CSE 190 – Intro to Deep RL
5/1 – RL and Search Combined

Prithviraj Ammanabrolu and Bosung Kim

Thanks to David Silver's DeepMind RL Course and Rich Sutton's RL Book. Some slides were adapted form there.

Logistics

- Project proposal due tonight

Lecture 5/1

- RL and Search Combined
- Model Based RL
- PyTorch Review

Temporal Difference

- With *Monte Carlo*, we update the value function from a complete episode, and so we **use the actual accurate discounted return of this episode.**

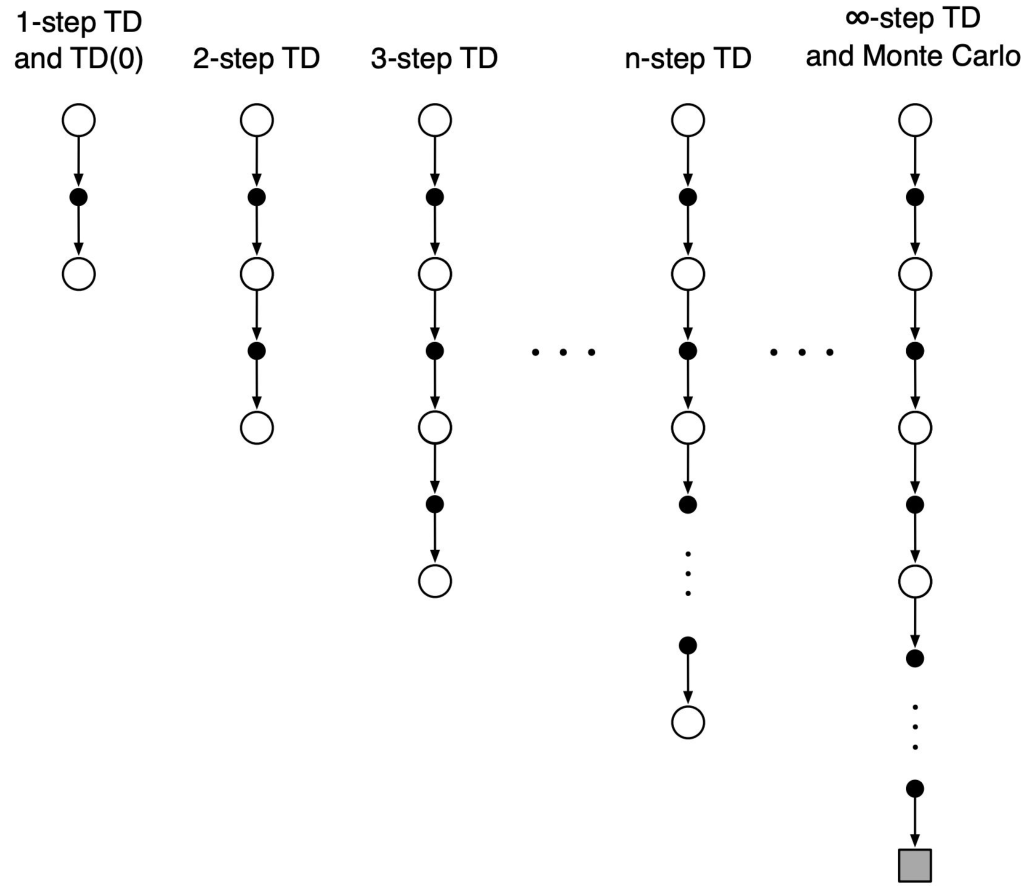
$$\text{Monte Carlo:} \quad V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

- With *TD Learning*, we update the value function from a step, and we replace G_t, which we don't know, with **an estimated return called the TD target – a bootstrapping method similar to DP**

$$\text{TD Learning:} \quad V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

TD(0) □ TD(∞)

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$



Off-policy Learning

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- While following behavior policy $\mu(a|s)$

$$\{S_1, A_1, R_2, ..., S_T\} \sim \mu$$

Why is this important?

- Learn from observing humans or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, ..., \pi_{t-1}$
- Learn about optimal policy while following exploratory policy
- Learn about multiple policies while following one policy

Q-Learning

- We now allow both behavior and target policies to improve
- The target policy π is greedy w.r.t. $Q(s, a)$
- $\pi(S_{t+1}) = \text{argmax}_{a'} Q(S_{t+1}, a')$
- The behavior policy μ is e.g. ϵ-greedy w.r.t. $Q(s, a)$
- The Q-learning target then simplifies:

$$R_{t+1} + \gamma Q(S_{t+1}, A0\ )$$
$$= R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}_{a'} Q(S_{t+1}, a'))$$
$$= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a')$$

Value Function Approximation

- So far we have represented value function by a lookup table
- Every state s has an entry V(s)
- Or every state-action pair s, a has an entry Q(s, a)
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with function approximation

    $$\hat{v}(s, w) \approx v_\pi(s) \text{ or } \hat{q}(s, a, w) \approx q_\pi(s, a)$$

  - Generalize from seen states to unseen states
  - Update parameter w using MC or TD learning

Can you do better if you have a Model?

