Planning and models

Matteo Hessel

2021

Recap

In the previous lectures:

- Bandits: how to trade-off exploration and exploitation.
- Dynamic Programming: how to solve prediction and control given full knowledge of the environment.
- Model-free prediction and control: how to solve prediction and control from interacting with the environment.
- Function approximation: how to generalise what you learn in large state spaces.

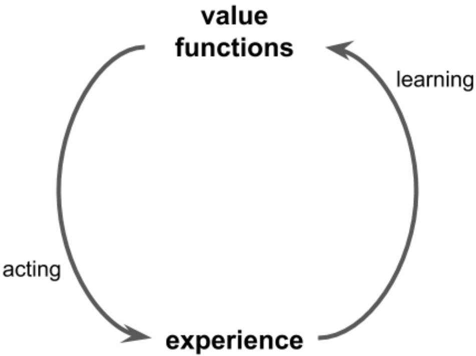
Dynamic Programming and Model-Free RL

- Dynamic Programming
  - Assume a model
  - Solve model, no need to interact with the world at all.
- Model-Free RL
  - No model
  - Learn value functions from experience.

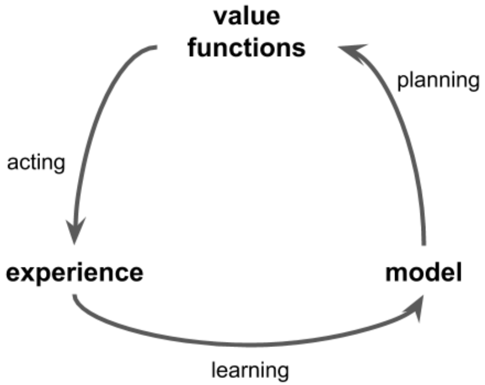
Model-Based RL

- Model-Based RL
  - Learn a model from experience
  - Plan value functions using the learned model.

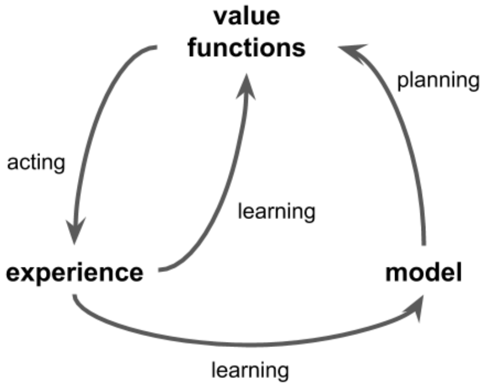
Model-Free RL



Model-Based RL



Model-Based RL



Why should we even consider this?

One clear disadvantage:

- First learn a model, then construct a value function
  $\Rightarrow$ two sources of approximation error
- Learn a value function directly
  $\Rightarrow$ only one source of approximation error

However:

- Models can efficiently be learned by supervised learning methods
- Reason about model uncertainty (better exploration?)
- Reduce the interactions in the real world (data efficiency? faster/cheaper?).

Learning a Model

Matteo Hessel

2021

What is a Model?

A model $\mathcal{M}_\eta$ is an approximate representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \hat{p} \rangle$,

- For now, we will assume the states and actions are the same as in the real problem
- That the dynamics , $\hat{p}_\eta$ is parametrised by some set of weights η
- The model directly approximates the state transitions and rewards $\hat{p}_\eta \approx p$:

$$R_{t+1}, S_{t+1} \sim \hat{p}_\eta(r, s' \mid S_t, A_t)$$

Model Learning - I

Goal: estimate model $\mathcal{M}_\eta$ from experience $\{S_1, A_1, R_2, ..., S_T\}$

- This is a supervised learning problem

$$S_1, A_1 \to R_2, S_2$$
$$\vdots$$
$$S_{T-1}, A_{T-1} \to R_T, S_T$$

- over a dataset of state transitions observed in the environment.

Model Learning - II

How do we learn a suitable function $f_\eta(s, a) = r, s'$?

