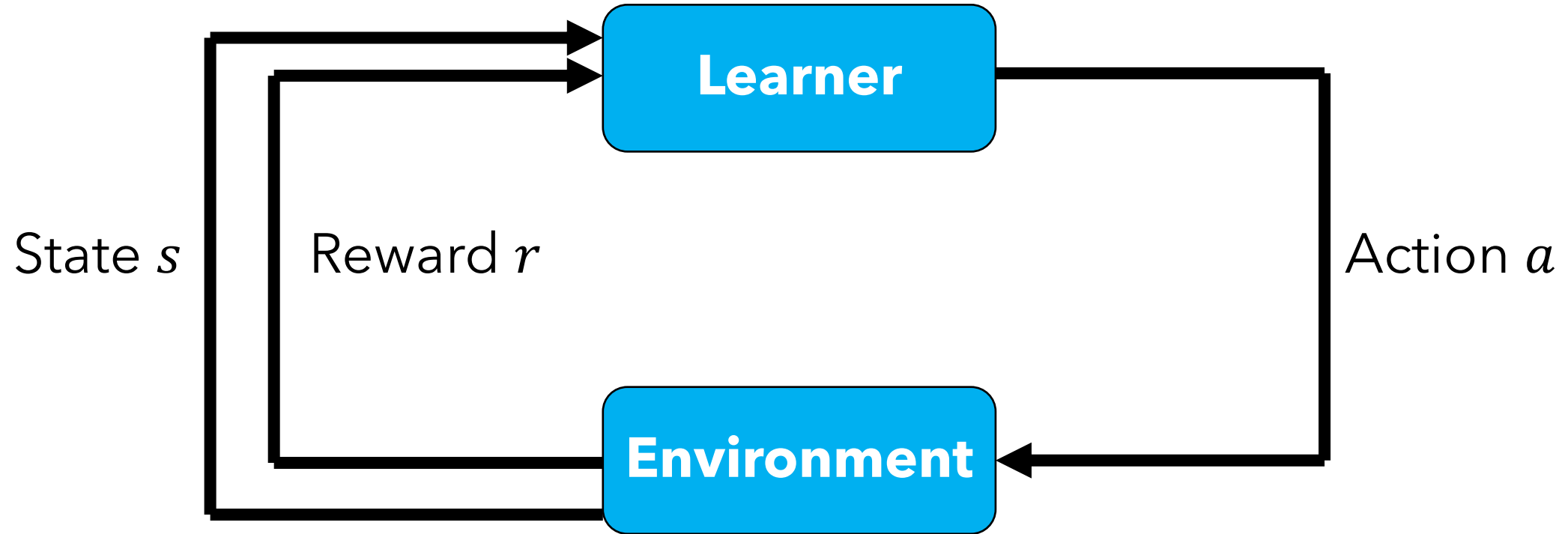


# Reinforcement learning refresher

Content draws on material by [Zico Kolter](#)

# Learner interaction with environment



# Outline

- 1. Markov decision processes**
2. Reinforcement learning
3. Branch-and-bound as an MDP

# Markov decision processes

- MDPs defined by:
  - States
  - Actions
  - Transition probabilities
  - Rewards
- **States**: encode how system will evolve when taking actions
- System governed by **transition probabilities**  $P(s_{t+1} \mid s_t, a_t)$ 
  - Only depend on **current** state and action (Markov assumption)
- **Agent's goal**: take actions that maximize expected reward

# Markov decision processes

$S$ : set of states (assumed for now to be discrete)

$A$ : set of actions

Transition probability distribution  $P(s_{t+1} \mid s_t, a_t)$

*Probability of entering state  $s'$  from state  $s$  after taking action  $a$*

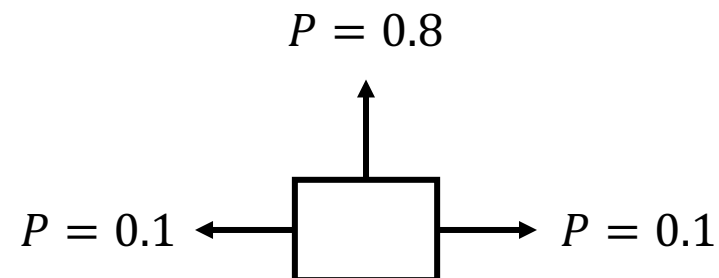
Reward function  $R: S \rightarrow \mathbb{R}$

**Goal:** Policy  $\pi: S \rightarrow A$  that maximizes total (discounted) reward

# Gridworld domain

- Goal state with reward 2
- "Bad state" with reward -100
- Actions move:
  - North with probability 0.8
  - East or west with probability 0.1
- Action that would bump into a wall leaves agent where it is

0	0	0	1
0		0	-100
0	0	0	0



# Policies and value functions

Policy is a mapping from states to actions  $\pi: S \rightarrow A$

## **Value function for a policy:**

Expected sum of discounted rewards

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, a_t = \pi(s_t), s_{t+1} \mid s_t, a_t \sim P \right]$$



Discount factor

# Bellman equation

Can also define  $V^\pi(s)$  recursively via the **Bellman equation**:

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P(s' \mid s, \pi(s)) V^\pi(s')$$



# Computing the policy value

- $\mathbf{v}^\pi \in \mathbb{R}^{|S|}$  is a vector of **values** for each state
- $\mathbf{r} \in \mathbb{R}^{|S|}$  is a vector of **rewards** for each state
- $P^\pi \in \mathbb{R}^{|S| \times |S|}$  contains the **transition probabilities** under  $\pi$

$$P_{ij}^\pi = P(s_{t+1} = i \mid s_t = j, a_t = \pi(s_t))$$

- **Bellman equation** can be written in vector form as

$$\begin{aligned}\mathbf{v}^\pi &= \mathbf{r} + \gamma P^\pi \mathbf{v}^\pi \\ \Rightarrow (I - \gamma P^\pi) \mathbf{v}^\pi &= \mathbf{r} \\ \Rightarrow \mathbf{v}^\pi &= (I - \gamma P^\pi)^{-1} \mathbf{r}\end{aligned}$$

i.e., computing the policy value requires solving a **linear system**

# Optimal policy and value function

**Optimal policy** achieves the highest value for every state

$$\pi^* = \operatorname{argmax}_{\pi} V^{\pi}(s)$$

Value function is written  $V^* = V^{\pi^*}$

There are an exponential number of policies

⇒ Formulation is not very useful

# Optimal policy and value function

Instead, define  $V^*(s)$  using the **Bellman optimality equation**

$$V^*(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' | s, a) V^*(s')$$

Optimal policy is simply the action that attains this max

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s' \in \mathcal{S}} P(s' | s, a) V^*(s')$$

