ATCI: Reinforcement Learning

Reinforcement Learning in zero-sum games

Mario Martin

CS-UPC

April 25, 2024

Agents in Zero sum games

Agents in Zero sum games

- We have seen agents in cooperative scenarios. Can we move to other kind of problems?
- Specific case of MARL competition: zero sum games
- Techniques we'll see:
  - Self-play
  - Monte-Carlo Tree Search and AlphaZero family of methods

Self-Play

Self-Play idea

- In competitive zero-sum games for 2 players (like go and chess) we can apply self-play
- Introduced years ago to play Backgamon (Tesauro, 92) and very appealing
- Consists in the agent playing against himself to increasingly learn good policies

Self-Play algorithm

- Consists in the following:
  1. Start with a random policy agent a
  2. Create a' as a copy of the agent a'
  3. Do:
     1. Play agent a against a'.
     2. Agent a learn from experiences of the game. a' is frozen
  4. Repeat until agent a wins consistently a'
  5. Copy a in a'
  6. Repeat 3-5 until desired performance

Self-Play: some notes

- Two copies of the same agent. Goal are not 2 competing agents, but one that plays well.
- Player a' is frozen to improve stability
- To improve stability in learning, play not against only last agent but collect agents in several iteration and play against them

Self-Play: application

- No previous knowledge needed (not human games, not even the rules of the game)
- Learning creates some kind of curriculum with increasing abilities
- You are always playing (and learning) at a competitive level
- Usually surpass Human game level play (no examples of expert playing)
- Can Learn surprising / unexpected (not human) strategies
- Applied successfully to Checkers, Backgammon, Chess, Go but also other games like Hide and Seek and soccer soccer.
- It can be applied to any kind of base RL algorithm but it works better with MCTS-based methods

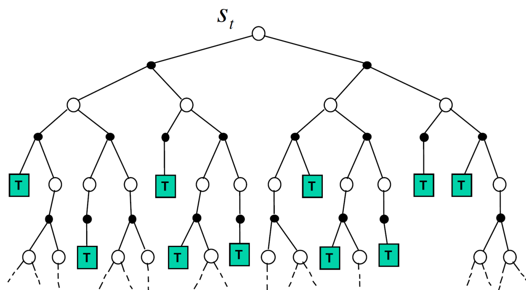
Monte Carlo tree-search

Agents in Zero sum games

- Can we apply RL in hard games?
- You can imagine than in interesting games the number of possible states in too big to apply Value based techniques (Go 10^{170}, Chess 10^{48})
- Hard to compute a good value function for each one of them... moreover you will never visit a lot of them in a game.
- We will focus on the sub-MDP that depends on the current state instead of solving all the game

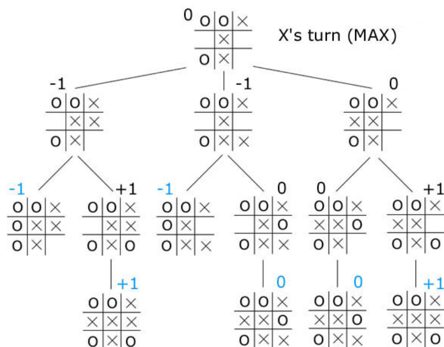
Planning: tree search

- Zero-sum games have been addressed from a long time ago: Checkers, Chess, Go...
- Usually there have been solved using **planning techniques** that generate search **trees**
- We will combine the learning of the policy with tree-search methods.



Monte-Carlo tree search

- In the case of Zero-sum games, the popular approach is `Mini-Max` and $\alpha - \beta$ algorithms

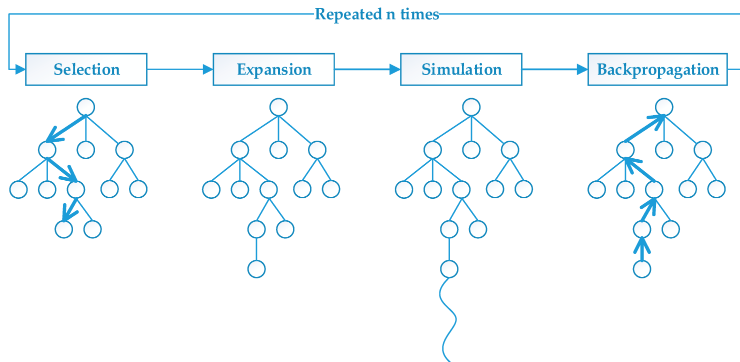


Monte-Carlo tree search

- First approach to play Go at a decent level
- We cannot generate the whole tree to apply minimax.
- Reduction of search of the tree at two levels:
  - Depth: We will stop growing of the tree at some point. We will use a criteria to evaluate the value of the leafs of the tree-
  - Breath: We will not explore all action with same probability. We will give more chances to promising actions

