A Simple Alpha(Go) Zero Tutorial

29 December 2017

This tutorial walks through a synchronous single-thread single-GPU (read malnourished) game-agnostic implementation of the recent AlphaGo Zero paper by DeepMind. It's a beautiful piece of work that trains an agent for the game of Go through pure self-play without *any* human knowledge except the rules of the game. The methods are fairly simple compared to previous papers by DeepMind, and AlphaGo Zero ends up beating AlphaGo (trained using data from expert games and beat the best human Go players) convincingly. Recently, DeepMind published a preprint of Alpha Zero on arXiv that extends AlphaGo Zero methods to Chess and Shogi.

The aim of this post is to distil out the key ideas from the AlphaGo Zero paper and understand them concretely through code. It assumes basic familiarity with machine learning and reinforcement learning concepts, and should be accessible if you understand neural network basics and Monte Carlo Tree Search. Before starting out (or after finishing this tutorial), I would recommend reading the original paper. It's well-written, very readable and has beautiful illustrations! AlphaGo Zero is trained by self-play reinforcement learning. It combines a neural network and Monte Carlo Tree Search in an elegant policy iteration framework to achieve stable learning. But that's just words- let's dive into the details straightaway.

The Neural Network

Unsurprisingly, there's a neural network at the core of things. The neural network f_θ is parameterised by θ and takes as input the state s of the board. It has two outputs: a continuous value of the board state $v_\theta(s) \in [-1, 1]$ from the perspective of the current player, and a policy $\vec{p}_\theta(s)$ that is a probability vector over all possible actions.

When training the network, at the end of each game of self-play, the neural network is provided training examples of the form $(s_t, \vec{\pi}_t, z_t)$. $\vec{\pi}_t$ is an estimate of the policy from state s_t (we'll get to how $\vec{\pi}_t$ is arrived at in the next section), and $z_t \in \{-1, 1\}$ is the final outcome of the game from the perspective of the player at s_t (+1 if the player wins, -1 if the player loses). The neural network is then trained to minimise the following loss function (excluding regularisation terms):

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t))$$

The underlying idea is that over time, the network will learn what states eventually lead to wins (or losses). In addition, learning the policy would give a good estimate of what the best action is from a given state. The neural network architecture in general would depend on the game. Most board games such as Go can use a multi-layer CNN architecture. In the paper by DeepMind, they use 20 residual blocks, each with 2 convolutional layers. I was able to get a 4-layer CNN network followed by a few feedforward layers to work for 6x6 Othello.

Monte Carlo Tree Search for Policy Improvement

Given a state s, the neural network provides an estimate of the policy $\vec{p}_\theta$. During the training phase, we wish to improve these estimates. This is accomplished using a Monte Carlo Tree Search (MCTS). In the search tree, each node represents a board configuration. A directed edge exists between two nodes $i \rightarrow j$ if a valid action can cause state transition from state i to j. Starting with an empty search tree, we expand the search tree one node (state) at a time. When a new node is encountered, instead of performing a rollout, the value of the new node is obtained from the neural network itself. This value is propagated up the search path. Let's sketch this out in more detail.

For the tree search, we maintain the following:

- $Q(s,a)$: the expected reward for taking action a from state s, i.e. the Q values
- $N(s,a)$: the number of times we took action a from state s across simulations
- $P(s,\cdot)=\vec{p}_\theta(s)$: the initial estimate of taking an action from the state s according to the policy returned by the current neural network.

From these, we can calculate $U(s,a)$, the upper confidence bound on the Q-values as

$$U(s,a)=Q(s,a)+c_{puct}\cdot P(s,a)\cdot\frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)}$$

Here c_{puct} is a hyperparameter that controls the degree of exploration. To use MCTS to improve the initial policy returned by the current neural network, we initialise our empty search tree with s as the root. A single simulation proceeds as follows. We compute the action a that maximises the upper confidence bound $U(s,a)$. If the next state s' (obtained by playing action a on state s) exists in our tree, we recursively call the search on s'. If it does not exist, we add the new state to our tree and initialise $P(s',\cdot)=\vec{p}_\theta(s')$ and the value $v(s')=v_\theta(s')$ from the neural network, and initialise $Q(s',a)$ and $N(s',a)$ to 0 for all a. Instead of performing a rollout, we then propagate $v(s')$ up along the path seen in the current simulation and update all $Q(s,a)$ values. On the other hand, if we encounter a terminal state, we propagate the actual reward (+1 if player wins, else -1).