- Everything so far was Model Free
  - No model
  - Learn value function (and/or policy) from experience
- If you know how the world will change in response to your action before you do it, can you use that somehow to influence your actions?
- This is the problem of “given a world model” how to use it.

Model Free vs Model Based RL

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience
- Model-Based RL
  - Learn a model from experience
  - **Plan value function (and/or policy) from model**

Sample Based Planning

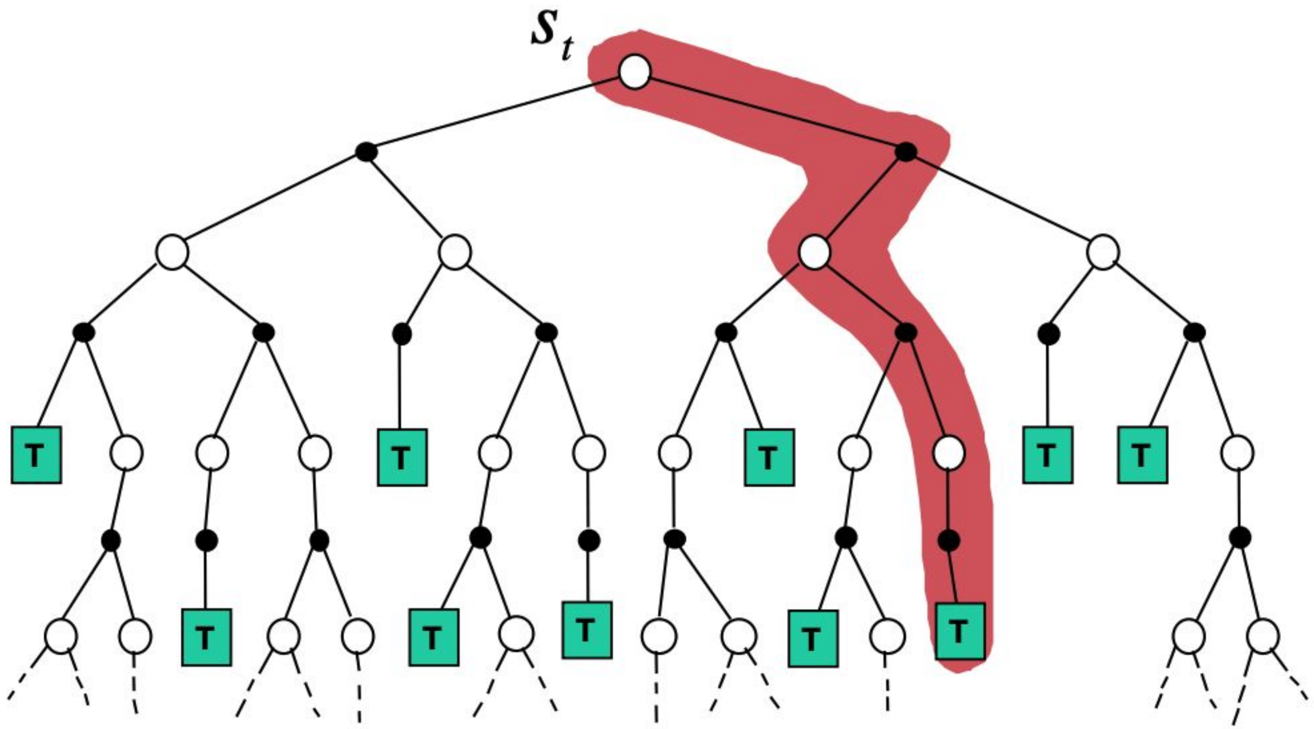
- A simple but powerful approach to planning
- Use the model only to generate samples
- Sample experience from model

  $S_{t+1} \sim T_\eta(S_{t+1} \mid S_t , A_t)$

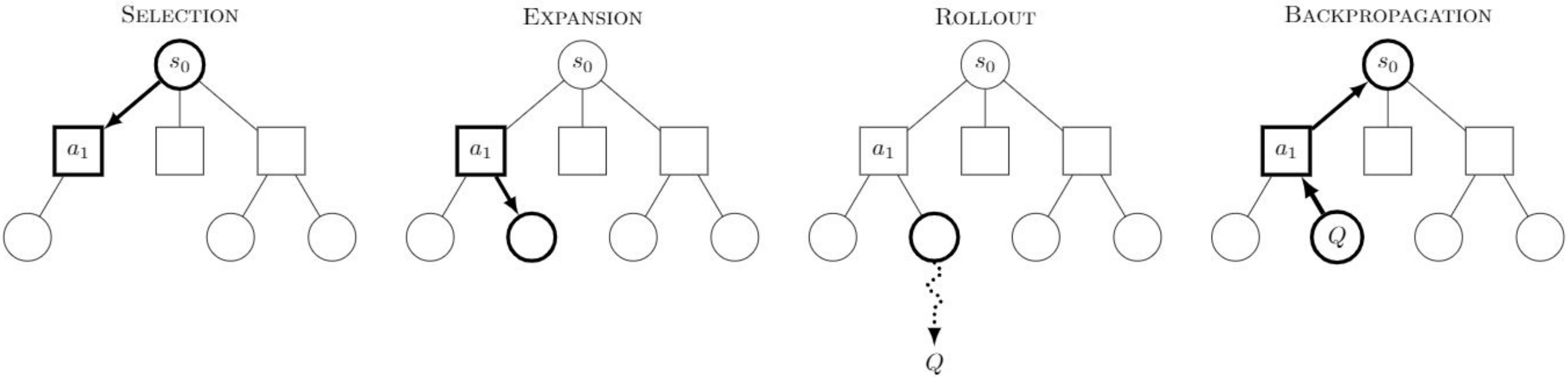
  $R_{t+1} = R_\eta(R_{t+1} \mid S_t , A_t)$
- Apply model-free RL to samples, e.g.: Monte-Carlo control Sarsa Q-learning
- Sample-based planning methods are often more efficient

