Learning to Stop: Dynamic Simulation Monte-Carlo Tree Search

Li-Cheng Lan[1], Meng-Yu Tsai[2], Ti-Rong Wu[2], I-Chen Wu[2,3], Cho-Jui Hsieh[1]

[1] Department of Computer Science, UCLA, Los Angeles, USA

[2] Department of Computer Science, National Chiao-Tung University Taiwan

[3] Research Center for IT Innovation, Academia Sinica, Taiwan

lclan@cs.ucla.edu, {adam0923686343,kds285}@gmail.com, icwu@csie.nctu.edu.tw, chohsieh@cs.ucla.edu

Abstract

Monte Carlo tree search (MCTS) has achieved state-of-the-art results in many domains such as Go and Atari games when combining with deep neural networks (DNNs). When more simulations are executed, MCTS can achieve higher performance but also requires enormous amounts of CPU and GPU resources. However, not all states require a long searching time to identify the best action that the agent can find. For example, in 19x19 Go and NoGo, we found that for more than half of the states, the best action predicted by DNN remains unchanged even after searching 2 minutes. This implies that a significant amount of resources can be saved if we are able to stop the searching earlier when we are confident with the current searching result. In this paper, we propose to achieve this goal by predicting the uncertainty of the current searching status and use the result to decide whether we should stop searching. With our algorithm, called Dynamic Simulation MCTS (DS-MCTS), we can speed up a NoGo agent trained by AlphaZero 2.5 times faster while maintaining a similar winning rate. Also, under the same average simulation count, our method can achieve 61% winning rate against the original program.

Introduction

Many of the state-of-the-art AI programs in games such as Go, Atari, shogi, chess, and NoGo (Silver et al. 2016, 2017, 2018; Schrittwieser et al. 2019; Lan et al. 2019) use a combination of deep neural networks (DNNs) (LeCun, Bengio, and Hinton 2015) and Monte Carlo tree search (MCTS) (Browne et al. 2012; Kocsis and Szepesvári 2006). Starting from 2015, DNNs have been used to extract high-level information from a given state and provide high-quality policy and state value for MCTS agents. Even the raw DNN, without any lookahead, can achieve a strong professional level (Silver et al. 2017). The well-trained networks can then used by policy value MCTS (PV-MCTS). PV-MCTS uses the policy as a heuristic to narrow down each node's search space in MCTS, while the state value is used to make each evaluation more accurate. After using PV-MCTS, AlphaGo Zero can achieve 5,185 Elo, which corresponds to more than 99% winning rate against the raw DNN.

Although achieving remarkable results, PV-MCTS requires enormous amounts of GPU resources in both training
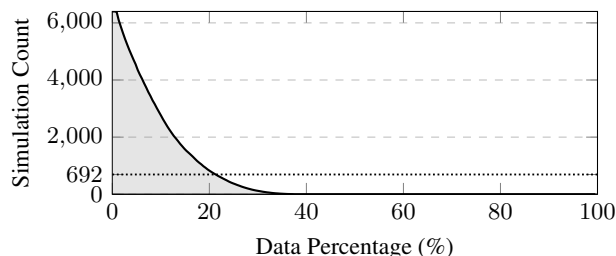
Figure 1: The minimum simulation count required to find the best move under a total of 6400 simulations in the 19x19 Go.

and testing. For example, AlphaGo used more than a thousand CPUs and 250 GPUs for playing games against Lee Sedol, not to mention the millions of self-play required during training time for AlphaZero. Many methods have been proposed to reduce the computation of PV-MCTS (Lan et al. 2019; Gao, Müller, and Hayward 2018) or decrease the inference time by parallelization (Liu et al. 2020). However, seldom of them consider terminating the search after the stable best action is found. Fig. 1 shows the minimum simulation count at which the best action remains the best choice using PV-MCTS with a total of 6400 simulations for each position for 19x19 Go. The results show that more than 62% of the states only require 1 MCTS simulation, and the average minimum simulation count is only 692. Moreover, KataGo (Wu 2019) speeds up AlphaZero training by randomly reducing the simulations on nearly 75% of states during self-play. We expect more acceleration if we reduce the simulation of each state carefully.