After a few simulations, the $N(s,a)$ values at the root provide a better approximation for the policy. The improved stochastic policy $\vec{\pi}(s)$ is simply the normalised counts $N(s,\cdot)/\sum_b(N(s,b))$. During self-play, we perform MCTS and pick a move by sampling a move from the improved policy $\vec{\pi}(s)$. Below is a high-level implementation of one simulation of the search algorithm.

```python
def search(s, game, nnet):
    if game.gameEnded(s): return -game.gameReward(s)

    if s not in visited:
        visited.add(s)
        P[s], v = nnet.predict(s)
        return -v

    max_u, best_a = -float("inf"), -1
    for a in game.getValidActions(s):
        u = Q[s][a] + c_puct*P[s][a]*sqrt(sum(N[s]))/(1+N[s][a])
        if u>max_u:
            max_u = u
            best_a = a
    a = best_a

    sp = game.nextState(s, a)
    v = search(sp, game, nnet)

    Q[s][a] = (N[s][a]*Q[s][a] + v)/(N[s][a]+1)
    N[s][a] += 1
    return -v
```

mcts.py hosted with ♡ by GitHub view raw

Note that we return the negative value of the state. This is because alternate levels in the search tree are from the perspective of different players. Since $v\in[-1,1]$, $-v$ is the value of the current board from the perspective of the other player.

Policy Iteration through Self-Play

Believe it or not, we now have all elements required to train our unsupervised game playing agent! Learning through self-play is essentially a policy iteration algorithm- we play games and compute Q-values using our current policy (the neural network in this case), and then update our policy using the computed statistics.

Here is the complete training algorithm. We initialise our neural network with random weights, thus starting with a random policy and value network. In each iteration of our algorithm, we play a number of games of self-play. In each turn of a game, we perform a fixed number of MCTS simulations starting from the current state s_t. We pick a move by sampling from the improved policy $\vec{\pi}_t$. This gives us a training example $(s_t, \vec{\pi}_t, \_)$. The reward _ is filled in at the end of the game: +1 if the current player eventually wins the game, else -1. The search tree is preserved during a game.

At the end of the iteration, the neural network is trained with the obtained training examples. The old and the new networks are pit against each other. If the new network wins more than a set threshold fraction of games (55% in the DeepMind paper), the network is updated to the new network. Otherwise, we conduct another iteration to augment the training examples.

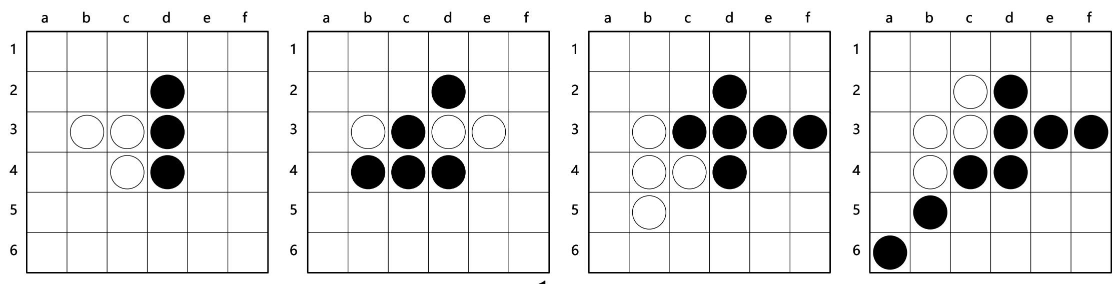
And that's it! Somewhat magically, the network improves almost every iteration and learns to play the game better. The high-level code for the complete training algorithm is provided below.

