Model-Based RL

Reinforcement Learning Summer School
Martha White
University of Alberta and AMII
Comments for the lecture

• Please ask questions (this is a summer school)

• I will pause a few times and get you to answer questions/exercises

• Outcomes: you will
  
  • understand how models can be used to learn optimal values/policies
  
  • understand in-depth one strategy, called Dyna, for online setting

  • recognize some of the other ways models can be used
What is model-based RL?
What is model-based RL?

Could mean RL when *given* the model
What is model-based RL?

Could mean RL when *given* the model

Could mean RL with a *learned* model
What is model-based RL?

Could mean RL when *given* the model

Could mean RL with a *learned* model
What is model-based RL?

Could mean RL when *given* the model

Could mean RL with a *learned* model
High-level Outline

- Part 1: Learning the optimal policy given the model (offline)
- Part 2: Moving to learned models (online)
  - Particularly looking at a formalism called Dyna
- Part 3: A brief discussion about other ways to use models
High-level Outline

• Part 1: Learning the optimal policy given the model (offline)
  
• Part 2: Moving to learned models (online)
  • Particularly looking at a formalism called Dyna
  
• Part 3: A brief discussion about other ways to use models
Imagine we have the model

- Joint transition and reward dynamics

\[ p(s', r | s, a) \]

- Then, we can learn offline without interacting with the world!
Bellman equations & Dynamic Programming to find the optimal policy

- We can directly solve for the (optimal) action-values, using Bellman equations
Bellman equations & Dynamic Programming to find the optimal policy

- We can directly solve for the (optimal) action-values, using Bellman equations

\[ q^*(s, a) = \sum_{s'} \sum_r p(s', r | s, a) \left[ r + \gamma \max_{a'} q^*(s', a') \right] \]
Bellman equations & Dynamic Programming to find the optimal policy

- We can directly solve for the (optimal) action-values, using Bellman equations

\[ q_\star(s, a) = \sum_{s'} \sum_r p(s', r | s, a) \left[ r + \gamma \max_{a'} q_\star(s', a') \right] \]

\[ q_{k+1}(s, a) = \sum_{s'} \sum_r p(s', r | s, a) \left[ r + \gamma \max_{a'} q_k(s', a') \right] \]

called Value Iteration
4.4. Value Iteration

This algorithm is called value iteration. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

\[
\begin{align*}
V_{k+1}(s) &= \max_a \left[ R_t + \gamma V_k(S_{t+1}) \right] \\
&= \max_a \left[ r + \gamma \max_{a'} Q(s', a') \right] \\
&\text{for all } s \in S.
\end{align*}
\]

For arbitrary \( v_0 \), the sequence \( \{V_k\} \) can be shown to converge to \( V^* \) under the same conditions that guarantee the existence of \( V^* \).

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing \( \pi^* \).

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to \( V^* \). In practice, we stop once the value function changes by only a small amount in a sweep.

The box below shows a complete algorithm with this kind of termination condition.

**Value Iteration, for estimating \( \pi \approx \pi^* \)**

*Algorithm parameter: a small threshold \( \theta > 0 \) determining accuracy of estimation*

*Initialize \( Q(s, a) = 0 \) for all \( s,a \)*

*Loop:*

| \( \Delta \leftarrow 0 \) |
| Loop for each \( s \in S, a \in A \) |
| \( v \leftarrow Q(s, a) \) |
| \( Q(s, a) \leftarrow \Sigma_{s',r} p(s', r \mid s, a) [r + \gamma \max_{a'} Q(s', a')] \) |
| \( \Delta \leftarrow \max(\Delta, |v - Q(s, a)|) \) |

until \( \Delta < \theta \)

*Output a deterministic policy, \( \pi \approx \pi^* \), such that*

\( \pi(s) = \arg\max_a Q(s, a) \)
High-level Outline

• Part 1: Learning the optimal policy given the model (offline)

• Part 2: Moving to learned models (online)

• Part 3: A brief discussion about other ways to use models
High-level Outline

• Part 1: Learning the optimal policy given the model (offline)

• Part 2: Moving to learned models (online)

• Part 3: A brief discussion about other ways to use models

RL with learned models can use a similar approach to Dynamic Programming but online
Online Reinforcement Learning