On the other hand, understanding the certainty of our prediction is critical for both humans and models in many domains (Amodei et al. 2016; Gal and Ghahramani 2016; Guo et al. 2017; Lakshminarayanan, Pritzel, and Blundell 2017; Liang, Li, and Srikant 2017). One of the reasons is that we can gather more information or even ask experts to avoid being wrong on highly uncertain data. The uncertainty estimation is even more important in reinforcement learning (Clements et al. 2019). Some use it for better exploration (Pathak et al. 2017; Burda et al. 2018), others use it for avoiding risky or unknown actions (García and Fernández 2015). However, to the best of our knowledge, non of previous works
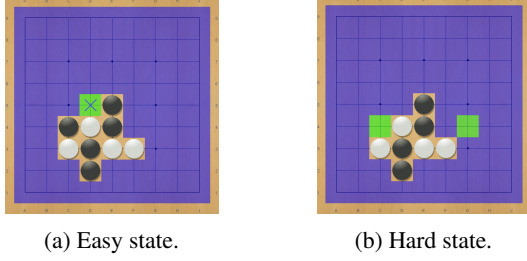
(a) Easy state.   (b) Hard state.

Figure 2: The illustration of the state uncertainty in the 9x9 Go. The color shows the policy output by the neural network. Purple through bright green ranges from the lower to the higher probability.

can be implemented on the tree searching directly since the uncertainty of tree searching is very different from modern neural networks. For example, given a state, we will become certain if we found a winning strategy after searching several simulations. Hence, we can predict the uncertainty based on different search stage.

In this paper, we first define the certainty of an MCTS searching status as the confidence of the current best action remains unchanged after reaching the given maximum simulations. Take Fig. 2a, an easy state of 9x9 Go, as an example. Even beginner Go players know that marked point is the only move to consider; DNN policies can also learn this. Therefore, we should be certain about our decision after evaluating the root state and stop the search immediately. On the contrary, Fig. 2b shows a rigid state where even an agent trained by the AlphaZero algorithm is uncertain about which action is better after a small amount of searching.

We proposed three ways to predict the uncertainty. The first method uses the policy of the original neural network and the confidence calibration technique (Guo et al. 2017). The second and the third methods train an auxiliary network to estimate the confidence. The first and second methods only use the given state's information, while the third method also includes information about the current search status. The results show that all the methods can achieve more than 92% accuracy.

Finally, we proposed *Dynamic Simulation MCTS* (DS-MCTS), which dynamically determines whether to continue the MCTS search according to the current search status's uncertainty. The experiment results on a NoGo AlphaZero agent show that the proposed algorithm leads to a speedup of 2.5 while maintaining a similar winning rate. The result is significant because playing at the level of AlphaZero; even a single lousy move will lead to losing immediately. Moreover, by increasing the maximum simulations, we achieve 61% win rate against the original PV-MCTS using the same number of average simulations. We also conduct experiments on 9x9 Go. Our method speeds up 1.66x even if we do not fine-tune our hyper-parameters.

Background

Policy Value Monte Carlo Tree Search (PV-MCTS), used in the AlphaGo Zero program (Silver et al. 2017), is similar

to MCTS (Browne et al. 2012; Kocsis and Szepesvári 2006) in that it is a best-first search algorithm that uses Monte Carlo simulation to evaluate state values. It incorporates into MCTS a two-headed neural network, which we refer to as a policy value neural network (PV-NN). Given a state s, PV-NN outputs a policy distribution $p(*|s)$ that represents the probability $p(a|s)$ of each action a, and a scalar $v(s)$, which estimates the state value.

