Model-Based Policy Learning

CS 285

Instructor: Sergey Levine
UC Berkeley
model-based reinforcement learning version 1.5:

1. run base policy $\pi_0(a_t|s_t)$ (e.g., random policy) to collect $D = \{(s, a, s')_i\}$
2. learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s'_i\|^2$
3. plan through $f(s, a)$ to choose actions
4. execute the first planned action, observe resulting state $s'$ (MPC)
5. append $(s, a, s')$ to dataset $D$
The stochastic open-loop case

\[ p_\theta(s_1, \ldots, s_T|a_1, \ldots, a_T) = p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) \]

\[ a_1, \ldots, a_T = \arg \max_{a_1, \ldots, a_T} E \left[ \sum_t r(s_t, a_t)|a_1, \ldots, a_T \right] \]

why is this suboptimal?
The stochastic **closed**-loop case

\[
p(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t)p(s_{t+1}|s_t, a_t)
\]

\[
\pi = \arg \max_{\pi} E_{\tau \sim p(\tau)} \left[ \sum_t r(s_t, a_t) \right]
\]
Backpropagate directly into the policy?

model-based reinforcement learning version 2.0:

1. run base policy \( \pi_0(\mathbf{a}_t|\mathbf{s}_t) \) (e.g., random policy) to collect \( \mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\} \)

2. learn dynamics model \( f(\mathbf{s}, \mathbf{a}) \) to minimize \( \sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2 \)

3. backpropagate through \( f(\mathbf{s}, \mathbf{a}) \) into the policy to optimize \( \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \)

4. run \( \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \), appending the visited tuples \( (\mathbf{s}, \mathbf{a}, \mathbf{s}') \) to \( \mathcal{D} \)
What’s the problem with backprop into policy?

$r(s_t, a_t)$

$a_t = \pi_\theta(s_t)$

$s_{t+1} = f(s_t, a_t)$

$r(s_{t+1}, a_{t+1})$

$a_{t+1} = \pi_\theta(s_{t+1})$

$s_{t+2} = f(s_{t+1}, a_{t+1})$

big gradients here

small gradients here
What’s the problem with backprop into policy?

\[ r(s_t, a_t) \]
\[ a_t = \pi_\theta(s_t) \]
\[ s_{t+1} = f(s_t, a_t) \]
\[ r(s_{t+1}, a_{t+1}) \]
\[ a_{t+1} = \pi_\theta(s_{t+1}) \]
\[ s_{t+2} = f(s_{t+1}, a_{t+1}) \]
What’s the problem with backprop into policy?

- Similar parameter sensitivity problems as shooting methods
  - But no longer have convenient second order LQR-like method, because policy parameters couple all the time steps, so no dynamic programming
- Similar problems to training long RNNs with BPTT
  - Vanishing and exploding gradients
  - Unlike LSTM, we can’t just “choose” a simple dynamics, dynamics are chosen by nature
What’s the solution?

• Use derivative-free (“model-free”) RL algorithms, with the model used to generate synthetic samples
  • Seem weirdly backwards
  • Actually works very well
  • Essentially “model-based acceleration” for model-free RL
Model-Free Learning With a Model
Model-free optimization with a model

Policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta (a_{i,t}|s_{i,t}) \hat{Q}_{i,t}^\pi$$

Backprop (pathwise) gradient:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \frac{d a_t}{d \theta} \frac{d s_{t+1}}{d a_t} \left( \sum_{t'=t+1}^{T} \frac{d r_{t'}}{d s_{t'}} \left( \prod_{t''=t+2}^{t'} \frac{d s_{t''}}{d a_{t''-1}} \frac{d a_{t''-1}}{d s_{t''-1}} + \frac{d s_{t''}}{d s_{t''-1}} \right) \right)$$

- Policy gradient might be more *stable* (if enough samples are used) because it does not require multiplying many Jacobians
- See a recent analysis here:
  - Parmas et al. ‘18: PIPP: Flexible Model-Based Policy Search Robust to the Curse of Chaos
Model-based RL via policy gradient

model-based reinforcement learning version 2.5:

1. run base policy \( \pi_0(a_t|s_t) \) (e.g., random policy) to collect \( D = \{(s, a, s')\} \)
2. learn dynamics model \( f(s, a) \) to minimize \( \sum_i \| f(s_i, a_i) - s'_i \|^2 \)
3. use \( f(s, a) \) to generate trajectories \( \{\tau_i\} \) with policy \( \pi_\theta(a|s) \)
4. use \( \{\tau_i\} \) to improve \( \pi_\theta(a|s) \) via policy gradient
5. run \( \pi_\theta(a_t|s_t) \), appending the visited tuples \( (s, a, s') \) to \( D \)

What’s a potential **problem** with this approach?
The curse of long model-based rollouts

How quickly does error accumulate?