- Choose a functional form for f
- Pick loss function (e.g. mean-squared error),
- Find parameters η that minimise empirical loss
- This would give an expectation model
- If $f_\eta(s, a) = r, s'$, then we would hope $s' \approx \mathbb{E}[S_{t+1} \mid s = S_t, a = A_t]$

Expectation Models

- Expectation models can have disadvantages:
  - Image that an action randomly goes left or right past a wall
  - Expectation models can interpolate and put you in the wall
- But with linear values, we are mostly alright:
  - Consider an expectation model $f_\eta(\phi_t) = \mathbb{E}[\phi_{t+1}]$ and value function $v_\theta(\phi_t) = \theta^\top \phi_t$

$$\begin{aligned}
\mathbb{E}[v_\theta(\phi_{t+1}) \mid S_t = s] &= \mathbb{E}[\theta^\top \phi_{t+1} \mid S_t = s] \\
&= \theta^\top \mathbb{E}[\phi_{t+1} \mid S_t = s] \\
&= v_\theta(\mathbb{E}[\phi_{t+1} \mid S_t = s]) \,.
\end{aligned}$$

- If the model is also linear: $f_\eta(\phi_t) = P\phi_t$ for some matrix P.
  - then we can even unroll an expectation model even multiple steps into the future,
  - and still have $\mathbb{E}[v_\theta(\phi_{t+n}) \mid S_t = s] = v_\theta(\mathbb{E}[\phi_{t+n} \mid S_t = s])$

Stochastic Models

- We may not want to assume everything is linear
- Then, expected states may not be right — they may not correspond to actual states, and iterating the model may do weird things
- Alternative: stochastic models (also known as generative models)

$$\hat{R}_{t+1}, \hat{S}_{t+1} = \hat{p}(S_t, A_t, \omega)$$

  where ω is a noise term
- Stochastic models can be chained, even if the model is non-linear
- But they do add noise

Full Models

- We can also try to model the complete transition dynamics
- It can be hard to iterate these, because of branching:

$$\mathbb{E}[v(S_{t+1}) \mid S_t = s] = \sum_a \pi(a \mid s) \sum_{s'} \hat{p}(s, a, s')(\hat{r}(s, a, s') + \gamma v(s'))$$

$$\begin{aligned}\mathbb{E}[v(S_{t+n}) \mid S_t = s] = \sum_a \pi(a \mid s) \sum_{s'} \hat{p}(s, a, s')\Bigg(&\hat{r}(s, a, s') + \\ &\gamma \sum_{a'} \pi(a' \mid s') \sum_{s''} \hat{p}(s', a', s'')\Bigg(\hat{r}(s', a', s'') + \\ &\gamma^2 \sum_{a''} \pi(a'' \mid s'') \sum_{s'''} \hat{p}(s'', a'', s''')\Bigg(\hat{r}(s'', a'', s''') + \ldots\Bigg)\Bigg)\Bigg)\end{aligned}$$

Examples of Models

We typically decompose the dynamics p_η into separate parametric functions

- for transition and reward dynamics

For each of these we can then consider different options:

- Table Lookup Model
- Linear Expectation Model
- Deep Neural Network Model

Table Lookup Models

- Model is an explicit MDP
- Count visits $N(s, a)$ to each state action pair

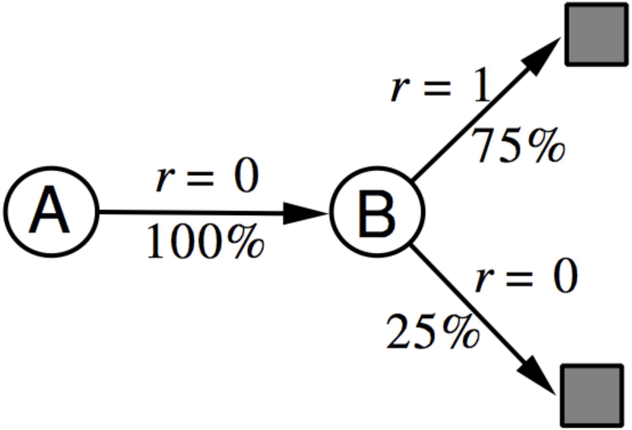
$$\hat{p}_t(s' \mid s, a) = \frac{1}{N(s, a)} \sum_{k=0}^{t-1} I(S_k = s, A_k = a, S_{k+1} = s')$$

$$\mathbb{E}_{\hat{p}_t}[R_{t+1} \mid S_t = s, A_t = a] = \frac{1}{N(s, a)} \sum_{k=0}^{t-1} I(S_k = s, A_k = a) R_{k+1}$$