Monte-Carlo tree search

- Four steps:



Monte-Carlo tree search: Selection

SELECTION:

- Let's assume that for each state we have a Q-value estimation $Q(s, a)$ obtained from experiences (it will be built during tree search).
- Choose action according to:

$$a_t^{UCB1} = \arg\max_{a \in \mathcal{A}} \left(Q(s, a) + C \sqrt{\frac{2 \log t(s)}{N_t(s, a)}} \right)$$

  where N_t is the number of times a has been tested in s, and $t(s)$ is the number of times state s has been visited. C is a hyper-parameter.
- We select action until we arrive to a node without Q-value estimation for at least one action[1]. This is the node **selected**.

[1] Or a terminal state!

Monte-Carlo tree search: Expansion and Simulation

Expansion:

- We **expand** the node selected in expansion step generating a new node from one action that has never tried before (without Q-values!)

Monte-Carlo tree search: Expansion and Simulation

EXPANSION:

- We **expand** the node selected in expansion step generating a new node from one action that has never tried before (without Q-values!)

SIMULATION:

- From the node created (expanded) in the previous step, we **simulate** a trajectory following a given policy (*Rollout Policy*).
- In MCTS, usually the uniform random policy is used (cheap, fast, no a priori knowledge required), that is, choose and apply random valid actions **until we arrive to a terminal state** so we have a final evaluation.
- We take note of the reward z in the terminal state.

Monte-Carlo tree search: reward Backpropagation

BACKPROPAGATION:

- We **Backpropagate** the reward z obtained in the terminal state
- We update the Q-value for all antecessor pairs state-action (from the expanded state to the root of the tree).
- Q-values are updated as follows:

$$
\begin{aligned}
t(s_t) &\leftarrow t(s_t) + 1 \\
N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\
Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \frac{z - Q(s_t, a_t)}{N(s_t, a_t)}
\end{aligned}
$$

Example

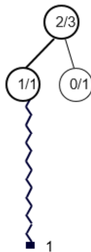
- 1 iteration

Example

- 2 iterations

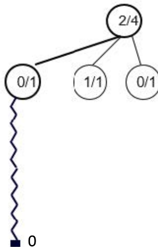
Example

- 3 iterations



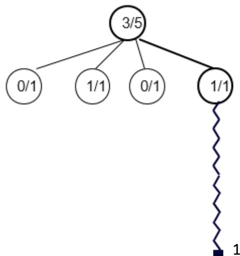
Example

- 4 iterations



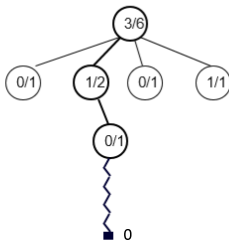
Example

- 5 iterations



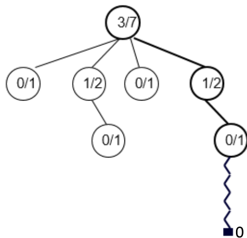
Example

- 6 iterations



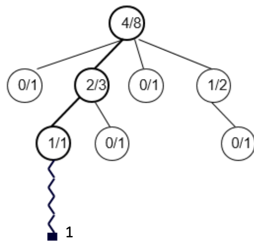
Example

- 7 iterations



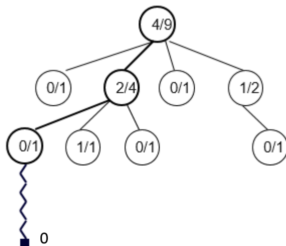
Example

- 8 iterations



Example

- 9 iterations



Monte-Carlo tree search: Overview

- This cycle is repeated a lot of times (as computational resources allow), so the tree grows with each iteration.
- When the limit of iterations allowed is reached, an action is chosen for the root state according to the **greedy criteria** or, in some implementations, **the action with more visits**).
- The resulting state from the action execution becomes the root of the new tree (may reuse statistics of subtree)
- From one game to another, the tree is started from scratch (usually).
- See a nice and simple implementation in python to play TIC-TAC-TOE

Conclusion MCTS

- It is a planing method based on some techniques of RL (see exploration slides)
- How is this related to Learning in competitive MultiAgent systems?

Conclusion MCTS

- It is a planing method based on some techniques of RL (see exploration slides)
- How is this related to Learning in competitive MultiAgent systems?
- If we estimate Q-values, we can learn them!