$O(\epsilon T^2)$
How to get away with short rollouts?

- huge accumulating error
+ much lower error
- never see later time steps
+ see all time steps
- wrong state distribution
Model-based RL with short rollouts

model-based reinforcement learning version 3.0:

1. run base policy $\pi_0(a_t|s_t)$ (e.g., random policy) to collect $D = \{(s, a, s')_i\}$
2. learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s'_i\|^2$
3. pick states $s_i$ from $D$, use $f(s, a)$ to make short rollouts from them
4. use both real and model data to improve $\pi_\theta(a|s)$ with off-policy RL
5. run $\pi_\theta(a_t|s_t)$, appending the visited tuples $(s, a, s')$ to $D$
Dyna-Style Algorithms
Model-based RL with short rollouts

model-based reinforcement learning version 3.0:

1. run base policy $\pi_0(a_t|s_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$

2. learn dynamics model $f(s, a)$ to minimize $\sum_i \| f(s_i, a_i) - s'_i \|^2$

3. pick states $s_i$ from $\mathcal{D}$, use $f(s, a)$ to make short rollouts from them

4. use both real and model data to improve $\pi_\theta(a|s)$ with off-policy RL

5. run $\pi_\theta(a_t|s_t)$, appending the visited tuples $(s, a, s')$ to $\mathcal{D}$
Model-free optimization with a model

Dyna

online Q-learning algorithm that performs model-free RL with a model

1. Given state $s$, pick action $a$ using exploration policy
2. Observe $s'$ and $r$, to get transition $(s, a, s', r)$
3. Update model $\hat{p}(s'|s, a)$ and $\hat{r}(s, a)$ using $(s, a, s')$
4. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s', r}[r + \max_{a'} Q(s', a') - Q(s, a)]$
5. Repeat $K$ times:
   6. Sample $(s, a) \sim B$ from buffer of past states and actions
   7. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s', r}[r + \max_{a'} Q(s', a') - Q(s, a)]$

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming.
General “Dyna-style” model-based RL recipe

1. collect some data, consisting of transitions \((s, a, s', r)\)
2. learn model \(\hat{p}(s'|s, a)\) (and optionally, \(\hat{r}(s, a)\))
3. repeat K times:
   4. sample \(s \sim \mathcal{B}\) from buffer
   5. choose action \(a\) (from \(\mathcal{B}\), from \(\pi\), or random)
   6. simulate \(s' \sim \hat{p}(s'|s, a)\) (and \(r = \hat{r}(s, a)\))
   7. train on \((s, a, s', r)\) with model-free RL
   8. (optional) take \(N\) more model-based steps

+ only requires short (as few as one step) rollouts from model
+ still sees diverse states
Model-accelerated off-policy RL

process 1: data collection

\((s, a, s', r)\)

\(\pi(a|s)\) (e.g., \(\epsilon\)-greedy)

process 2: target parameter update

current parameters \(\phi\)

target parameters \(\phi'\)

process 3: Q-function regression

process 4: model training

process 5: model data collection

buffer of model-based transitions

dataset of transitions ("replay buffer")

rollout start state \(s \sim B\)

evict each time model changes

evict old data

Model-accelerated off-policy RL

The diagram illustrates the flow of data and processes in model-accelerated off-policy reinforcement learning (RL). The process begins with data collection (process 1) where states, actions, next states, and rewards are gathered. This data is then stored in a dataset referred to as the “replay buffer” (process 3), which is used for Q-function regression. The model training (process 4) uses the replay buffer to learn a model that is subsequently used to collect new data (process 5). The target parameters are updated (process 2) to improve model performance, and the model training continuously adapts to changes in the environment or model.
Model-Based Acceleration (MBA)
Model-Based Value Expansion (MVE)
Model-Based Policy Optimization (MBPO)

1. take some action $a_i$ and observe $(s_i, a_i, s'_i, r_i)$, add it to $B$
2. sample mini-batch $\{s_j, a_j, s'_j, r_j\}$ from $B$ uniformly
3. use $\{s_j, a_j, s'_j\}$ to update model $\hat{p}(s'|s,a)$
4. sample $\{s_j\}$ from $B$
5. for each $s_j$, perform model-based rollout with $a = \pi(s)$
6. use all transitions $(s, a, s', r)$ along rollout to update $Q$-function

\textbf{+ why is this a good idea?}
\textbf{- why is this a bad idea?}

Gu et al. Continuous deep Q-learning with model-based acceleration. ‘16
Feinberg et al. Model-based value expansion. ’18
Janner et al. When to trust your model: model-based policy optimization. ‘19
Multi-Step Models & Successor Representations
What kind of model do we need to evaluate a policy?

