Mastering the Game of Go without Human Knowledge

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert et al. "Mastering the game of go without human knowledge." ***nature*** **550, no. 7676 (2017): 354-359.**

Paper review by Eric Allen

AlphaGo

- First program to achieve superhuman performance at the game of Go

AlphaGo Fan

- Played against Fan Hui in October 2015
- Distributed over many machines and 176 GPUs

AlphaGo Lee

- Defeated Lee Sodol 4-1 in October 2016
- Value network trained from outcomes of fast games of self-play
    - Iterated many times
- 12 convolutional layers of 256 *planes*
- Distributed over many machines, using 48 TPUs
- Handcrafted features and *rollouts*

AlphaGo Master

- An intermediate (unpublished) upgrade
- Defeated top human players 60-0 in January 2016
- Same neural network architecture, reinforcement learning algorithm, and MCTS algorithm as AlphaGo Zero
- Uses the same handcrafted features and rollouts as AlphaGo Lee
- Training initialized by supervised learning from human data

AlphaGo Zero

- Learned entirely from self-play reinforcement learning, starting from random weights for the neurons

- No rollouts

- No human supervision

- Only the raw board (history) was used as input features

- Used a single machine on the Google Cloud, with 4 TPUs (the smallest and simplest approach that worked)

Domain Knowledge Incorporated in AlphaGo Zero

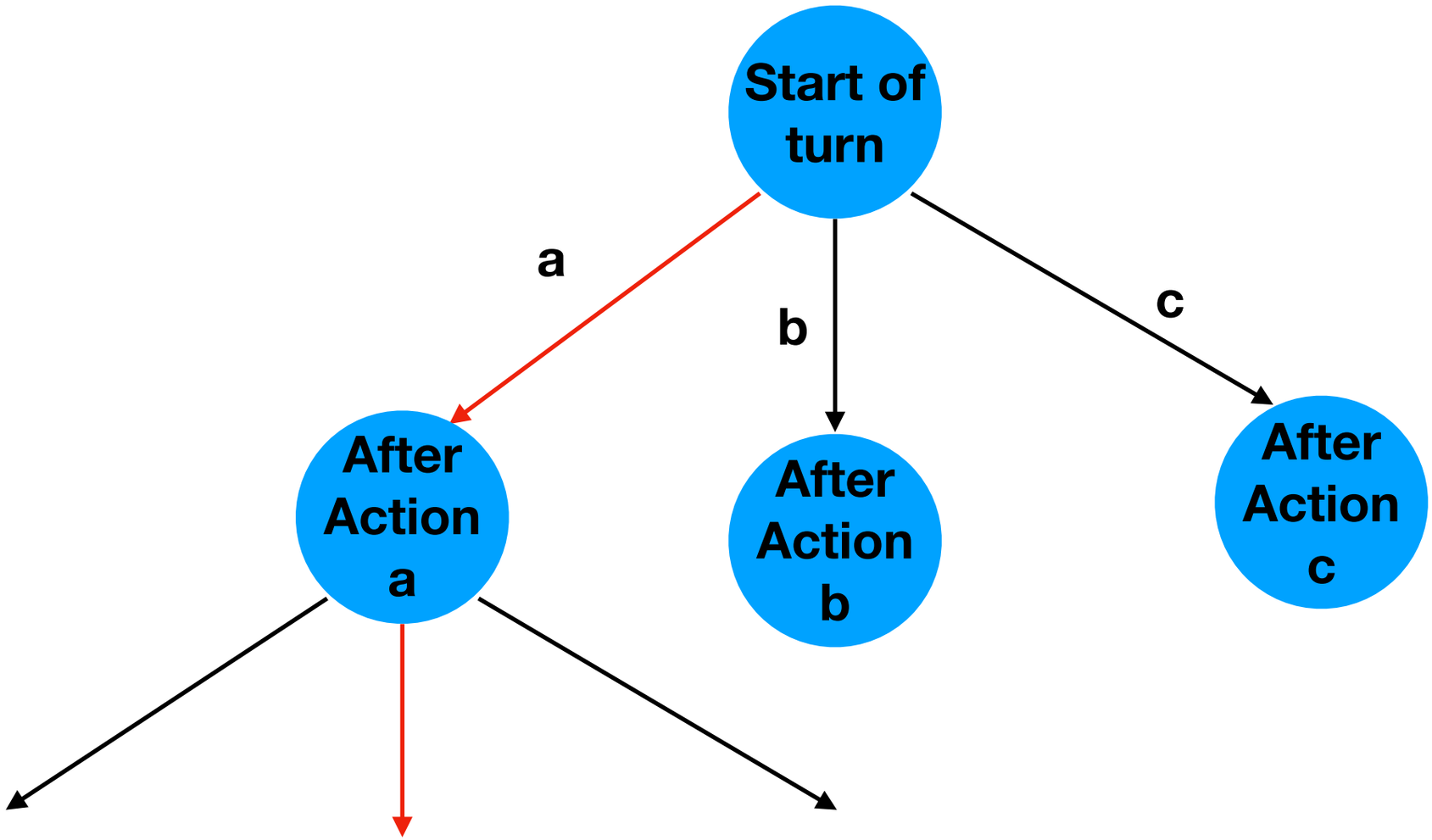
- Perfect knowledge of the rules of Go
- Uses “Tromp-Taylor” scoring during MCTS
    - Avoids some ambiguities in Chinese, Japanese, and Korean rules
- Board evaluation is invariant with respect to reflections and rotations, and virtually symmetric with respect to color
    - These properties are used to augment the dataset

