Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

Part V

RL, AlphaGo and Beyond

455 Today - Lecture 21

- Introduction to Reinforcement Learning (RL)
- TD-Gammon, an early example of reinforcement learning with neural nets

Coursework

- Assignment 4 (due **Tue, Dec 14**)
- Reading and activities: Sutton RL tutorial + slides
- Quiz 11: Neural Networks and Deep Learning (double length)

Reinforcement Learning

- Reinforcement Learning (RL) introduction
- Credit assignment problem
- Learning from rewards and temporal differences
- TD-gammon as early example
- Training by RL
- Deep RL

Reinforcement Learning (RL)

- Activity - watch the tutorial and slides by Rich Sutton
- Brief review in class only
- Focus on what we need for AlphaGo
- Discuss Gerry Tesauro's TD-Gammon program
  - Early big success story for RL in heuristic search
  - Early example of neural nets in games

Basic Concepts of RL

- Observe input S_t (state of game at time t)
- Produce move, action A_t
- Observe reward (quality of action) R_{t+1}
  - Note that reward occurs at *next* timestep
  - Often, the reward is *delayed*
  - Games: reward only at end of game

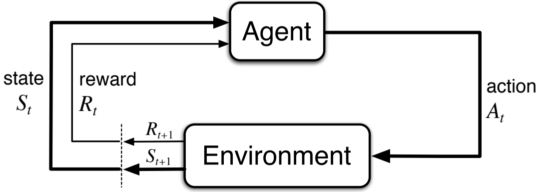


Figure 3.1: The agent–environment interaction in a Markov decision process.

Image source: [Sutton & Barto 2020]

Basic Concepts of RL

- Observe input S_t (state of game at time t)
- Produce move, action A_t
- Observe reward (quality of action) R_{t+1}
  - Note that reward occurs at *next* timestep
  - Often, the reward is *delayed*
  - Games: reward only at end of game
- Interaction produces a trajectory: $S_0, A_0, R_1, S_1, A_1, \ldots$

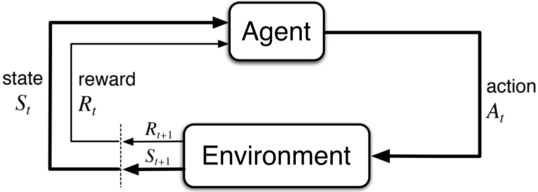


Figure 3.1: The agent–environment interaction in a Markov decision process.

Image source: [Sutton & Barto 2020]

RL vs Supervised Learning in Games

Supervised Learning

- Label for each move
    - Good/bad, expert move/not expert move
- Learn - minimize prediction error on given data set
- Can use mathematical optimization techniques, e.g. gradient descent

Reinforcement Learning

- Reward for whole game sequence only
- Learn - try to improve gameplay by trial and error
- Need to solve the credit assignment problem

Credit Assignment Problem

- Reward for (possibly long) sequence of decisions
- No direct reward for each single move decision
- How can we tell which moves are good or bad?
- Distribute reward from end of game over all actions
- Difficult problem
- RL provides the most popular answers
- Main idea: if same action happens in many different sequences, we can learn if it leads to more wins or losses

Value Functions

- Extremely widespread approach to solving the credit assignment problem: value-based reinforcement learning
- Estimate one or both of:
  - State-value function:

$$v_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \,\middle|\, S_0 = s, A_t \sim \pi(S_t)\right]$$

  - Action-value function:

$$q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \,\middle|\, S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

where $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ is a *stochastic policy*

Value Functions

- Extremely widespread approach to solving the credit assignment problem: value-based reinforcement learning
- Estimate one or both of:
  - State-value function:

$$v_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t \,\middle|\, S_0 = s, A_t \sim \pi(S_t)\right]$$

  - Action-value function:

$$q_\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t \,\middle|\, S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

where $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ is a *stochastic policy*

Monte Carlo Reinforcement Learning

$$q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \;\middle|\; S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

- In lecture 12 we already saw how to estimate an expectation using simulations
- Play out a bunch of games using policy π
- Find the average total return from every trajectory that starts from s, a

Monte Carlo Reinforcement Learning

$$q_\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \;\middle|\; S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

- In lecture 12 we already saw how to estimate an expectation using simulations
- Play out a bunch of games using policy π
- Find the average total return from every trajectory that starts from s, a
- In fact, we can do better!
- Find average total return from every **sub-trajectory** that starts from s, a

Policy Improvement Theorem

$$q_\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \,\middle|\, S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

- Problem: This procedure only estimates the value of a state-action pair assuming that all other moves are chosen according to a known policy π
- If we already knew the optimal π we would be done!

Policy Improvement Theorem

$$q_\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{T} R_t \;\middle|\; S_0 = s, A_0 = a, A_{t>0} \sim \pi(S_t)\right]$$

- Problem: This procedure only estimates the value of a state-action pair assuming that all other moves are chosen according to a known policy π
- If we already knew the optimal π we would be done!
- It turns out that greedily optimizing with respect to any policy π will produce a new policy that is guaranteed to be *weakly better at every state*.