Simulation Search

- Forward search paradigm using sample-based planning
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



Revisit MCTS



MCTS (contd)

- Given a model M_v and a simulation policy π
- For each action $a \in A$
  - Simulate K episodes from current (real) state

    $s_t \{s_t, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, \ldots, S^k_T\}^K_{k=1} \sim M_{v,\pi}$
  - Evaluate actions by mean return (Monte-Carlo evaluation)

    $Q(s_t, a) = 1/K \sum_{k=1}^{K} G_t \rightarrow q_\pi(s_t, a)$
- Select current (real) action with maximum value

  $a_t = \text{argmax}_{a \in A} Q(s_t, a)$

MCTS Evaluation

- Given a model M_v
- Simulate K episodes from current state st using current simulation policy π $\{s^t, A^k{}_t, R^k{}_{t+1}, S^k{}_{t+1}, A^k{}_{t+1}, \ldots, S^k{}_T\}^K{}_{k=1} \sim M_{v,\pi}$
- Build a search tree containing visited states and actions
- Evaluate states Q(s, a) by mean return of episodes from s, a

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

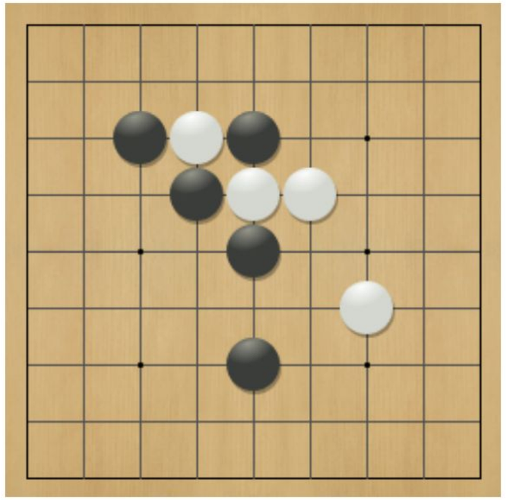
- After search is finished, select current (real) action with maximum value in search tree $a_t = \text{argmax}_{a \in A} Q(s_t, a)$

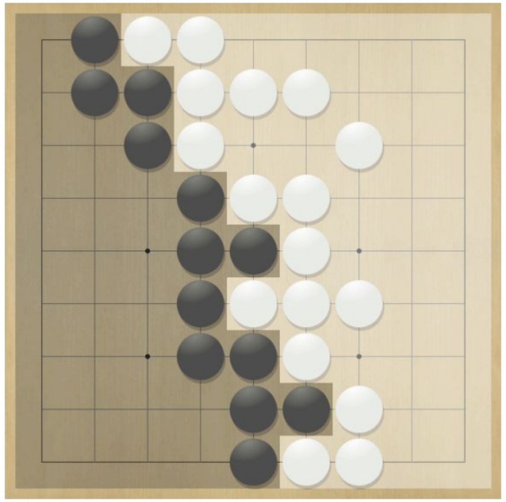
MCTS Simulation

- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximize $Q(S, A)$
  - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
  - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
  - Improve tree policy, e.g. by − greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q^*(S, A)$

Go Case Study

- Usually played on 19x19, also 13x13 or 9x9 board
- Simple rules, complex strategy
- Black and white place down stones alternately
- Surrounded stones are captured and removed
- The player with more territory wins the game