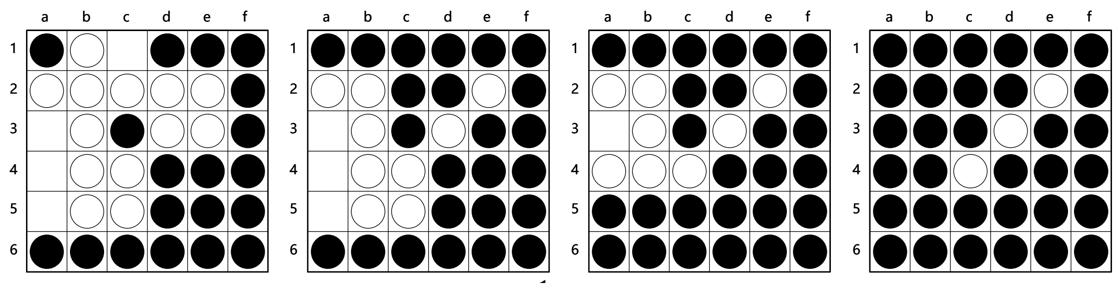
```python
def policyIterSP(game):
    nnet = initNNet()                                       # initialise random neural network
    examples = []
    for i in range(numIters):
        for e in range(numEps):
            examples += executeEpisode(game, nnet)          # collect examples from this game
        new_nnet = trainNNet(examples)
        frac_win = pit(new_nnet, nnet)                      # compare new net with previous net
        if frac_win > threshold:
            nnet = new_nnet                                 # replace with new net
    return nnet

def executeEpisode(game, nnet):
    examples = []
    s = game.startState()
    mcts = MCTS()                                           # initialise search tree

    while True:
        for _ in range(numMCTSSims):
            mcts.search(s, game, nnet)
        examples.append([s, mcts.pi(s), None])              # rewards can not be determined yet
        a = random.choice(len(mcts.pi(s)), p=mcts.pi(s))    # sample action from improved policy
        s = game.nextState(s,a)
        if game.gameEnded(s):
            examples = assignRewards(examples, game.gameReward(s))
            return examples
```

politer-selfplay.py hosted with ♡ by GitHub view raw

Experiments with Othello

We trained an agent for the game of Othello for a 6x6 board on a single GPU. Each iteration consisted of 100 episodes of self-play and each MCTS used 25 simulations. Note that this is orders of magnitude smaller than the computation used in the AlphaGo paper (25000 episodes per iteration, 1600 simulations per turn). The model took around 3 days (80 iterations) for training to saturate on an NVIDIA Tesla K80 GPU. We evaluated the model against random and greedy baselines, as well as a minimax agent and humans. It performed pretty well and even picked up some common strategies used by humans.


The agent (Black) learns to capture walls and corners in the early game


The agent (Black) learns to force passes in the late game

A Note on Details

This post provides an overview of the key ideas in the AlphaGo Zero paper and excludes finer details for the sake of clarity. The AlphaGo paper describes some additional details in their implementation. Some of them are:

- **History of State**: Since Go is not completely observable from the current state of the board, the neural network also takes as input the boards from the last 7 time steps. This is a feature of the game itself, and other games such as Chess and Othello would only require the current board as input.
- **Temperature**: The stochastic policy obtained after performing the MCTS uses exponentiated counts, i.e. $\vec{\pi}(s) = N(s,\cdot)^{1/\tau}/\sum_b(N(s,b)^{1/\tau})$, where τ is the temperature and controls the degree of exploration. AlphaGo Zero uses $\tau = 1$ (simply the normalised counts) for the first 30 moves of each game, and then sets it to an infinitesimal value (picking the move with the maximum counts).
- **Symmetry**: The Go board is invariant to rotation and reflection. When MCTS reaches a leaf node, the current neural network is called with a reflected or rotated version of the board to exploit this symmetry. In general, this can be extended to other games using symmetries that hold for the game.
- **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.
- **Compute Power**: Each neural network was trained using 64 GPUs and 19 CPUs. The compute power used for executing the self-play in unclear from the paper.
- **Neural Network Design**: The authors tried a variety of architectures including networks with and without residual networks, and with and without parameter sharing for the value and policy networks. Their best architecture used residual networks and shared the parameters for the value and policy networks.

This code presented in this tutorial provides a high-level overview of the algorithms involved. A complete game and framework independent implementation can be found in this GitHub repo. It contains an example implementation for the game of Othello in PyTorch, Keras and TensorFlow.

Feel free to leave questions/comments/suggestions below :).