Policy Improvement Theorem

Let π and π' be any pair of (deterministic**) policies.
If $q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \forall s \in \mathcal{S}$, then $v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$.

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation: Compute estimates $V(s)$ for state values

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation: Compute estimates $V(s)$ for state values
3. Policy Improvement: New policy chooses action that leads to highest value of V

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation: Compute estimates $V(s)$ for state values
3. Policy Improvement: New policy chooses action that leads to highest value of V
4. If policy is stable, stop; else goto 2 using new policy

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation: Compute estimates $V(s)$ for state values
3. Policy Improvement: New policy chooses action that leads to highest value of V
4. If policy is stable, stop; else goto 2 using new policy

The new policy is "stable" if it chooses the same actions as the old one at every state.

Policy Iteration

The Policy Improvement Theorem means that you can construct a new policy by solving credit assignment problem for an old policy:

1. Initialization: Set $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation: Compute estimates $V(s)$ for state values
3. Policy Improvement: New policy chooses action that leads to highest value of V
4. If policy is stable, stop; else goto 2 using new policy

The new policy is “stable” if it chooses the same actions as the old one at every state.

Question: What complication am I glossing over here?

Self-Play

- Standard reinforcement learning is a single-agent problem
- "Expected reward from following a policy" is ill-defined, because it depends on the other player's policy
- Solution: self-play
- Each policy is part of "the environment" for the other
- Train policies simultaneously

Monte Carlo Advantages and Disadvantages

Advantages:

- Conceptually straightforward
- Very parallelizable (**Why?**)

Monte Carlo Advantages and Disadvantages

Advantages:

- Conceptually straightforward
- Very parallelizable (**Why?**)
  - No dependence at all between state estimates

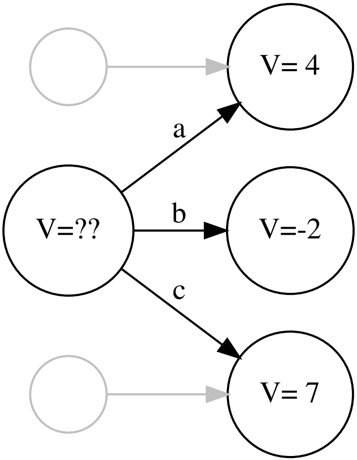
Monte Carlo Advantages and Disadvantages

Advantages:

- Conceptually straightforward
- Very parallelizable (**Why?**)
    - No dependence at all between state estimates

Disadvantages:

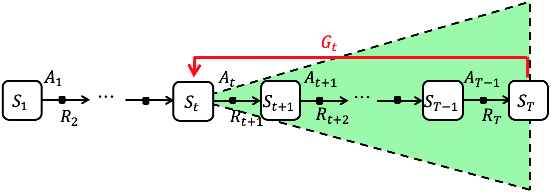
- Estimates of one state's value are not used to improve estimates of another
- Can only estimate the value of states and actions that are visited sufficiently often in some trajectory
- $\Longrightarrow$ Slow, data-inefficient



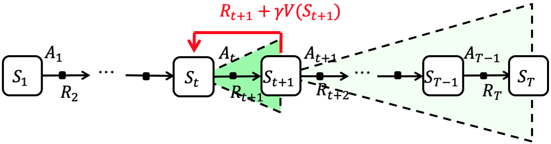
Temporal Difference (TD) Learning and $TD(\lambda)$

- Sutton (1988)
- Learn a model - a function from inputs to outputs
- Given only action sequences and rewards
- Learns a prediction (what is the best move?)
- Samples the environment (plays games)
- Compares learned estimate in each state with reward
- Learns from the difference
- Discount factor λ for future rewards
- The sooner after the current state the reward happens, the higher the effect

MC vs. TD



MC learning



TD learning

TD High-level Ideas

- Usually, predictions from states closer to the end are more reliable
- We can adjust earlier predictions, “trickle down”
- Bootstrapping - learn predictions from other predictions
- Whole process is grounded in the true final rewards
- This is one successful approach to solving the credit assignment problem in practice

Function Approximation

- Tabular learning: Value of each state / state-action is tracked separately
- Function approximation: Learn a model of values instead
  - Based on features of the state / state-action
  - Can use either Monte Carlo or TD updates

Function Approximation

- Tabular learning: Value of each state / state-action is tracked separately
- Function approximation: Learn a model of values instead
  - Based on features of the state / state-action
  - Can use either Monte Carlo or TD updates
- Advantage: Generalization. The model can guess values for similar states that it has never visited before.