Agent

Environment

actions

reward

states
Online Reinforcement Learning

Agent

Environment

actions

reward

states

$S_0$
Online Reinforcement Learning

Agent

Environment

S_0, A_0

actions

reward

states
Online Reinforcement Learning

Agent

Environment

$S_0$, $A_0$, $R_1$, $S_1$.
Online Reinforcement Learning

Agent

Environment

S₀  A₀  R₁  S₁  A₁

actions

reward

states
Online Reinforcement Learning

Agent

Environment

S₀  A₀  R₁  S₁  A₁  R₂  S₂
Online Reinforcement Learning

Agent

Environment

actions

reward

states

$S_0$, $A_0$, $R_1$, $S_1$, $A_1$, $R_2$, $S_2$, $A_2$, ...
Online Reinforcement Learning

Agent

Environment

Tuples of experience:

\((S_0, A_0, R_1, S_1)\)
\((S_1, A_1, R_2, S_2)\)
\((S_2, A_2, R_3, S_3)\)

\(\ldots\)
Q-learning update:
\[ Q(S, A) = Q(S, A) + \alpha \left[ R + \gamma \max Q(S', A') - Q(S, A) \right] \]
Online RL without a Model

- Agent
- Environment
- Actions
- Reward
- States
Online RL without a Model

Agent
\((S_t, A_t, R_{t+1}, S_{t+1})\)

Environment

actions

reward

states
Online RL without a Model

Agent

Environment

actions

reward

states
Online RL without a Model

Agent

\((S_t, A_t, R_{t+1}, S_{t+1})\)

actions

Environment

reward

states
Online RL without a Model

Agent

Environment

- actions
- reward
- states
Online RL with a Model

Agent

Environment

actions

reward

states
Online RL with a Model

Agent

Policy

Model

Environment

actions

reward

states
Online RL with a Model

Agent

Policy

Model

Environment

\((S_t, A_t, R_{t+1}, S_{t+1})\)
Online RL with a Model

Agent

Policy

Model

Environment

One Goal: Improve Sample Efficiency

$(S_t, A_t, R_{t+1}, S_{t+1})$
What are possible learned models?

- **Most obvious answer:** \( \hat{p}(s', r | s, a) \)
- **Realistically:** models with state abstraction and temporal abstraction
- **For now:** let’s assume we learn approximation \( \hat{p}(s', r | s, a) \)
Outline for Part 2: Moving to Learned Models

- Introduce a **planning** framework called **Dyna**
- Explain how Experience Replay is a simple instance of Dyna
- Discuss two key choices in Dyna: **Model** and **Search Control**
- Discuss different choices for the **Model**
- Discuss different choices for **Search Control**
What is Dyna?
What is Dyna?

\[ w = w + \alpha \delta x(s) \]
e.g., Q-learning
What is Dyna?

\[ w = w + \alpha \delta x(s) \]

e.g., Q-learning
What is Dyna?

\[ w = w + \alpha \delta x(s) \]

e.g., Q-learning

\[ s', r \sim M(s,a) \]

(s,a)
What is Dyna?

$$w = w + \alpha \delta x(s)$$

$$s', r \sim M(s,a)$$

$$(s,a)$$

**Example:** Q-learning
What is Dyna?

w = w + \alpha \delta x(s)

e.g., Q-learning

s', r ~ M(s,a)
What is Dyna?

**Key Idea**: Use RL updates on simulated experience from a model as if it is the real world

\[ w = w + \alpha \delta x(s) \]

- Policy/value functions
- Planning update
- Environment
- Model
- e.g., Q-learning
Pseudocode

Dyna-Q
**Pseudocode**

**Dyna-Q**

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
Pseudocode

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$
Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
**Pseudocode**

**Dyna-Q**

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$-greedy($S, Q$)

(c) Take action $A$; observe resultant reward, $R$, and state, $S'$

(d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

(e) $Model(S, A) \leftarrow R, S'$

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in $S$
Pseudocode

**Dyna-Q**

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$

$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in $S$
$R, S' \leftarrow Model(S, A)$
Pseudocode

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$
Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
(e) $Model(S, A) \leftarrow R, S'$

$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in $S$
$R, S' \leftarrow Model(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
**Pseudocode**

**Dyna-Q**

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat $n$ times:
   - $S \leftarrow$ random previously observed state
   - $A \leftarrow$ random action previously taken in $S$
   - $R, S' \leftarrow Model(S, A)$
   - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Let’s see how much better an agent can do with Dyna
Agent in the first episode
Agent in the first episode

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \right)$$
Agent’s knowledge after the first episode
Agent’s knowledge after the first episode
Agent using many planning steps in Dyna
Agent using many planning steps in Dyna
Agent has the optimal policy after just one episode
Agent has the optimal policy after just one episode.
Interleaving planning and acting