PV-MCTS contains four phases for each simulation: (1) *Selection*: starting from the root state s_0, traverse downwards, choosing the best move until reaching a leaf state s_T. At every time step t, AlphaZero uses the PUCT algorithm (Rosin 2011) as the tree search policy to select the action for state s_t:

$$a_t = \arg\max_a \left\{ Q(s_t, a) + c_{PUCT} \times p(a|s_t) \times \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)} \right\},$$
(1)

where $Q(s, a)$ and $N(s, a)$ represents the estimated value of the search tree and visit count of action a for state s, c_{PUCT} is a constant value for exploration, and $P(s, a)$ is the prior probability provided by PV-NN. (2) *Expansion*: Expand the search tree from the selected leaf s_T. (3) *Evaluation*: The neural network evaluates the selected leaf state s_T using the PV-NN (4) *Update*: The evaluation result of PV-NN is then used to update the prior probability $P(s_T, *)$ of s_T and the action values $Q(s_t, a_t)$ on the selected path $(s_0, s_1, ..., s_T)$. After maximum simulations have been executed, PV-MCTS returns a policy $\pi(s_0, a) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$ of root state s_0, where τ is a temperature parameter that can be adjusted by users. A typical choice is to set $\tau \to 0^+$, which means the move with the maximum visit count is always selected.

Method

This section will first discuss the proposed uncertainty estimation algorithm for MCTS and then show how to use it to conduct a Dynamic Simulation MCTS (DS-MCTS).

Uncertainty estimation

Evaluating the uncertainties is challenging since the "ground truth" of uncertainty estimates are usually not available. Therefore, we try to define our "uncertainty" of current search status based on the following observations:

- Although there is no "ground truth" for playing games, PV-MCTS will converge to a decent result after hundreds of simulations have been executed. On the other hand, it is much easier to predict the uncertainty when a large enough maximum simulation counts N_{max} exists, since we can use the policy and action values after N_{max} simulations to establish our "ground truth".

- There could be more than one best action (e.g., two possible winning moves), and it is sufficient to find any of these actions. The state should be considered certain for the agent if it is sure that the action it found is nearly the best. The near-best actions can be defined as actions with similar highest action value after N_{max} simulations.

- Even if the agent can predict the best policy at the beginning of the search, a state should be considered uncertain if another sub-optimal move is considered the best action during the search since luck plays a factor in the beginning stages of a search. Action can be considered sub-optimal if its action value is lower than the best action after N_{max} simulations.

Based on the above observations, we propose the following definition of our uncertainty.

Let $\pi(a|s, n), Q(s, a, n)$ be the policy and the action value provided by the PV-MCTS agent after n simulations on the state s. We use $\tau \to 0^+$ to select π (only select the best action). If an action a' hasn't been visited, the action value $Q(s, a', n)$ will be -1, which means losing. Given a maximum simulation N_{max}, we use the action value $Q(s, a, N_{max})$ to approximate the true action value. Therefore, we have the approximate reward R of current policy as follows:

$$R_{N_{max}}(s, n) = \sum_a (\pi(a|s, n) * Q(s, a, N_{max})). \quad (2)$$

We define an MCTS tree status after n simulations is uncertain $U(s, n) = 1$ if:

$$\exists n' \geq n(R_{N_{max}}(s, N_{max}) - R_{N_{max}}(s, n') \geq \epsilon), \quad (3)$$

where ϵ is a small constant that we can tolerate. We are certain about the tree status $U(s, n) = 0$, when every $n' \geq n$ select one of the near-best actions. This definition satisfies our three observations. First, the uncertainty is based on the given N_{max}. Second, it is sufficient to find any of the near-best action within the ϵ threshold. Finally, the search status after nth simulations will be considered uncertain ($U(s, n) = 1$) if any sub-optimal action be chosen at any $n' \geq n$.

For convenience sake, we define the minimum simulation count $M(s)$ of a state s as: after $M(s)$ simulations, the searching status becomes *certain*. In another words, $U(s, n) = 1$ if $n < M(s)$, otherwise, $U(s, n) = 0$ if $M(s) \leq n \leq N_{max}$. Fig. 1 is drawn by $M(s)$. The state shown in Fig. 2a has minimum simulation count $M(s_{easy}) = 1$ and the state of Fig. 2b has a minimum simulation count $M(s_{hard})$ that is close to N_{max}.