# Outline

1. Markov decision processes
  - i. **Computing the optimal policy**
    - a. **Value iteration**
    - b. Policy iteration
2. Reinforcement learning
3. Branch-and-bound as an MDP

# Computing the optimal policy

## Approach #1: value iteration

Repeatedly update estimate of the optimal value function  
(according to Bellman optimality equation)

1.  $\hat{V}(s) \leftarrow 0, \forall s \in S$
2. Repeat:

$$\hat{V}(s) \leftarrow R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in S} P(s' | s, a) \hat{V}(s')$$

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s' \in S} P(s' | s, a) V^*(s')$$

# Computing the optimal policy

## Approach #1: value iteration

Repeatedly update estimate of the optimal value function  
(according to Bellman optimality equation)

1.  $\hat{V}(s) \leftarrow 0, \forall s \in S$
2. Repeat:

$$\hat{V}(s) \leftarrow R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in S} P(s' | s, a) \hat{V}(s')$$

**Theorem:** Value iteration converges to optimal value:  $\hat{V} \rightarrow V^*$

# Illustration of value iteration

Running value iteration with  $\gamma = 0.9$

0	0	0	1
0		0	-100
0	0	0	0

Original reward function

# Illustration of value iteration

Running value iteration with  $\gamma = 0.9$

0	0	0.72	1.81
0		0	-99.91
0	0	0	0

$\hat{V}$  at 1 iteration



# Illustration of value iteration

Running value iteration with  $\gamma = 0.9$

0.809	1.598	2.475	3.745
0.268		0.302	-99.59
0	0.034	0.122	0.004

$\hat{V}$  at 5 iterations

# Illustration of value iteration

Running value iteration with  $\gamma = 0.9$

2.686	3.527	4.402	5.812
2.021		1.095	-98.82
1.390	0.903	0.738	0.123

$\hat{V}$  at 10 iterations

# Illustration of value iteration

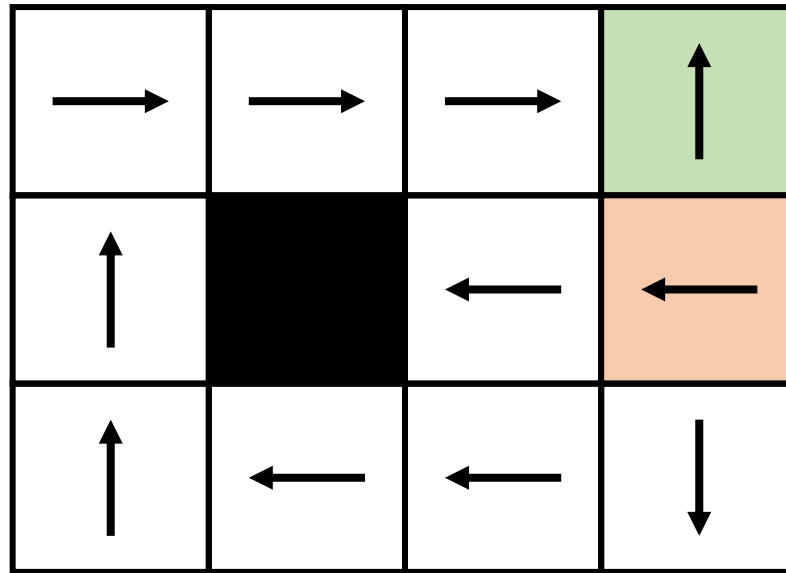
Running value iteration with  $\gamma = 0.9$

5.470	6.313	7.190	8.669
4.802		3.347	-96.67
4.161	3.654	3.222	1.526

$\hat{V}$  at 1000 iterations

# Illustration of value iteration

Running value iteration with  $\gamma = 0.9$



Resulting policy after 1000 iterations

# Outline

1. Markov decision processes
  - i. Computing the optimal policy
    - a. Value iteration
    - b. Policy iteration**
2. Reinforcement learning
3. Branch-and-bound as an MDP

# Policy iteration

1. Initialize policy  $\pi$  randomly
2. Compute value of policy  $V^\pi$  (e.g., by solving linear system)
3. Update  $\pi$  to be greedy policy with respect to  $V^\pi$

$$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s' \in S} P(s' | s, a) V^\pi(s')$$

4. If policy  $\pi$  changed in last iteration, return to step 2

**Theorem:** Policy iteration converges to optimal policy:  $\pi \rightarrow \pi^\star$

# Illustration of policy iteration

Running policy iteration with  $\gamma = 0.9$ , initialize with  $\pi(s) = \text{North}$

0	0	0	1
0		0	-100
0	0	0	0

Original reward function

# Illustration of policy iteration

Running policy iteration with  $\gamma = 0.9$ , initialize with  $\pi(s) = \text{North}$

0.418	0.884	2.331	6.367
0.367		-8.610	-105.7
-0.168	-4.641	-14.27	-85.05

$V^\pi$  at iteration 1



# Illustration of policy iteration

Running policy iteration with  $\gamma = 0.9$ , initialize with  $\pi(s) = \text{North}$

5.414	6.248	7.116	8.634
4.753		2.881	-102.7
2.251	1.977	1.849	-8.701

$V^\pi$  at iteration 2

# Illustration of policy iteration

Running policy iteration with  $\gamma = 0.9$ , initialize with  $\pi(s) = \text{North}$