AB Example

Two states A, B; no discounting; 8 episodes of experience

$A, 0, B, 0$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 1$
$B, 0$



We have constructed a table lookup model from the experience

Linear expectation models

In linear expectation models

- we assume some feature representation ϕ is given
- so that we can encode any state s as $\phi(s)$
- we then parametrise separately rewards and transitions
- each as a linear function of the features

Linear expectation models

- expected next states are parametrised by a square matrix T_a, for each action a

$$\hat{s'}(s,a) = T_a \phi(s)$$

- the rewards are parametrised by a vector w_a, for each action a

$$\hat{r}(s,a) = w_a^T \phi(s)$$

- On each transition (s, a, r, s') we can then apply a gradient descent step
- to update w_a and T_a so as to minimise the loss:

$$L(s,a,r,s') = (s' - T_a \phi(s))^2 + (r - w_a^T \phi(s))^2$$

Planning for Credit Assignment

Matteo Hessel

2021

Planning

In this section we investigate planning

- This concept means different things to different communities
- For us planning is the process of investing compute to improve values and policies
- Without the need to interact with the environment
- Dynamic programming is the best example we have seen so far
- We are interested in planning algorithms that don't require privileged access to a perfect specification of the environment
- Instead, the planning algorithms we discuss today use learned models

Dynamic Programming with a learned Model

Once learned a model $\hat{p}_\eta$ from experience:

- Solve the MDP $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$
- Using favourite dynamic programming algorithm
  - Value iteration
  - Policy iteration
  - ...

Sample-Based Planning with a learned Model

A simple but powerful approach to planning:

- Use the model only to generate samples
- Sample experience from model

$$S, R \sim \hat{p}_\eta(\cdot \mid s, a)$$

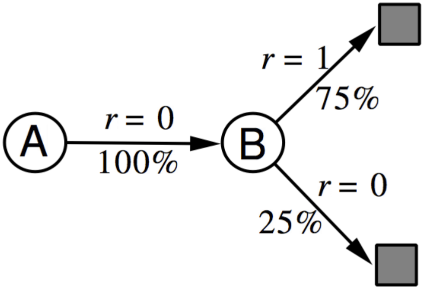
- Apply model-free RL to samples, e.g.:
  - Monte-Carlo control
  - Sarsa
  - Q-learning

Back to the AB Example

- Construct a table-lookup model from real experience
- Apply model-free RL to sampled experience

Real experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Sampled experience

B, 1
B, 0
B, 1
A, 0, B, 1
B, 1
A, 0, B, 1
B, 1
B, 0

e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

Limits of Planning with an Inaccurate Model - I

Given an imperfect model $\hat{p}_\eta \neq p$:

- The planning process may compute a suboptimal policy
- Performance is limited to optimal policy for approximate MDP $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$
- Model-based RL is only as good as the estimated model

Limits of Planning with an Inaccurate Model - II

How can we deal with the inevitable inaccuracies of a learned model?

- Approach 1: when model is wrong, use model-free RL
- Approach 2: reason about model uncertainty over η (e.g. Bayesian methods)
- Approach 3: Combine model-based and model-free methods in a single algorithm.

Real and Simulated Experience

We consider two sources of experience

Real experience Sampled from environment (true MDP)

$$r, s' \sim p$$

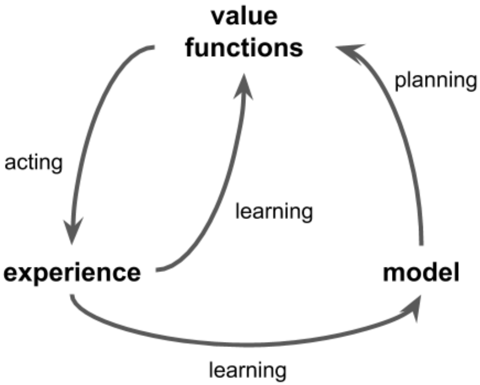
Simulated experience Sampled from model (approximate MDP)

$$r, s' \sim \hat{p}_\eta$$

Integrating Learning and Planning

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
  - Learn a model from real experience
  - Plan value function (and/or policy) from simulated experience
- Dyna
  - Learn a model from real experience
  - Learn AND plan value function (and/or policy) from real and simulated experience
  - Treat real and simulated experience equivalently. Conceptually, the updates from learning or planning are not distinguished.