Number of actions taken: 184
Interleaving planning and acting

Number of actions taken: 184
Interleaving planning and acting

Number of steps planned: 100
Number of actions taken: 185
Interleaving planning and acting

Number of steps planned: 100
Number of actions taken: 185
Dyna = Background Planning

• Given unlimited computation, each planning update in the background could essentially solve the Bellman equation for the current model

• Loop over all states and actions many times

• At extreme of computation, behaves like Dynamic Programming

• In practice, have limited computation

• Use any extra computation for background planning: do as many updates to the value function or policy as computation allows
Advantages of Dyna

- **Anytime** planning (asynchronous, occurs in the background)
  - contrasts Decision-time planning
- Can take advantage of **parallelism**
- Naturally enables **partial models**
- Can still do **long-term planning** use temporal abstraction, but avoids multi-step rollouts
Now let’s dive into specific instances of Dyna
Important choices

- The type of model
- Search-control

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat $n$ times:
   - $S \leftarrow$ random previously observed state
   - $A \leftarrow$ random action previously taken in $S$
   - $R, S' \leftarrow Model(S, A)$
   - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Important choices: Model

- The type of model
- Search-control

**Dyna-Q**

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$-greedy$(S, Q)$

(c) Take action $A$; observe resultant reward, $R$, and state, $S'$

(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

(e) $Model(S, A) \leftarrow R, S'$

(f) Loop repeat $n$ times:

   $S \leftarrow$ random previously observed state

   $A \leftarrow$ random action previously taken in $S$

   $R, S' \leftarrow Model(S, A)$

   $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Important choices: Search Control

- The type of model
- Search-control

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat $n$ times:

$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in $S$
$R, S' \leftarrow Model(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Let’s look at a simple example of Dyna: Experience Replay
Experience Replay

- Essentially using a batch method in an online setting
- Store buffer of recent transitions \((s, a, s', r)\)
  - e.g., sliding window buffer
- Sample mini-batch updates from the buffer, for updates to the value function or policy

**Exercise:** How can ER be seen as an instance of Dyna? What is the choice for the **Model** and for **Search Control**?
Exercise: How can ER be seen as an instance of Dyna? What is the choice for the Model and for Search Control?
Experience Replay: A Simple Example of Dyna

\[ s', r \sim M(s, a) \]

\((s, a)\)
Experience Replay:
A Simple Example of Dyna

Model is tuples of experience:
(s₀, a₀, r₁, s₁)
(s₁, a₁, r₂, s₂)
(s₂, a₂, r₃, s₃)
...
Experience Replay: A Simple Example of Dyna

Model is tuples of experience:

\((s_0, a_0, r_1, s_1)\)
\((s_1, a_1, r_2, s_2)\)
\((s_2, a_2, r_3, s_3)\)
\(...\)
Experience Replay: A Simple Example of Dyna

Model is tuples of experience:

\((s_0, a_0, r_1, s_1)\)
\((s_1, a_1, r_2, s_2)\)
\((s_2, a_2, r_3, s_3)\)

...
Experience Replay: A Simple Example of Dyna

Model is tuples of experience:

\[(s_0, a_0, r_1, s_1)\]
\[(s_1, a_1, r_2, s_2)\]
\[(s_2, a_2, r_3, s_3)\]
\[\ldots\]

We should be able to get a better Model and smarter Search Control
Advantages of a Learned Model over a Transition Buffer

- **Compactness**: summarizes experience
- **Coverage**: cannot store all experience, so in ER common to use most recent experience (does not cover space)
- **Querying**: can query a model from a particular (s,a)
Important choices: Model

- The type of model
- Search-control

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat $n$ times:
   $S \leftarrow$ random previously observed state
   $A \leftarrow$ random action previously taken in $S$
   $R, S' \leftarrow Model(S, A)$
   $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
What are possible learned models?

$$\hat{p}(s', r | s, a)$$

Increasing Abstraction and/or Simplicity
What are possible learned models?

- Most obvious answer: $\hat{p}(s', r | s, a)$

Increasing Abstraction and/or Simplicity
What are possible learned models?

- Most obvious answer: $\hat{p}(s', r | s, a)$

- Transition Model with **agent state**
  - agent state = constructed vector summarizing key information

Increasing Abstraction and/or Simplicity
What are possible learned models?