5.470	6.313	7.190	8.669
4.803		3.347	-96.67
4.161	3.654	3.222	1.526

$V^\pi$  at iteration 3 (converged)

# Gridworld results

## Approximation of value function

- Policy iteration: exact value function after three iterations
- Value iteration: after 100 iterations,  $\|V - V^*\|_2 = 7.1 \cdot 10^{-4}$

## Calculation of optimal policy

- Policy iteration: three iterations
- Value iteration: 12 iterations

VI converges to  $\pi^*$  long before it converges to  $V^*$  in this MDP

*But this property is highly MDP-specific*

# Policy iteration or value iteration?

Policy iteration requires **fewer iterations** than value iteration

- But each iteration requires solving a linear system
- Only need to apply Bellman operator for value iteration

In practice, policy iteration is often **faster**

- Especially if the transition probabilities are structured (e.g., sparse)  
⇒ Solving linear system is efficient

# Outline

1. Markov decision processes
- 2. Reinforcement learning**
3. Branch-and-bound as an MDP

# Challenge of RL

## **MDP** ( $S, A, P, R$ ):

- $S$ : set of states (assumed for now to be discrete)
- $A$ : set of actions
- Transition probability distribution  $P(s_{t+1} \mid s_t, a_t)$
- Reward function  $R: S \rightarrow \mathbb{R}$

**RL twist:** We don't know  $P$  or  $R$ , or too big to enumerate

# Model-based RL

- A simple approach: just **estimate the MDP** from data
- Agent acts according to some policy, observes

$$s_1, r_1, a_1, s_2, r_2, a_2, \dots, s_m, r_m, a_m$$

- We form the **empirical estimate** of the MDP:

$$\hat{P}(s' | s, a) = \frac{\sum_{i=1}^{m-1} \mathbf{1}\{s_i = s, a_i = a, s_{i+1} = s'\}}{\sum_{i=1}^{m-1} \mathbf{1}\{s_i = s, a_i = a\}}$$

$$\hat{R}(s) = \frac{\sum_{i=1}^m \mathbf{1}\{s_i = s\} r_i}{\sum_{i=1}^m \mathbf{1}\{s_i = s\}}$$

- Now solve the MDP  $(S, A, \hat{P}, \hat{R})$

# Model-based RL

Will **converge** to correct MDP (and hence correct policy)

How can we ensure we get the “right” samples?

**Advantages** (informally): makes “efficient” use of data

## **Disadvantages:**

- Requires we build the the actual MDP models
- State space may be too large



# Outline

1. Markov decision processes
2. Reinforcement learning
  - i. **Model-free RL**
    - a. **Temporal difference methods**
    - b. Q-learning
    - c. Function approximation
  - ii. Exploration vs exploitation
3. Branch-and-bound as an MDP

# Model-free RL

**Temporal difference methods** (TD, SARSA, Q-learning):

Directly learn value function  $V^\pi$

# Temporal difference (TD) methods

- Consider computing  $V^\pi$  via the update

$$\hat{V}^\pi(s) \leftarrow R(s) + \gamma \sum_{s' \in S} P(s' | s, \pi(s)) \hat{V}^\pi(s'), \quad \forall s \in S$$

- We're in state  $s_t$ , receive  $r_t$ , take action  $a_t = \pi(s_t)$ , end in  $s_{t+1}$
- Can't update  $\hat{V}^\pi$  for all  $s$ , but can we update **just for  $s_t$** ?

$$\hat{V}^\pi(s_t) \leftarrow r_t + \gamma \sum_{s' \in S} P(s' | s_t, a_t) \hat{V}^\pi(s')$$

- ...No, still can't compute this sum

# Temporal difference (TD) methods

But,  $s_{t+1}$  is a sample from the distribution  $P(s' | s_t, a_t)$

Could perform the update  $\hat{V}^\pi(s_t) \leftarrow r_t + \gamma \hat{V}^\pi(s_{t+1})$

- Too “harsh” an assignment
- Assumes that  $s_{t+1}$  is the only possible next state

Instead “smooth” the update using some  $\alpha < 1$

$$\hat{V}^\pi(s_t) \leftarrow (1 - \alpha) \hat{V}^\pi(s_t) + \alpha (r_t + \gamma \hat{V}^\pi(s_{t+1}))$$

This is the **temporal difference (TD) algorithm**

# Temporal difference (TD) algorithm

**algorithm**  $\hat{V}^\pi = \text{TD}(\pi, \alpha, \gamma)$

**initialize**  $\hat{V}^\pi(s) \leftarrow 0$

**repeat**

    Observe state  $s$  and reward  $r$

    Take action  $a = \pi(s)$  and observe next state  $s'$

$$\hat{V}^\pi(s) \leftarrow (1 - \alpha)\hat{V}^\pi(s) + \alpha \left( r + \gamma \hat{V}^\pi(s') \right)$$

return  $\hat{V}^\pi$

Will converge to  $\hat{V}^\pi(s) \rightarrow V^\pi(s)$  (for all  $s$  visited often enough)

