AlphaZero Algorithm - NOTES

02.26.2021

Vincent Manier

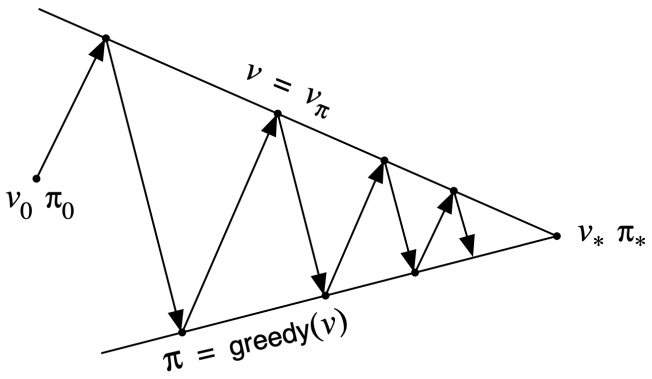
A. Main Concepts

1. An Approximate Policy Iteration Scheme

A classic algorithm in reinforcement learning and dynamic programming is the Generalized Policy Iteration (GPI) where a policy is constantly improved by alternating a policy evaluation step and a policy improvement step. Each step can include many iterations.

Policy evaluation consists in computing the state-value function of an arbitrary policy. It is also called the prediction problem and makes the value function consistent with the current policy.

Policy improvement consists in using the current value function to improve the current policy (e.g. by acting greedily by choosing the highest value). Improving the policy also causes an inconsistency with the current value function.



In Sutton (2018) : “Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal. Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In

either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly."

In AlphaGo Zero, MCTS is used for both policy improvement and policy evaluation, that is why the "self-play algorithm can similarly be understood as an approximate policy iteration scheme". The integration of MCTS as a "powerful policy improvement operator" is one of the remarkable innovations of AlphaGo Zero. In other words, it seems that the list of recommended probabilities of taking some actions in a given state (policy for a given state) output by MCTS is always more precise than the raw move probabilities output by the neural network.

- **Evaluation**: the "select, expand and evaluate and back-up" sequences select moves according to statistics in the search tree (PUCT algorithm). State-value of every visited node is stored in memory during a simulation.
- **Improvement**: this is the "play" decision made according to exponentiated visit count. At every time step t, the probability vector of every node is stored in memory during a simulation.

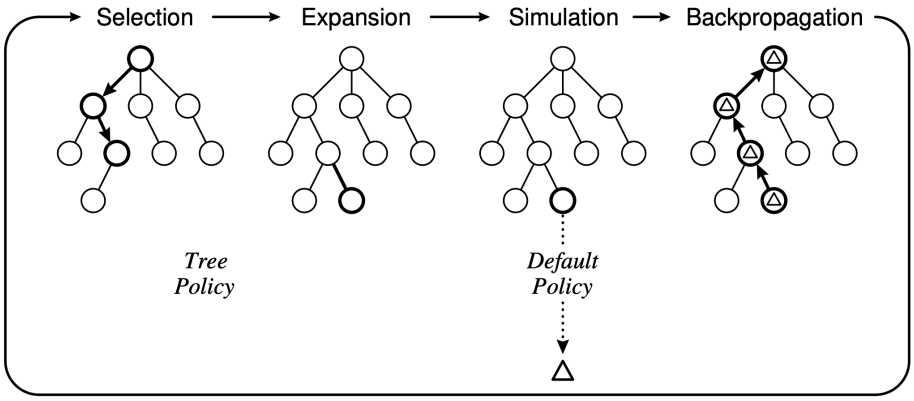
"The MCTS search outputs probabilities π of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities p of the neural network fθ(s); MCTS may therefore be viewed as a powerful policy improvement operator. Self-play with search – using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value – may be viewed as a powerful policy evaluation operator."

Note that in the [Mastering the game of Go with deep neural networks and tree search] paper, the preference of selecting action with visit count compared to action-value is justified by this comment: "this is less sensitive to outliers than maximizing action-value".

2. Monte Carlo Tree Search (MCTS)

Plain vanilla MCTS has basically 2 phases by move:

- **Search phase**, i.e. building the tree (many iterations)
    - consists in iteratively building a tree where a node represents a state and links to child nodes represent actions leading to subsequent states.
    - Four steps are applied per iteration: selection, expansion, simulation and back-up or backpropagation.
    - We call tree policy the rules for selection and expansion; and default policy the rules for running a simulation (usually random)
- **Selection phase**, i.e. selecting the best action - 4 criteria were identified in this MCTS Survey: *1) Max child*: Select the root child with the highest reward. 2) *Robust child*: Select the most visited root child. 3) *Max-Robust child*: Select the root child with both the highest visit count and the highest reward. If none exist, then continue searching until an acceptable visit count is achieved 4) *Secure child*: Select the child which maximises a lower confidence bound.