- Most obvious answer: \( \hat{p}(s', r | s, a) \)

- Transition Model with agent state
  - agent state = constructed vector summarizing key information

- Predictions about some observations/features in the future

Increasing Abstraction and/or Simplicity
What are possible learned models?

• Most obvious answer: \( \hat{p}(s', r | s, a) \)

• Transition Model with agent state
  • agent state = constructed vector summarizing key information

• Predictions about some observations/features in the future

• Predictions about (cumulative) rewards in the future
What are possible learned models?

• Most obvious answer: \( \hat{p}(s', r | s, a) \)

• Transition Model with agent state
  • agent state = constructed vector summarizing key information

• Predictions about some observations/features in the future

• Predictions about (cumulative) rewards in the future

• ...or even \( Q(s,a) \)?
So is Sarsa a model-based RL algorithm?

- This question only arises due to being imprecise
- Let’s try to be more precise
What does the model do?

- The **agent** uses knowledge/predictions about the world (a **model**) to
  - improve estimates of the **optimal** value function/policy
  - learn about **new** things **faster**
    - e.g., learn new option policies (new skills)
    - e.g., help agent re-visit parts of the space in non-stationary problems

**Notice now that Q(s,a) does not really count as a model**
What does the model do?

• The agent uses knowledge/predictions about the world (a model) to
  • improve estimates of the optimal value function/policy
  • learn about new things faster
    • e.g., learn new option policies (new skills)
    • e.g., help agent re-visit parts of the space in non-stationary problems

Notice now that Q(s,a) does not really count as a model
But the model does not have to be the transition dynamics
Important aspects of the model

- State-to-State vs Observation-to-Observation
Models on Agent State

- Construct agent state $\hat{s}$
  - e.g., recurrent neural network to summarize history (POMDPs)
  - e.g., remove unnecessary detail from an image, only keep key info in the agent state needed to make predictions
- Learn one-step model for agent state $\hat{p}(\hat{s}', r | \hat{s}, a)$
- Only model what the agent thinks is important, avoid pixel-to-pixel models
Important aspects of the model

- State-to-State vs Observation-to-Observation
- Expectation vs Sample Models
Sample Models

- Given \((s, a)\), obtain a **sample** of \(s'\) and \(r\)

- Examples:
  - Conditional Gaussian distribution
  - Conditional Mixture Model
  - Mixture Density Network
Expectation Model

- Given \((s,a)\), output expected next state and reward

- **Exercise**: Imagine you train a feedforward NN with input-output pairs \((s,a), (s', r)\), with a squared error

- Would this result in a Sample Model or Expectation Model

- or something else?
Expectation Model

• Given \((s,a)\), output expected next state and reward

• **Exercise**: Imagine you train a feedforward NN with input-output pairs \(( (s,a), (s’, r) )\), with a squared error

• Would this result in a Sample Model or Expectation Model

• or something else?

• **Answer**: Expectation Model
Expectation Model

- Given \((s,a)\), output \textbf{expected} next state and reward

- Examples:
  - Linear function of (features of) \((s,a)\)
  - Neural Network

\[
\text{(s, a)} \rightarrow \mathbb{E}[S' | s, a]
\]
Potential Issues with an Expectation Model

For illustration purposes, consider the domain in Figure 2.2. Assume that after moving forward from $(x, y)$ the agent ends up in either $(x+1, y+1)$, $(x+1, y)$, or $(x+1, y-1)$, each $\frac{1}{3}$ from Cosmin Paduraru's nice thesis, “Planning with Approximate and Learned Models of Markov Decision Processes”
Potential Issues with an Expectation Model

\[ \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \]
\[ = \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[v(S_{t+1})|S_t = s] \]
\[ \neq \mathbb{E}[R_{t+1}|S_t = s] + \gamma v(\mathbb{E}[S_{t+1}|S_t = s]) \]

* from Cosmin Paduraru's nice thesis, “Planning with Approximate and Learned Models of Markov Decision Processes”
Potential Issues with an Expectation Model

\[
\begin{align*}
\mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] = & \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[v(S_{t+1})|S_t = s] \\
\neq & \mathbb{E}[R_{t+1}|S_t = s] + \gamma v(\mathbb{E}[S_{t+1}|S_t = s])
\end{align*}
\]

* from Cosmin Paduraru's nice thesis, “Planning with Approximate and Learned Models of Markov Decision Processes”
Some Benefits of an Expectation Model

• Likely simpler to learn

• Modeling entire distributions more difficult than just statistics like the mean