AlphaGo family

AlphaGo family

- An evolution of methods proposed by DeepMind:
  - **AlphaGo** (Silver et al. 16) where the authors describe a MCTS method with RL and self-play that learns to play Go beating the Human World Master of the game
  - **AlphaZero** (Silver et al. 17), an evolution where agent learns purely using RL without any previous knowledge of the game
  - **Muzero** (Schrittwieser et al. 19) that learns to play *without a model of the game* (model-free RL). It can be extended to any kind of problem in RL (we will see it in Model-based methods)

AlphaZero

- Simpler than AlphaGo and applicable to other games
- In AlphaZero there is only a Neural Network f_θ that outputs both, the value of a state $v_\theta(s)$ and the distribution of probabilities for each action of a stochastic policy $P_\theta(a|s)$
- It applies self-play schema together with a variation of MCTS with learning of f_θ

AlphaZero details

- It uses MCTS where:
  - Selection step done according:

$$a_t^{UCB1} = \arg\max_{a_i \in \mathcal{A}} Q\left(s_t, a_i\right) \; + \; c \; P_\theta\left(a_i | s_t\right) \frac{\sqrt{t(s_t)}}{1 + N\left(s_t, a_i\right)}$$

AlphaZero details

- It uses MCTS where:
  - Selection step done according:

$$a_t^{UCB1} = \arg\max_{a_i \in \mathcal{A}} Q(s_t, a_i) \; + \; c \; P_\theta(a_i | s_t) \frac{\sqrt{t(s_t)}}{1 + N(s_t, a_i)}$$

  - Compared with MCTS there is no simulation! **The prediction of the value of the expanded node is used to backpropagate results**

AlphaZero details

- It uses MCTS where:
  - Selection step done according:

$$a_t^{UCB1} = \arg\max_{a_i \in \mathcal{A}} Q(s_t, a_i) \; + \; c \; P_\theta(a_i|s_t) \frac{\sqrt{t(s_t)}}{1 + N(s_t, a_i)}$$

  - Compared with MCTS there is no simulation! **The prediction of the value of the expanded node is used to backpropagate results**
  - Action executed while self-play is according to sampling distribution:

$$\pi(s, a) = \frac{N(s_t, a_i)}{t(s_t)}$$

AlphaZero details

- Learning of f_θ is done with cases collected from the play of the kind

$$(s_t, \pi(s_t), z_t)$$

  where for each state in the trajectory s_t we store the policy distribution $\pi(s_t)$ and z is the final outcome of the trajectory (win or lose)

- Loss for f_θ is simply:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \pi(s_t) \cdot \log\left(\vec{P}_\theta(s_t)\right)$$

- Loss minimize at the same time the prediction on final game and mismatch between policy used and the predicted by the network

AlphaZero discussion

- Some numbers for Go from a nice cheatsheet (not mine) of the paper:
  - Self-play of about 4.9 million games
  - At each iteration of SelfPlay the agent *a* plays 25.000 games against itself
  - We continue untill in the last 400 games agent *a* wins 55% of games
  - Number of iterations for growing a MTCS: 1600 simulations
  - Training of the neural network is done with batchsize 2048 from buffer containing 500.000 last games
  - In the case of go, input is 17 boards (19x19) stacked representing current and the last 7 boards per player (x2) plus a board to represent the turn
  - Neural Network if composed of 40 residual convolutional layers

AlphaZero discussion

- General algorithm for zero sum games
- Very effective and state of the art is most zero-sum games (even in chess![2]).
- No examples of playing required. Learns from scratch.
- Still needs to know the rules of the game (model of the world). Muzero solves this problem.
- When playing in production still generate a tree

[2] See LeelaZero

AlphaZero discussion

- General algorithm for zero sum games
- Very effective and state of the art is most zero-sum games (even in chess![2]).
- No examples of playing required. Learns from scratch.
- Still needs to know the rules of the game (model of the world). Muzero solves this problem.
- When playing in production still generate a tree Why? Could not we use the policy learn?

[2] See LeelaZero

AlphaZero discussion

- General algorithm for zero sum games
- Very effective and state of the art is most zero-sum games (even in chess![2]).
- No examples of playing required. Learns from scratch.
- Still needs to know the rules of the game (model of the world). Muzero solves this problem.
- When playing in production still generate a tree Why? Could not we use the policy learn? In practice better performance.

[2] See LeelaZero

AlphaZero discussion

- General algorithm for zero sum games
- Very effective and state of the art is most zero-sum games (even in chess![2]).
- No examples of playing required. Learns from scratch.
- Still needs to know the rules of the game (model of the world). Muzero solves this problem.
- When playing in production still generate a tree Why? Could not we use the policy learn? In practice better performance.
- On the dark side: Time to learn by self-play is high. Not easy to find NN architectures for each game. Large amount of resources to play (still uses MCTS)

[2] See LeelaZero