Lecture 14 - Monte-Carlo Tree Search

Guiliang Liu

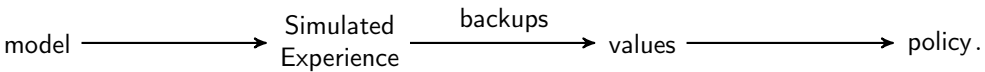
The Chinese University of Hong Kong, Shenzhen

DDA4230: Reinforcement Learning
Course Page: [Click]

Model-based Planning and Learning

Planning: A computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment.

State-space Planning: A search through the state space for an optimal policy or an optimal path to a goal. It computes value functions by updates or backup operations applied to simulated experience.



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Model Learning

"What if we do not have the model?" $\rightarrow$ "Learn the model!"

In the model learning regime we assume that the state and action spaces $\mathcal{S}, \mathcal{A}$ are known and typically we also assume conditional independence between state transitions and rewards, i.e.,

$$\mathbb{P}(s_{t+1}, r_t \mid s_t, a_t) = \mathbb{P}(s_{t+1} \mid s_t, a_t)\mathbb{P}(r_t \mid s_t, a_t).$$

Hence learning a model consists of two main parts of learning the reward function $R(\cdot \mid s, a)$ (assumed to be a deterministic) and the transition distribution $\mathbb{P}(\cdot \mid s, a)$.

Model Learning

Given a set of real trajectories $\{s_t^k, a_t^k, r_t^k, \ldots, s_T^k\}_{k=1}^K$, the model learning can be posed as a supervised learning problem.

1) Learning the reward function $R(s, a) \rightarrow$ a regression problem.

2) Learning the transition function $\mathbb{P}(s' \mid s, a) \rightarrow$ a density estimation problem. The key steps are:

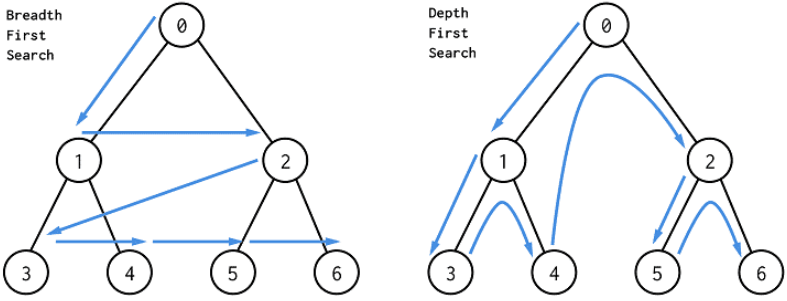
- Pick a suitable family of parametrization models (e.g., Neural Network).
- Choose an appropriate loss function (e.g., Mean Squared Error (MSE)).
- Update the models based on the trajectories data.

Simulation-based Search

Given access to a model of the world, simulation-based search methods seek to identify the best action to take based on forward search and simulations.

Simulation-based Search

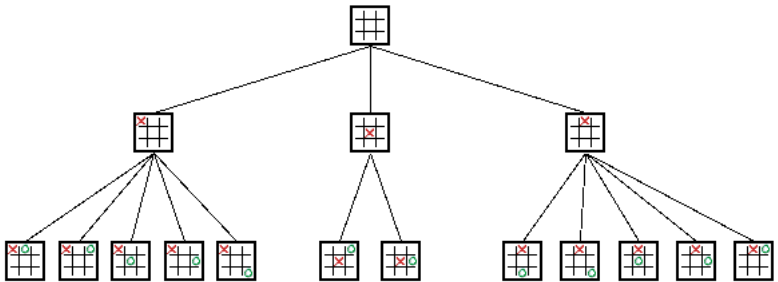
Uninformed search is a kind of generic search algorithm which are given no extra information other than an abstract problem definition.



Source of the image: link

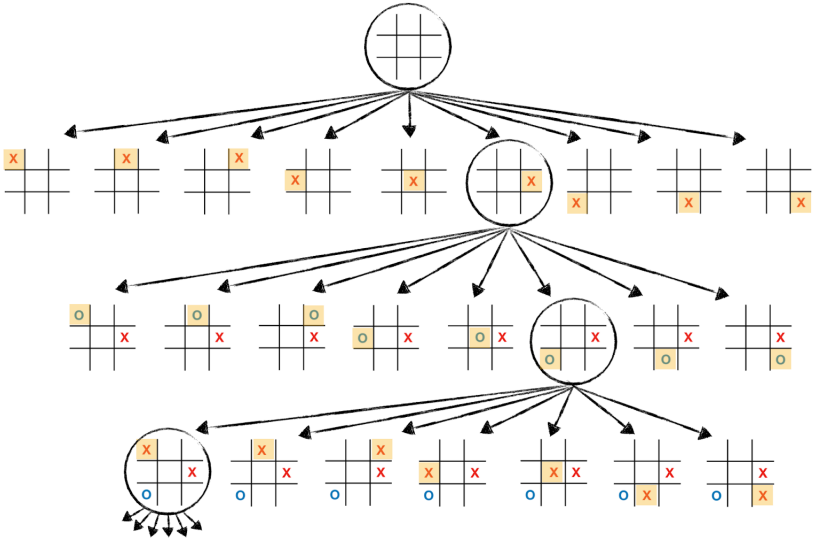
Simulation-based Search

During the research, **a search tree** is built with the current state as the root and its children nodes as possible next states generated using the model.

