A short introduction to (model-based) reinforcement learning.

Wenda Zhou

April 7, 2021

Markov Decision Process (MDP)

A Markov Decision Process is a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, R)$ where:

- $\mathcal{S}$ denotes the state space
- $\mathcal{A}$ denotes the action space
- $\mathcal{T}(s, a) \to \mathcal{M}_1(S)$ denotes the transition function, a probability measure giving the probability of transitioning to a state s' from state s using action a.
- $R(s, a, s')$ denotes the reward from going from state s to s' using action a.

Example: Chess

- $\mathcal{S}$: position of the pieces (+ castling rights and en passant)
- $\mathcal{A}$: all legal piece moves
- $\mathcal{T}(s, a)$: deterministic transition of moving a piece
- $R(s, a)$: 0 for all states, except $+1$ for winning, -1 for losing.

Policies and Problem Formulation

We consider an agent which acts in the environment according to a policy $\pi : \mathcal{S} \to \mathcal{M}_1(\mathcal{A})$. We can then consider a trajectory $(s_t, a_t, r_t)_t$ of the agent as follows:

- $s_0 \sim p_0(\mathcal{S})$,
- $a_t \sim \pi(s_t)$,
- $s_{t+1} \sim \mathcal{T}(s_t, a_t)$,
- $r_t = R(s_t, a_t, s_{t+1})$.

Given a discount factor $0 < \gamma \leq 1$, we wish to optimize the (expected) cumulative rewards:

$$J = \mathbb{E}_{\pi,\mathcal{T}}\left[\sum_{k=0}^{\infty} \gamma^k r_k\right]$$

Q-function and Bellman's equation

To look at the cumulative reward in a more analytical fashion, we define the action-value (or quality) function $Q^{\pi}(s, a)$:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi,\mathcal{T}}\left[\sum_{k=0}^{\infty} \gamma^k r_k \mid s_0 = s, a_0 = a\right].$$

The recursive definition of the Q-function is known as Bellman's equation:

$$Q^{\pi}(s, a) = \mathbb{E}_{s' \sim \mathcal{T}(s,a)}\Big[R(s, a, s') + \gamma \mathbb{E}_{a' \sim \pi(s')}\big[Q^{\pi}(s', a')\big]\Big].$$

A related quantity is also often used, the value function:

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(s)} Q^{\pi}(s, a).$$

Applications of Reinforcement Learning

- Robotics and control problems
- Combinatorial optimization problems: many combinatorial optimization problems can be expressed by building or modifying the solution in an iterative fashion [2].
  - E.g. Optimization of molecular properties by modifying molecules iteratively.
  - My current project: generating optimized (computer) programs for computing simple programs (arithmetic circuits).
- Heuristics and meta-heuristics: reinforcement learning can be seen as a learning framework to implement learning for heuristics, e.g. tuning of search parameters, hyper-parameters etc.

Sythesizing optimized programs from arithmetic circuits.

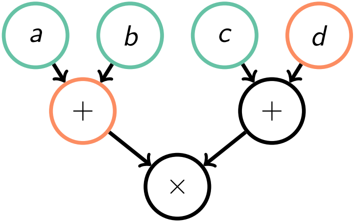
Act in the environment by emitting instructions:

```
load a, %1
load b, %2
load c, %3
```

Valid actions are given by boundary of arithmetic circuit, e.g. here, can consider:

```
load d, %4
add %1, %2, %4
```

Reward given by (negative) time taken to execute the program.



Model-based vs. Model-free reinforcement learning

- Applications of reinforcement learning differ substantially depending on their access (and knowledge) of $\mathcal{T}$ and R.
  - Chess: $\mathcal{T}$ is fully known, cheap to compute.
  - Recommendation system (e.g. Amazon/Netflix): $\mathcal{T}$ and R highly complex (human behavior), unknown.
  - Optimizing molecules: R may be require significant computational expense to access, or only approximate access available.

Model-free learning
: Learning only has forward access to the model (must execute action to observe reward).

Model-based learning
: Learning has reversible / counter-factual access to the model (*what if I execute this action?*).

Note: possibility (and large amount of research) on how to use model-based techniques in model-free contexts: we can learn a model!

Model-based reinforcement learning

I will try to explore three related facets of model-based reinforcement learning techniques (but also see surveys [3, 4]):

1. Dynamics Model Learning: Dyna (Dyna-Q) [7]
2. Learning and planning with a known model: AlphaZero [6].
3. Implicit model-based reinforcement learning: MuZero [5].

Q-learning

Q-learning is a *model-free* approach which attempts to directly estimate the Q-function by fixed-point iteration.

