AML in depth study: Reinforcement Learning

Gianluca Giudice

University of Milano-Bicocca

g.giudice2@campus.unimib.it

January 23, 2022

Overview

Characteristics of Reinforcement Learning

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor, only a reward signal
  - Interactions between agent and environment
- Feedback is delayed
- Time really matters (sequential, non i.i.d data)
- Agent's actions affect the subsequent data it receives
  - Greedy good actions could be penalizing in the future

Key concepts in RL

Agent and Environment

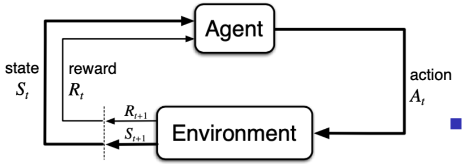

Figure: Typical RL scenario. [11]

- At each step t the agent:
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}

States and observations

- A **state** s is a complete description of the state of the world
- An **observation** o is a partial description of a state
- Markov decision processes are a mathematical way to describe RL problems

Definition

A **MDP** is a tuple (S, A, P, R, γ), where:

- S is a (finite) set of Markov states $s \in S$
- A is a (finite) set of actions $a \in A$
- P is dynamics/transition model for each action, that specifies

$$P(s_{t+1} = s' | s_t = s, a_t = a) \tag{1}$$

- R is a reward function

$$R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a] \tag{2}$$

- γ is a discount factor $\gamma \in [0, 1]$

Action spaces

- Different environments allow different kinds of actions
- The set of all valid actions in a given environment is called the **action space**
- Depending on the environment, the action spaces could be **discrete** or **continuous**

Policies

- A **policy** defines the learning agent's way of behaving, it is a mapping from perceived states of the environment (i.e. observations) to actions to be taken when in those states
- A policy fully defines the behavior of an agent

Definition

A **policy** π is a probability distribution over actions given states

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{3}$$

The MDP and agent's policy together give rise to a **trajectory** τ, which is a sequence of states and actions in the world:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1 ...)$$

Reward and return

- The reward function R depends on the current state of the world and action just taken:
$$r_t = R(s_t, a_t)$$
- The agent aim to maximize the cumulative reward over a trajectory
- The sum of all rewards obtained by the agent is discounted by how far off in the future they are obtained
- The discount factor is $\gamma \in [0, 1]$
- The cumulative reward over a trajectory is:

$$R(\tau) = \sum_{t=0}^{H-1} \gamma^t r_t \tag{4}$$

RL optimization problem

- The goal in RL is to **maximize the expected return** when an agent acts according to a policy
- The policy influences the probability distribution over trajectories
- The probability of a T-step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{5}$$

- The expected return, denoted by $J(\pi)$, is then:

$$J(\pi) = \mathbb{E}_{\tau \sim \pi}[R(\tau)] \tag{6}$$

- The central optimization problem in RL can then be expressed by:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{7}$$

with π^* being the optimal policy.

Value function

Crucial aspects of RL are state-value function and action-value function:

Definition

The **State-Value Function** of a policy, $V^{\pi}(s)$ is the expected return by starting in state s and always acting according to policy π:

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau) | s_0 = s] \tag{8}$$

Definition

The **Action-Value Function** of a policy, $Q^{\pi}(s, a)$ is the expected return by starting in state s and taking an arbitrary action a (which may not have come from the policy), and then forever after acting according to the policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau \sim \pi}[R(\tau) | s_0 = s, a_0 = a] \tag{9}$$

Optimal value function and optimal policy

Value functions define a partial ordering over policies, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. There is always at least one policy that is better than or equal to all other policies, this is the optimal policy π^*.

Definition

The **Optimal State-Value Function**, $V^*(s)$ is the expected return by starting in state s and always acting according to the *optimal* policy in the environment:

$$V^*(s) = \max_\pi \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s] = \max_\pi V^\pi(s) \tag{10}$$

Definition

The **Optimal Action-Value Function**, $Q^*(s, a)$ is the expected return by starting in state s, taking an arbitrary action a, and then forever after acting according to the *optimal* policy in the environment:

$$Q^*(s, a) = \max_\pi \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] = \max_\pi Q^\pi(s, a) \tag{11}$$

Optimal value function and optimal policy

By having $Q^*(s,a)$, it is possible to directly obtain the optimal action $a^*(s)$:

$$a^*(s) = \arg\max_{a \in A} Q^*(s,a) \tag{12}$$

Therefore an optimal policy can be found by maximizing over $Q^*(s,a)$:

$$\pi^*(a|s) = \begin{cases} 1 & \textit{if } a = \arg\max_{a \in A} Q^*(s,a) \\ 0 & \textit{otherwise} \end{cases} \tag{13}$$

RL algorithms

Taxonomy of RL algorithms

The final objective of RL is to find the best policy, i.e.:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{14}$$

There are a variety of algorithms to do so, depending on what to learn and how to learn it.