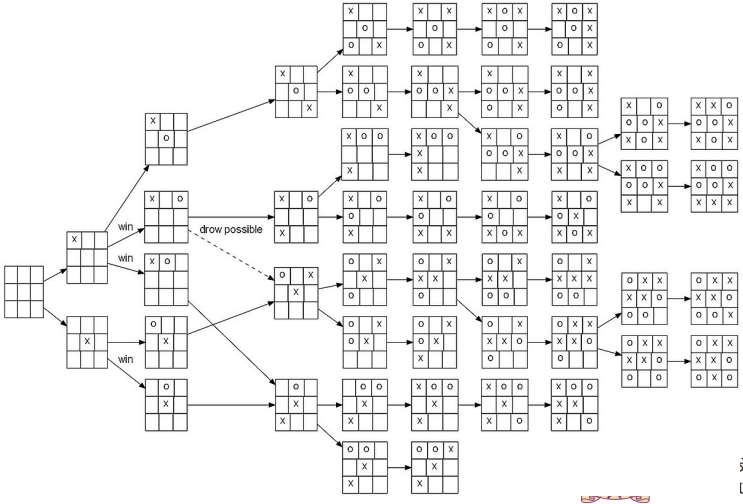


Source of the image: link

Simulation-based Search



Simulation-based Search



Source of the image: link

Simulation-based Search

However, for some complex games like Go (which has an average branch factor of 250), using an uninformed search becomes computationally intractable:

- At step 1: 250
- At step 2: $250^2 = 62500$
- At step 3: $250^3 = 15{,}625{,}000$
- At step 4: $250^4 = 3{,}906{,}250{,}000$
- At step 5: $250^5 = 976{,}562{,}500{,}000$
- a ...
- At step 10: $250^{10} = 953{,}674{,}316{,}406{,}250{,}000{,}000{,}000$

A brief guide to Go: link

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is based on two principles:

- The true value of a state can be estimated using average returns of random simulations.
- These values can be used to iteratively adjust the policy to focus on high-value regions of the search space.

Monte-Carlo Tree Search

MCTS progressively constructs a partial search tree starting out with the current node set as the root.

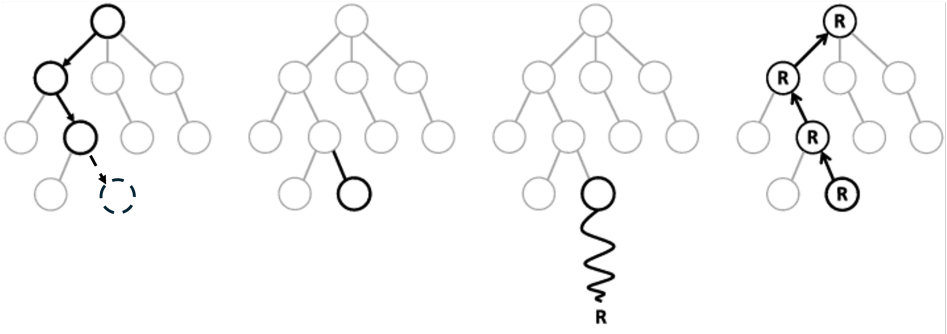
- The tree consists of nodes corresponding to states s.
- Each node stores statistics such as a count for each state-action pair $N_{s,a}$ and the Monte-Carlo action value estimates $Q(s, a)$.

At its core, MCTS consists of repeated iterations of 4 steps(in practice constrained by computing time and resources):

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

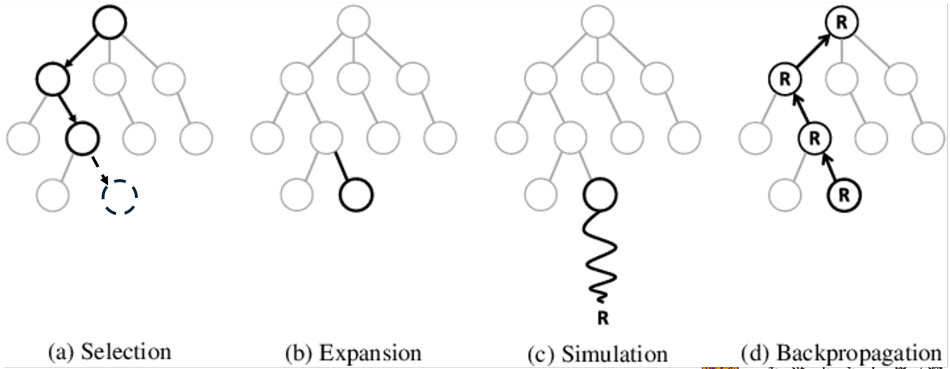
Monte-Carlo Tree Search

Selection. Starting at the root node, we select child nodes recursively in the tree till a non-terminal leaf node is reached.



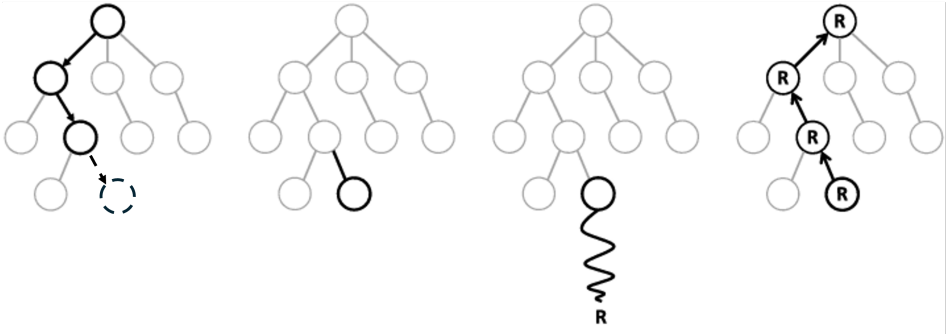
Monte-Carlo Tree Search

Expansion. The chosen leaf node is added to the search tree.