Dyna Architecture



Dyna-Q Algorithm

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

Do forever:

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

(b) $a \leftarrow \varepsilon$-greedy(s, Q)

(c) Execute action a; observe resultant state, s', and reward, r

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

(e) $Model(s,a) \leftarrow s', r$ (assuming deterministic environment)

(f) Repeat N times:

$s \leftarrow$ random previously observed state

$a \leftarrow$ random action previously taken in s

$s', r \leftarrow Model(s,a)$

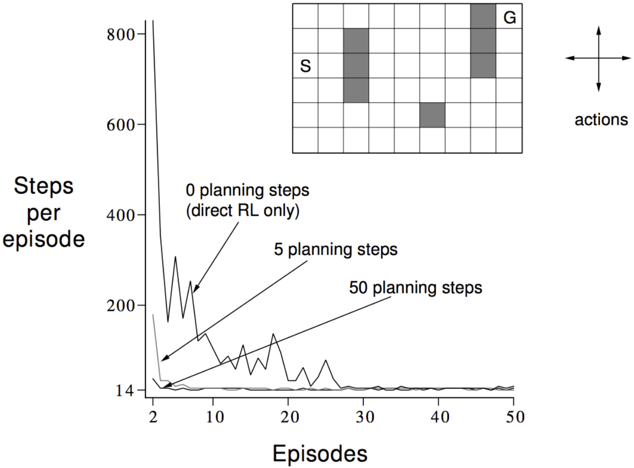
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Advantages of combining learning and planning.

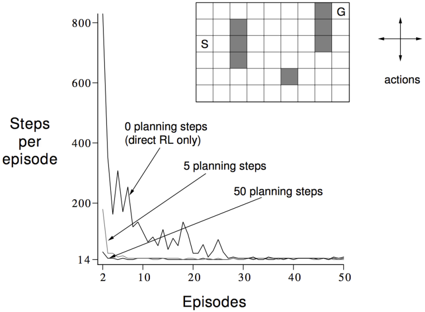
What are the advantages of this architecture?

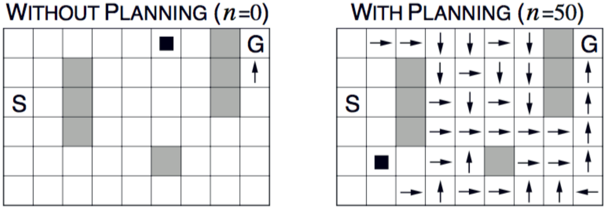
- We can sink in more compute in order to learn more efficiently.
- This is especially important when collecting real data is
  - expensive / slow (e.g. robotics)
  - unsafe (e.g. autonomous driving)

Dyna-Q on a Simple Maze



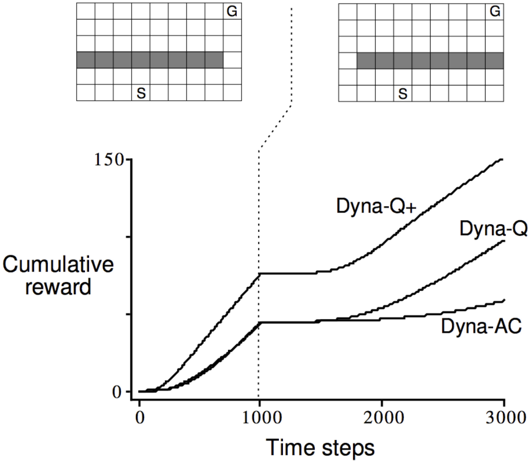
Dyna-Q on a Simple Maze





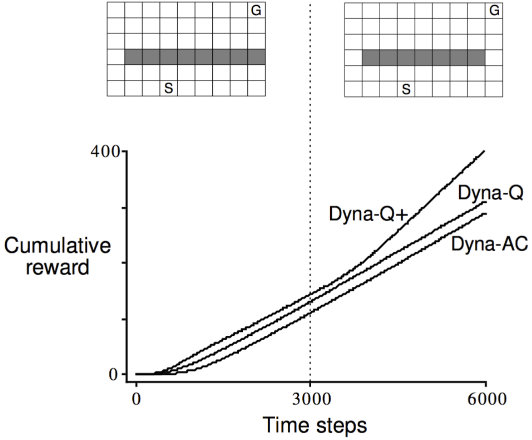
Dyna-Q with an Inaccurate Model

- The changed environment is harder



Dyna-Q with an Inaccurate Model (2)

- The changed environment is easier



Planning and Experience Replay

Matteo Hessel

2021

Conventional model-based and model-free methods

Traditional RL algorithms did not explicitly store their experiences,
It was easy to place them into one of two groups.