• If the world is deterministic, then expectation model = sample model

• …maybe this is not so unreasonable

• If the value function is linear in agent state, then there is no disadvantage to using an expectation model

• See “Planning with Expectation Models”, Wan et al, 2019
Important aspects of the model

• State-to-State vs Observation-to-Observation

• Expectation vs Sample Models

• Rollouts vs Temporal Abstraction
Issues with Rollouts

*image from Erin Talvitie, “Self-Correcting Models for Model-Based Reinforcement Learning”, 2017*
Temporal Abstraction

• Use options to define **macro-actions**

• e.g., Imagine a navigation robot. It could have a policy that tells it how to get to the door (policy defined by option, Andre will talk about this more)

• Agent can plan over options, \( \hat{p}(s', r | s, \pi) \)

• e.g., can ask: “What is the resulting agent-state and (accumulated) reward from a given agent-state when following the option policy?”

• **Advantage:** Can reason about longer horizons (multiple steps into future)

• Without rolling out the model many steps
Important aspects of the model

• State-to-State vs Observation-to-Observation

• Expectation vs Sample models

• Rollouts vs Temporal abstraction

• Full transition dynamics or a subset of predictions about the future

• Whether model outputs certainty estimates

• Sample efficiency in learning the model

• Computational efficiency for querying/sampling from the model
Important aspects of the model

• State-to-State vs Observation-to-Observation
• Expectation vs Sample models
• Rollouts vs Temporal abstraction
• Full transition dynamics or a subset of predictions about the future
• Whether model outputs certainty estimates
• Sample efficiency in learning the model
• Computational efficiency for querying/sampling from the model

Any other suggestions?
Important choices: Search Control

- The type of model
- Search-control

Dyna-Q

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Take action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat $n$ times:

- $S \leftarrow$ random previously observed state
- $A \leftarrow$ random action previously taken in $S$
- $R, S' \leftarrow Model(S, A)$
- $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Search Control strategies:
How to pick (s,a)
Search Control strategies: How to pick \((s,a)\)

- Prioritize samples with high error
- e.g., store observed \((s,a)\) and associated TD-error
Search Control strategies: How to pick \((s,a)\)

- Prioritize samples with high error
  - e.g., store observed \((s,a)\) and associated TD-error
- Update backwards from “important” states
  - e.g., generate predecessor states from state with high TD-error
The utility of updating with predecessor states

- Imagine agent initializes values to zero
- Updates in the center are all zero!
- Then imagine it reaches the Goal State and transitions back to the Start
- What happens if the agent updates around the start state now?
- What happens if the agent updates predecessors around the goal?
Search Control strategies:
How to pick (s,a)
Search Control strategies: How to pick \((s,a)\)

- Prioritize samples with high error
  - e.g., store observed \((s,a)\) and associated TD-error

- Update backwards from “important” states
  - e.g., generate predecessor states from state with high TD-error
Search Control strategies: How to pick (s,a)

- Prioritize samples with high error
  - e.g., store observed (s,a) and associated TD-error

- Update backwards from “important” states
  - e.g., generate predecessor states from state with high TD-error

- Update with on-policy transitions
  - e.g., store observed states s and query current policy for action a
Search Control strategies: How to pick \((s,a)\)

- Prioritize samples with high error
  - e.g., store observed \((s,a)\) and associated TD-error

- Update backwards from “important” states
  - e.g., generate predecessor states from state with high TD-error

- Update with on-policy transitions
  - e.g., store observed states \(s\) and query current policy for action \(a\)

- See “Organizing experience: a deeper look at replay mechanisms for sample-based planning in continuous state domains”, Pan et al, 2018
Search Control strategies: How to pick (s,a)

- Prioritize samples with high error
- Update backwards from “important” states
- Update with on-policy transitions
- **Update current state or nearby region around current state**
  - improve values right before they are used
- see Dyna-2 (Silver et al., 2016), “Hill Climbing on Value Estimates for Search-control in Dyna”, Pan et al., 2019
Search Control strategies: How to pick (s,a)

- Prioritize samples with high error
- Update backwards from “important” states
- Update with on-policy transitions
- Update current state or nearby region around current state
Search Control strategies: How to pick (s,a)

- Prioritize samples with high error
- Update backwards from “important” states
- Update with on-policy transitions
- Update current state or nearby region around current state

Any other suggestions?
So, are we done?
So, are we done?

• Can we just learn an accurate model with a deep NN and use Dyna?
So, are we done?

• Can we just learn an accurate model with a deep NN and use Dyna?