3. Leveraging deep neural networks in MCTS

According to *Rémi Coulom*, one of the main difficulties identified in solving Go was "the creation of an accurate static position evaluator". Monte Carlo evaluation was already seen as a solution that would fit the dynamic nature of Go. Interestingly, AlphaZero combines a powerful static evaluator (neural network) and a dynamic search.

AlphaZero MCTS introduced the following changes to the plain-vanilla MCTS:

- The tree policy uses the PUCT algorithm and a neural network that outputs the prior probabilities.
- The default policy is replaced by a neural network that outputs an estimated state-value of a position.
- The action selection is proportional to an exponentiated visit count (version of Robust child).

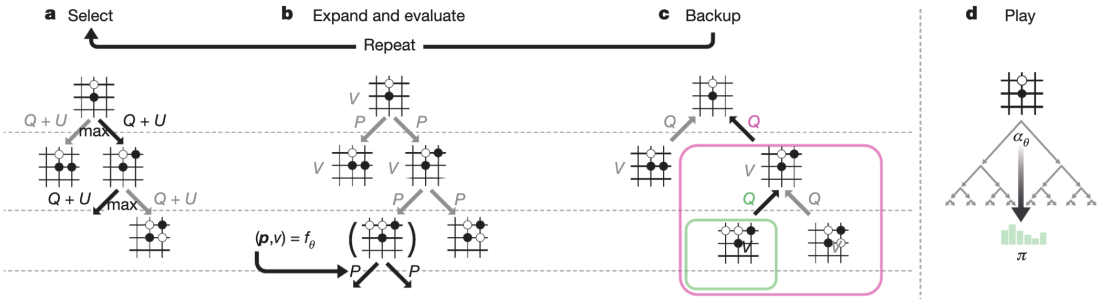


Figure 2 | MCTS in AlphaGo Zero. a, Each simulation traverses the tree by selecting the edge with maximum action value *Q*, plus an upper confidence bound *U* that depends on a stored prior probability *P* and visit count *N* for that edge (which is incremented once traversed). **b**, The leaf node is expanded and the associated position *s* is evaluated by the neural network $(P(s, \cdot), V(s)) = f_\theta(s)$; the vector of *P* values are stored in the outgoing edges from *s*. **c**, Action value *Q* is updated to track the mean of all evaluations *V* in the subtree below that action. **d**, Once the search is complete, search probabilities π are returned, proportional to $N^{1/\tau}$, where *N* is the visit count of each move from the root state and τ is a parameter controlling temperature.

C. Game Representation, MCTS and Network Parameters

AlphaZero Game	Notations	Description
Self-play game	$s_1, s_2, \ldots, s_{n-1}, s_T$	A succession of position from start to terminal stat
Terminal state	T	End of the game. Both players pass, search value drops below a resignation threshold, or when the game exceeds a max length
Final reward	r_T	The score of the game
State	S = s + H	Raw board representation s of the position **and its history H**
Policy represented by a move probability vector for a given position s = s_t	p_S	Probability distribution over moves (inc. pass), p_a = Pr (a / S) for every possible a
Scalar evaluation of the score for a given position s = s_t	v_S	Estimated score of the current player from position s [note: sometimes considered as the winning probability from the current player perspective]
MCTS-guided search outputs for a given position s = s_t	$\boldsymbol{\pi}_t$	MCTS-guided policy or recommended moves to play after search (the search improves the existing policy)
Winner = {-1,1}	z	Losing or Winning
Observed score at time t from the perspective of the current player	z_t = +/-r_T	The final score r_T is propagated back to all the past positions of the winner leading to the win.