Q-learning

1. Initialize $Q(s, a)$
2. Act in environment to obtain tuples (r, s, a, s'), where $s' \sim \mathcal{T}(s, a)$, and $r = R(s, a, s')$.
3. Update Q-function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \Big).$$

In classical setup, Q is encoded as a table. Recent resurgence of *deep q-learning*, where Q is encoded as a neural network.

Dyna-Q

Augment Q-learning with a learned model $M(s, a)$.

Dyna-Q

1. Initialize $Q(s, a)$
2. Act in environment to obtain tuples (r, s, a, s'), where $s' \sim \mathcal{T}(s, a)$, and $r = R(s, a, s')$.
3. Update Q-function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big).$$

4. Update Model:

$$M(s, a) \leftarrow (s', r)$$

5. Repeat n times: generate tuples (r, s, a, s') from model, and update Q according to (3).

Learned dynamics model and science

- In many problems at FI, dynamics are (at least approximately) known, but may be expensive to compute.
- Learned surrogates can help speed-up inner loop.
- Multi-fidelity computation: optimize computational expense by choosing accuracy of oracle to query.
- Bayesian Optimization: jointly learn model and optimize to increase sample effectiveness.

Learning and planning with a known model

In some cases (e.g. board games), the model (i.e. $\mathcal{T}$ and R) is fully known. We could thus (in principle) compute Q through its definition.

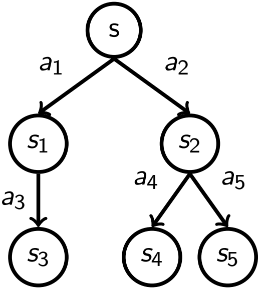
$$Q^\pi(s,a) = \mathbb{E}_{s' \sim \mathcal{T}(s,a)}\Big[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi(s')}\big[Q^\pi(s',a')\big]\Big].$$

Problem: this is typically not a computationally tractable quantity.

A detour into high-performance planning

The previously posed problem, with known $\mathcal{T}$ and R, can be seen as a purely computational problem. There has been significant work in obtaining tractable approximations to the computation through tree search methods.

$$Q^\pi(s,a) = \mathbb{E}_{s' \sim \mathcal{T}(s,a)}\Big[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi(s')}\big[Q^\pi(s',a')\big]\Big].$$



Monte-Carlo Tree Search

- Trees are exponentially large, computationally intractable to search them completely.
- How to decide which nodes to explore? (Breadth vs. depth, exploration / exploitation trade-offs).

Monte-Carlo Tree Search

Bayesian formalism to decide on exploration-exploitation trade-off.

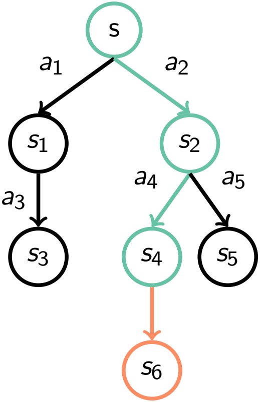
Exploration Search actions that have not been evaluated much,

Exploitation Search actions that are the most promising so far.

Monte-Carlo Tree Search

MCTS is usually formulated with four steps:

1. Select. Find out which node to explore next.
2. Expand. Execute an action from that node.
3. Simulate. Compute a value estimate for the new node (e.g. by playing the game until the end).
4. Backpropagate. Update value estimates of the tree.



Monte-Carlo Tree Search: UCT selection rule

Selection is the crucial step in the algorithm. It controls the trade-off between exploration and exploitation.

Upper-Confidence Bound for Trees (UCT) [1]

At a given state s, select the action a which maximizes:

$$\hat{Q}(s,a) + c\sqrt{\frac{\log N}{n_a}},$$

where we have:

- $\hat{Q}(s,a)$, the current estimate of the value of that action (average value of all simulations).
- n_a: number of simulations containing the action a.
- N: number of simulations containing the parent node.
- $c > 0$: coefficient adjusting exploration / exploitation trade-off.

See also: *optimism under uncertainty* (multi-armed bandits).

Learning and planning: AlphaZero

Leverage learning to more efficiently explore the tree.

AlphaZero selection rule

At a given state s, select the action a which maximizes:

$$\hat{Q}(s,a) + cP(s,a)\frac{\sqrt{N}}{1+n_a}.$$

where here, $P(s,a)$ is a prior policy.

Idea: learn prior policy P through policy refinement.