• **Two key take-aways:**

  • How we use the model (search-control) can have a huge impact on how useful it is (can have very little impact, just a waste of computation)
So, are we done?

• Can we just learn an accurate model with a deep NN and use Dyna?

• **Two key take-aways:**

• How we use the model (search-control) can have a huge impact on how useful it is (can have very little impact, just a waste of computation)

• Small errors in the model can result in big errors in the policy
Examples of Bad Errors

Agent
Unreachable States
Goal State
Reachable States

Figure 3.1: Bordered Gridworld. The black square is the agent's position. The agent cannot transition into any states in the grey border. However, the learned environment dynamics models may produce state predictions to or within this border.

To do this, the agent may move in the relative directions (up, down, left, or right) in the region marked 'Reachable States' in Figure 3.1. The feature distinguishing Bordered Gridworld from typical Gridworlds is a region of 'Unreachable States', defined by the set $S_r$, surrounding the 'Reachable States', i.e., the set $S_r$. Note these are disjoint. The agent cannot transition to these 'border' states and is never initialised to start in them either.

3.1.2 How Dyna Planning on Bordered Gridworld can go Wrong

Consider the planning TD update shown in Figure 3.2. An agent performs a planning updates with an imperfect model (which generates simulated transitions from reachable to border states) from state $s$ to next states $s_0$ for all four possible actions (up, left, down, right). Two of these $s_0$ will be in the unreachable border (due to imperfect predictions by the model). If the value function updates $Q(s, \text{left})$ and $Q(s, \text{up})$ to the states inside the border, the values of these states will be moved values of the $s_0$. For example, $s'$ is the state inside the border and $r$ is some prediction of the reward on the transition from $s$ to $s_0$. The values of the states inside the border, $Q(s_0, \cdot, a)$, are arbitrary. If these states have high value, this update may cause the agent to think that the best action is to 'run' into the walls. Consequently, the next time the agent is in state $s$, it will repeatedly either take the actions up or left and get 'stuck'. Moreover, as there is no way for the values of states in the border to be reduced (as the agent cannot transition out from these states), the values $s_0$ will remain at their arbitrary initialisation values, thus perpetually misleading the agent.

With this intuition in mind, in the next section we present the hypothesis around which this thesis is developed.

3.2 The Hallucinated Value Hypothesis

We claim: Any planning update that results in the values of real states being updated towards the value of simulated states may impair learning of an optimal control policy. This hypothesis depends on the type of TD update performed during planning (cf. "...values of real states being updated towards the value of...", page 22).
High-level Outline

• Part 1: Learning the optimal policy given the model (offline)

• Part 2: Moving to learned models (online)

• Part 3: A brief discussion about other ways to use models
Models are useful. They have been used in a variety of ways in RL.

- **Most related**: Learn a model and then use dynamic programming on this learned model to obtain approximate values
  - e.g., KBRL, KBSF, Compressed CME, Pseudo-MDPs

- Decision-time Planning
  - Model Predictive Control (see work from Byron Boots), MCTS

- Use model to improve exploration

- Use model to obtain better estimates of policy gradients (PILCO)

- Use model as inductive bias on value function (e.g., Predictron)
KBRL, KBSF, and CCME

• Learn values only for a representative set of points

• Define smaller (pseudo)-MDP only on these states

• Use value iteration (dynamic programming) on this smaller MDP, which is reasonably efficient

• Value function for whole state-space a simple weighting of the values for these representative states
Exploration with models

- Huge research area using learned models for sound exploration
  - Often consider an optimistic model in the set/distribution of models
  - Most algorithms though are very computationally expensive
- Reward bonuses: accuracy of learned models to incentivize exploration
  - Only indirectly using model, no planning
Implicit Planning

- Optimize model and planner based on the reward the agent receives, using end-to-end learning

- Contrasts learning the model using a separate objective and updating using explicit planning steps

- Can be seen as an inductive bias on value function architecture

- Examples:
  - Predictron (DeepMind)
  - TreeQN and ATreeC (Whiteson and others)
High-level Outline

- Part 1: Learning the optimal policy given the model (offline)
- Part 2: Moving to learned models (online)
  - Dyna for background planning
  - Search-control and Model choices
- Part 3: A brief discussion about other ways to use models
High-level Outline

- Part 1: Learning the optimal policy given the model (offline)
- Part 2: Moving to learned models (online)
  - Dyna for background planning
  - Search-control and Model choices
- Part 3: A brief discussion about other ways to use models

Questions?