# TD experiments

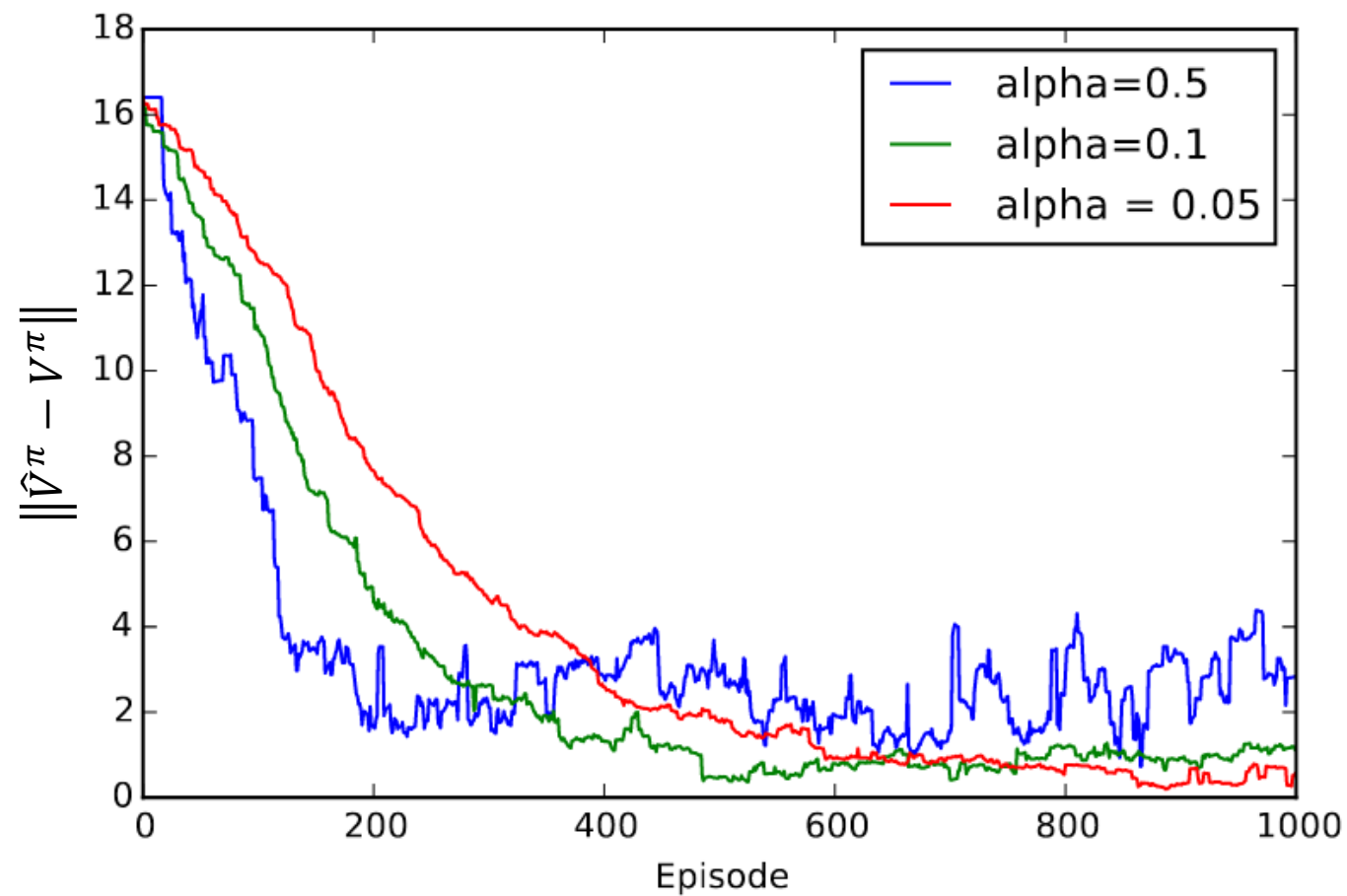
Run TD on gridworld domain for 1000 episodes. Each episode:

- 10 steps
- Sampled according to policy  $\pi$
- Starting at a random state

Initialize with  $\hat{V} = R$

0	0	0	1
0		0	-100
0	0	0	0

# TD progress



# Temporal difference (TD) algorithm

TD lets us **learn the value function** of a policy  $\pi$  directly  
*Don't ever need to construct the MDP*

But is this really that helpful?

Consider trying to execute greedy policy w.r.t. estimated  $\hat{V}^\pi$

$$\pi'(s) = \max_a \sum_{s'} T(s, a, s') \hat{V}^\pi(s')$$

We **need a model** anyway



# Outline

1. Markov decision processes
2. Reinforcement learning
  - i. Model-free RL
    - a. Temporal difference methods
    - b. Q-learning**
    - c. Function approximation
  - ii. Exploration vs exploitation
3. Branch-and-bound as an MDP

# Q-learning

## **Q functions:**

Like value functions but defined over state-action pairs

$$Q^{\pi}(s, a) = R(s) + \gamma \sum_{s' \in S} P(s' | s, a) Q^{\pi}(s', \pi(s'))$$

I.e., Q function is the value of:

1. Starting in state  $s$
2. Taking action  $a$
3. Then acting according to  $\pi$

# Q-learning

$$\begin{aligned} Q^*(s, a) &= R(s) + \gamma \sum_{s' \in S} P(s' \mid s, a) \max_{a'} Q^*(s', a') \\ &= R(s) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^*(s') \end{aligned}$$

$Q^*$  is the value of:

1. Starting in state  $s$
2. Taking action  $a$
3. Then acting optimally

# Q-learning

As with TD:

1. Observe  $s$  and reward  $r$
2. Take action  $a$  (but not necessarily  $a = \pi(s)$ )
3. Observe next state  $s'$

Estimate  $Q^*(s, a)$  as

$$\hat{Q}^*(s, a) \leftarrow (1 - \alpha)\hat{Q}^*(s, a) + \alpha \left( r + \gamma \hat{Q}^*(s', a') \right)$$

$\hat{Q}^* \rightarrow Q$  if all state-action pairs seen frequently enough

# Q-learning

As with TD:

1. Observe  $s$  and reward  $r$
2. Take action  $a$  (but not necessarily  $a = \pi(s)$ )
3. Observe next state  $s'$

Estimate  $Q^*(s, a)$  as

$$\hat{Q}^*(s, a) \leftarrow (1 - \alpha)\hat{Q}^*(s, a) + \alpha \left( r + \gamma \hat{Q}^*(s', a') \right)$$