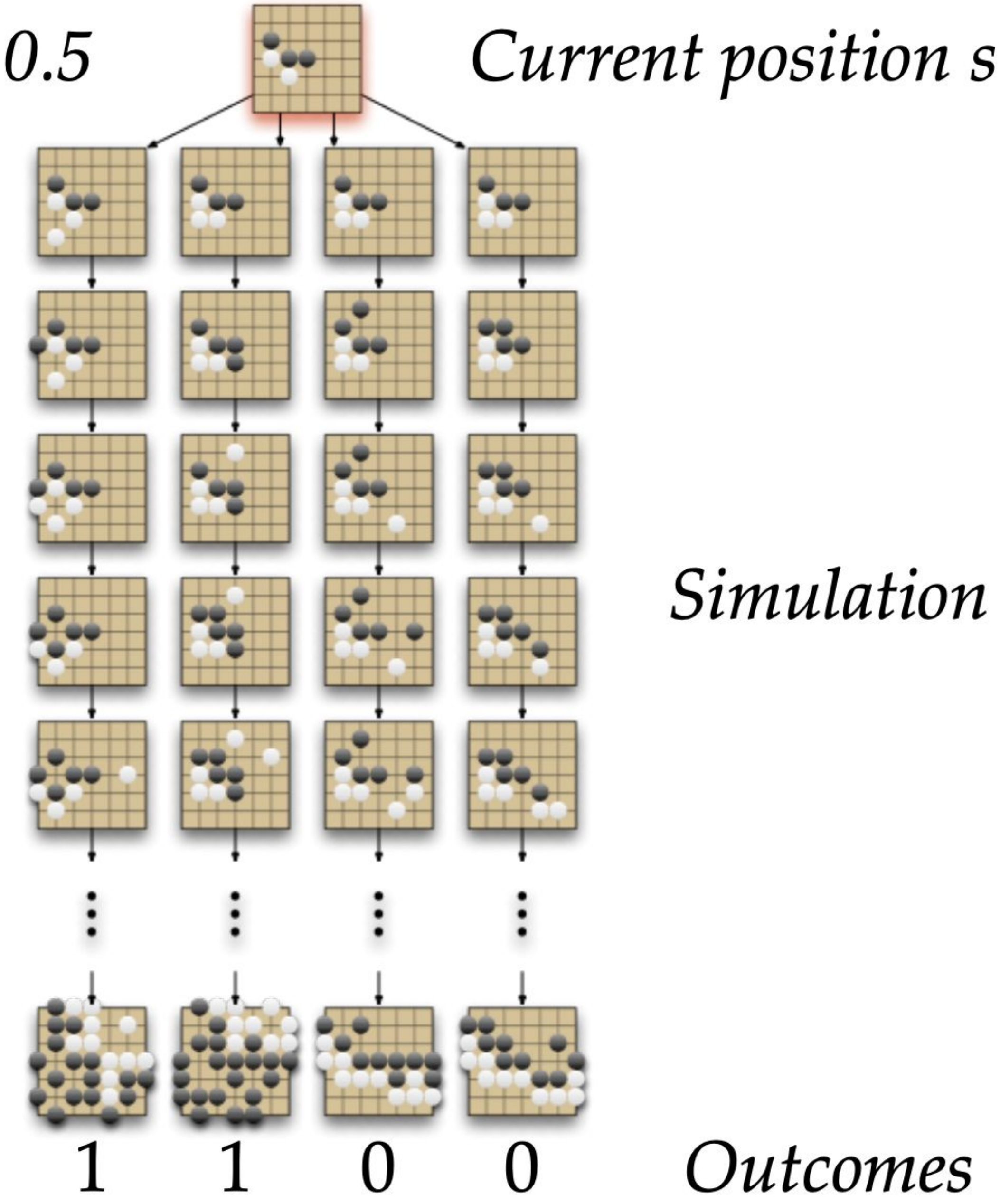
Go Case Study

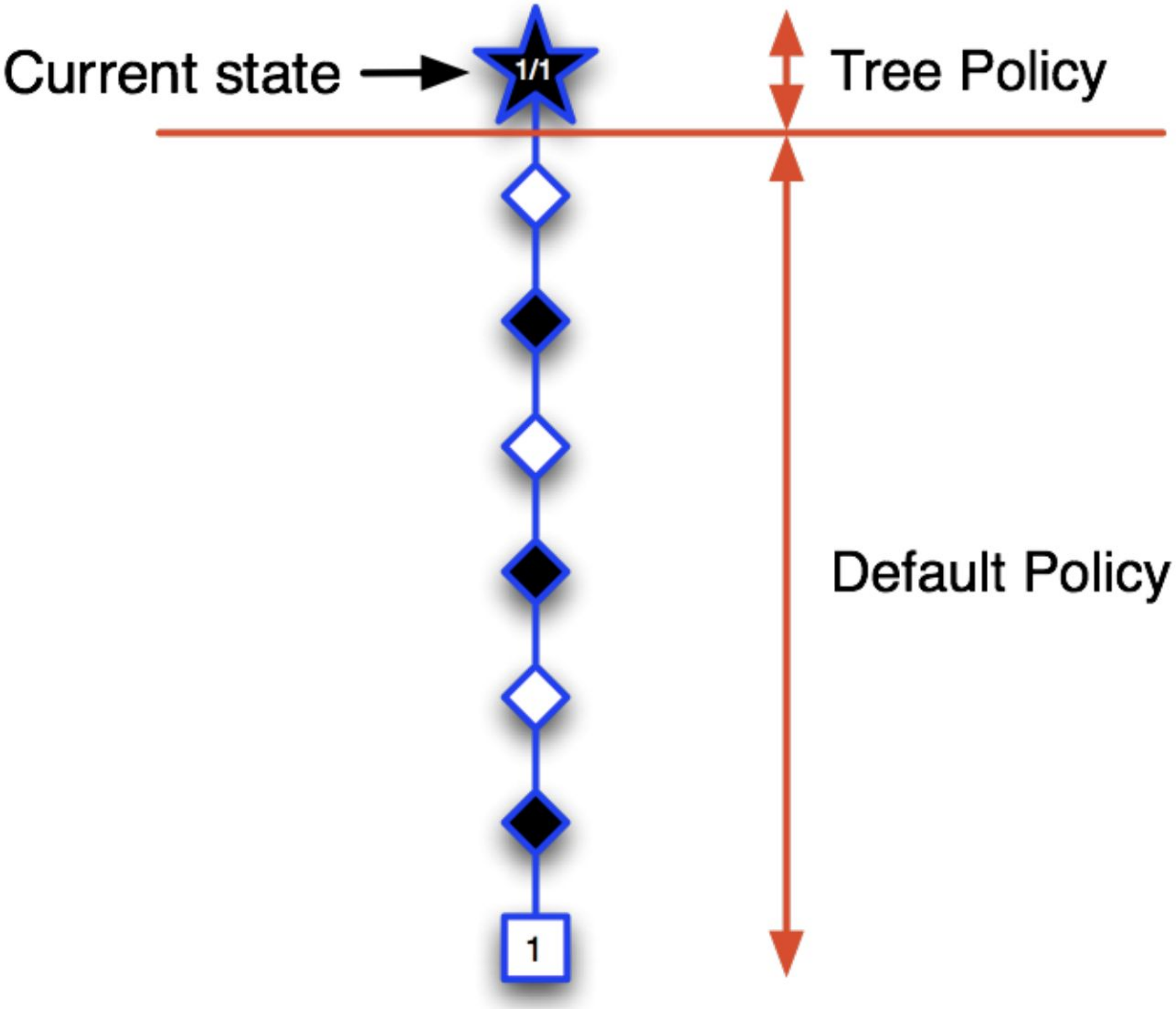
- How good is a position s?
- Reward function (undiscounted):
  - $R_t = 0$ for all non-terminal steps $t < T$
  - $R_T = 1$ if Black wins
  - $R_T = 0$ if White wins
- Policy $\pi = \langle \pi_B , \pi_W \rangle$ selects moves for both players, Self Play
- Value function (how good is position s):