MCTS	Notations	Description
Edge	S, a	State, (legal) action
Edge statistics	N, W, Q, P	Statistics stored for each edge after the root node
Visit count	N(S,a)	Number of times a node was traversed during the search phase
Total action-value	W(S,a) +=v	Cumulated action-values or total score propagated up to the tree during simulation.
Mean action-value	Q(s,a) = $\frac{W(s,a)}{N(s,a)}$	
PUCT	U(s, a) = c_{puct}x P(s, a) x N'	Polynomial Upper Confidence Tree
Low visit count indicator	N' = $\frac{[\sum_b N(s,b)]^{1/2}}{1+N(s,a)}$	Favors low visit count
constant c_{puct}	c_{puct}	Controls the level of exploration
Action selector during search	$a_t = argmax_a$ (Q(s_t,a) + U(s_t,a))	Action selection prior expansion / evaluation
Prior probability	P(s,a)	Probability queried from the neural network p,v = f_θ(S)
Root State, Leaf Node	S_0, S'	
Upper-Confidence Bound	UCB Q(s,a) + U(s,a)	U(s,a) ∝ P(s,a) / (1 + N(s,a))
Temperature	τ	Controls the level of exploration [0->deterministic;1->exploration]
Search parameters		selected by Gaussian process optimisation
Dirichlet noise	P(s, a) = (1 −)p_a + $\epsilon\eta_a$	η ~ Dir(0.03) and ϵ = 0.25 - adds exploration
Virtual loss		Ensures each thread evaluates different nodes (parallel implementation)
Best action selector post-search	π(a/s_0)=$[\frac{N(s0,a)/}{\sum_b N(s0,b)}]^{1/\tau}$	Action selected proportionally to exponentiated (tau) visit count [Q: we use the same distribution for training target?]
Resign test	$v_{root} < v_{resign}$ **and** $v_{child} < v_{resign}$	Test for a player to resign

Network	Notations	Description
Architecture: ResNet	/	Residual blocks 4 of convolutional layers with batch normalisation and rectifier non-linearities Note: Not studied yet in details
Neural network with parameters **θ**	$p,v = f_{\theta}(S)$	The policy and scalar evaluation of the score. mapping a state to a move probability vector (incl. pass) and a value
Input: State	S	See game representation
Outputs: policy and value function	p_S, v_S	See below
Policy represented by a move probability vector for a given position $s = s_t$	p_S	Probability distribution over moves (inc. pass), $p_a = Pr(a / S)$ for every possible a
Scalar evaluation of the score for a given position $s = s_t$	v_S	Estimated score of the current player from position s
Loss function	$l = MSE + CE + c.L2$	See below
Mean-Square Error (MSE)	$MSE = (z - v)^2$	MSE between self-play winner v and predicted value v
Cross-Entropy (CE)	$CE = -\pi^T \log p$	Maximise the similarity of the neural network move probabilities p to the search probabilities π
L2 weight regularization	$L2 = \|\|\theta\|\|^2$	Encourages small weights
Parameter c	c	hyperparameter controlling the intensity of L2 penalty to avoid overfitting
Learning rate	$\boldsymbol{\alpha}$	Cyclical learning rate
Momentum		Set to 0.9

Notes: In the original paper, s denotes both the position and the state

D. Implementations - Review Status

1. General

So far, I have not studied examples of AlphaZero implementations in detail.

2. Oracle posts

I reviewed the Oracle post and summarized below my main questions.

Oracle Series	Comfort level	Questions
Part I	9 / 10	Can we train the network during self-play, i.e. before reaching a terminal state? z not available. Why don't we start close to a terminal state (assuming we know some) to ease the learning and accelerate backpropagation of z?
Part II	9 / 10	Does MCTS always improve the policy? How do we treat impossible states? Is there a benefit training the impossible states?
Part III	6 / 10	C and alpha are both related to exploration - why 2 indicators tuned separately? Could it be simplified? Not clear on inference optimization tools / 8-bit quantization Not clear why duplicated positions would have different priors (if the network is not trained during a self-play)
Part IV	9 / 10	"Ultimately the falloff approach is able to achieve a slightly lower error percentage than averaging"??
Part V	2 / 10	Model deployment process? Distributed inference in practice? Inference bucket size? calibration? int8 quantization?
Part VI	5 / 10	Learning rate schedule "When repeated positions are found in your training window, they are likely from different model generations" We found that using 2 epochs of training per window sample provided a good bump in learning over single epoch training, without bottlenecking our synchronous training cycle for too long

4. Asynchronous MCTS and parallel implementation

I do not fully understand yet the asynchronous MCTS process or parallel simulations implementation.

- *Asynchronous MCTS: AlphaGo Zero uses an asynchronous variant of MCTS that performs the simulations in parallel. The neural network queries are batched and each search thread is locked until evaluation completes. In addition, the 3 main processes: self-play, neural network training and comparison between old and new networks are all done in parallel.*
- *Self-Play Training Pipeline AlphaGo Zero's self-play training pipeline consists of three main components, all executed asynchronously in parallel. Neural network parameters θi are continually optimised from recent self-play data; AlphaGo Zero players αθi are continually evaluated; and the best performing player so far, αθ∗ , is used to generate new self-play data.*
- *In Alpha Zero on the other hand there is only one network f0 that is both Value and Policy Network. It is trained entirely via self-play starting from random initialization. There is a number of networks trained in parallel and the best one is chosen for training data generation every checkpoint after evaluation against best current neural network. (int8 blog)*