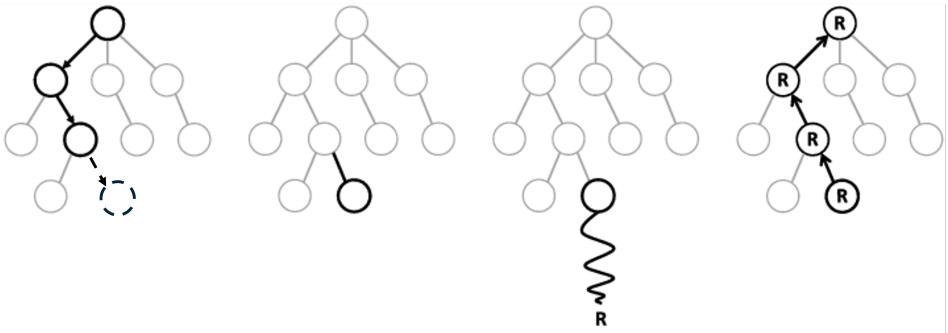
Monte-Carlo Tree Search

Simulations. Simulations are run from this node to produce an estimate of the outcomes.



Monte-Carlo Tree Search

Simulations. The values obtained in the simulations are backpropagated through the tree by following the path from the root to the chosen leaf in reverse and updating the statistics of the encountered nodes.



Implementation for MCTS

MCTS iteratively grows the search tree by the above four steps. The algorithm is:

Algorithm 1: General MCTS algorithm

Input s_0
Create root node s_0
while *within computational budget* **do**
  $s_k \leftarrow TreePolicy(s_0)$
  $\Delta \leftarrow Simulation(s_k)$
  $Backprop(s_k, \Delta)$
Return $\arg\max_a Q(s_0, a)$

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Implementation for MCTS

Variants of MCTS generally contain modifications to the two main policies involved.

- **Tree policy.** To chose actions for nodes in the tree based on the stored statistics. Variants include greedy, UCB.
- **Rollout policy.** For simulations from leaf nodes in the tree. Variants include random simulation, default policy network in the case of AlphaGo.

Implementation for MCTS

In the backdrop stages, we increment the visit count ($N_{s,a}$) and update the action values $Q(s,a)$ on all the traversed edges (s,a):

- Update the state-action value:

$$Q(s,a) = \frac{Q(s,a) * N_{s,a} + R(s_L)}{N_{s,a} + 1}$$

  where s_L denotes the leaf note and $R(s_L)$ denotes the average cumulative rewards calculated during simulation.

- Increment the visit count: $N_{s,a} = N_{s,a} + 1$.

Implementation for MCTS

Greedy Tree Search. To implement MCTS, we can 1) choose actions greedily amongst the tree nodes in the first stage and 2) generate rollouts using a random policy in the simulation stage.

Algorithm 2: Greedy MCTS algorithm

Input s_0
Create root node s_0
while *within computational budget* **do**
  $s_{next} \leftarrow s_0$
  while $|Children(s_{next})| \neq 0$ **do**
    $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s_{next}, a)$
    $s_{next} \leftarrow NextState(s_{next}, a)$
  $\Delta \leftarrow RandomSimulation(s_{next})$
  $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$

Implementation for MCTS

Upper Confidence Tree Policy. Using a greedy policy may avoid actions after observing even one bad outcome despite the significant uncertainty about its true value. To handle it, we can pick the action that maximizes the upper confidence bound on the value of the action, which is:

Algorithm 3: Upper Confidence MCTS algorithm

Input s_0
Create root node s_0
while *within computational budget* **do**
  $s_{next} \leftarrow s_0$
  while $|Children(s_{next})| \neq 0$ **do**
    $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + \sqrt{\frac{2 \log \sum_b N_{s,b}}{N_{s,a}}}$
    $s_{next} \leftarrow NextState(s_{next}, a)$
  $\Delta \leftarrow RandomSimulation(s_{next})$
  $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$

Implementation for MCTS

Predictor Upper Confidence Tree. When given a prior policy (predictor) π_0, we can have the following updates:

- Instead of doing random simulation, we can do policy rollout with the policy π.
- We can exploit the prior knowledge by applying the predictor UCB as follows:

$$a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + c_{puct} \pi_0(a|s) \frac{\sqrt{\sum_b N_{s,b}}}{N_{s,a}+1}$$

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Implementation for MCTS

Algorithm 4: Predictor Upper Confidence MCTS algorithm

Input s_0, π_0
Create root node s_0
while *within computational budget* **do**
  $s_{next} \leftarrow s_0$
  while $|Children(s_{next})| \neq 0$ **do**
    $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + c_{puct}\pi_0(a|s)\frac{\sqrt{\sum_b N_{s,b}}}{N_{s,a}+1}$
    $s_{next} \leftarrow NextState(s_{next}, a)$
  $\Delta \leftarrow StrategicSimulation(s_{next}, \pi_0)$
  $Backprop(s_{next}, \Delta)$
Return $\arg\max_a Q(s_0, a)$