How AlphaGo Zero Chooses Moves to Play During a Game

AlphaGo Zero Uses Monte Carlo Tree Search

- Consider a game tree for Go:
  - Every game state / board state is a node in the tree
  - The root of the tree is the starting game state
  - There is an edge from vertex *s* to vertex *t* iff there is a legal action that transforms the game state from *s* to *t*

AlphaGo Zero Uses Monte Carlo Tree Search



AlphaGo Zero Uses Monte Carlo Tree Search

- Each search begins at a root node s_0 (the current actual game state)

- The search selects a single action at each step for L time steps, arriving at a state s_L

- Many games are simulated in parallel in this fashion from root node s_0

- Statistics on the potential actions are collected by these simulations

- The “best” potential action is actually played

AlphaGo Zero Uses Monte Carlo Tree Search

- Each action in the search tree is annotated with:
  - *N(s,a)* the number of times that *a* has been visited
  - *W(s,a)* the sum of values that have resulted from performing action *a*
  - *Q(s,a)* the mean action value
  - *P(s,a)* the prior probability of selecting edge *a*

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Exploitation **Exploration**

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Informed by a neural network (the *value network*)

Informed by a neural network (the *policy network)*

Search Algorithm

- At each step in a simulation:
    - The next action is selected in proportion to the scores of the potential actions

Value and Policy Networks

- The value network estimates the value (desirability) of an action
- The policy network estimates the probability of an action being taken
- We will see that AlphaGo Zero uses two heads on the same network for value and policy
- Thus, a single network is trained to output both quantities

Board Evaluation

- When reaching the end of a path:
  - The value v of board s_L is computed using our value network applied to a random reflection or rotation of s_L
  - Each new edge a from our end state s_L is initialized as:

$$N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a$$

- The value p_a is computed using our policy network
- The network(s) are shared resources:
  - They evaluate actions for eight boards at a time
  - All threads waiting on those board evaluations are locked until they complete

Backup

- The statistics of each board s_t in the path from s_0 to s_L is now updated:

$$
\begin{aligned}
N(s_t, a_t) &\quad += &&\quad 1 \\
W(s_t, a_t) &\quad += &&\quad v \\
Q(s_t, a_t) &\quad = &&\quad \frac{W(s_t, a_t)}{N(s_t, a_t)}
\end{aligned}
$$

Search Algorithm

- The tree nodes along the selected path are kept
- All other tree nodes are discarded

- If the root value and best child value are below a threshold, AlphaGo Zero resigns

Rollouts

- When game states are encountered for the first time, how do we decide which action to take?

- Rollouts: Select a move based on some default policy (e.g., random play, or something smarter)

- AlphaGo Zero does not use rollouts; it simply evaluates the game state based on the current value and policy networks

Action Selection

- At the end of the search, an action is selected for actual play

- Action is chosen probabilistically, in proportion to its "exponentiated visit count":

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

Action Selection

- At the end of the search, an action is selected for actual play

- Action is chosen probabilistically, in proportional to its "exponentiated visit count":

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

Temperature.

Higher values result in more exploration.