Paulo Martel • 2 years ago

Dear Surag,
first of all thank you for the excellent article and code, really useful to understand the workings of AlpahZero. However, I'm confused about one thing: since there is no stochasticity in MCTS when pitting two networks againsts each other, shouldn't each game be exactly like any other? (or better, only two games depending on which net is playing first). Every time I run "pit.py" with two nets I get the same N games... but these games are all different, how is that happening ?...

33 ^ | ˅ 2 • Reply • Share ›

> **Surag Nair** Mod → Paulo Martel • 2 years ago • edited
>
> Thanks Paulo! The original paper mentions that Dirichlet noise is added to prior probabilities for additional exploration in the self-play stage, but at least from the paper it looks like the noise was not added in the evaluation stage. If that is the case, then there is no stochasticity when pitting the networks.
>
> In my implementation, I do not clear the MCTS trees after each game in the pit stage. This explains why you are seeing different games. This probably works for 6x6 Othello because we're using ~50 simulations per step, and so additional simulations may change the policy.
>
> ^ | ˅ • Reply • Share ›

> **Gergely Papp** → Paulo Martel • 2 years ago
>
> I think in the original algorithm it was achieved by adding dirichlet noise to the probabilities at the root (and it was not removed at self-play). On the other hand, I see no noise added in this code.
>
> ^ | ˅ • Reply • Share ›

Rolf Engstrand • 2 years ago

Very good article! One little criticism though: It states that one (single) board state is sufficient for Chess. But in a few minor cases, knowledge of earlier board states are needed for interpreting the rules. I am thinking of en passant (in which a pawn can be captured on a square where it currently is not) and castling (which can be performed only if none of the two pieces has moved before.

3 ^ | ˅ • Reply • Share ›

> **Sharan G** → Rolf Engstrand • 2 months ago • edited
>
> We simply allow/disallow those moves, as per the history. Let the network provide outputs for such moves, but those wont be considered during move selection if invalid.
>
> ^ | ˅ • Reply • Share ›

Fernando Mir • 4 months ago

"for a in range(game.getValidActions(s)):" seems like a bug. Why use range here? The value "a" is already the action index.

1 ^ | ˅ • Reply • Share ›

> **Surag Nair** Mod → Fernando Mir • 3 months ago
>
> Corrected, thanks!
>
> 1 ^ | ˅ • Reply • Share ›

Vaibhav Gupta • 7 months ago

This is the most beautifully written blog I found on this subject. Thanks so much

1 ^ | ⌄ • Reply • Share ›

Surag Nair Mod ➦ Vaibhav Gupta • 3 months ago

Thank you :)

^ | ⌄ • Reply • Share ›

unhandyandy • 2 years ago

I'm new to neural nets. My understanding is they require very precise specification of dimensions at all levels. But most games, including Chess, Go, and Othello, have different numbers of possible moves in different positions. So how do you get a NN to return a policy vector whose length depends on the input? Or does it just return a policy for all conceivable moves, which is then restricted to those actually playable?

1 ^ | ⌄ • Reply • Share ›

Surag Nair Mod ➦ unhandyandy • 2 years ago

We design a neural neural network for each game, that at the very least differs at the input and output dimensions :).

^ | ⌄ • Reply • Share ›

unhandyandy ➦ Surag Nair • 2 years ago • edited

But my point is that even for a specific game, the number of possible moves varies from position to position.

^ | ⌄ • Reply • Share ›

Willahelm Bhavna ➦ unhandyandy • 2 years ago

You are right the number of outputs of a neural network is fixed. Therefore when you have a policy network you have an output for all possible moves and simply discard the illegal ones. The other possibility is a value network which gives a score to an entire board position and then you iterate for all possible moves and choose the one that produces the position with the highest score. AlphaGo Zero uses a combination of both methods.

1 ^ | ⌄ • Reply • Share ›

unhandyandy ➦ Willahelm Bhavna • 2 years ago

So when training a net to play chess the policy vector must have thousands of entries - there are 896 conceivable rook moves alone - even more than in Go!

^ | ⌄ • Reply • Share ›

Willahelm Bhavna ➦ unhandyandy • 2 years ago

I checked the AlphaZero chess paper and it says the policy representation contains 4,672 possible moves. It encodes as a combination of starting square and move so most moves are just a subset of the possible moves for a queen.

https://arxiv.org/pdf/1712....