We can now learn an optimal policy without an MDP model

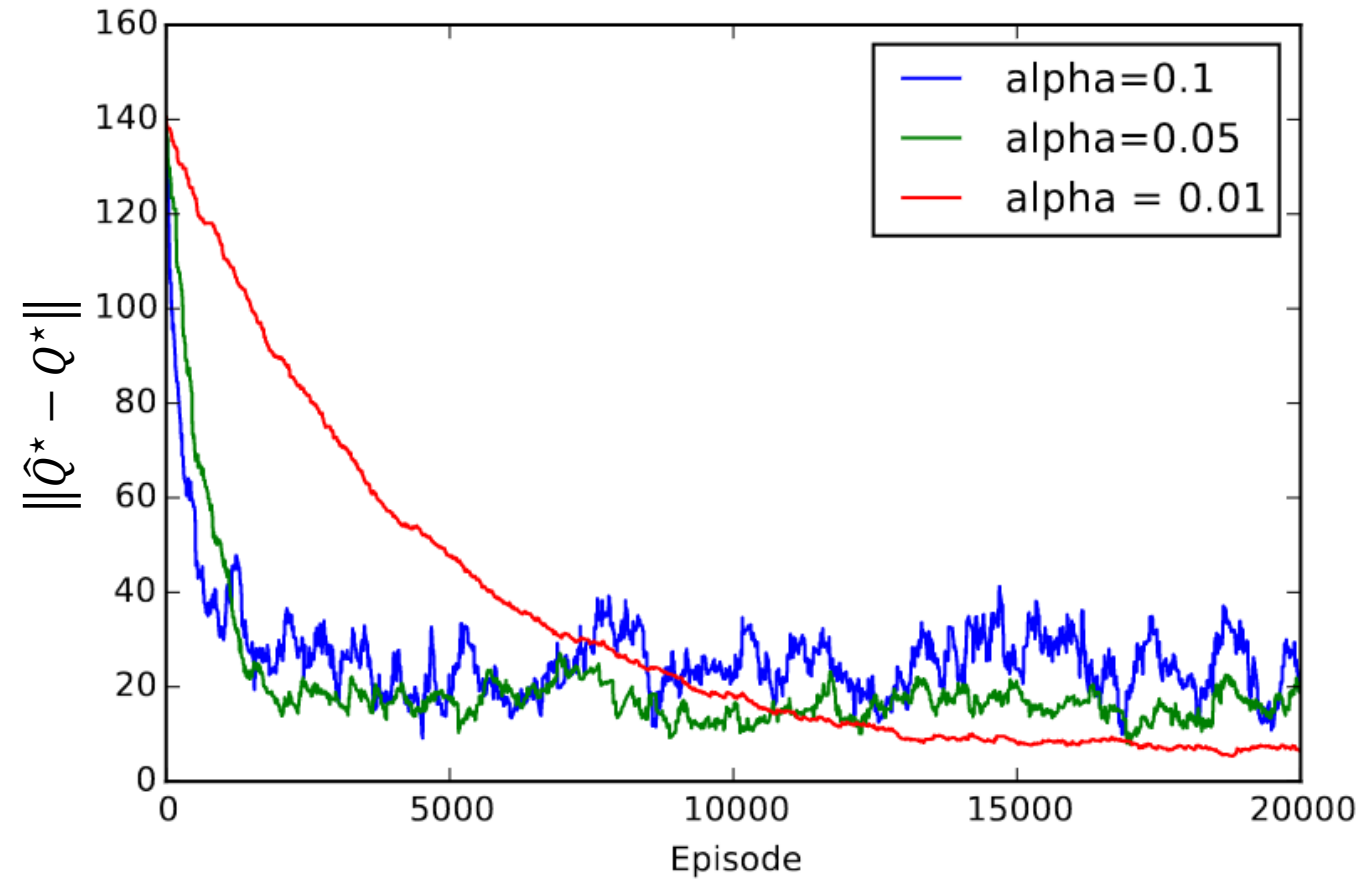
$$\pi^*(s) = \max_a \hat{Q}^*(s, a)$$

# Q-learning experiments

- Run Q-Learning on gridworld for 20000 episodes
  - 10 step per episode
- Initialize with  $\hat{Q}^*(s, a) = R(s)$
- Policy (epsilon-greedy): act according to current optimal
$$\pi^*(s) = \max \hat{Q}^*(s, a)$$
with probability 0.9, else act randomly

0	0	0	1
0		0	-100
0	0	0	0

# Q-learning progress



# Outline

1. Markov decision processes
2. Reinforcement learning
  - i. Model-free RL
    - a. Temporal difference methods
    - b. Q-learning
    - c. Function approximation**
  - ii. Exploration vs exploitation
3. Branch-and-bound as an MDP



# Function approximation

- How to avoid keeping track of each state?
- Major advantage to model-free RL methods:  
Can use **function approximation** to represent  $\hat{V}^\pi$  compactly
- Let  $\hat{V}^\pi(s) = f_\theta(s)$  be our approximator parameterized by  $\theta$
- TD update:  $\hat{V}^\pi(s) \leftarrow (1 - \alpha)\hat{V}^\pi(s) + \alpha(r + \gamma\hat{V}^\pi(s'))$
- Update  $\theta$ : ideally  $\operatorname{argmin}_\theta \left( \hat{V}^\pi(s) - f_\theta(s) \right)^2$
- Instead,  $\operatorname{argmin}_\theta \left( (1 - \alpha)f_\theta(s) + \alpha(r + \gamma f_\theta(s')) - f_\theta(s) \right)^2$   
(using gradient descent)

# Function approximation

- How to avoid keeping track of each state?
- Major advantage to model-free RL methods:  
Can use **function approximation** to represent  $\hat{V}^\pi$  compactly
- Let  $\hat{V}^\pi(s) = f_\theta(s)$  be our approximator parameterized by  $\theta$

Can use similar approximators for the  $Q$  function

# Outline

1. Markov decision processes
2. Reinforcement learning
  - i. Model-free RL
  - ii. Exploration vs exploitation**
3. Branch-and-bound as an MDP

# Exploration/exploitation problem

All the methods discussed so far had some condition like:

- “assuming we visit each state enough”, or
- “taking actions according to some policy”

**Fundamental question:** should we

1. Take **exploratory** actions to get more information, or
2. **Exploit** current knowledge to perform as best we can?

# Exploration/exploitation

## **Epsilon-greedy policy:**

$$\pi(s) = \begin{cases} \max_a \hat{Q}^\pi(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{otherwise} \end{cases}$$

Want to decrease  $\epsilon$  as we see more examples, e.g.:

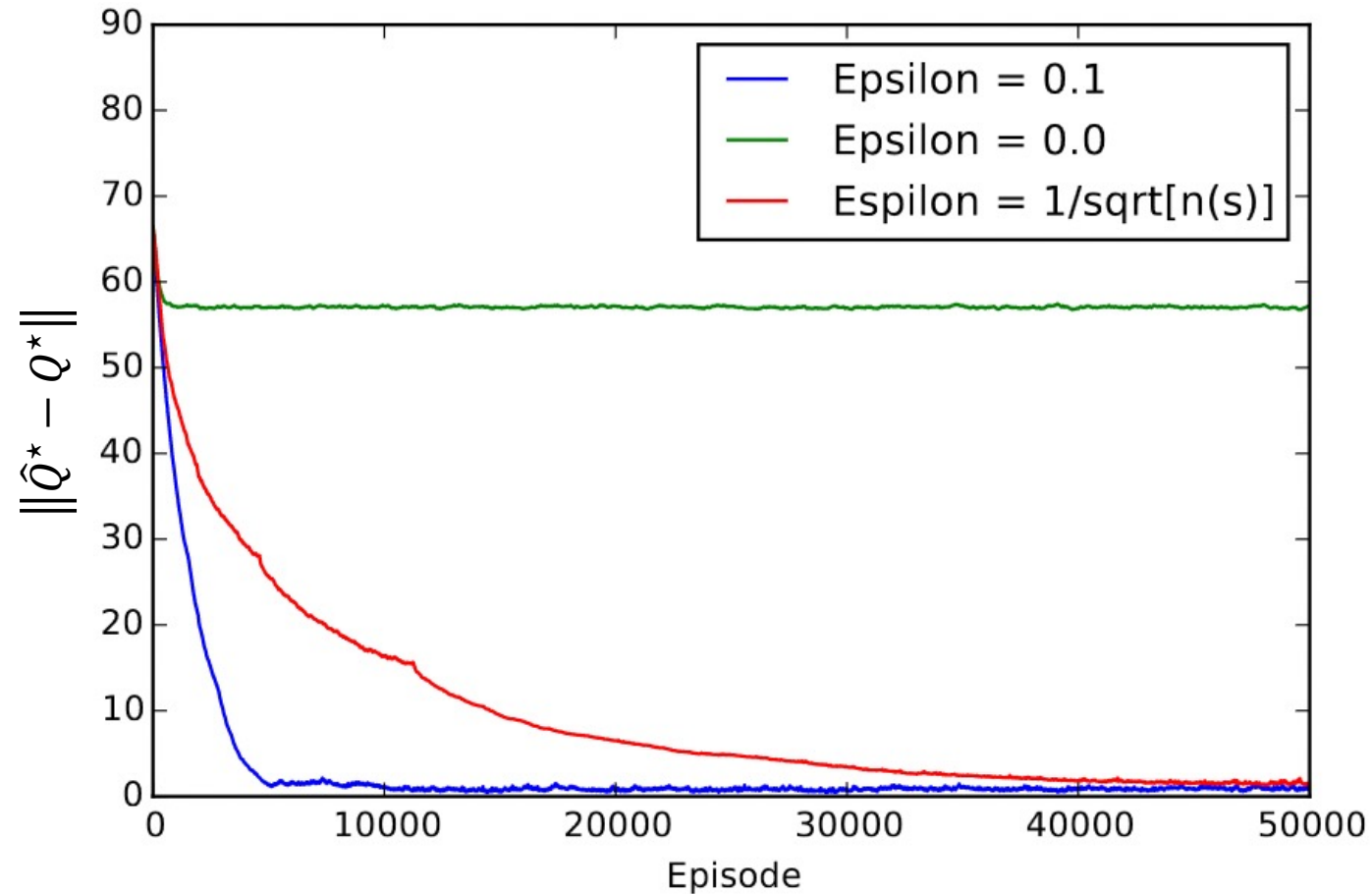
$\epsilon = \frac{1}{\sqrt{n(s)}}$  where  $n(s)$  is the number of times we've visited state  $s$