Our main goal is to stop the search as soon as $M(s)$ simulations have been executed. However, predicting $M(s)$ is hard for neural networks since this requires the models to predict the uncertainty of the future. Hence, we let our models predict the probability of being uncertain $U(s, n) = 1$ after n simulation have been executed on state s by providing a value u. If u is large, it is more likely $U(s, n) = 1$. With the value u, we can stop the searching if u is smaller than a tuned threshold.

Using the original PV-NN Although our ultimate goal is predicting the uncertainty based on both the given state and the current tree status, it is still important to predict without any tree status at the beginning of the search since more than half of the states can be certain about its choice once the root state is evaluated. Part of the reason is the high quality of the original PV-NN, and its policy will directly influence the search. According to the PUCT algorithm, if an action's

prior probability $p(a|s)$ is large, the selection algorithm of PUCT (Equ. 1) will encourage the agent to explore the action, and as a result, it will have a higher chance to end up with the most simulation count and be selected as the best action. Therefore, in this section, we use the original PV-NN to predict the uncertainty at the beginning of the search.

One straightforward way to use the original PV-NN is by using the probability of best action. If the probability of the best action is 0.99, then it is almost certain that it is the best action the agent will choose after the search. However, according to Guo et al. (2017), the softmax prediction of, say, a ResNet (He et al. 2016) may not match its true confidence. It needs to be calibrated. For instance, Guo et al. (2017) showed that Platt Scaling (Platt et al. 1999) (temperature scaling) is a simple but surprisingly effective way to calibrate the prediction. It searches for a proper temperature τ to adjust the prediction π. For untrustworthy models it will set $\tau \to \infty$, while for trustworthy models $\tau \to 0^+$. Based on this idea, we approximate the value u with the wrong prediction probability, shown as follows:

$$u = 1 - \max_a \frac{p(a|s)^{1/\tau}}{\sum_{a'} (p(a'|s)^{1/\tau})}. \quad (4)$$

In the experiments, we show that this approach can moderately reduce the average search count, but there is still a large room for improvement. In addition, although using the original PV-NN can provide a usable u, it does not match our uncertainty definition. For example, it will be uncertain if the final best action is different from its prediction, even if its prediction is a near-best action. Also, the original PV-NN does not make use of the information of the tree states. We thus propose estimating uncertainty using an auxiliary neural network in the following section.

Using auxiliary neural networks In this subsection, we propose two kinds of auxiliary neural networks to provide the value u, which is both trained by predicting our uncertainty $U(s, n)$ directly. The first auxiliary neural network takes only the state as input and is used at the beginning of the tree search. We call this the state uncertainty network (State-UN). The second auxiliary neural network takes both the state and the current MCTS tree status information as input, which is called the MCTS uncertainty network (MCTS-UN).

State uncertainty network (State-UN) We generate training data by the following steps. First, to increase the diversity, we generate self-play records using a small simulation count with Dirichlet noise. Second, we re-evaluate the states of those self-play records with a higher simulation count N_{max} without any noise to get the uncertainty $U(s, n)$ of each simulation.

The State-UN uses the same state input features as original PV-NN. The outputs of state Uncertainty Network are StateUN(s) = $(u, \mathbf{p}, v)$, which are used to predict: 1) the beginning uncertainty $U(s, 1)$ defined in Equation 3, 2) MCTS policy $\pi(*|s, N_{max})$ after N_{max} simulations, and 3) the final result $z \in \{1, -1\}$ of the game record. We then get the following loss function

$$l = (u - U(s, 1))^2 + c_1((v-z)^2 - \boldsymbol{\pi}^T log(\mathbf{p})) + c_2 \|\theta\|^2, \quad (5)$$

where c_1, c_2 are weighting constants. The first term of the loss function is to make u close to $U(s, 1)$ using mean squared error. The second term of loss function, controlled by c_1, is the same as AlphaGo Zero (Silver et al. 2017) training its policy and value. Both $\boldsymbol{\pi}$ and $\mathbf{p}$ are vectors and $\boldsymbol{\pi}^T log(\mathbf{p})$ is their cross entropy. We add this term in order to help the network extract more information from state features. The experiment results shows that with a small $c_1 = 0.1$ can truly help the performance of u and don't really need to tune c_1. The last term is for regularization, where θ is the parameters of our model.