The job of the model is to evaluate the policy (if you can evaluate it, you can make it better)

$$J(\pi) = E_{s \sim p(s_1)}[V^\pi(s_1)]$$

$$V^\pi(s_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} E_{p(s_{t'}|s_t)} E_{a_{t'} \sim \pi(a_{t'}|s_{t'})}[r(s_{t'}, a_{t'})]$$

$$= \sum_{t'=t}^{\infty} \gamma^{t'-t} E_{p(s_{t'}|s_t)}[r(s_{t'})]$$

$$= \sum_{t'=t}^{\infty} \gamma^{t'-t} \sum_s p(s_{t'} = s|s_t) r(s)$$

$$= \sum_s \left( \sum_{t'=t}^{\infty} \gamma^{t'-t} p(s_{t'} = s|s_t) \right) r(s)$$

fit model $f(s, a)$

generate samples (i.e. run the policy)

fit a model to estimate return

improve the policy
What kind of model do we need to evaluate a policy?

$$V^\pi(s_t) = \sum_{t=t'}^{\infty} \gamma^{t'-t} E_{p(s_{t'}|s_t)}[r(s_{t'})]$$

$$= \sum_s \left( \sum_{t=t'}^{\infty} \gamma^{t'-t} p(s_{t'} = s|s_t) \right) r(s)$$

$$p_\pi(s_{\text{future}} = s|s_t) = (1 - \gamma) \sum_{t'=t}^{\infty} \gamma^{t'-t} p(s_{t'} = s|s_t)$$

just to ensure it sums to 1

(if you can evaluate it, you can make it better)

fit model \( f(s, a) \)

fit a model to estimate return

generate samples (i.e. run the policy)

improve the policy
What kind of model do we need to **evaluate** a policy?

\[ p_\pi(s_{\text{future}} = s | s_t) = (1 - \gamma) \sum_{t' = t}^{\infty} \gamma^{t' - t} p(s_{t'} = s | s_t) \]

\[ V^\pi(s_t) = \frac{1}{1 - \gamma} \sum_s p_\pi(s_{\text{future}} = s | s_t) r(s) \]

\[ \mu^\pi(s_t) = p_\pi(s_{\text{future}} = i | s_t) \]

(if you can evaluate it, you can make it better)

fit model \( f(s, a) \)

fit a model to estimate return

generate samples (i.e. run the policy)

improve the policy

This is called a **successor representation**

Successor representations

\[
\mu_i^\pi(s_t) = (1 - \gamma) \sum_{t'=t}^{\infty} \gamma^{t'-t} p(s_{t'} = i | s_t)
\]

\[
= (1 - \gamma) \delta(s_t = i) + \gamma E_{a_t \sim \pi(a_t | s_t), s_{t+1} \sim p(s_{t+1} | s_t, a_t)} [\mu_i^\pi(s_{t+1})]
\]

like a Bellman backup with “reward” \( r(s_t) = (1 - \gamma) \delta(s_t = i) \)

in practice, we can use vectorized backups for all \( i \) at once

A few issues...

- Not clear if learning successor representation is easier than model-free RL
- How to scale to large state spaces?
- How to extend to continuous state spaces?
Successor features

\[ \mu_i^\pi(s_t) = (1 - \gamma) \sum_{t' = t}^{\infty} \gamma^{t' - t} p(s_{t'} = i | s_t) \]

\[ \psi_j^\pi(s_t) = \sum_s \mu_s^\pi(s_t) \phi_j(s) \]

\[ \psi_j^\pi(s_t) = \mu^\pi(s_t)^T \vec{\phi}_j \]

\[ V^\pi(s_t) = \mu^\pi(s_t)^T \vec{r} \]

so what?

If the number of features is much less than the number of states, learning them is much easier!
Successor features

\[
\mu^\pi_i(s_t) = (1 - \gamma)\delta(s_t = i) + \gamma E_{a_t \sim \pi(a_t|s_t), s_{t+1} \sim p(s_{t+1}|s_t, a_t)} \left[ \mu^\pi_i(s_{t+1}) \right]
\]

\[
\psi^\pi_j(s_t) = \phi_j(s_t) + \gamma E_{a_t \sim \pi(a_t|s_t), s_{t+1} \sim p(s_{t+1}|s_t, a_t)} \left[ \psi^\pi_j(s_{t+1}) \right]
\]

can also construct a “Q-function-like” version:

\[
\psi^\pi_j(s_t, a_t) = \phi_j(s_t) + \gamma E_{s_{t+1} \sim p(s_{t+1}|s_t, a_t), a_{t+1} \sim \pi(a_{t+1}|s_{t+1})} \left[ \psi^\pi_j(s_{t+1}, a_{t+1}) \right]
\]

\[
Q^\pi(s_t, a_t) \approx \psi^\pi(s_t, a_t)^T w \quad \text{when} \ r(s_t) \approx \phi(s_t)^T w
\]

special case with \( \phi_i(s_t) = (1 - \gamma)\delta(s_t = i) \)
Using successor features