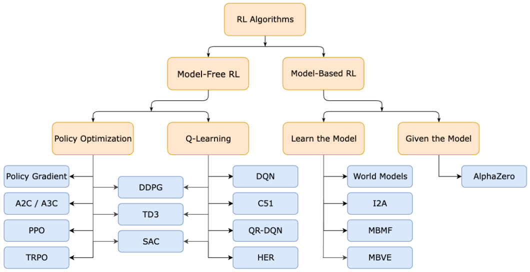

Figure: A non-exhaustive taxonomy of RL algorithms.[10]

Q-Learing

An example of RL algorithm is Q-learning:

- Q-Learning methods learn an approximation $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$
- The Q-learning algorithm keeps an estimate $Q(s,a)$ of $Q^*(s,a)$ for each state-action pair $(s,a) \in S \times A$
- By observing $(s_t, a_t, r_{t+1}, s_{t+1})$ the estimates are updated

The idea behind Deep Reinforcement Learning is that since Q-Learning is simply function approximation, a Deep Neural Network could be used to approximate this function.

AlphaGo Zero

Motivation of AlphaGo Zero

- The game of Go is one of the most challenging games for artificial intelligence
  - Enormous search space and the difficulty of evaluating board positions and moves
- Search algorithms that use minimax, alpha-beta pruning and evaluation function can not be used in the game of Go due to its huge complexity.
  - This techniques were used in DeepBlue[6] to beat Garry Kasparov in the game of chess

Game	Board size (positions)	Game-tree complexity (log to base 10)	Complexity class
Tic-tac-toe	9	5	PSPACE-complete[2]
Connect Four	42	21	in PSPACE [5]
Checkers	32	40	EXPTIME-complete [4]
Chess	64	123	EXPTIME-complete [1]
Go (19x19)	361	360	EXPTIME-complete [3]

Table: Complexity of the most famous board game.

AlphaGo Zero

Characteristics of AlphaGo Zero:

- AlphaGo Zero [9], is an algorithm based on reinforcement learning
- It started from zero, without human knowledge beyond game rules
- AlphaGo Zero achieved superhuman performance in Go.

A neural network is used for value-function approximation:

- AlphaGo Zero uses a deep neural network f_θ with parameters θ
- The neural network takes as input the raw board representation s
- The first layer of the neural network is a convolutional layer, followed by 40 convolutional blocks with skip-connections
- The neural network has two outputs (multitask learning):
  - Vector p: the vector of move probabilities p represents the probability of selecting each move, $p_a = P(a|s)$
  - Scalar v: the value v is a scalar evaluation, estimating the probability of the current player winning from position s

Self-play in AlphaGo Zero

- The neural network in AlphaGo Zero is trained from games of self-play
- In each position s, a MCTS search is executed, guided by the neural network f_θ
- The MCTS search outputs probabilities π of playing each move
- The MCTS output probabilities select much stronger moves than the raw move probabilities p of the neural network $f_\theta(s)$
    - MCTS can then be viewed as a policy improvement operator
- During the self-play the MCTS-based policy is used to select each move, and the game winner z is used as a sample of the value v (the winning player)
- Parameters are updated to make the move probabilities and value $f_\theta(s) = (p, v)$ match the search probabilities and self play winner (π, z).

Self-play in AlphaGo Zero

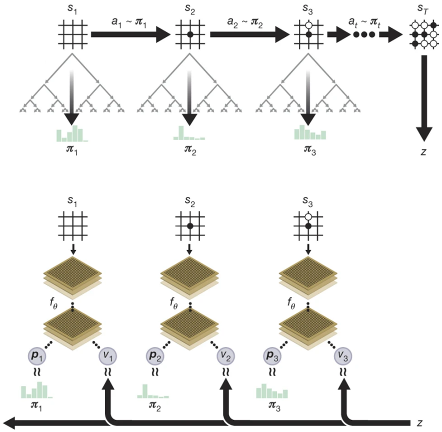

Figure: **Self-play** and **training** AlphaGo Zero.[8]

Self-play in AlphaGo Zero

- The MCTS uses the neural network f_θ to guide its simulations.