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Implementation for MCTS

Predictor Upper Confidence Tree with Value Priors. When given a prior value function V_0, we can further update our algorithm:

- Instead of applying the results with random simulation, we can combine them with our value outputs:

$$V(s) = (1-\lambda)V_0(s) + \lambda R(s)$$

where $R(s)$ is the estimation from simulation (policy rollout)

Implementation for MCTS

Algorithm 5: Predictor Upper Confidence MCTS algorithm with Priors

Input s_0, π_0, V_0
Create root node s_0
while *within computational budget* **do**
  $s_{next} \leftarrow s_0$
  while $|Children(s_{next})| \neq 0$ **do**
    $a \leftarrow \arg\max_{a \in \mathcal{A}} Q(s,a) + c_{puct}\pi_0(a|s)\frac{\sqrt{\sum_b N_{s,b}}}{N_{s,a}+1}$
    $s_{next} \leftarrow NextState(s_{next}, a)$
  $\Delta \leftarrow StrategicSimulation(s_{next}, \pi_0)$
  $V(s_{next}) = (1-\lambda)V_0(s_{next}) + \lambda\Delta$
  $Backprop(s_{next}, V(s_{next}))$
Return $\arg\max_a Q(s_0, a)$

Advantages of MCTS

The main advantages of MCTS include

1. States are evaluated dynamically, that is, MDP is only solved from the current state, unlike dynamic programming.
2. It efficiently combines planning and sampling to break the curse of dimensionality in complicated games like Go.

AlphaGo: A Case Study

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a world champion, and is arguably the strongest Go player in history.

Introducing the Deepmind AlphaGo: link

How Google Deepmind AlphaGo works? link

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

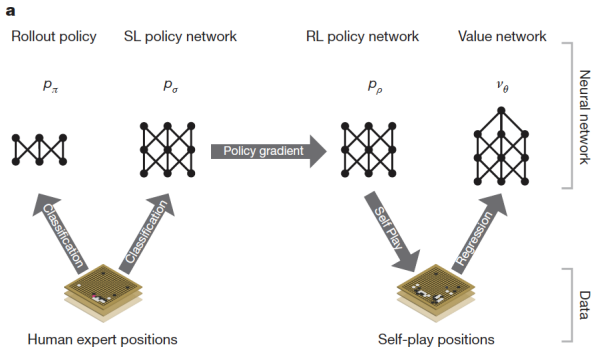
AlphaGo: A Case Study

AlphaGo is a computer program that combines Monte Carlo simulation with value and policy networks.

- It uses value networks to evaluate board positions and policy networks to select moves. These deep neural networks are trained by combining supervised learning from human expert games, and reinforcement learning from games of self-play.
- Without any lookahead search, the neural networks play Go at the level of state of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play by combining Monte Carlo simulation with value and policy networks

AlphaGo: A Case Study

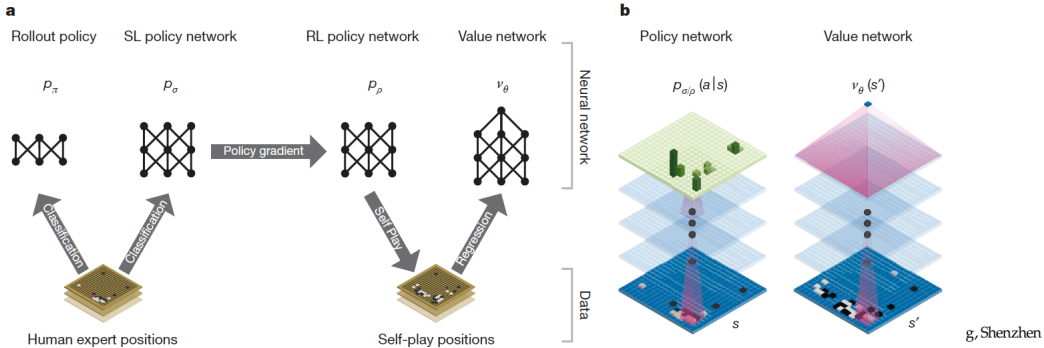
Supervised learning of policy networks. The SL policy network $p_\sigma(a|s)$ (13-layer, 57.0% accuracy) and rollout policy network $p_\pi(a|s)$ (2-layer, 24.2% accuracy) predicts expert moves in the game of Go using supervised learning.



g, Shenzhen

AlphaGo: A Case Study

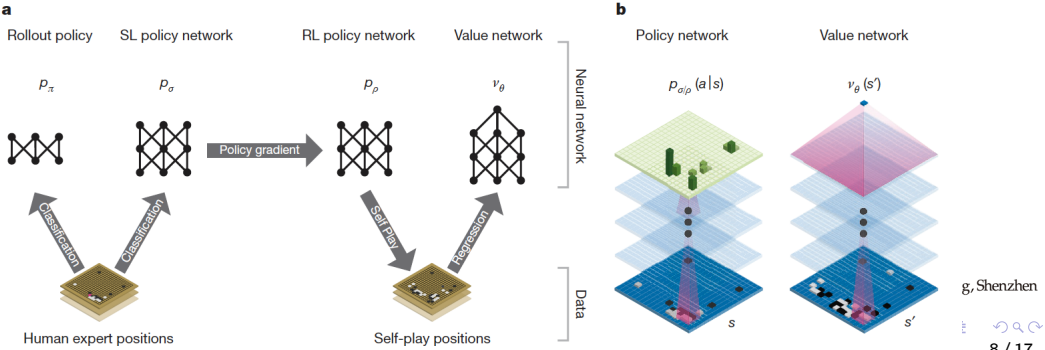
Reinforcement learning of policy networks. 1) The RL policy network p_ρ is identical in structure to the SL policy network, and its weights ρ are initialized to the same values, $\rho = \sigma$.



