Reinforcement Learning:
Q-learning and Tic-Tac-Go

BIOE 498/598 PJ

Spring 2021

Review

- Discount factors shorten the horizon of RL problems, causing the agent to focus on rewards in the near future.
- Temporal Difference (TD) learning incrementally updates value functions using a new experience.
- Learning Q-factors eliminates the need to predict the next state given an action; however, the number of Q-factors is much greater than the number of states.

Review

- Discount factors shorten the horizon of RL problems, causing the agent to focus on rewards in the near future.
- Temporal Difference (TD) learning incrementally updates value functions using a new experience.
- Learning Q-factors eliminates the need to predict the next state given an action; however, the number of Q-factors is much greater than the number of states.
- **Today:**
  - Review SARSA
  - Q-learning
  - Tic-Tac-Go

Learning Q-factors

Using Q-factors, the policy problem at state s_i

$$\max_a \mathbb{E}\left\{r_i + \gamma V(s_{i+1})\right\}$$

becomes

$$\max_a \mathbb{E}\left\{Q(s_i, a)\right\}.$$

Learning Q-factors

Using Q-factors, the policy problem at state s_i

$$\max_a \mathbb{E}\left\{r_i + \gamma V(s_{i+1})\right\}$$

becomes

$$\max_a \mathbb{E}\left\{Q(s_i, a)\right\}.$$

- **Pro:** We do not need a model or a way to predict s_{i+1}.
- **Con:** We need to learn a Q-factor for every state/action pair.

Learning Q-factors

Using Q-factors, the policy problem at state s_i

$$\max_a \mathbb{E}\left\{r_i + \gamma V(s_{i+1})\right\}$$

becomes

$$\max_a \mathbb{E}\left\{Q(s_i, a)\right\}.$$

- **Pro:** We do not need a model or a way to predict s_{i+1}.
- **Con:** We need to learn a Q-factor for every state/action pair.

We can learn Q-factors using a TD approach given a trajectory s_0, a_0, r_0, s_1, a_1, r_1 …, s_T, r_T:

$$\hat{Q}(s_i, a_i) = r_i + \gamma Q(s_{i+1}, a_{i+1}) \qquad \text{target}$$

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha\left[\hat{Q}(s_i, a_i) - Q(s_i, a_i)\right] \qquad \text{update}$$

This approach is called *SARSA*.

SARSA follows a trajectory, not an optimal path

The SARSA update equation is

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[\underbrace{r_i + \gamma Q(s_{i+1}, a_{i+1})}_{\text{target}} - Q(s_i, a_i) \Big].$$

SARSA follows a trajectory, not an optimal path

The SARSA update equation is

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[\underbrace{r_i + \gamma Q(s_{i+1}, a_{i+1})}_{\text{target}} - Q(s_i, a_i) \Big].$$

Our estimate of $Q(s_i, a_i)$ is based on

- The reward r_i experienced by selecting action a_i in state s_i.
- The future reward $Q(s_{i+1}, a_{i+1})$ based on the action a_{i+1} from the trajectory.

SARSA follows a trajectory, not an optimal path

The SARSA update equation is

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[\underbrace{r_i + \gamma Q(s_{i+1}, a_{i+1})}_{\text{target}} - Q(s_i, a_i) \Big].$$

Our estimate of $Q(s_i, a_i)$ is based on

- The reward r_i experienced by selecting action a_i in state s_i.
- The future reward $Q(s_{i+1}, a_{i+1})$ based on the action a_{i+1} from the trajectory.

The policy that generated the trajectory is not optimal, so it is likely that a_{i+1} was not the best action to take.

Selecting a suboptimal action underestimates the reward to go, and therefore the value $Q(s_i, a_i)$.

Q-learning

The Q-learning algorithm changes the SARSA update

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)\right]$$

to use the optimal action in state s_{i+1}:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma \max_a Q(s_{i+1}, a) - Q(s_i, a_i)\right].$$

Q-learning

The Q-learning algorithm changes the SARSA update

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)\right]$$

to use the optimal action in state s_{i+1}:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma \max_a Q(s_{i+1}, a) - Q(s_i, a_i)\right].$$

Q-learning can converge faster to an optimal policy. However, it has two drawbacks:

1. If the number of available actions is large, the maximization operator can be expensive to evaluate.
2. The maximization operator is biased.

Records were meant to be broken.

- Imagine that the quality of professional basketball players was fixed over time.
- In this case, scoring records would still be broken.
- Basketball includes stochastic elements, so as more games are played the chance of observing outliers increases.