Function Approximation

- Tabular learning: Value of each state / state-action is tracked separately
- Function approximation: Learn a model of values instead
  - Based on features of the state / state-action
  - Can use either Monte Carlo or TD updates
- Advantage: Generalization. The model can guess values for similar states that it has never visited before.
- Disadvantage: Over-generalization. Different states can be conflated if the features are insufficiently detailed.

Review - Backgammon

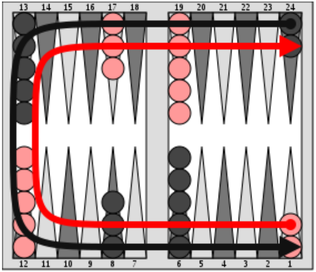


Image source: `https://en.wikipedia.org/wiki/Backgammon`

- Racing game played with dice
- Players race in opposite directions on the 24 *points*
- Single pieces can be captured and have to start from the beginning
- Doubling cube - play for double stakes, or resign
- Gammon and backgammon - win counts more if opponent is far behind

Tesauro's Neurogammon and TD-Gammon

- Neurogammon (Tesauro 1989)
- Plays backgammon using neural networks
- First program to reach "strong intermediate" human level, close to expert
- Beat all (non-learning) opponents at 1989 Computer Olympiad
- Beat many intermediate level humans,
  lost to an expert player

Neurogammon Architecture

- Six separate networks, for different phases of the game
- Fully connected feed-forward nets
- One hidden layer
- Trained with backprop
  - Supervised learning from 400 expert games
- One more network to make doubling cube decisions
  - Trained on 3000 positions, hand-labeled

Limitations of Neurogammon

- Hand-engineered features are difficult to create
- Human experts cannot explain much of what they are doing in a form that can be programmed
- Human expert games are difficult to collect, and are not perfect

TD-Gammon

- TD-Gammon (Tesauro 1992, 1994, 1995)
- Training by self-play
- Learns from the outcome of games
- Uses Temporal Difference (TD) Learning

TD-Gammon Architecture

- 198 inputs - 8 per point, 6 extra information (pieces off the board, toPlay)
- Single hidden layer, tried 10..80 hidden units
- Sigmoid activation function
- Output: one number, winning probability of input position
- Trained by $TD(\lambda)$ with $\lambda = 0.7$, learning rate $\alpha = 0.1$
- 200,000 training games, 2 weeks on high-end workstation
- Small (1-3-ply) Alphabeta search

TD-Gammon - Examples of Weights Learned

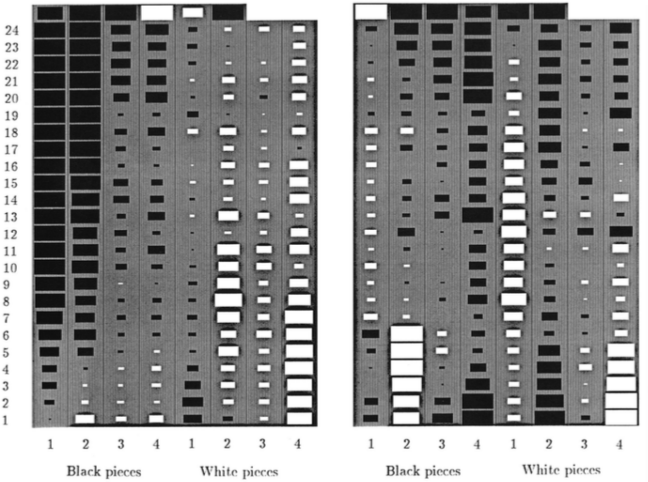


Image source: Tesauro, Practical Issues in Temporal Difference Learning, Machine Learning, 1992

TD-Gammon - Examples of Weights Learned

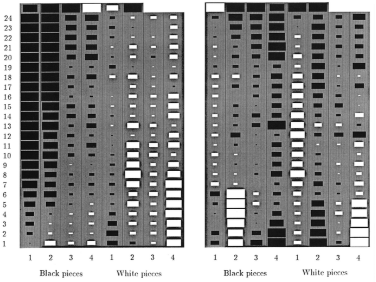


Image source: Tesauro, Practical Issues in Temporal Difference Learning, Machine Learning, 1992

- Weights from input to two of the 40 hidden units
- Both make sense to human expert players
- Left: corresponds to who is ahead in the race
- Right: probability that attack will be successful

TD-Gammon Impact

- Much stronger than Neurogammon
- Close to top human players
- Changed opening theory
- Changed the way the game is played by human experts
- For many years, the most impressive application of RL

Computer Backgammon Now

- Programs generally follow the TD-Gammon architecture
- Bigger, faster, longer training
- Endgame databases with exact winning probabilities
- Considered almost perfect
- Much stronger than humans

Summary of RL Introduction

- Reinforcement learning for learning from self-play
- TD-Gammon as early success story
- Very small (for todays standard) net with 1 hidden layer
- World class performance
- Trained by RL, more specifically the $TD(\lambda)$ algorithm