g, Shenzhen

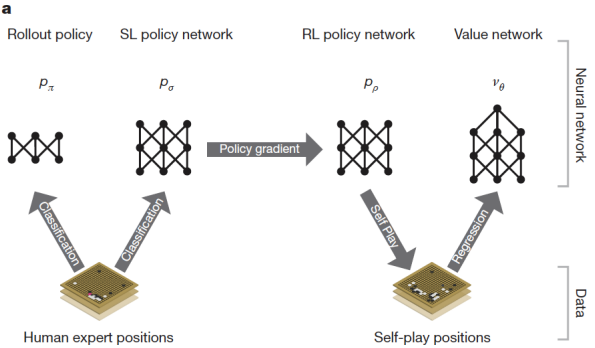
AlphaGo: A Case Study

Reinforcement learning of policy networks. 2) It plays games between the current policy network p_ρ and a randomly selected previous iteration of the policy network. The reward function $r(s)$ is zero for all non-terminal time steps $t < T$. At the end of the game, $r(s)$ is +1 for winning and -1 for losing. p_ρ is updated by policy gradient.



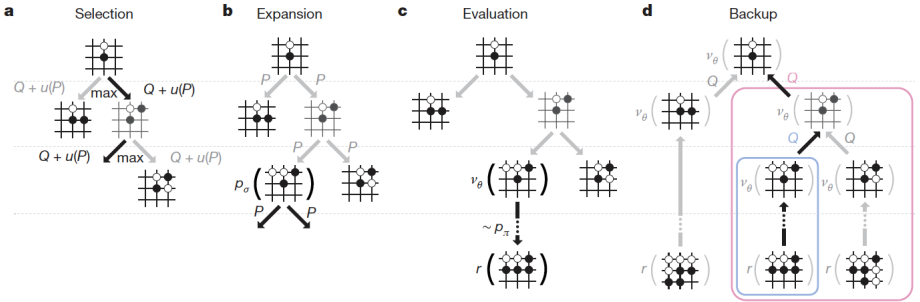
AlphaGo: A Case Study

Reinforcement learning of value networks. Value function $v_\theta(s)$ that predicts the outcome from positions of games played by RL policy network p_ρ. This neural network has a similar architecture to the policy network, but outputs a single prediction. The value network is trained by regression on state-outcome pairs (Monte Carlo method).



AlphaGo: A Case Study

Searching with policy and value networks. **Tree Structure.** Each edge (s, a) of the search tree stores an action value $Q(s, a)$, visit count $N(s, a)$, and prior probability $P(s, a)$. The tree starts from the root state

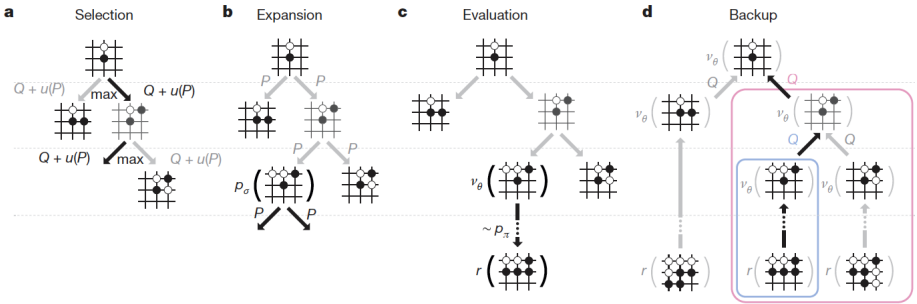


AlphaGo: A Case Study

Searching with policy and value networks. **1) Selection.** At each time step t of each simulation, an action a_t is selected from state s_t:

$$a_t = \arg\max_a (Q(s_t, a) + u(s_t, a))$$

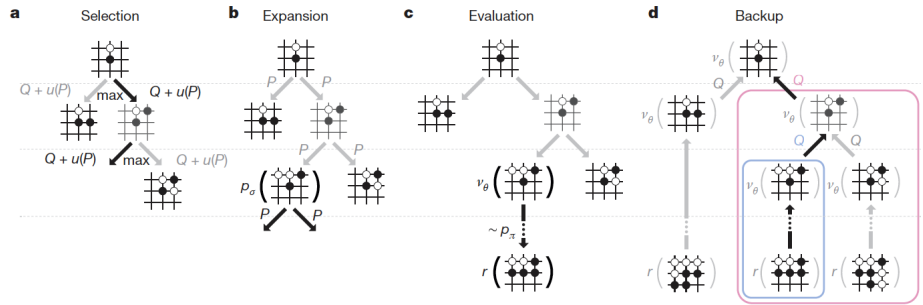
where $u(s, a) \propto \frac{p_\sigma(a|s)}{1+N(s,a)}$.



, Shenzhen

AlphaGo: A Case Study

Searching with policy and value networks. **2) Expansion.** When the traversal reaches a leaf node s_L at step L, the leaf node may be expanded. The leaf position s_L is processed just once by the SL policy network p_σ. The output probabilities are stored as prior probabilities P for each legal action a.