The Self-Play Training Pipeline

Neural Network Architecture

- A single network with two heads is used to output both value and policy quantities

- Input is a 19 x 19 x 17 “image stack” consisting of 17 binary feature planes:

  - 8 feature planes indicate presence of the player’s stones for past eight moves

  - 8 feature planes indicate presences of opponent’s stones for past eight moves

  - One feature plane indicates the color to play

Neural Network Architecture

- Input features are processed by a convolutional block followed by either 19 or 39 residual blocks (depending on the size of the run)

Convolutional Block

- Consist of:
  - 256 filters of size 3 x 3 with stride 1
  - Batch normalization
  - Rectifier non-linearity

Residual Blocks

- Each consists of:
    - Its own convolutional block
    - Batch normalization
    - Rectifier non-linearity
    - Another convolutional block
    - Another batch normalization
    - A skip connection adding back the input
    - Another rectifier non-linearity

Heads for Computing Policy and Value

- Policy head consists of:
  - Convolution of 2 kernels of size 1x1 of stride 1
  - Batch normalization
  - Rectifier non-linearity
  - A fully connected linear layer with output of size $19^2 + 1$ providing the probability of a move to each position or a pass

Heads for Computing Policy and Value

- Value head consists of:
  - Convolution of 2 kernels of size 1x1 of stride 1
  - Batch normalization
  - Rectifier non-linearity
  - A fully connected linear layer to a hidden layer of size 256
  - Another rectifier non-linearity
  - A fully connected linear layer to a scalar
  - A *tanh* non-linearity outputting a scalar in the range [-1, 1]

Parameter Initialization

- Parameters initialized by Gaussian process optimization for a small run (20 blocks, 3 days)

- The resulting trained network was then used to select parameters for a larger run (40 blocks, 40 days)

Neural Network Training

Neural network is optimized with stochastic gradient descent with momentum and learning rate annealing

$$l = (z - v)^2 - \pi^T log p + c||\theta||^2$$

Neural Network Training

Neural network is optimized with stochastic gradient descent with momentum and learning rate annealing

$$l = (z - v)^2 - \pi^T logp + c||\theta||^2$$

mean-squared error for the value head

cross-entropy for the policy head

Weight regularization to prevent overfitting

Optimization

- Each network is optimized on Google Cloud using 64 GPU workers and 19 CPU “parameter servers”

- Minibatch size is 2048 (32 per worker)

- Minibatches sampled uniformly from most recent 500,000 games

Self-Play Training Pipeline

- Neural network parameters are continually optimized from “recent” self-play
- Many players are constantly run and evaluated in parallel
- The best performing player at any moment is used to generate new training data

Evaluation at Checkpoints

- Optimization process stops at a checkpoint every 1000 training steps
- At a checkpoint, every network is tested against the best network on 400 games
- During testing, each game uses MCTS with no exploration and 1600 simulations to select each move
- If a new player wins at least 55% then it becomes the new best player

Training Steps

- At each training step, the current best player plays 25,000 games of self-play, using 1600 simulations in MCTS to select each move
    - Requires about 0.4 s per search
- Temperature set to 1 for first 30 games
- Temperature set to near 0 for remainder of games
- “Dirichlet noise” added to prior probabilities to further aid in exploration:

$$(1 - \epsilon)p + \epsilon\eta$$

“Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, proverbs, and books.

“In the space of a few days, starting tabula rosa, AlphaGo Zero was able to rediscover much of this Go knowledge, as well as novel strategies that provide new insights into the oldest of games.”

Algorithm 2 UCT

procedure UCTSEARCH(s_0)
  while time remaining **do**
    $\{s_0, ..., s_T\}, R = \text{SIMULATE}(s_0)$
    BACKUP($\{s_0, ..., s_T\}, R$)
  end while
  return $\underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s_0, a)$
end procedure

procedure SIMULATE(s_0)
  $t = 0$
  $R = 0$
  repeat
    if $s_t \in \mathcal{T}$ **then**
      $a = \text{UCB1}(s_t)$
    else
      NEWNODE(s_t)
      $a_t = \text{DEFAULTPOLICY}(s_t)$
    end if
    $s_{t+1} = \text{SAMPLETRANSITION}(s_t, a_t)$
    $r_{t+1} = \text{SAMPLEREWARD}(s_t, a_t, s_{t+1})$
    $R = R + r_{t+1}$
    $t \mathrel{+}= 1$
  until $Terminal(s_t)$
  return $\{s_0, ..., s_t\}, R$
end procedure

procedure UCB1(s)
  $a^* = \underset{a}{\text{argmax}}\ Q(s, a) + c\sqrt{\frac{2 \log N(s)}{N(s,a)}}$
  return a^*
end procedure

procedure BACKUP($\{s_0, ..., s_T\}, R$)
  for $t = 0$ **to** $T - 1$ **do**
    $N(s_t) \mathrel{+}= 1$
    $N(s_t, a_t) \mathrel{+}= 1$
    $Q(s_t, a_t) \mathrel{+}= \frac{R - Q(s_t, a_t)}{N(s_t, a_t)}$
  end for
end procedure

procedure NEWNODE(s)
  $N(s) = 0$
  for all $a \in \mathcal{A}$ **do**
    $N(s, a) = 0$
    $Q(s, a) = \infty$
  end for
  $\mathcal{T}.Insert(s)$
end procedure