$$f_\theta(s) = (P(s,\cdot),\ V(s))$$

- Each edge (s,a) in the search tree stores a prior probability $P(s,a) = p_\theta(a|s)$, a visit count $N(s,a)$, and an action-value $Q(s,a)$.
- Each simulation starts from the root state and selects moves:

$$a_t = \arg\max_a Q(s_t,a) + U(s_t,a) \tag{15}$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s' \mid s,\, a \rightarrow s'} V(s'), \tag{16}$$

$$U(s,a) = c_{puct} P(s,a) \frac{\sum_b N(s,b)}{1 + N(s,a)} \tag{17}$$

- $U(s_t,a)$ acts as an exploration factor governed by c_{puct}
- This strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

Self-play in AlphaGo Zero

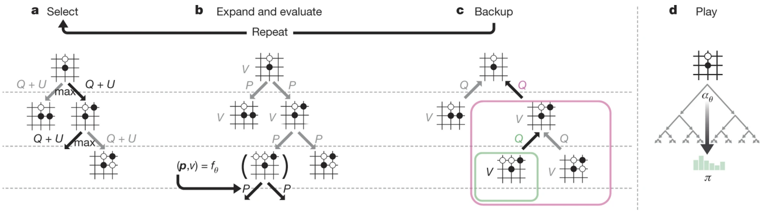

Figure: **MCTS procedure** in AlphaGo Zero.[8]

The parameters θ are adjusted by gradient descent on the loss function L:

$$f_\theta(s) = (p, v), \quad L = (z - v)^2 - \pi_t log(p) + c\,\|\theta\|^2 \tag{18}$$

Training method

The training consists in:

1. Self-play
   - N iteration of training phase
   - For each N iteration, K self-play games are played by the player α_θ using MCTS guided by the neural network f_θ.
     - In each K game the player α_θ plays against itself in order to collect new training data.
   - In each K game, for every move to choose, J simulations of MCTS are performed in order to collect more confident outcomes of the games.
2. Training the neural network:
   - Use the old and new training data collected in the K self-play games of the I-th iteration to fit the data in a fixed number of epochs using the new data collected in the K self-play games of the N-th iteration, and the data from the previous iterations
   - Decrease of the loss of the neural network eventually leads to better search in the MCTS guided by f_θ, hence leading to higher quality training examples in the next iteration, and so on.

Evaluation

1. MCTS is guided by the neural network f_θ, we want to be sure that the neural network f_θ is improving over the iterations
2. Starting with the best neural network so far $f_{\theta'}$, at the end of each iteration, the latest neural network is evaluated
3. Two players α_θ and $\alpha_{\theta'}$ play P games against each other
4. If $\alpha_{\theta'}$ win a number of games higher than a threshold, then the new neural network $f_{\theta'}$ is considered the best network so far.

Note: The number of games for evaluation are performed letting each player start the game half of the times.

Empirical results of AlphaGo Zero

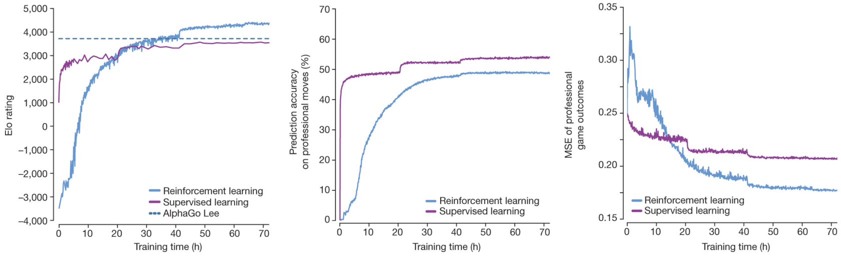

Figure: AlphaGo Zero performance. [8]

- AlphaGo zero performs worse in predicting master Go player moves
- It is better in predicting the actual outcome of a game (lower MSE)
- This is due to nature of reinforcement learning: since we are not in charge of giving any kind of human expertise to the algorithm, there is no human knowledge ceiling to the performances.

Hands-on: Connect4 Zero

Connect4 Zero

- Use the techniques of AlphaGo Zero [9] to achieve master level in Connect4
- I considered the game of Connect4 due to the limited computational resources at my disposal
- Starting from this code[1], I changed it to support the game of Connect4. I also added:
    - A graphical user interface to play against the agent
    - A simpler neural network architecture. (In order to being able to perform the training on my machine)
    - Episodes of self-play are performed in parallel to drastically reduce the training time (taking inspiration form the work of David Silver "Playing Atari with Deep Reinforcement Learning" [7]).

[1] `https://github.com/suragnair/alpha-zero-general`

Hyperparameters

Training hyperparameters:

- Number of iterations: 200
- Self-play games per each iteration: 100
- Number of MCTS: 25
- Number of games against previous agent: 10
- Update threshold: 60%
- Number of training epochs: 10

Connect4 Zero

Live Demo

Connect4 Zero

Thank you for your attention!