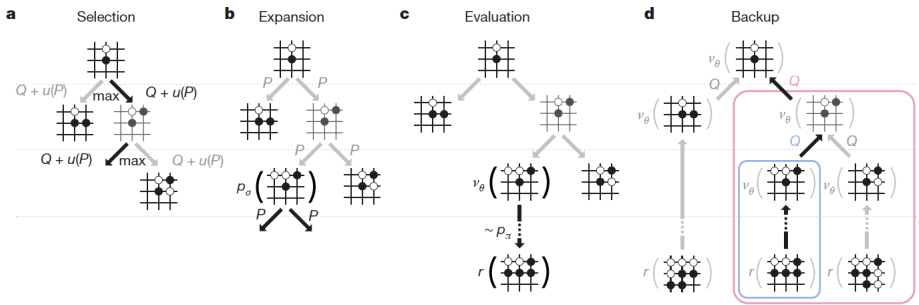


, Shenzhen

AlphaGo: A Case Study

Searching with policy and value networks. **3) Evaluation.** The leaf node is evaluated by the value network $v_\theta(s_L)$ and the outcome $R(s_L)$ of a random rollout played out until terminal step T using the fast rollout policy p_π; these evaluations are combined:

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda R(s_L)$$

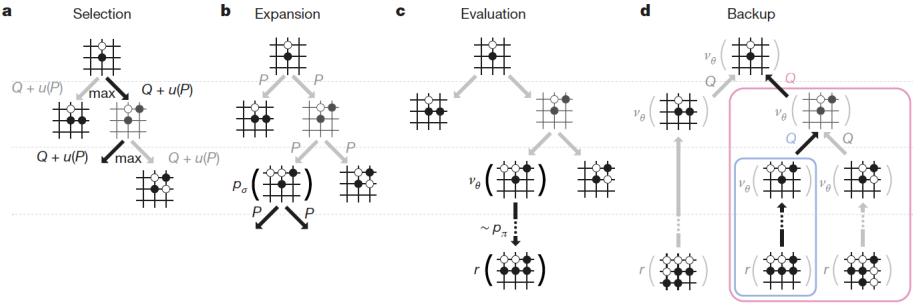


, Shenzhen

AlphaGo: A Case Study

Searching with policy and value networks. **4) Backup.** The action values and visit counts of all traversed edges are updated. Each edge accumulates the visit count and mean evaluation of all simulations passing through that edge

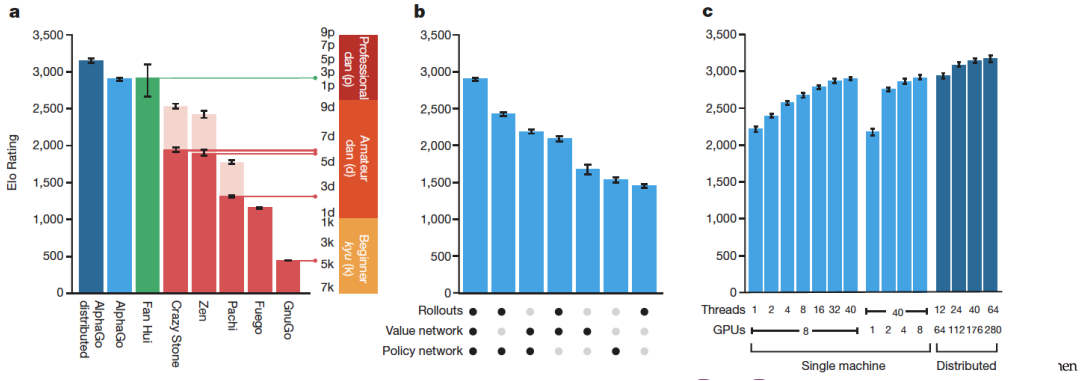
$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i) \text{ and } Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$



, Shenzhen

AlphaGo Performance

Evaluating the playing strength of AlphaGo.



AlphaGo Performance

An early version of AlphaGo was tested on hardware with various numbers of CPUs and GPUs, running in asynchronous or distributed mode.

Configuration and performance

Configuration	Search threads	No. of CPU	No. of GPU	Elo rating
Distributed	64	1,920	280	3,168
Distributed	40	1,202	176	3,140
Distributed	24	764	112	3,079
Distributed	12	428	64	2,937
Single	40	48	8	2,890
Single	40	48	4	2,850
Single	40	48	2	2,738
Single[4] p. 10–11	40	48	1	2,181

AlphaGo Performance

Performance of AlphaGo and its family.

Configuration and strength[62]

Versions	Hardware	Elo rating	Date	Results
AlphaGo Fan	176 GPUs,[53] distributed	3,144[52]	Oct 2015	5:0 against Fan Hui
AlphaGo Lee	48 TPUs,[53] distributed	3,739[52]	Mar 2016	4:1 against Lee Sedol
AlphaGo Master	4 TPUs,[53] single machine	4,858[52]	May 2017	60:0 against professional players; Future of Go Summit
AlphaGo Zero (40 block)	4 TPUs,[53] single machine	5,185[52]	Oct 2017	100:0 against AlphaGo Lee 89:11 against AlphaGo Master
AlphaZero (20 block)	4 TPUs, single machine	5,018[63]	Dec 2017	60:40 against AlphaGo Zero (20 block)

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

AlphaGo Zero: Starting from scratch

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

AlphaGo Zero: Starting from scratch: link

AlphaGo Zero: Discovering new knowledge: link

AlphaGo Zero: Starting from scratch

AlphaGo Zero differs from the previous AlphaGo in several important aspects.