# Exploration experiments

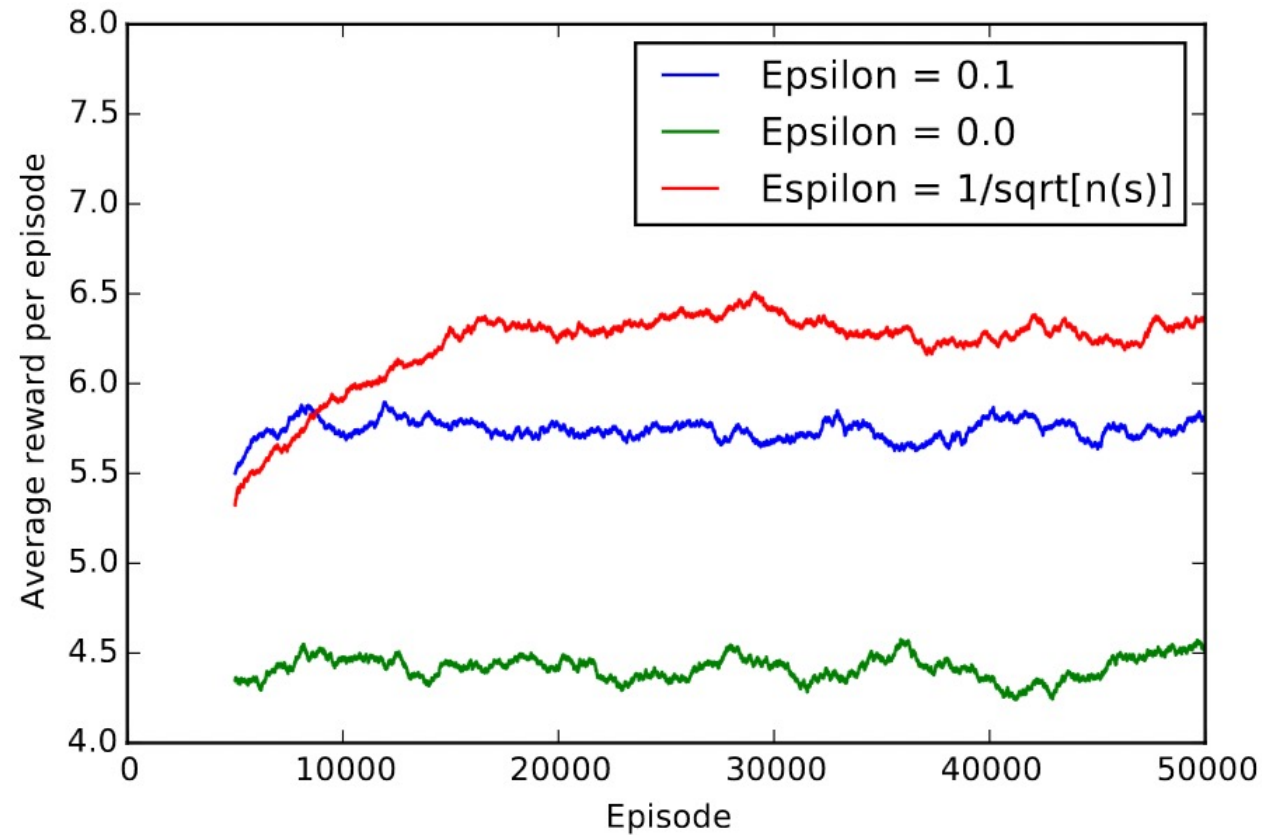
0	0	0	1
0		0	-100
0	0	0	0

- Gridworld but with  $U([0, 1])$  rewards instead of rewards above
- Initialize Q function with  $\hat{Q}(s, a) = 0$
- Run with  $\alpha = 0.05, \epsilon = 0.1, \epsilon = 0$  (greedy),  $\epsilon = \frac{1}{\sqrt{n(s)}}$

# Exploration experiments



# Exploration experiments



Average reward (sliding average over past 5000 episodes) for different strategies

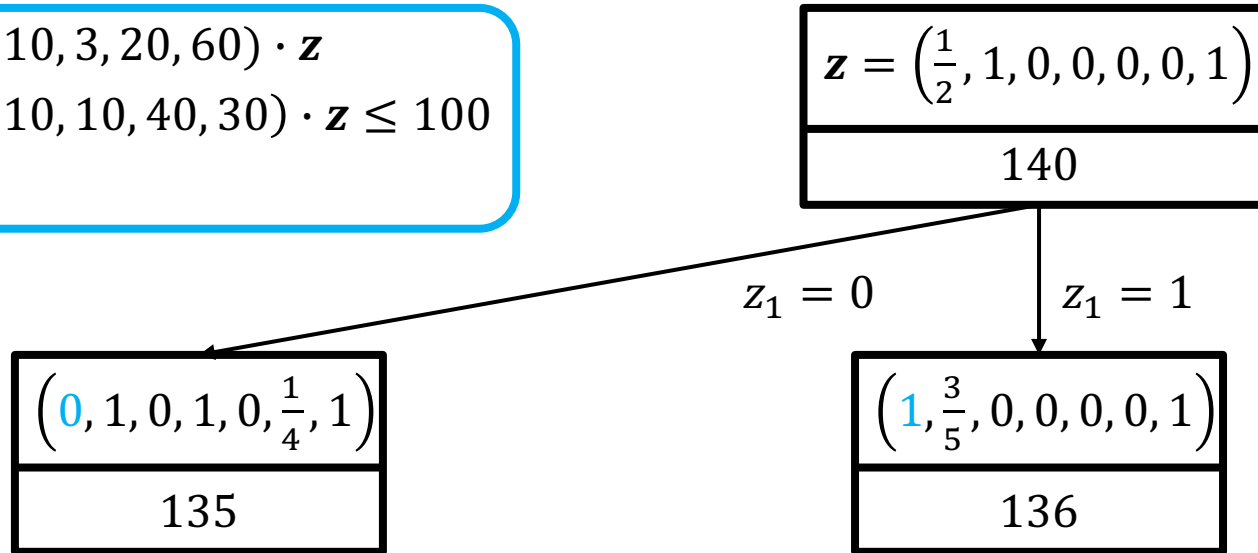


# Outline

1. Markov decision processes
2. Reinforcement learning
- 3. Branch-and-bound as an MDP**

# Branch and bound (B&B)

$$\begin{array}{ll}\max & (40, 60, 10, 10, 3, 20, 60) \cdot \mathbf{z} \\ \text{s.t.} & (40, 50, 30, 10, 10, 40, 30) \cdot \mathbf{z} \leq 100 \\ & \mathbf{z} \in \{0,1\}^7\end{array}$$



# Branch and bound (B&B)

$$\begin{array}{ll}\max & (40, 60, 10, 10, 3, 20, 60) \cdot \mathbf{z} \\ \text{s.t.} & (40, 50, 30, 10, 10, 40, 30) \cdot \mathbf{z} \leq 100 \\ & \mathbf{z} \in \{0, 1\}^7\end{array}$$

