a\}$

Notes

- CEM performs well for most control tasks
- Instead of searching for action sequence, we can also search for the parameters of the network
- General "Gradient Descent"
- CEM and Random shooting work poorly for very long horizons T or dense reward

Performance of CEM

- When the model is known

Comparisons

- On-policy methods: Policy
- Off-policy methods: Value/Q
- Model-based methods: Model

$$\text{Model} \Rightarrow \text{Value} \Rightarrow \text{Policy}$$

Comparisons

- How difficult to model it
  - Q Value > Policy
  - Model depends on the priors

- Robustness
  - Model < Q Value < Policy

- Time complexity
  - Model > Q/Policy

- Data-efficient/Generalization
  - Model > Q Value > Policy

Conclusion

- Very few data / We know the model well
  - Model-based methods
- We can’t model the environment and we don’t want to sample too much
  - Off-policy methods
- We have enough time/money
  - Off-policy + On-policy methods