Planning and Learning
Week #9

Introduction... 1

- Two types of methods in RL
  - *Planning methods:* Those that require an environment model
    - *Dynamic programming, heuristic search*
  - *Learning methods:* Those that function without a model
    - *Monte Carlo (MC) & Temporal Difference (TD)* methods

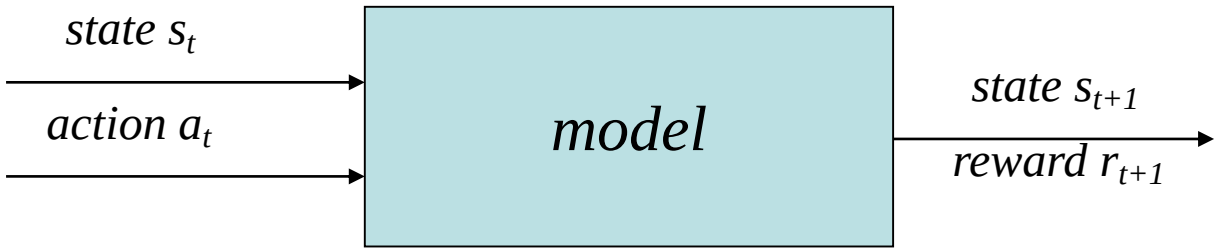
Introduction... 2

- *MC* methods and *TD* methods are distinct alternatives in RL and they are related by using eligibility traces (ETs).

- In a similar way, *planning methods* and *learning methods* may be shown to be related to each other.

Models

- A *model* is anything that helps the agent predict its environment's responses to the actions it takes.



Model Types from Planning Perspective

- Two types of models in RL from the planning perspective
  - *Distribution Models:* Those that generate a description of all possibilities and their probabilities
  - *Sample Models:* Those that produce only one of the possibilities, sampled according to probabilities at hand.

Distribution and Sample Models

- Distribution models are stronger than sample models since they may be used to generate samples
- They are harder to obtain since
  - Either there is not enough information to build one,
  - Or the information is there, but it is too complex to analytically obtain the model's parameters.
- Both types of models generate ***simulated experience***.

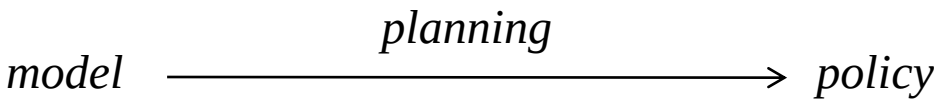
Simulated Experience in Models

- Given an initial state and an action
  - *Distribution model* can generate
    - All possible transitions and their occurrence probabilities
    - All possible episodes and their probabilities

  Sample model can generate
    - A possible transition
    - An entire episode

Planning

- In RL context, planning is defined as
  - any computational process that, once having a model presented, produces/improves a policy to interact with the environment represented by this model.



Planning

- Two distinct approaches to planning
  - *State space planning*:
    - involves a search through the state space for an optimal policy or a path to a goal.
  - *Plan-space planning*
    - Performs a search through a space of plans.
- We will use *state space planning*.

State Space Planning

- All state space planning methods share a common structure.
- Two ideas are
  - All state space planning methods involve *computing value functions* as a key intermediate step *toward improving the policy*,
  - They compute their value functions *by backup operations applied to simulated experience*.

Diagram of State-Space Planning

How learning and planning relate…

- The ***core of both learning and planning*** is the *estimation of value functions* by backup operations.
- The *difference*
  - *real experience* is used *in learning* that originates from the environment;
  - *simulated experience* generated by the ***model*** is employed *in planning*.

Further differences of Planning and Learning

- Origins of experience in learning and planning are different.
  - Simulated experience in planning and
  - Real experience in learning.
- This in turn leads to other differences such as
  - the different ways of performance assessment and
  - how flexibly experience is generated,

Advantages of Planning

- A learning algorithm may be replaced for the key backup step of a planning method.

- Learning methods can be applied to simulated experience as well as to real experience.