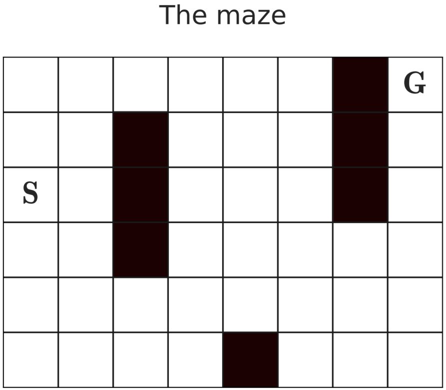
- Model-free methods update the value function and/or policy and do not have explicit dynamics models.
- Model-based methods update the transition and reward models, and compute a value function or policy from the model.

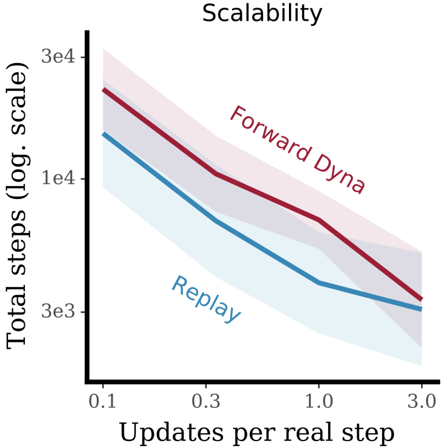
Moving beyond model-based and model-free labels

The sharp distinction between model-based and model-free is now less useful:

1. Often agents store transitions in an *experience replay buffer*
2. Model-free RL is then applied to experience sampled from the replay buffer,
3. This is just Dyna, with the experience replay as a non-parametric model
    - we plan by sampling an entire transition (s, a, r, s'),
    - instead of sampling just a state-action (s, a) and inferring r, s' from the model.
    - we can still sink in compute to make learning more efficient,
    - by making many updates on past data for every new step we take in the environment.

Scalability





Comparing parametric model and experience replay - I

- For tabular RL there is an exact output equivalence between some conventional model-based and model free algorithms.
- If the model is perfect, it will give the same output as a non-parametric replay system for every (s, a) pair
- In practice, the model is not perfect, so there will be differences
- Could model inaccuracies lead to better learning?
- Unlikely if we only use the model to sample imagined transitions from the actual past state-action pairs.
- But a parametric model is more flexible than a replay buffer

Comparing parametric model and experience replay - II

- Plan for action-selection!
  - query a model for action that you *could* take in the future
- Counterfactual planning.
  - query a model for action that you *could* have taken in the past, but did not

Comparing parametric model and experience replay - III

- Backwards planning
  - model the inverse dynamics and assign credit to different states that *could* have led to a certain outcome
- Jumpy planning for long-term credit assignment,
  - plan at different timescales

Comparing parametric model and experience replay - IV

Computation:

- Querying a replay buffer is very cheap!
- Generating a sample from a learned model can be very expensive
- E.g. if the model is large neural network based generative model.

Memory:

- The memory requirements of a replay buffer scale linearly with its capacity
- A parametric model can achieve goods accuracy with a fixed and comparably small memory footprint