$$v_\pi(s) = \mathbb{E}_\pi [R_T \mid S = s] = P [\text{Black wins} \mid S = s]$$

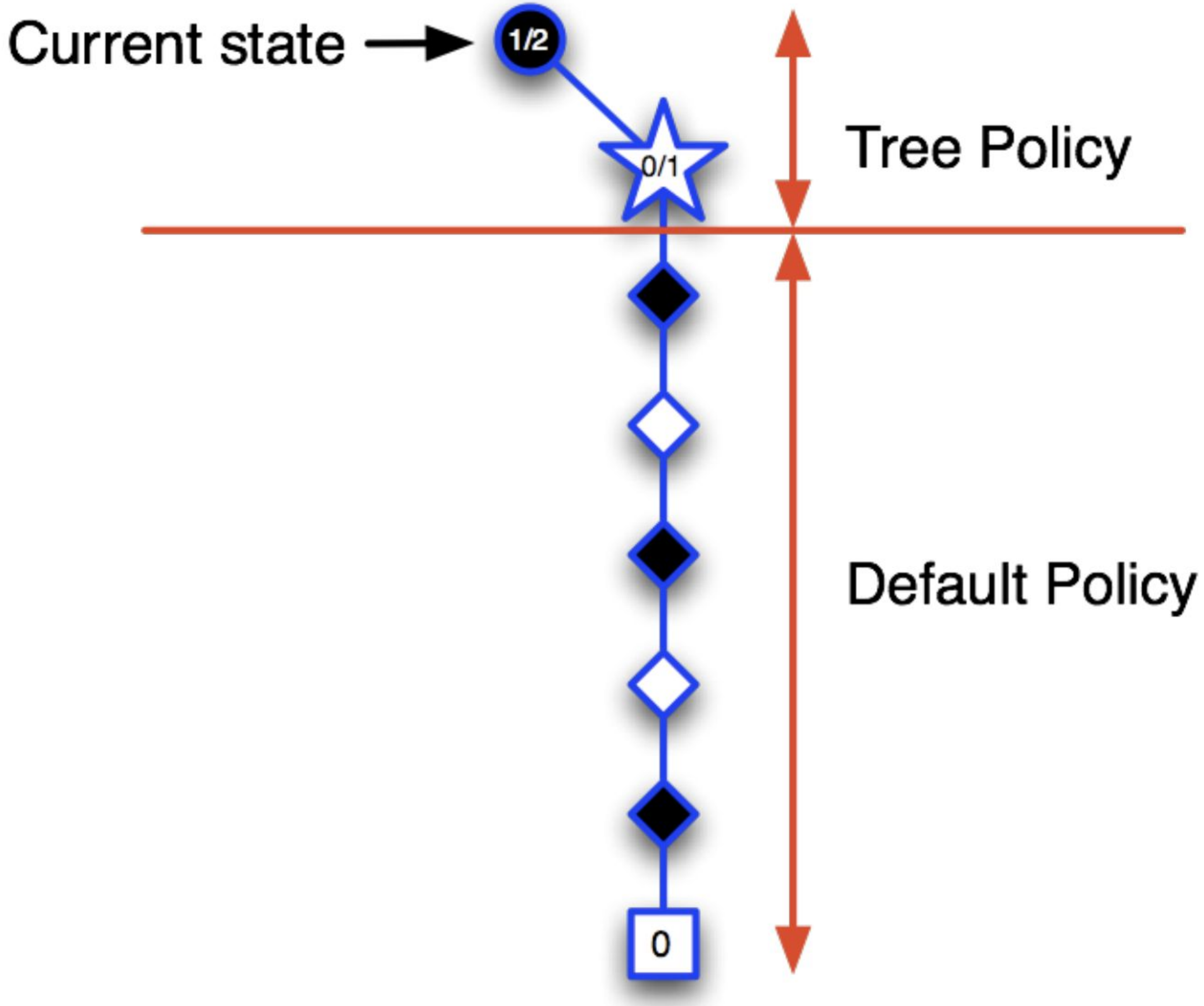
$$v^*(s) = \max_{\pi B} \min_{\pi W} v_\pi(s)$$