Records were meant to be broken.

- Imagine that the quality of professional basketball players was fixed over time.
- In this case, scoring records would still be broken.
- Basketball includes stochastic elements, so as more games are played the chance of observing outliers increases.

Any algorithm with a $\max$ operator will drift upwards over time, *even if the mean value remains fixed*.

For Q-learning, we need to combat the bias in the $\max$ operator.

Double Q-learning

One solution to the $\max$ bias is using two separate Q functions (networks), called Q_1 and Q_2.

Both Q_1 and Q_2 are trained with separate experiences. (Or, one network can *lag* behind the other in experiences.)

Double Q-learning

One solution to the $\max$ bias is using two separate Q functions (networks), called Q_1 and Q_2.

Both Q_1 and Q_2 are trained with separate experiences. (Or, one network can *lag* behind the other in experiences.)

When updating, we use one network to select the action, and the other network to compute its value.

$$Q_1(s_i, a_i) \leftarrow Q_1(s_i, a_i) + \alpha \left[r_i + \gamma\, Q_2(s_{i+1}, a_1) - Q_1(s_i, a_i)\right]$$
$$a_1 \equiv \arg\max_a Q_1(s_{i+1}, a)$$

$$Q_2(s_i, a_i) \leftarrow Q_2(s_i, a_i) + \alpha \left[r_i + \gamma\, Q_1(s_{i+1}, a_2) - Q_2(s_i, a_i)\right]$$
$$a_2 \equiv \arg\max_a Q_2(s_{i+1}, a)$$

Double Q-learning

One solution to the $\max$ bias is using two separate Q functions (networks), called Q_1 and Q_2.

Both Q_1 and Q_2 are trained with separate experiences. (Or, one network can *lag* behind the other in experiences.)

When updating, we use one network to select the action, and the other network to compute its value.

$$Q_1(s_i, a_i) \leftarrow Q_1(s_i, a_i) + \alpha\, [r_i + \gamma\, Q_2(s_{i+1}, a_1) - Q_1(s_i, a_i)]$$
$$a_1 \equiv \arg\max_a Q_1(s_{i+1}, a)$$

$$Q_2(s_i, a_i) \leftarrow Q_2(s_i, a_i) + \alpha\, [r_i + \gamma\, Q_1(s_{i+1}, a_2) - Q_2(s_i, a_i)]$$
$$a_2 \equiv \arg\max_a Q_2(s_{i+1}, a)$$

Even if a_1 was selected because $Q_1(s_{i+1}, a_1)$ was aberrantly high, the value $Q_2(s_{i+1}, a_1)$ will not share this bias.

Deep Q-learning

- Currently, the most common method for approximating Q-factors is *deep learning* with artificial neural networks.
- We're going to learn to play a simple board game called Tic-Tac-Go.
- We want a game that is simple enough to be computationally tractable, but not easily solved.

Deep Q-learning

- Currently, the most common method for approximating Q-factors is *deep learning* with artificial neural networks.
- We're going to learn to play a simple board game called Tic-Tac-Go.
- We want a game that is simple enough to be computationally tractable, but not easily solved.
- Tic-Tac-Toe is simple, but solved. If both players follow an optimal strategy, the game will always end in a draw.

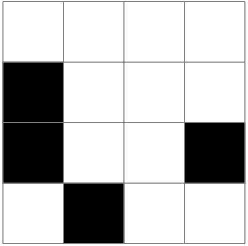
Deep Q-learning

- Currently, the most common method for approximating Q-factors is *deep learning* with artificial neural networks.
- We're going to learn to play a simple board game called Tic-Tac-Go.
- We want a game that is simple enough to be computationally tractable, but not easily solved.
- Tic-Tac-Toe is simple, but solved. If both players follow an optimal strategy, the game will always end in a draw.
- Go is unsolved, but approximating Q-factors is ridiculously expensive.

Tic-Tac-Go

- Tic-Tac-Go is played on a 4×4 grid.
- Before playing, 4 squares are randomly "blocked".
- Two players, `X` and `O`, alternate placing pieces in open squares.
- Players receive points for making horizontal or vertical "chains".
- A chain of length k is worth $(k-1)^2$ points.

$$\text{length } 2 = 1 \text{ point}$$
$$\text{length } 3 = 4 \text{ points}$$
$$\text{length } 4 = 9 \text{ points}$$