Planning for Action Selection

Matteo Hessel

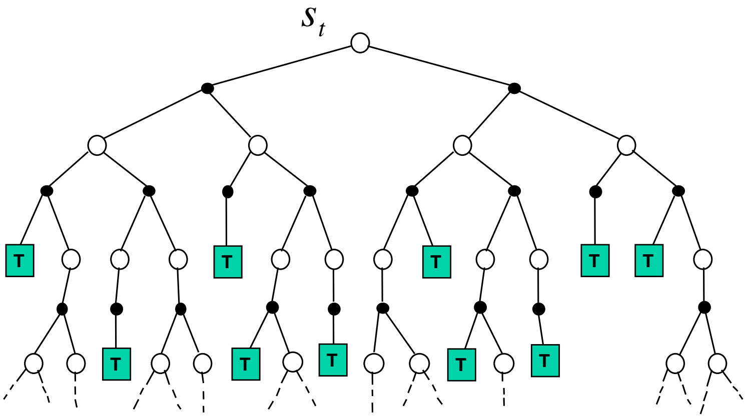
2021

Planning for Action Selection

- We considered the case where planning is used to improve a global value function
- Now consider planning for the near future, to select the next action
- The distribution of states that may be encountered from now can differ from the distribution of states encountered from a starting state
- The agent may be able to make a more accurate local value function (for the states that will be encountered soon) than the global value function
- Inaccuracies in the model may result in interesting exploration rather than in bad updates.

Forward Search

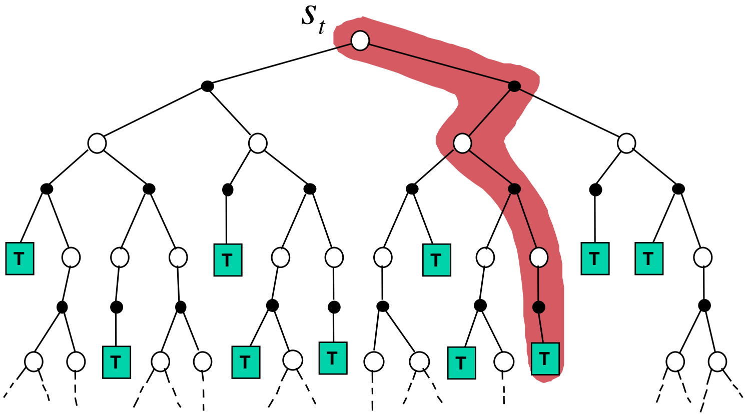
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- No need to solve whole MDP, just sub-MDP starting from now

Simulation-Based Search

- Sample-based variant of Forward search
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



Prediction via Monte-Carlo Simulation

- Given a parameterized model $\mathcal{M}_\eta$ and a simulation policy π
- Simulate K episodes from current state S_t

$$\{S_t^k = S_t, A_t^k, R_{t+1}^k, S_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \hat{p}_\eta, \pi$$

- Evaluate state by mean return (Monte-Carlo evaluation)

$$v(S_t) = \frac{1}{K}\sum_{k=1}^{K} G_t^k \rightsquigarrow v_\pi(S_t)$$

Control via Monte-Carlo Simulation

- Given a model $\mathcal{M}_\eta$ and a simulation policy π
- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s

$$\{S_t^k = s, A_t^k = a, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

  - Evaluate actions by mean return (Monte-Carlo evaluation)

$$q(s, a) = \frac{1}{K} \sum_{k=1}^{K} G_t^k \rightsquigarrow q_\pi(s, a)$$

- Select current (real) action with maximum value

$$A_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ q(S_t, a)$$

Monte-Carlo Tree Search - I

In MCTS, we incrementally build a search tree containing visited states and actions, Together with estimated action values $q(s, a)$ for each of these pairs

- Repeat (for each simulated episode)
  - Select Until you reach a leaf node of the tree, pick actions according to $q(s, a)$.
  - Expand search tree by one node
  - Rollout until episode termination with a fixed simulation policy
  - Update action-values q(s,a) for all state-action pairs in the tree

$$q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(S_u^k, A_u^k = s, a) G_u^k \rightsquigarrow q_\pi(s, a)$$

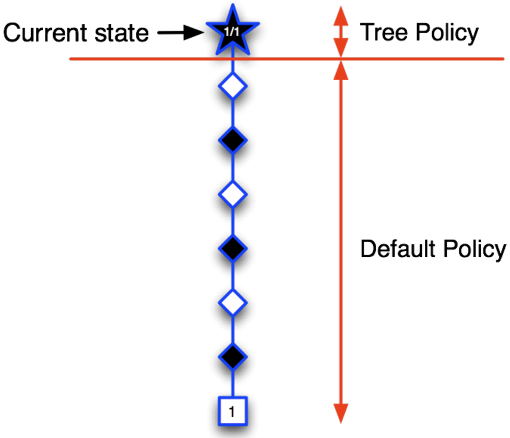
- Output best action according to $q(s, a)$ in the root node when time runs out.