- Zero is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data.
- Zero uses only the black and white stones from the board as input features without hand-engineered features.
- Zero uses a single neural network, rather than separate policy and value networks.
- Zero uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts

Go v.s., Zero: Different Input Features

Input features from AlphaGo:

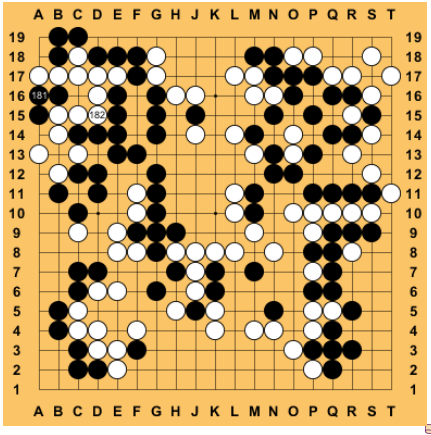
Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Feature planes used by the policy network (all but last feature) and value network (all features).

Go v.s., Zero: Different Input Features

Input features from AlphaGo Zero:



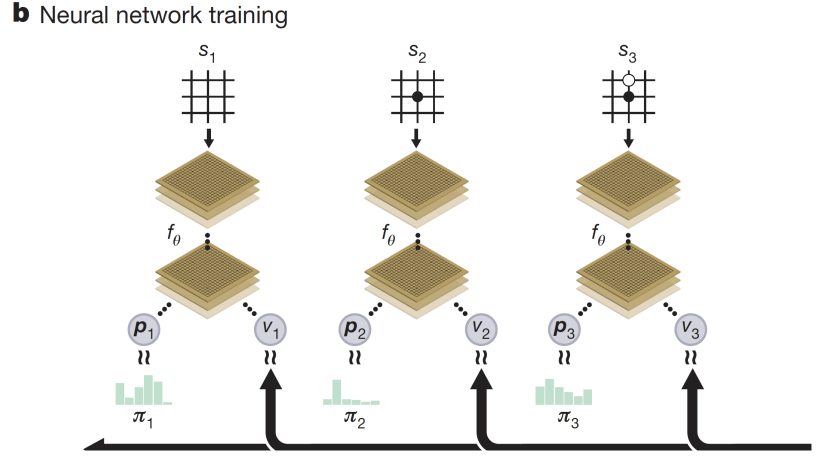
Go v.s., Zero: Different Neural Networks

This neural network outputs both move probabilities and a value, $(p, v) = f_\theta(s)$. This neural network combines the roles of both policy network and value network into a single architecture.

- The vector of move probabilities p represents the probability of selecting each move a (including pass), $p_a = Pr(a|s)$.
- The value v is a scalar evaluation, estimating the probability of the current player winning from position s.

Go v.s., Zero: Different Neural Networks

Neural Network Structure in AlphaGo Zero:



Go v.s., Zero: Different Neural Networks

Neural network training. The neural network parameters θ are updated to:

- Maximize the similarity of the policy vector p_t to the search probabilities π_t.
- Minimize the error between the predicted winner v_t and the game winner R.

$$\ell = (R - v_t)^2 - \pi_t \log p_t + c\|\theta\|^2 \tag{1}$$

  where $R \in \{-1, +1\}$ denotes the winner of this game.

- The new parameters are used in the next iteration of self-play.

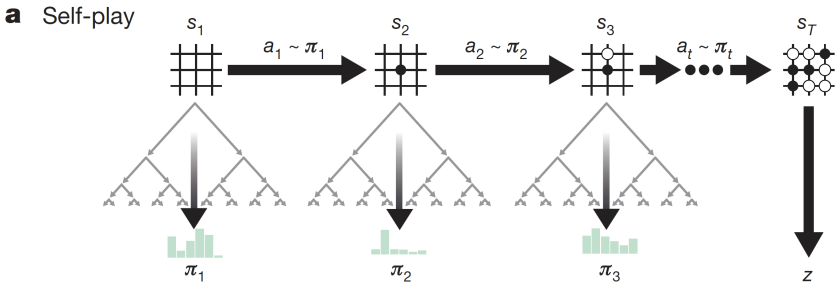
Go v.s., Zero: Different Learning Methods

Self-play RL: The program plays a game $s_1, ..., s_T$ against itself.

- In each position st, an MCTS is executed using the latest neural network f_θ.
- Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$.
- The terminal position s_T is scored according to the rules of the game to compute the game winner R.

where $z = R$ denotes the final winner.

Go v.s., Zero: Different Learning Methods

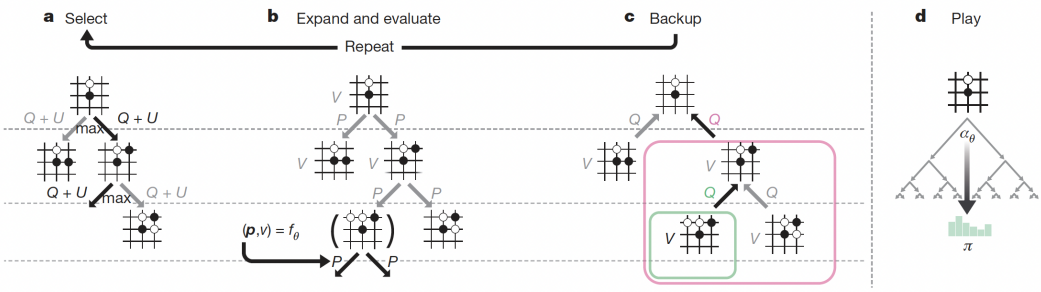


where $z = R$ denotes the final winner.