$$\mathbf{z} = \left( \frac{1}{2}, 1, 0, 0, 0, 0, 1 \right)$$

140

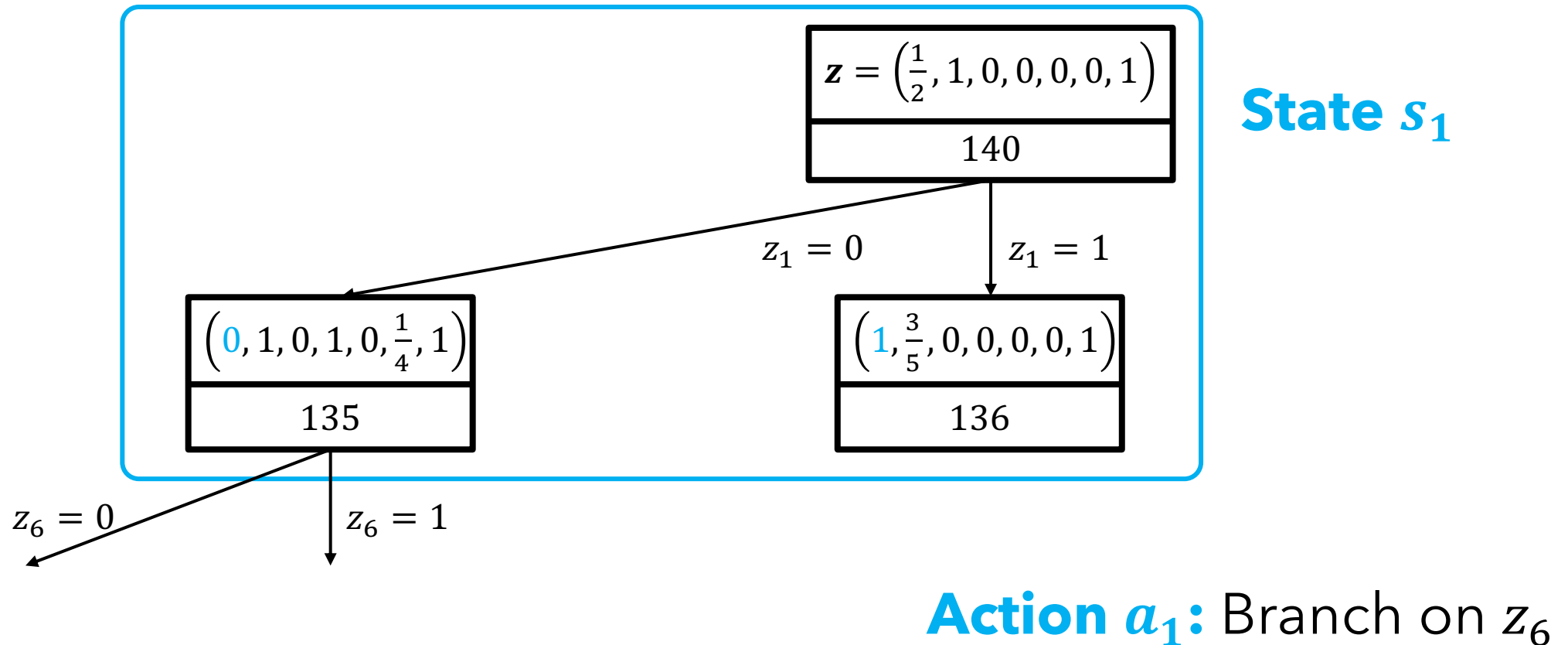
**State  $s_0$**

$z_1 = 0$

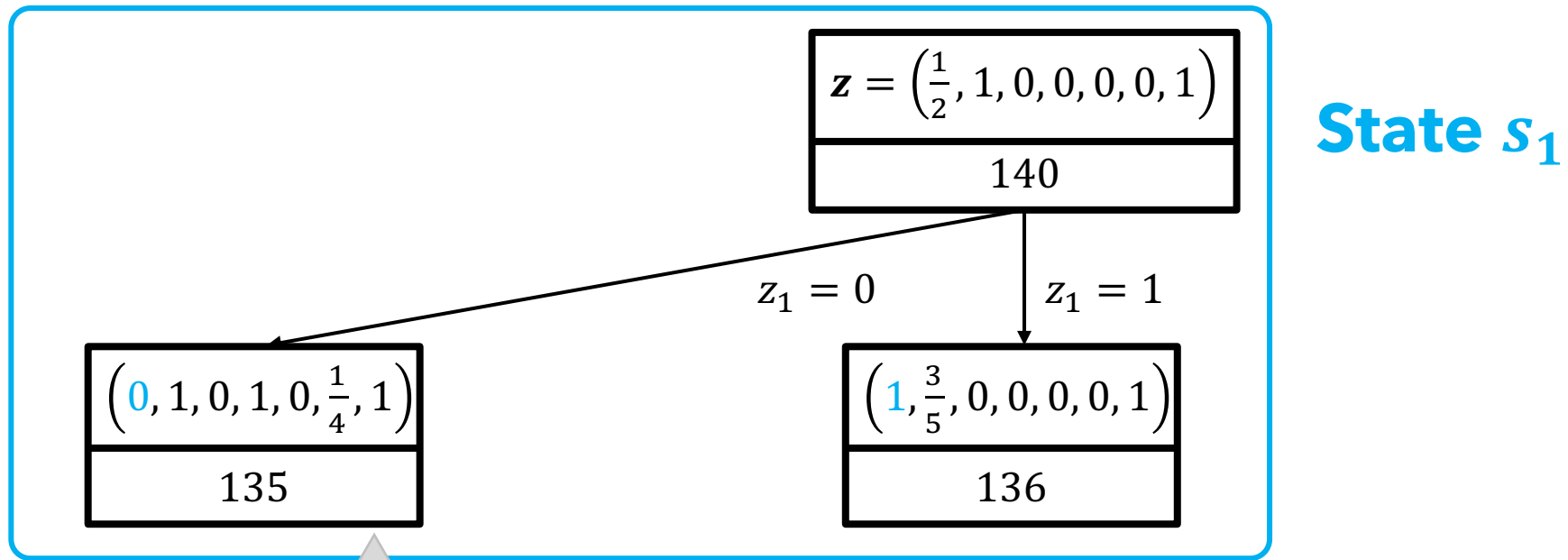
$z_1 = 1$

**Action  $a_0$ :** Branch on  $z_1$

# Branch and bound (B&B)



# Branch and bound (B&B)



**Action  $a_1$ :** Explore this node

# Papers we'll read

Gasse, Maxime, et al. "Exact combinatorial optimization with graph convolutional neural networks." *NeurIPS*. (2019).

- Frame B&B **variable selection** as an MDP
- Use **GNNs** to design variable selection policies

Dai, Hanjun, Khalil, Elias, et al. "Learning combinatorial optimization algorithms over graphs." *NeurIPS'17*.

- Develop **RL algorithms** for a variety of combinatorial problems
- Suggest RL could be used for **algorithm discovery**