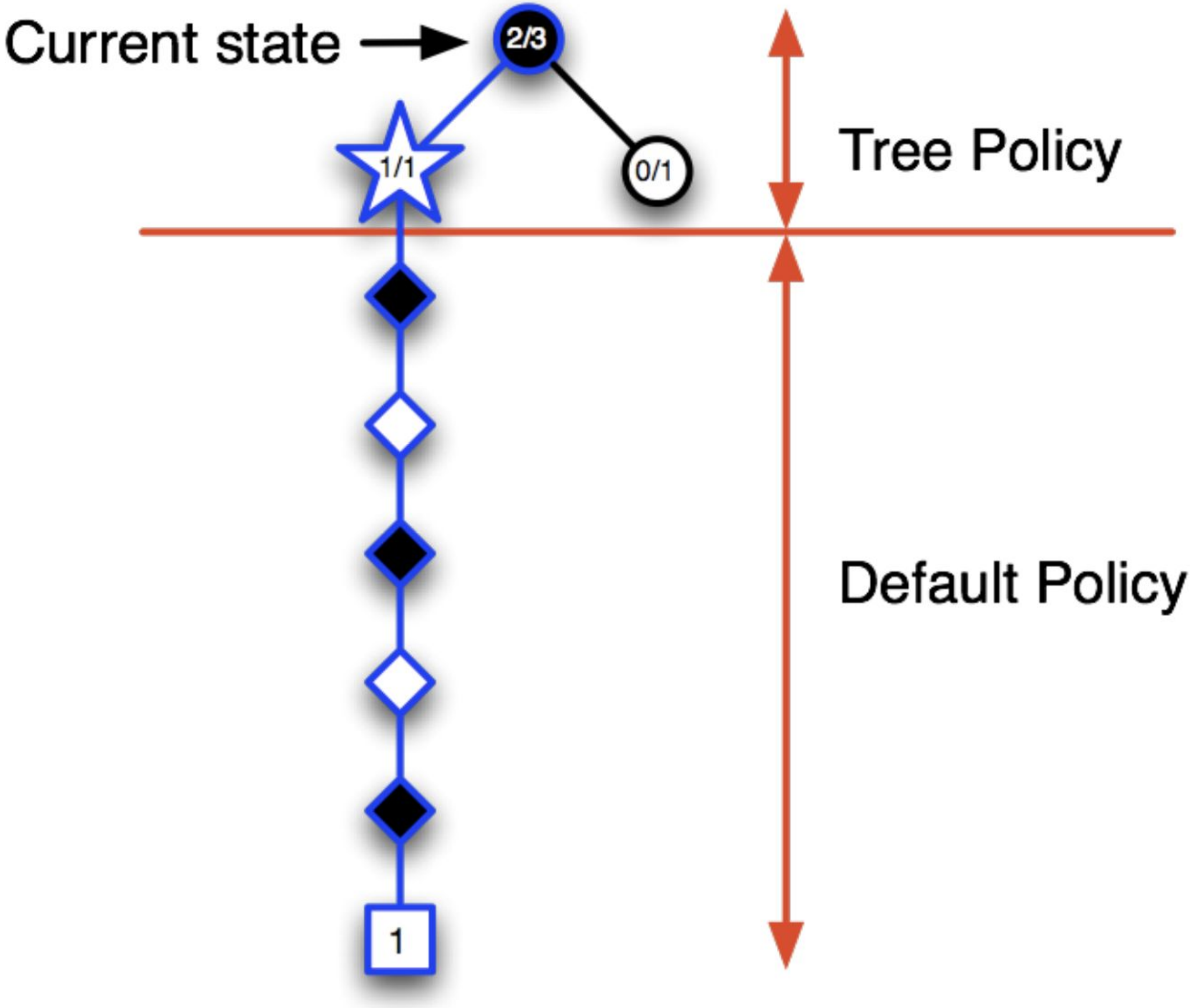
$V(s) = 2/4 = 0.5$



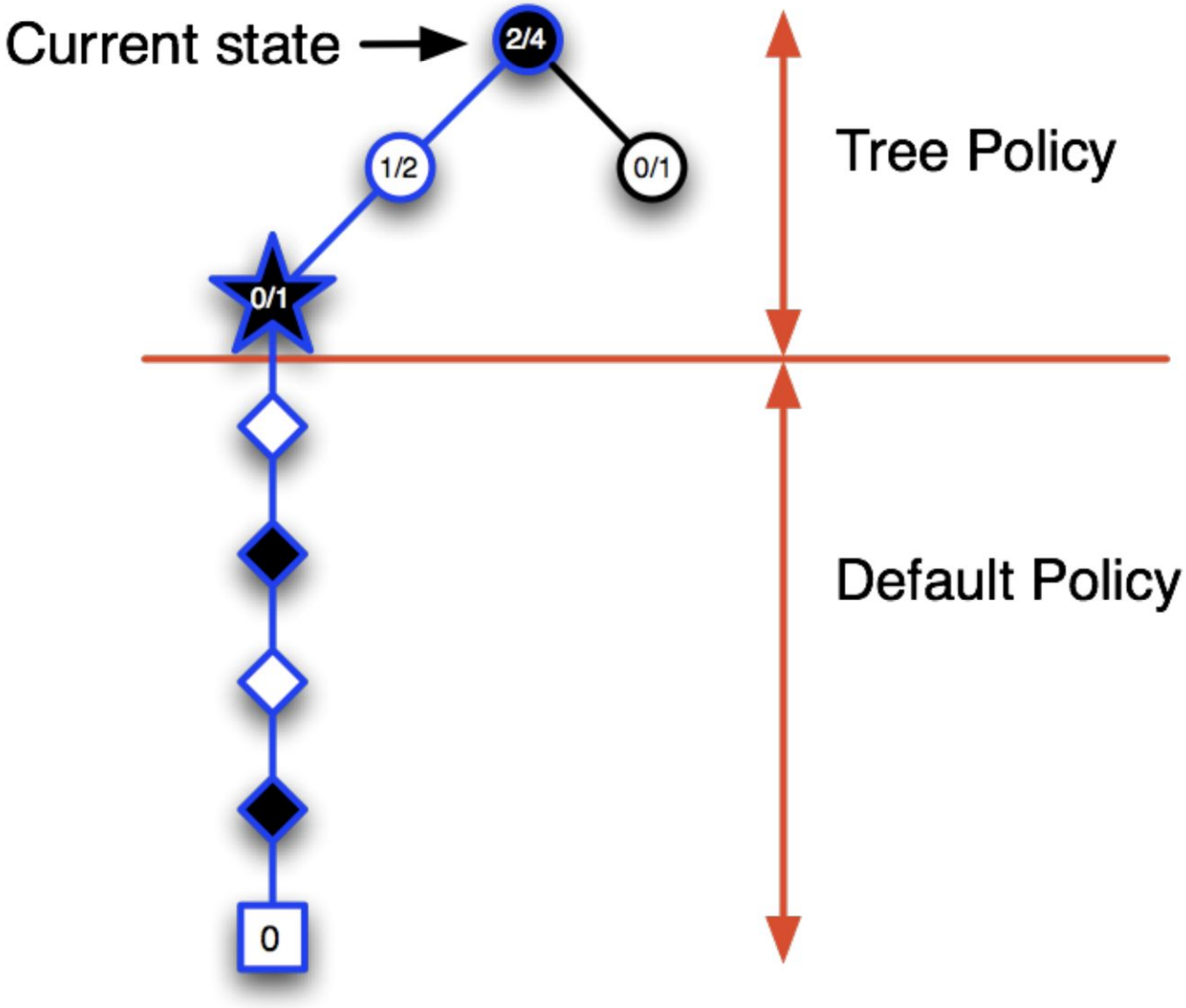


Current state
1/1
Tree Policy
Default Policy
1


Current state
1/2
0/1
0
Tree Policy
Default Policy



Current state
2/3
1/1
0/1
Tree Policy
Default Policy
1



Current state
2/4
1/2
0/1
0/1
0
Tree Policy
Default Policy

TD Search

- Simulate episodes from the current (real) state s_t
- Estimate action-value function Q(s, a)
- For each step of simulation, update action-values by

  $\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$
- Select actions based on action-values Q(s, a) e.g. -greedy
- May also use function approximation for Q

AlphaGo

- Same exact MC method as what we just talked about
- Just use neural nets to learn the probabilities using self play and outcome rewards
- Needed a lot of human games to train the initial value networks
- Also had some hand crafted features to bake in knowledge about the game

AlphaZero

- Relaxed the constraint of requiring a lot of human data and constraints up front by just scaling
- Just do pure online RL

Model Free vs Model Based RL

- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience
- Model-Based RL
  - **Learn a model from experience**
  - Plan value function (and/or policy) from model

What is a Model?