Monte-Carlo Tree Search - II

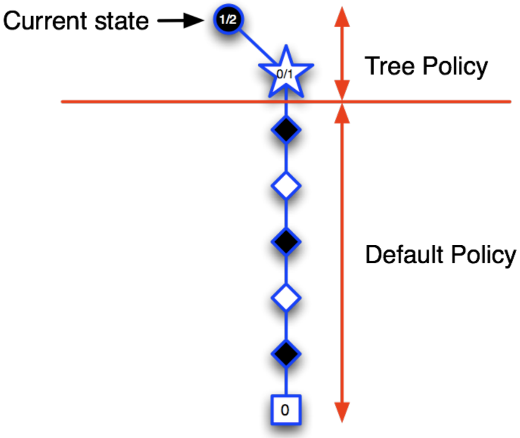
Note that we effectively have two simulation policies:

- a Tree policy that improves during search.
- a Rollout policy that is held fixed: often this may just be picking actions randomly.

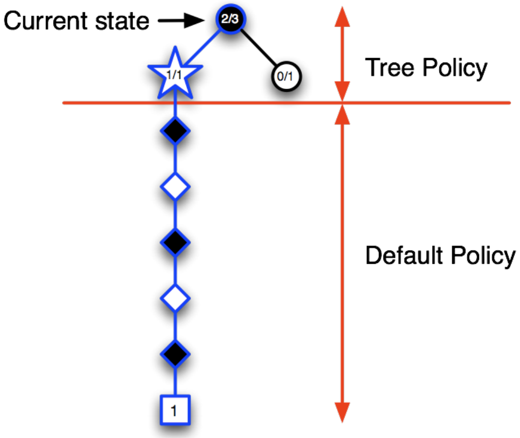
Applying Monte-Carlo Tree Search (1)



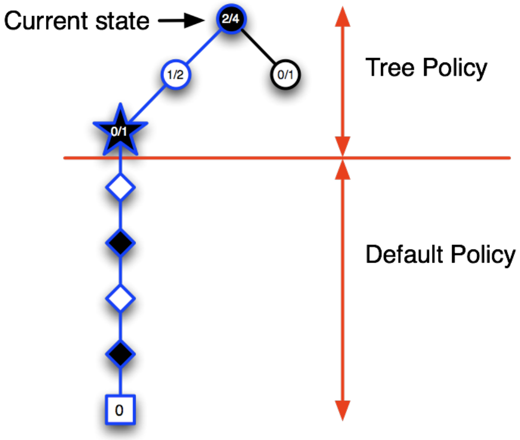
Applying Monte-Carlo Tree Search (2)



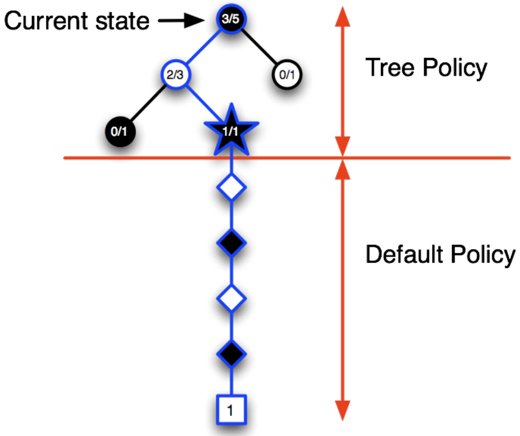
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



Applying Monte-Carlo Tree Search (5)



Advantages of Monte-Carlo Tree Search

- Highly selective best-first search
- Evaluates states dynamically (unlike e.g. DP)
- Uses sampling to break curse of dimensionality
- Works for “black-box” models (only requires samples)
- Computationally efficient, anytime, parallelisable

Search tree and value function approximation - I

- Search tree is a table lookup approach
- Based on a partial instantiation of the table
- For model-free reinforcement learning, table lookup is naive
  - Can't store value for all states
  - Doesn't generalise between similar states
- For simulation-based search, table lookup is less naive
  - Search tree stores value for easily reachable states
  - But still doesn't generalise between similar states
  - In huge search spaces, value function approximation is helpful