Go v.s., Zero: Different Research Tree

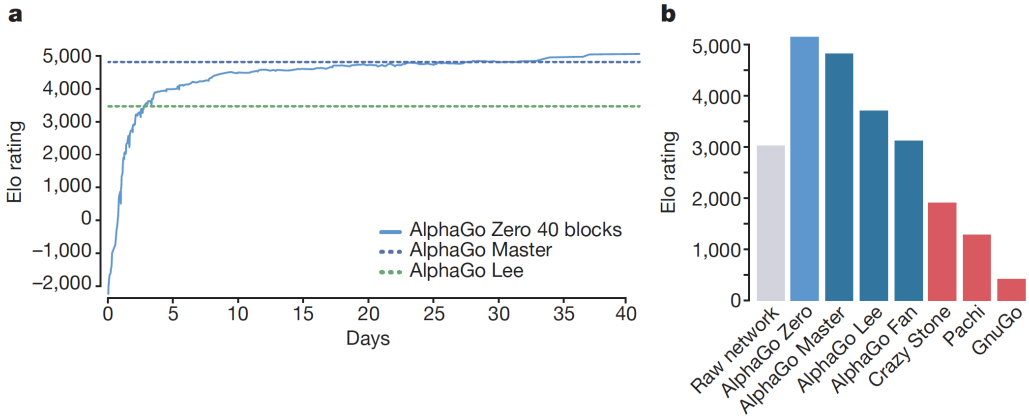
Zero uses a simpler tree search in the following ways:

- Rely upon this single neural network to evaluate positions and sample moves.
- Do not perform any Monte Carlo rollouts.



AlphaGo Zero Performance

Learning curve for AlphaGo Zero using a larger 40-block residual network over 40 days (Left), and Final performance of AlphaGo Zero (Right).



AlphaZero: Masters Chess, Shogi,and Go

COMPUTER SCIENCE

A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

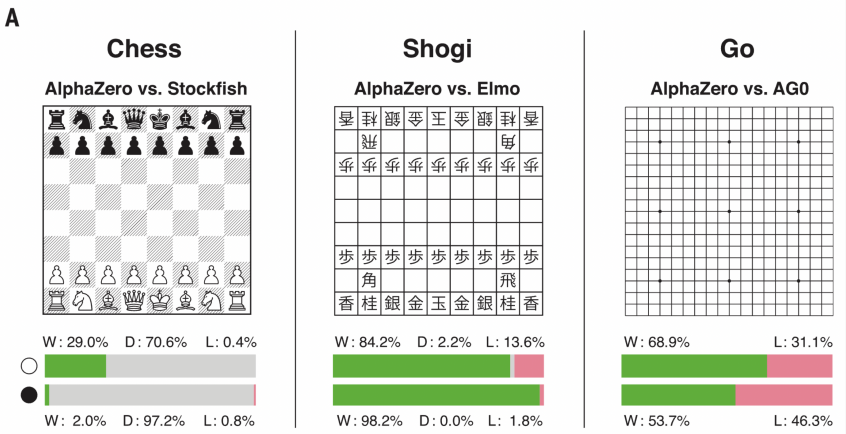
David Silver[1,2]*†, Thomas Hubert[1]*, Julian Schrittwieser[1]*, Ioannis Antonoglou[1], Matthew Lai[1], Arthur Guez[1], Marc Lanctot[1], Laurent Sifre[1], Dharshan Kumaran[1], Thore Graepel[1], Timothy Lillicrap[1], Karen Simonyan[1], Demis Hassabis[1]†

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

AlphaZero: Masters Chess, Shogi,and Go

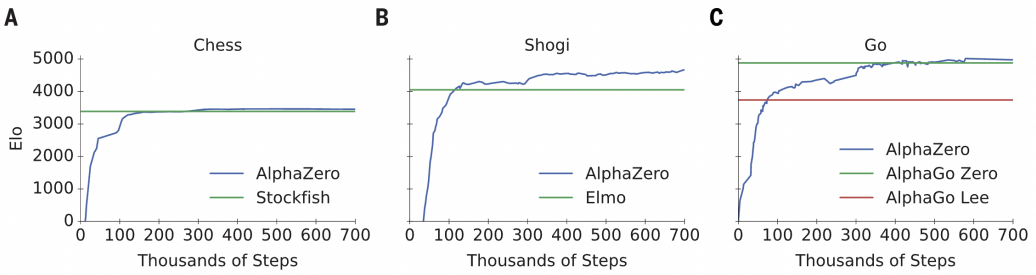
- Generalize the self-play approach into a single AlphaZero algorithm that can achieve superhuman performance in many challenging games.
- Starting from random play and given no domain knowledge except the game rules, AlphaZero convincingly defeated a world champion program in the games of chess and shogi (Japanese chess), as well as Go.

AlphaZero: Masters Chess, Shogi,and Go



The Chinese University of Hong Kong, Shenzhen

AlphaZero: Masters Chess, Shogi,and Go



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Question and Answering (Q&A)