- A model M is a representation of an MDP <S, A,T, R>, parametrized by η
- We will assume state space S and action space A are known
- So a model M = $<T_\eta, R_\eta>$ represents state transitions $T_\eta \approx T$ and rewards $R\eta \approx R$

  $S_{t+1} \sim P\eta(S_{t+1} \mid S_t , A_t)$
  $R_{t+1} = R\eta(R_{t+1} \mid S_t , A_t)$
- Typically assume conditional independence between state transitions and rewards

  $P [S_{t+1}, R_{t+1} \mid S_t , A_t] = P[S_{t+1} \mid S_t , A_t] P[R_{t+1} \mid S_t , A_t]$

Learning a Model

- Goal: estimate model M_η from experience $\{S_1, A_1, R_2, ..., S_T\}$
- This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$
$$S_2, A_2 \rightarrow R_3, S_3$$
$$\dots S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- Learning $s, a \rightarrow r$ is a regression problem
- Learning $s, a \rightarrow s'$ is a density estimation problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
  Find parameters η that minimizes empirical loss

Model Based RL

- Pick your fav simulation search algo from before and do planning with your model
- Key difference here is that the Model has errors, uncertainty
- What does this mean for how many steps you need to take in an env?

Model Based RL

- Pick your fav simulation search algo from before and do planning with your model
- Key difference here is that the Model has errors, uncertainty
- It will take a lot longer! (Why?)
- This is the overall concept behind MuZero, simultaneously learn both model and policy
  - work well in enviroments where we don’t know the true dynamics
  - e.g., video games, robotics, or real-world systems.

Models and Simulation and Reality

- Traditionally we consider two sources of experience
- Real experience: Sampled from environment (true MDP)

  $S' \sim T^{a}_{s,s'}$

  $R = R^{a}_{s}$
- Simulated experience: Sampled from model (approximate MDP)

  $S' \sim T_{\eta}(S' \mid S, A)$

  $R = R_{\eta}(R \mid S, A)$
- What’s the issue with World Models learned inside a simulation?

Pros and Cons of MBRL

- Pros
  - Can do all the (self, un) supervised learning tricks to learn from large scale data
  - Can reason about uncertainty
- Cons
  - Need model of T first
  - Will build estimate of value from that
  - Two(+) sources of error
    - error in your model
    - error in your value estimation that builds on top of the model

PyTorch Review

Neural Networks with PyTorch

PyTorch is a popular deep learning framework used for building neural networks

Matrix Multiplication

```python
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
result = [[0, 0], [0, 0]]

for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]
```

Python List

```python
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

result = np.matmul(A, B)
```

Numpy

```python
import torch

A = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
B = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

result = torch.matmul(A, B)
```

PyTorch

Neural Networks with PyTorch

PyTorch is a popular deep learning framework used for building neural networks

PyTorch uses the `nn.Module` class to define models

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.linear = nn.Linear(10, 1)

    def forward(self, x):
        return self.linear(x)
```

`nn.Linear()`: Linear transformation

$$y = xW + b$$

```
layer = nn.Linear(10, 1)
out = layer(torch.randn(5, 10))  # batch of 5 samples
```

Neural Networks with PyTorch

PyTorch is a popular deep learning framework used for building neural networks

PyTorch uses the `nn.Module` class to define models

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.linear = nn.Linear(10, 1)

    def forward(self, x):
        return self.linear(x)
```

`forward()`: how input flows through the network

DQN with PyTorch

`forward()`: how input flows through the network

```
class DQN(nn.Module):
  def __init__(self, config):
    super(DQN, self).__init__()
    self.state_network = StateNetwork(config)
    self.act_scorer = nn.Linear(config.hidden_size, config.act_size)

  def forward(self, state):
    """
      the output should be (BATCH_SIZE, ACTION_SIZE): the estimated Q-values
    """
    ### YOUR CODE BELOW HERE
    raise NotImplementedError
    ### YOUR CODE ABOVE HERE
```

Training loop

Steps to train a model:

1. model
2. loss function
3. optimizer
4. loop over epochs and batches

```
model = MyModel()
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(10):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch}, Loss: {loss.item()}')
```

`optimizer.zero_grad()`: reset gradients

`loss.backward()`: compute gradients

`optimizer.steps()`: update weights

Training DQN

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

```python
class DQNAgent:
  def __init__(self,
               action_set,
               dqn_config,
               gamma,
               epsilon,
               learning_rate=0.0005,
               epsilon_decay=0.995,
               epsilon_min=0.01,
               batch_size=64,
               memory_size=100000,
               update_freq=4,
               update_freq_target=1000):
    self.act2id = {a: i for i, a in enumerate(action_set)}
    self.id2act = {i: a for i, a in enumerate(action_set)}

    self.update_freq = update_freq
    self.update_freq_target = update_freq_target
    self.max_seq_len = 256  # DO NOT CHANGE `max_seq_len`
    self.tokenizer =  AutoTokenizer.from_pretrained('gpt2')

    self.gamma = gamma
    self.epsilon = epsilon
    self.epsilon_decay = epsilon_decay
    self.epsilon_min = epsilon_min
    self.batch_size = batch_size
    self.replay_buffer = deque(maxlen=memory_size)
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.model = DQN(dqn_config).to(self.device)
    self.target_model = DQN(dqn_config).to(self.device)
    self.optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
```

Training DQN

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

`optimizer.zero_grad()`: reset gradients
`loss.backward()`: compute gradients
`optimizer.steps()`: update weights