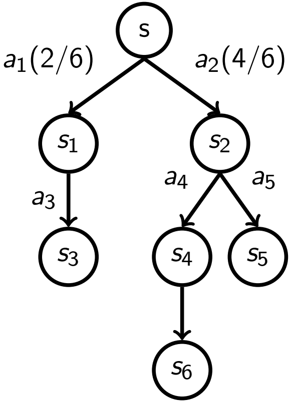
Learning and planning: AlphaZero

MCTS as policy refinement

Given a prior policy with weights θ $P_\theta(s, a)$, we wish to obtain a better policy.

- Given a state s
- Run MCTS from s for K iterations
- Obtain new policy as normalized counts $\hat{P}(s, a) = n_a/K$.

Update θ to better approximate $\hat{P}(s, a)$, repeat.



How effective is learning to plan?

- In general, evaluating the prior policy P can be expensive (it's a neural network).
  - LeelaChess (MCTS + NN) evaluates 30k positions per second.
  - Stockfish (Alpha-Beta search) evaluates 200M positions per second.
  - Large continuum between the two (e.g. recent innovations in Stockfish NNUE for efficiently updateable networks to evaluate a neural network at 100M positions per second).
- Compared to heuristics / meta-heuristics in combinatorial optimization, often faster but similar quality solutions.
- Cost of policy P is only part of the picture, dynamics $\mathcal{T}$ and R may also be potentially expensive to evaluate.

Implicit / Latent Models

- Can we combine learning of both dynamics and planning?
- Wish to tackle problems which require both high performance (learning to plan) with complex environments (learn dynamics).
- Wish to bypass necessity to encode state s (e.g. complex / not completely observed state).

Value equivalent models / implicit dynamics

Observation: we only require the value of each state, not the full state.

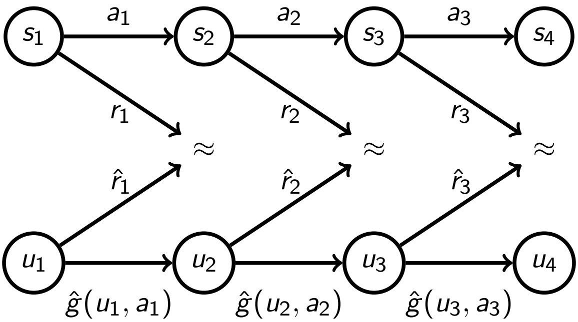
- Learn a reduced model and implicit dynamics to predict value (and policy).

MuZero

Learning the reduced model

Suppose that we observe transitions (s_t, a_t, r_t). We wish to obtain latents $u_t \in \mathbb{R}^d$, by learning a function $g(u, a)$ and a function $\hat{r}(u, a)$ such that:

$$u_{t+1} = \hat{g}(u_t, a_t) \text{ and } \hat{r}(u_t, a_t) \approx r_t.$$



MuZero

- Use learned dynamics $g(u, a)$ to plan action using MCTS.
- Learn value and policy functions from latent u.
- Achieves state of the art performance in board games (e.g. Chess / Go) without “knowing” the rules.

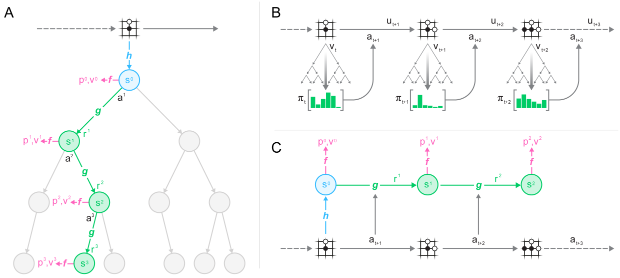


Figure: Figure taken from MuZero paper [5]

References I

[1] Hyeong Soo Chang et al. "An Adaptive Sampling Algorithm for Solving Markov Decision Processes". In: *Oper. Res.* 53.1 (2005), pp. 126–139.

[2] Nina Mazyavkina et al. "Reinforcement Learning for Combinatorial Optimization: a Survey". In: *arXiv e-print* (2020).

[3] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. "Model-based Reinforcement Learning: A Survey". In: *arXiv e-print* (2020).

[4] Aske Plaat, Walter A. Kosters, and Mike Preuss. "Model-Based Deep Reinforcement Learning for High-Dimensional Problems, a Survey". In: *arXiv e-print* (2020).

References II

[5] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609. ISSN: 0028-0836. DOI: 10.1038/s41586-020-03051-4.

[6] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nat.* 550.7676 (2017), pp. 354–359. URL: https://doi.org/10.1038/nature24270.

[7] Richard S. Sutton. "Dyna, an Integrated Architecture for Learning, Planning, and Reacting". In: *SIGART Bull.* 2.4 (1991), pp. 160–163. DOI: 10.1145/122344.122377.