1 ^ | ⌄ • Reply • Share ›

unhandyandy ➦ Willahelm Bhavna • 2 years ago

Interesting - it seems extremely inefficient, remarkable it worked so well.

^ | ⌄ • Reply • Share ›

Totoka ➦ unhandyandy • 4 months ago

You can think of it as this: when a chess master looks at the board, they can immediately disregard 99% of the possible moves as "obviously wrong" (against the rules, supremely damaging, etc). Similarly, once AlphaZero developed the same kind of intuition, the simulations no longer have to visit all those thousands potential moves, only the promising few.

^ | ⌄ • Reply • Share ›

unhandyandy ➦ Totoka • 4 months ago

Well, I'm guessing that in the future much more efficient algorithms will be found.

^ | ˅ • Reply • Share ›

Christopher Jernigan • 2 years ago

Dumb question, but what is the benefit of having both a policy and value network? Can't you estimate the policy based off of the values (which should be closely proportional to the visit counts of each node)?

1 ^ | ˅ • Reply • Share ›

Surag Nair Mod ➦ Christopher Jernigan • 2 years ago • edited

That's a pretty good question, actually. My hunch is that the policy network allows you to prune certain actions faster when the action space is huge. If you look at the formula for the upper confidence bound, it includes a P(s,a) term which acts as a prior for the exploration component. This presumably cuts down the action space at each state significantly by ruling out poor actions.

2 ^ | ˅ • Reply • Share ›

Lucas Simon • 2 years ago • edited

Why do you take the update

on line 19 of the first code snippet?

1 ^ | ˅ • Reply • Share ›

tbischel ➦ Lucas Simon • 2 years ago

Q(s,a) is the average valuation (v) of all searches traversing node s via action a, and N(s,a) is the number of searches traversing node s via action a. This works out to an inline update of the average valuation. The paper actually tracks another parameter W(s,a) which is the sum of all valuations through s -> a... so that updating the average value is a slightly less expensive calculation.

1 ^ | ˅ • Reply • Share ›

Piotr • 2 years ago • edited

Hi!

Thank you guys for clear explanation and nice paper, (0.) have you published it somewhere? :) My friends and I try to reproduce Alpha(Go) Zero for Othello and Connect4 on our own to practice and I have couple of questions, maybe someone could help us:
1. How did you find NN architecture and hyper-parameters? Some kind of automated hyper-parameter tuning or you found it by hand? Especially number of episodes you train NN after each self-play, it seems really arbitrary and if chosen to low the NN would never train enough to win a tournament, but set it to high and it will overfit.
2. Did you try to use validation set and then stop training when val_loss doesn't improve anymore? We did it, but I don't know if it makes sens, for now it's disabled (for root causing problem from question 3.). Do you have an opinion on that one?
3. We have a problem with our implementation, it seems unstable in a sens that models trained more can (and very often do) loose when pitting with earlier ones. Have you encountered similar problems? NN seems to forget what it learned after more training and even beside it still improves and beats current best it can be worse then some models before.
We currently use this kind of NN:
conv 5x5x256/2 -> 3x residual block (conv 3x3x256) with bottleneck (128 channels) -> avgpool -> dense (512 neutrons) -> heads value and pi

I really appreciate any help and discussion!

1 ^ | ˅ 1 • Reply • Share ›

Li, Wenhan • 17 days ago

another question, i thought that random choice serves to pick out a random sample, why using it in this case to another question, i thought that random.choice serves to pick out a random sample, why using it in this case to determine action?

Reply • Share ›

Li, Wenhan • 17 days ago

thanks a lot for the explanation and code, but I am confused a little: 1. what if most games in generated in an iteration are identical? would that be very inefficient? 2.how exactly should the neural net be trained? how many epochs and batch size?

Reply • Share ›

Li, Wenhan • 17 days ago

Thanks a lot for the explanation and code, but I am a little confused: what if all games in an iteration are played in an exact same way?

Reply • Share ›

Pilou45 • 2 months ago

The best blog written on this subject so far thank you !
I was just wondering what kind of small changes would you make if the player can execute multiple actions (from 1 to 5 different actions) during one single turn ? (because in board game you take an action and then its the opponent turn) I guess there would be a change in the way of treating the returned values for example

Reply • Share ›

Amir Hefetz • 3 months ago

Hi, Thanks a lot for that post!