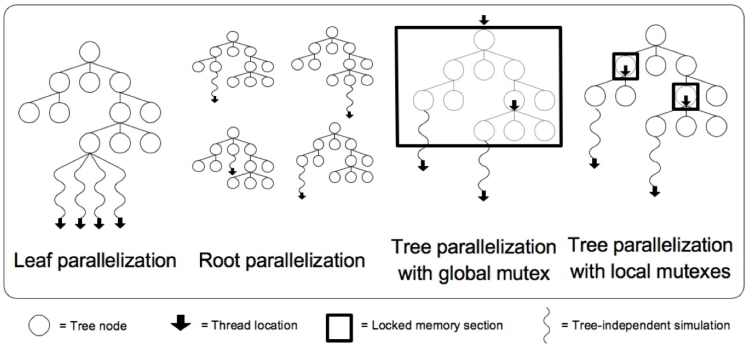
5. MCTS improvements (MCTS Survey)

a. Tree Policy

i. Bandit-based
ii. Selection enhancement
iii. All Moves As First (AMAF)
iv. Game-theoretic enhancements
v. Move pruning
vi. Expansion enhancements (none!)

b. Other enhancements

i. Simulation enhancements
ii. Backpropagation enhancements
iii. Parallelisation



E. Some Personal Comments

One of the most exciting promises of artificial intelligence (AI) is to build systems able to autonomously learn new domains beyond human intelligence by simply interacting with the environment and figuring out optimal strategies to achieve a given goal. A board game, like Go, is a good application for testing AI systems as it cannot be solved by brute force, but its environment can be perfectly simulated (simple rules of the game) and its goal is straight-forward (win a game). Solving Go was also considered as the pinnacle of AI research or just impossible until recently, which makes it an even more interesting challenge.

The beauty of AlphaZero algorithm lies in its surprising ability to teach an agent new domain knowledge from scratch through an iterative process of searching (MCTS) and gradual learning by pure self-play (no prior human knowledge). However, even if the underlying concepts of AlphaGo algorithm are intuitively simple to understand and demonstrate, its implementation in a generic form proves to be quite challenging judging by the number of hyper-parameters and game-specific features!

I am wondering how the search algorithm could also learn by better interacting with the learning process and the neural network architecture. It appears to me that the search process does not evolve much as a function of the learning curve, the quality or type of the learning (v and p). Also the learning contained within the neural network is so dependent on its architecture (the bigger, the better?) that it raises the question of the possibility of improving or scaling the architecture during the learning process.

F. Possible Next steps

1. Understand examples of implementations
2. Study AlphaZero.jl
3. Continue reading about implementation details

G. Interesting Sources

- DeepMind papers
    - Mastering the Game of Go without Human Knowledge
    - Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm
    - Mastering the game of Go with deep neural networks and tree search
- AlphaZero.jl repository jonathan-laurent and documentation
    - Surag Nair's tutorial.
    - Monte Carlo Tree Search (MCTS) tutorial.
    - Posts from Oracle
- Multiple Policy Value Monte Carlo Tree Search : a balance needs to be reached between accurate state estimation and more MCTS simulations.
- Learning to Play Othello Without Human Knowledge
- Thinking Fast and Slow with Deep Learning and Tree Search [TO READ]
- 8-bit Inference with TensorRT [TO STUDY]
- TensorRT Integration Speeds Up TensorFlow Inference [TO STUDY]
- Taking the Human Out of the Loop: A Review of Bayesian Optimization [TO STUDY]
- MCTS by Rémy Coulom [TO STUDY]
- MCTS Survey [TO STUDY]

H. History

"In 2016, researchers at DeepMind announced a new breakthrough -- the development of a new AI engine, alphago for the game of go.The AI was able to beat a professional player LeeSedol. The breakthrough was significant, because go was far more complex than chess: the number of possible games is so high, that a professional go engine was believed to be way out of reach at that point, and human intuition was believed to be a key component in professional play. Still, performance in alphago depends on expert input during the training step, and so the algorithm cannot be easily transferred to other domains. This changed in 2017, when the team at DeepMind updated their algorithm, and developed a new engine called alphago zero. This time, instead of depending on expert gameplay for the training, alphago zero learned from playing against itself, only knowing the rules of the game. More impressively, the algorithm was generic

enough to be adapted to chess and shogi (also known as japanese chess). This leads to an entirely new framework for developing AI engine, and the researchers called their algorithm, simply as the alphazero. The best part of the alphazero algorithm is simplicity: it consists of a Monte Carlo tree search, guided by a deep neural network. This is analogous to the way humans think about board games -- where professional players employ hard calculations guides with intuitions.” Udacity