Random-sample One-step tabular Q-Planning

- *Select a state s* $\in$ S *and an action a* $\in$ *A(s) at random*
- *Send s,a to a sample model, and obtain a sample next state s′ and a sample reward r.*

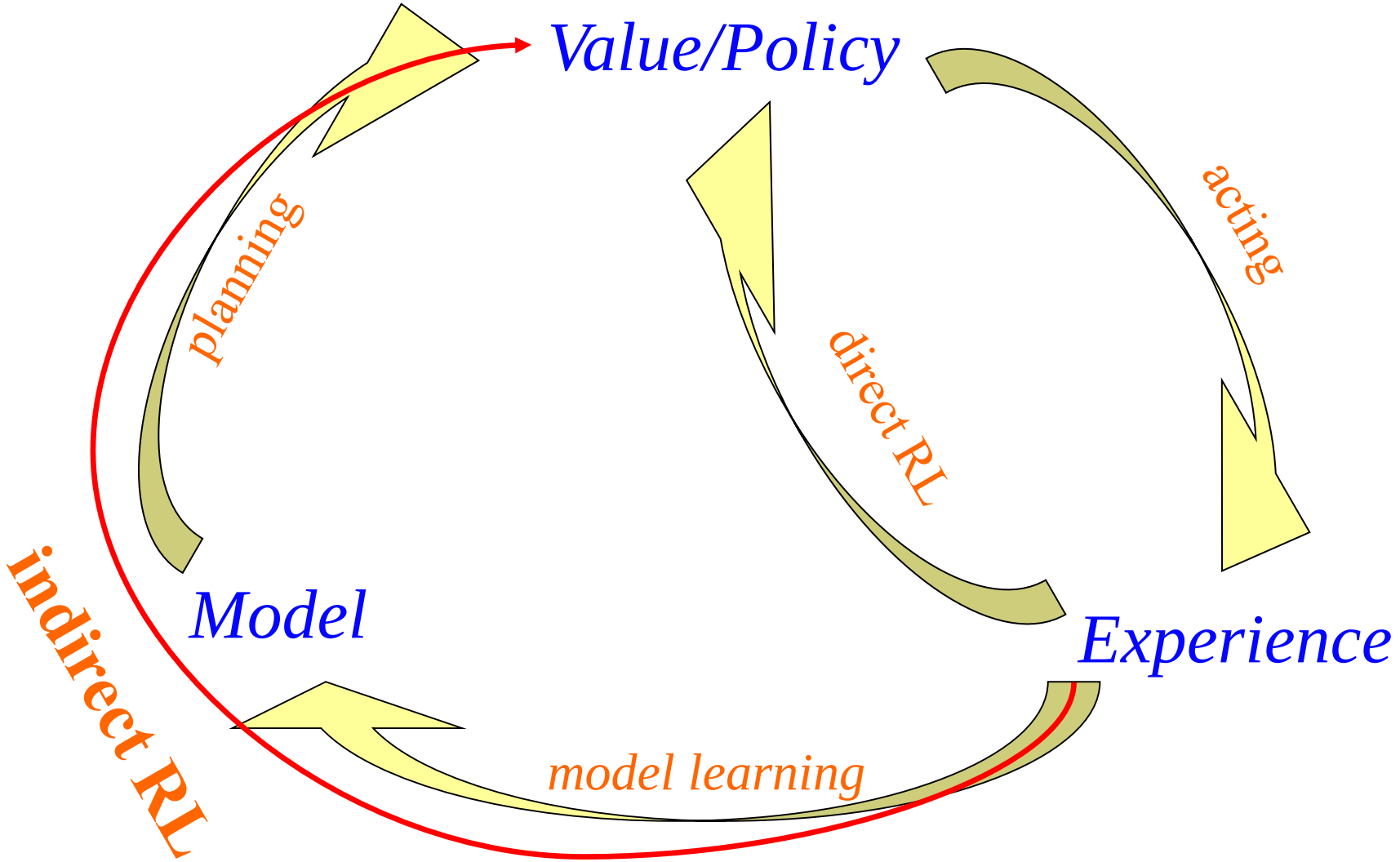
- *Apply one-step tabular Q-learning to s,a,s′,r*

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Integrating Planning, Acting and Learning

- *Learning* may be achieved *together with planning* where,
  - while interacting with the environment,
  - *the model is continuously improved/updated*.

Learning, Planning and Acting



Dyna-Q Architecture

- A planner (agent) uses *experience* for …
  - … *improving the model*, and
  - … *improving state-action values/policy*.

- Dyna-Q architecture uses both learning and planning together.

- Dyna agent achieves both *direct* and *indirect RL* indicated on page 17 in the algorithm on the next page.

Algorithm for Dyna-Q-learning

- *Initialize Q(s,a) and Model(s,a) for all $s \in S$ and $a \in A(s)$*
- *Do forever*
  - *s ← current (non-terminal) state*
  - *a ← ε-greedy(s,Q)*
  - *Execute a; observe s′ and r*

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

  - *Model(s,a) ← s′,r //assuming deterministic environments*
  - *Repeat N times*
    - *s ← random previously observed state;*
    - *a ← random action previously taken in s*
    - *s′,r ← Model(s,a)*

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Algorithm's parameter N

- With the growing values of N, the modification history of the model gets deeper.