I have a question though:
The MCTS node Q score is basically an AVERAGE of all subtree nodes V score (probability of winning in this state)
This is in contrast to minimax algorithms that calculate min\max of subtrees.
in my personal project-i encountered cases where after move X, the player can immediately win, with V score of 99%, but in the alternative subtrees ,it loses, its like getting 1 chance to checkmate and win but all alternative scenarios are a loss,
this causes MCTS to rule out this move because of low score, although with correct play it wins right away in the next move.
In go it can occur the same-the next move can be capturing a group to win, but all other scenarios are a loss because it lives. after checking all subtrees, the average is very low and will suffer because of lower N visits-and therefore won't be selected?
What am i missing here?
Thanks!!

Reply • Share ›

> **Surag Nair** Mod ➜ Amir Hefetz • 3 months ago
>
> This should not happen. With sufficient sampling, the MCTS will eventually visit the most promising path, and the average should be sufficiently positive. Do have a closer look at the algorithm!
>
> Reply • Share ›
>
> > **Amir Hefetz** ➜ Surag Nair • 3 months ago • edited
> >
> > Hi @Surag Nair ,
> > I checked it really carefully, in my tic tac toe implementation - the next move can be a win, but ALL other optional subtrees lead to a lose\draw, the averaging between 20 subtrees that only 1 of them has a high V value, gets very low.
> >
> > I solved it eventually.
> > I saw in the alphazero documentation that they use domain knowledge of game termination. so i added a new rule in my MCTS that is using "domain knowledge" and help the model learn (and eventually solved it perfectly):
> > - If any of actions within MCTS search is found, that terminated the game with a win - pick it automatically regardless of the Q\V scores. this propogates only the high V value when a win is encountered, and stops searching within subtrees that directly "ignore" playing a winning move.

I hope this is clear.
Thanks!
^ | ˅ • Reply • Share ›

Surag Nair Mod ➦ Amir Hefetz • 9 days ago • edited
This bug may explain what you were seeing (if you were using `alpha-zero-general`).
^ | ˅ • Reply • Share ›

Yazeed Mohi Eldeen • 4 months ago
Thank you for posting this :)
Very good explanation, Though you could've made it a bit more beginner-friendly ;)
^ | ˅ • Reply • Share ›

Surag Nair Mod ➦ Yazeed Mohi Eldeen • 3 months ago • edited
I apologise for that, it is hard to get the balance right.
^ | ˅ • Reply • Share ›

Peter Geiger • 6 months ago
@Surag Nair -- Your codebase is by far the easiest to extend that I've played with. I am modeling Hive - a game with no board. The pieces have relationships to one another and I modeled this through an x,y,z vector system. -- There are 22 pieces x 3 = 66 doubles in the representation for the board state. I haven't played with NNs in a LONG time, so I'm catching back up. I was hoping you might have some suggestions on how to modify your original NN definition for a representation like this. Appreciate any advice.
^ | ˅ • Reply • Share ›

BlackRoll • 7 months ago
Dear Surag,
I am quite late to the party. Nevertheless, I hope you still get notified about this:
I am a student and trying to implement this on my own - your post has helped me quite a lot, thanks for that.
Here is my question:
You avoided the canonical form in this post, but this is the last part I am struggling with.

I want to use this for Connect4. To make it time-efficient, I implemented my Connect4 as Bitboard.
My Connect4Board Object has an array of 2 boards, which each store the pieces of one player.

I am unsure about the input dim for the NN and how to tell the NN to differentiate between the two players, since I see so many different approaches in doing so.

The approaches I consider but cannot decide on and would really appreciate guidance:
1) I just input the dimension 7x6 (width x height) and each turn, only pass the pieces of 1 player to the neuronal net
2) I increase the input dimension to 7x6x2, so the net has always information on all pieces (first 42 (7x6) are player one pieces, last 42 are player two pieces)
3) I choose 7x6 + 2, 2 signaling which player's turn it is currently.

The second one seems the one I would take, since signaling which player's turn it is is not really necessary in my opinion. Every state is unique and automatically tells u whose players turn it is by just counting the pieces.