**Idea 1:** recover a Q-function very quickly

1. Train $\psi^\pi(s_t, a_t)$ (via Bellman backups)
2. Get some reward samples $\{s_i, r_i\}$
3. Get $w \leftarrow \arg\min_w \sum_i \|\phi(s_i)^T w - r_i\|^2$
4. Recover $Q^\pi(s_t, a_t) \approx \psi^\pi(s_t, a_t)^T w$

$$\pi'(s) = \arg\max_a \psi^\pi(s, a)^T w$$

Is this the **optimal** Q-function?

Equivalent to **one step** of policy iteration

Better than nothing, but **not** optimal
Using successor features

**Idea 2:** recover many Q-functions

1. Train $\psi^{\pi_k}(s_t, a_t)$ for many policies $\pi_k$ (via Bellman backups)

2. Get some reward samples $\{s_i, r_i\}$

3. Get $w \leftarrow \text{arg min}_w \sum_i \|\phi(s_i)^T w - r_i\|^2$

4. Recover $Q^{\pi_k}(s_t, a_t) \approx \psi^{\pi_k}(s_t, a_t)^T w$ for every $\pi_k$

$$\pi'(s) = \max_a \max_k \psi^{\pi_k}(s, a)^T w$$

Finds the highest reward policy in each state

Continuous successor representations

\[ \mu_i^\pi(s_t) = (1 - \gamma)\delta(s_t = i) + \gamma \mathbb{E}_{a_t \sim \pi(a_t | s_t), s_{t+1} \sim p(s_{t+1} | s_t, a_t)}[\mu_i^\pi(s_{t+1})] \]

always zero for any sampled state if states are continuous

Framing successor representation as \textit{classification}:

\[ p^\pi(F = 1 | s_t, a_t, s_{\text{future}}) = \frac{p^\pi(s_{\text{future}} | s_t, a_t)}{p^\pi(s_{\text{future}} | s_t, a_t) + p^\pi(s_{\text{future}})} \]

binary classifier

\( F = 1 \) means \( s_{\text{future}} \) is a future state from \( s_t, a_t \) under \( \pi \)

\( \mathcal{D}_+ \sim p^\pi(s_{\text{future}} | s_t, a_t) \quad \mathcal{D}_- \sim p^\pi(s) \)
Continuous successor representations

\[ D_+ \sim p^\pi(s_{\text{future}}|s_t, a_t) \quad D_- \sim p^\pi(s) \]

\[ p^\pi(F = 1|s_t, a_t, s_{\text{future}}) = \frac{p^\pi(s_{\text{future}}|s_t, a_t)}{p^\pi(s_{\text{future}}|s_t, a_t) + p^\pi(s_{\text{future}})} \]

\[ p^\pi(F = 0|s_t, a_t, s_{\text{future}}) = \frac{p^\pi(s_{\text{future}})}{p^\pi(s_{\text{future}}|s_t, a_t) + p^\pi(s_{\text{future}})} \]

\[ \frac{p^\pi(F = 1|s_t, a_t, s_{\text{future}})}{p^\pi(F = 0|s_t, a_t, s_{\text{future}})} = \frac{p^\pi(s_{\text{future}}|s_t, a_t)}{p^\pi(s_{\text{future}})} \]

\[ \frac{p^\pi(F = 1|s_t, a_t, s_{\text{future}})}{p^\pi(F = 0|s_t, a_t, s_{\text{future}})} p^\pi(s_{\text{future}}) = p^\pi(s_{\text{future}}|s_t, a_t) \]

constant independent of \( a_t, s_t \)
The C-Learning algorithm

\[ \mathcal{D}_+ \sim p^\pi(s_{\text{future}}|s_t, a_t) \quad \mathcal{D}_- \sim p^\pi(s) \]

\[ p^\pi(F = 1|s_t, a_t, s_{\text{future}}) = \frac{p^\pi(F = 1|s_t, a_t, s_{\text{future}})}{p^\pi(F = 1|s_t, a_t, s_{\text{future}}) + p^\pi(s_{\text{future}})} \]

To train:
1. Sample \( s \sim p^\pi(s) \) (e.g., run policy, sample from trajectories)
2. Sample \( s \sim p^\pi(s_{\text{future}}|s_t, a_t) \) (e.g., pick \( s_{t'} \) where \( t' = t + \Delta, \Delta \sim \text{Geom}(\gamma) \))
3. Update \( p^\pi(F = 1|s_t, a_t, s) \) using SGD with cross entropy loss

This is an on policy algorithm

Could also derive an off policy algorithm