- The *deeper the update history of the model, the faster the convergence to the optimal policy.*[*]

[*] *Check Example 9.1 in [1] pp 233-235 and Fig. 9.5 and 9.6!*

What if the model is wrong…

- …or what if the environment is *non-stationary*?
- If the environment has changed such that the current policy does not end with success anymore, then the agent discovers the optimal policy sooner or later provided sufficiently many episodes are involved.
- If, on the other hand, the environment changes so a new optimal policy arises, but the old (sub-optimal) policy is still available for access to goal state, then the optimal policy might go undetected even with an ε-greedy policy.*

* Check Example 9.2 and 9.3 in [1] pp 236-238

What if the model is wrong…2

- As in direct RL, there is no perfect and practical solution for the exploration/exploitation dilemma.
- Using heuristics may in such cases be effective.
- In the corresponding example* in [1], for each state-action pair, the time that has elapsed since the pair was last examined in a real interaction with the environment was recorded.
- This heuristic indicates the extent of the chance the model is incorrect.
- To give more chance to long-unexamined actions, a special "bonus reward" ($r+\kappa\sqrt{n}$; r being the regular reward, n the time steps the state-action pair is not examined and $\kappa \in R$ small) is given in simulated experience to these actions. Agent keeps encouraged to test such pairs and has a good possibility of getting to the optimal policy.

* Check Example 9.2 and 9.3 in [1] pp 236-238

Prioritized Sweeping

- In real world cases, where the *state-action space* is usually *tremendously large*, *unnecessary value updates* (i.e., those that do not contribute to optimal or near optimal policy) *in the model must be minimized*.
- *Prioritized sweeping* is the improved version of the Dyna-Q algorithm, where *only values of those state-action pairs get updated which lead to state-action pairs with currently updated values*.
- *Effective* method *as long as* model has *discrete states*.

Prioritized Sweeping Algorithm

- *Initialize Q(s,a) and Model(s,a) for all s,a and PQueue to empty*
- *Do forever*
  - *s ← current (non-terminal) state*
  - *a ← policy(s,Q)*
  - *Execute a; observe s′ and r*
  - *Model(s,a) ← s′,r //assuming deterministic environments*
  - $p \leftarrow \left| r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right|$
  - *if (p>θ) then insert s,a into PQueue with priority p*
  - *Repeat N times, while not isEmpty(PQueue)*
    - *s,a ← first(PQueue)*
    - *s′,r ← Model(s,a)*
    - $Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
    - *Repeat for all s″,a″ (from within all previously experienced pairs) predicted to lead to s*
      - *r″ ← predicted reward*
      - $p \leftarrow \left| r'' + \gamma \max_{a} Q(s,a) - Q(s'',a'') \right|$
      - *if (p>θ) then insert s″,a″ into PQueue with priority p*

Prioritized Sweeping Algorithm with a specific clustering algo (e.g., EM)

- *Initialize Q(s,a) and Model(s,a) for all s,a and PQueue to empty*
- *Do forever*
  - *s ← current (non-terminal) state; discretize s into cl(s);*
  - *a ← policy(cl(s),Q)*
  - *Execute a; observe s' and r; discretize s' into cl(s')*
  - *Model(cl(s),a) ← cl(s'),r //assuming deterministic environments*
  - $p \leftarrow \left| r + \gamma \max_{a'} Q(cl(s'), a') - Q(cl(s), a) \right|$
  - *if (p>θ) then insert cl(s),a into PQueue with priority p*
  - *Repeat N times, while not isEmpty(PQueue)*
    - *cl(s),a ← first(PQueue)*
    - *cl(s'),r ← Model(cl(s),a)*
    - $Q(cl(s), a) = Q(cl(s), a) + \alpha\left[r + \gamma \max_{a'} Q(cl(s'), a') - Q(cl(s), a) \right]$
    - *Repeat for all cl(s"),a" (from within all previously experienced pairs) predicted to lead to cl(s)*
      - *r" ← predicted reward*

      $p \leftarrow \left| r'' + \gamma \max_{a} Q(cl(s), a) - Q(cl(s''), a'') \right|$
      - *if (p>θ) then insert cl(s"),a" into PQueue with priority p*

References

- [1] Sutton, R. S. and Barto A. G., "*Reinforcement Learning: An introduction*," MIT Press, 1998