I really hope this reaches you and thank you so much for your time.
^ | ˅ • Reply • Share ›

James • 8 months ago
Hi,

I'm implementing a custom game where is similar to chess, but you attack the piece one square over, instead of moving on top of the enemy piece.
I'm running into a MCTS max recursion, i believe due to not being able to find a leaf node, maybe because every state with the current board is at one point played already, and the game hasn't ended yet. So that s is never not-in self.Ps ...

Is there a way to either set max-tree depth? if so would we just do another self.Ps[s], v = self.nnet.predict(canonicalBoard) and return -v among condition?

or can we set game end after max-tree depth?

How would chess handle a similar situation where each side only has a king left and the game can never end due to running forever?

ps. I assume we can't just limit the game moves and call it game end after 1000 moves, since the board doesn't represent this in a np.array form?

Would this still train the agent in a meaningful way?

• Reply • Share ›

Surag Nair Mod ➦ James • 8 months ago

Hi James, could you please post this as a GitHub issue. Will look into it there. Also it would be great if you could explain your chess variant in a bit more detail in the post.

• Reply • Share ›

James ➦ Surag Nair • 8 months ago

will do, thanks :)

• Reply • Share ›

James • 10 months ago

Hi, Would having a non-symmetry board mess with the training?

• Reply • Share ›

Surag Nair Mod ➦ James • 9 months ago

No, you shouldn't have any difficulties.

• Reply • Share ›

James ➦ Surag Nair • 7 months ago

update: just to prove it still works on non-sym boards, disabled random locations on the board, and it still wins most of the games.

• Reply • Share ›

Jan Konopka • a year ago

I tried to recreate this from scratch, but sadly the random initial neural network assigns probability 0 to some moves and thus never explores them. (Unless we get lucky with a backpropagation which rarely happens.) So my net is simply beaten by choosing the move it never considered.

• Reply • Share ›

Jan Konopka ➦ Jan Konopka • a year ago

I looked at the AlphaZero paper and it said

> In AlphaZero we reuse the same hyper-parameters for all games without game-specific tuning. The sole exception is the noise that is added to the prior policy to ensure exploration (29); this is scaled in proportion to the typical number of legal moves for that game type.

So I think I just have to add a constant 5% to the probability distribution function or so

• Reply • Share ›

Jan Konopka ➦ Jan Konopka • a year ago

I found a different version of the paper with more info:

> without game-specific tuning. The only exceptions are the exploration noise and the learning rate schedule

> Dirichlet noise Dir(α) was added to the prior probabilities in the root node; this was scaled in

> inverse proportion to the approximate number of legal moves in a typical position, to a value of α = {0.3, 0.15, 0.03} for chess, shogi and Go respectively.

^ | ˅ • Reply • Share ›

Addy R • a year ago

Hi Surag, Thanks for the tutorial! I have one lingering question/confusion. During the policy iteration, it is stated that the neural net is used to predict v(s) at a search leaf node instead of doing a traditional MCTS rollout, does this mean you restrict MCTS to only have selection phase, expansion phase and backup phase? My question is how does MCTS reach the end game with out doing rollout?

^ | ˅ • Reply • Share ›

Surag Nair Mod ➦ Addy R • a year ago

Hi Addy- yes that is correct. If the search leaf node is indeed an end game node, it is assigned the true value. In other cases, the neural network is used to predict. So in that sense, MCTS is indeed restricted to the three phases you described.

^ | ˅ • Reply • Share ›

Addy R ➦ Surag Nair • a year ago • edited

Thanks Surag!

^ | ˅ • Reply • Share ›

Fu Jun • 2 years ago

Thanks Surag, great work!
Is the MCTS still needed for playing (testing)? If the P and V network get well trained at last, a further MCTS could help very less, right?
Hope reply, thanks again.

^ | ˅ • Reply • Share ›

Surag Nair Mod ➦ Fu Jun • a year ago

I agree with your point, but it seems like using MCTS during testing will not adversely impact performance. It would be great if you could try out both the versions and report the difference!

^ | ˅ • Reply • Share ›

unhandyandy • 2 years ago

From which paper did you get your formula for U(s,a)? I couldn't find it in the two papers linked in the first paragraph.

^ | ˅ • Reply • Share ›

Load more comments