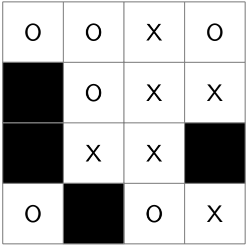
Tic-Tac-Go

- Tic-Tac-Go is played on a 4×4 grid.
- Before playing, 4 squares are randomly "blocked".
- Two players, `X` and `O`, alternate placing pieces in open squares.
- Players receive points for making horizontal or vertical "chains".
- A chain of length k is worth $(k-1)^2$ points.

  length $2 = 1$ point

  length $3 = 4$ points

  length $4 = 9$ points



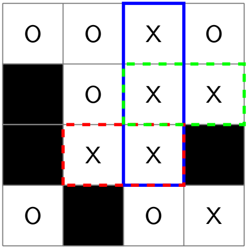
Tic-Tac-Go

- Tic-Tac-Go is played on a 4×4 grid.
- Before playing, 4 squares are randomly "blocked".
- Two players, `X` and `O`, alternate placing pieces in open squares.
- Players receive points for making horizontal or vertical "chains".
- A chain of length k is worth $(k-1)^2$ points.

length $2 = 1$ point

length $3 = 4$ points

length $4 = 9$ points



`X` score:
$$(2-1)^2+(2-1)^2+(3-1)^2 = 6$$

`O` score:
$$(2-1)^2 + (2-1)^2 = 2$$

States for Tic-Tac-Go

- States s_i are configurations of the board.
- Each of the 16 squares can be empty, blocked, `X`, or `O`.
- There are ${}_{16}C_4 \times 3^{12} = 967,222,620$ possible board configurations.
- Each configuration has, on average, 6 possible moves, so there are more than 5.8×10^9 Q-factors to learn!

States for Tic-Tac-Go

- States s_i are configurations of the board.
- Each of the 16 squares can be empty, blocked, `X`, or `O`.
- There are ${}_{16}C_4 \times 3^{12} = 967,222,620$ possible board configurations.
- Each configuration has, on average, 6 possible moves, so there are more than 5.8×10^9 Q-factors to learn!

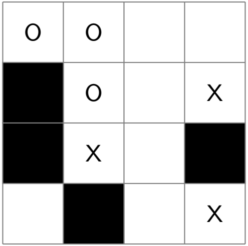
How do we encode the states for a function approximator?

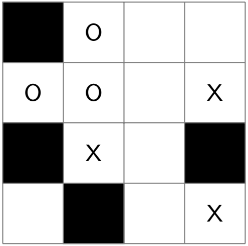
Option 1: Ignore the blocked states, -1, 0, $+1$

- Let's ignore the blocked squares since we can't play on them.
- A square with `X` is -1, `O` is $+1$, and empty squares are 0.
- Each state s_i is a 12×1 trinary vectory.

Option 1: Ignore the blocked states, -1, 0, $+1$

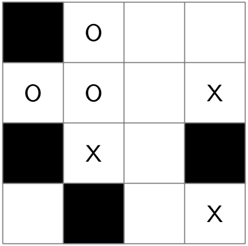
- Let's ignore the blocked squares since we can't play on them.
- A square with `X` is -1, `O` is $+1$, and empty squares are 0.
- Each state s_i is a 12×1 trinary vectory.





Option 2: One-hot encoding

state



blocked

1			
1			1
	1		

player `O`

	1		
1	1		

empty

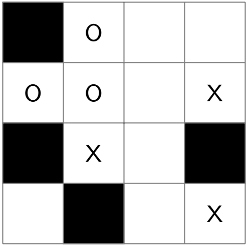
		1	1
		1	
		1	
1		1	

player `X`

			1
	1		
			1

Option 2: One-hot encoding

state



Each state is a
16×4 matrix
or a
64×1 vector.

blocked

1			
1			1
	1		

player `O`

	1		
1	1		

empty

		1	1
		1	
		1	
1		1	

player `X`

			1
	1		
			1

Our plan for Tic-Tac-Go

- We'll start with a one-hot encoding, as it simplifies the neural network.
- The four features are not independent, but we leave them to accelerate learning.
- However, *state unrolling* loses spatial information about the board. We will eventually avoid unrolling via *convolution*.
- Convolution will also exploit symmetry in the game board.

Summary

- Q-learning is a state-of-the-art technique for RL.
- Double Q-learning counteracts the bias in the $\max$ operator.
- Defining the state space for simple board games in not trivial. Some state space representations are better for learning.

Summary

- Q-learning is a state-of-the-art technique for RL.
- Double Q-learning counteracts the bias in the $\max$ operator.
- Defining the state space for simple board games in not trivial. Some state space representations are better for learning.
- **Next time:** Artificial neural networks.