MCTS Uncertainty Network (MCTS-UN) As more simulations have been executed, more states will switch from uncertain to certain. It is worthy to evaluate the uncertainty according to the latest search tree status. Hence, we propose MCTS Uncertainty Network (MCTS-UN) method to identify those states using both the state and current MCTS information at any time during the search.

For efficiency, given an N_{max}, we hope to train our MCTS-UN on a single simulation n and be used on other simulations n'. In this way, we can check the uncertainty using only one model (MCTS-UN) at any time of the search and stop the search as soon as possible. It is critical to select a proper n to ensure the model can be generalized to different simulations. We select the n that minimizes the average MCTS count with only one checkpoint for identifying the uncertain states during the search. We also assume that the accuracy of negative samples ($U(s, n) = 0$) will have a limitation α when we make the accuracy of positive samples $U(s, n) = 1$ close to 100%. This can be done by using a very small threshold thr and only predict negative when the output u is smaller than the small threshold. With this predictor, the minimum of our average simulation count if we perform the uncertainty at n will be:

$$f(\alpha, n) = (n \times \alpha + N_{max} \times (1 - \alpha))p(M(s) < n) \\ + N_{max}p(M(s) \geq n) \quad (6)$$

where $p(M(s) < n)$ is the probability of the state s in training data after being pruned by State-DN that its $M(s) < n$. For example, if our predictor is perfect, $\alpha = 1$, all the states with $M(s) \leq n$ will stop at n simulations, while others will continue to finish N_{max} simulations.

Lemma 1 *The problem* $\arg\min_n f(\alpha, n)$ *have the same solution for any* $\alpha \in [0, 1]$.

We provide the detailed proof in the appendix. According to Lemma 1, we can safely select the cut point n for MCTS-DN to decide whether the agent should keep searching or not by assuming it is a perfect predictor ($\alpha = 1$), hence we have:

$$n = \arg\min_{n'}(f(1, n')), \quad (7)$$

which can be solved exactly in a linear scan.

Since training MCTS-UN needs MCTS features $T(s, n)$ of a state s (root state of the search tree) after n simulation ($n \leq N_{max}$) as input, we collect extra data while generating the training records of State-UN. The followings are the

extra data we collected. More data can be stored for more complicated MCTS features $T(s, n)$.

For each state s_0 in training record, we collect a pair (a_i, v_i) from each simulation, where a_i is the first action on the selection path in the ith simulation, and v_i is the evaluation result from the view of root state. For example, assume the root state s_0 is black's turn. If the selected leaf state s_T is also black's turn, then $v_i = v(s_T)$; otherwise if s_T is white's turn, $v_i = -v(s_T)$. Hence, for each game state, we end up collecting a sequence of pairs $(a_2, v_2), (a_3, v_3), ..., (a_{N_{max}}, v_{N_{max}})$. Note that the first simulation evaluates the root state and does not select any action, hence there is no (a_1, v_1). We also prune some of the training data that are certain in the beginning $U(s, 1) = 0$ and can be identified by State-UN easily. This helps us solve the data imbalance issue for training MCTS-UN.

With the extra data, we can generate MCTS features $T(s, n)$ of a state s (root state of the search tree) after any n simulation ($n \leq N_{max}$), which is described as followings:

MCTS features $T(s, n)$ is a tensor with dimensions of $channel\_num \times board\_size \times board\_size$, where our $channel\_num = 7$. Each channel contains certain types of information about each action. Take Go as an example: channel 0 ($T(s, n)[0]$) stores the prior probability provided by the original PV-NN. Hence, $T(s, n)[0][1][2]$ stores the prior probability of playing at position $(1, 2)$. Channel 1 to 3 store the information of the first layer of the search tree, including the policy ($N(s, a)/N(s)$) with $\tau = 1$, the action value ($Q(a|s)$), and the standard deviation of action value. If an action a' haven't been visited after n simulations, then action value $Q(a|s) = -1$ and the standard deviation equals to 1. Channel 4 to 6 generated by the latest half ($n/2$ to n) of the simulations to provide MCTS-UN the temporal information about the search, which also includes the policy, the action value, and the standard deviation of action value. For instance, the latest half policy can be calculated by $\pi(*|s, n) \times 2 - \pi(*|s, (n/2))$. The temporal information can show the trend of the search. For example, if the current best action is not selected in the last half of simulations, this may indicate that the MCTS is trying to switch to an alternative solution. Note that our MCTS features $T(s, n)$ do not have any information of n, since we hope to train our MCTS-UN on a single n and use it on any others. If there are information of n in $T(s, n)$, the features $T(s, n')$ will be out of the training distribution. For the same reason, $T(s, n)$ also does not have the information of N_{max} since we also want to scale up to a larger N_{max}.

Finally, MCTS-UN$(s, T(s, n)) = (u, \mathbf{p}, v)$ is trained as follows:

$$l = (u - U(s, n))^2 + c_1((v - z)^2 - \boldsymbol{\pi}^T log(\mathbf{p})) + c_2 \|\theta\|^2, \quad (8)$$

Same as State-UN, we also train it to learn the policy and value so that more information can be extracted. During training, we randomly dropout MCTS features or state features to ensure that MCTS-UN will not overly rely on any one of the features.

Dynamic Simulation MCTS

Although our predictors' accuracy can achieve more than 90% accuracy, it is still non-trivial to use in practice since every wrong prediction can lead to losing directly. Moreover, due to the game's high complexity, it is possible to encounter many unseen data. Therefore, we need to be as careful as possible by making the accuracy of positive samples ($U(s, n) = 1$) close to 100%. Base on this idea, we propose Dynamic Simulation MCTS, which only stops the search when it is very certain. Otherwise, we will continue the search and recheck it at the next checking point.

Algorithm 1: Dynamic Simulation MCTS

Input: State s, Maximum simulation N_{max}, Check points c, Thresholds thr
Output: policy π
1 **if** state_UN $(s) < thr[0]$ **then**
2 $\quad$ **return** $p(*|s)$
3 $i \leftarrow 1$
4 **for** $n = 1 : N_{max}$ **do**
5 $\quad$ **if** $i <$ len(c) *and* $n = c[i]$ **then**
6 $\quad\quad$ **if** MCTS_UN $(s, \text{T}(s,n)) < thr[i]$ **then**
7 $\quad\quad\quad$ **return** $\pi(*|s, n)$
8 $\quad\quad$ $i \leftarrow i + 1$
9 $\quad$ MCTS_simulation $()$
10 **end**
11 **return** $\pi(*|s, N_{max})$

Dynamic Simulation MCTS is shown in Algorithm 1. The input includes the given state s, maximum simulation count N_{max}, a list of checkpoints c, and a list of thresholds thr. Checkpoints list c is sorted and indicates when we should check our uncertainty. For example, the first checkpoint $c[0] = 0$ is at the beginning of the search. The threshold list thr is the threshold for each checkpoint.

At the beginning of the search, we obtain u by the original PV-NN or the State-UN at line 1. If the output u is $< thr[0]$ (line 1), we will return the policy provided by PV-NN (line 2). Otherwise, the search starts (line 4). During the search, when the current simulation count n reaches a checkpoint $c[i]$ (line 5), we then evaluate the current uncertainty by using MCTS-UN$(s, T(s, c[i]))$. If the output u of MCTS-UN is smaller than $thr[i]$ (line 6), then the search will stop (line 7). Otherwise, the search continues (line 9) and wait for the next checkpoint reaches (line 8).

The number of checkpoints should not be too many since there is still cost for forwarding MCTS-UN. We normally selects the second checkpoint same as the training data $c[1] = n$, where n is selected by Equ. 7. For other checkpoints, we use $c[i] = 2 \times c[i - 1]$. Normally, when more simulations are executed, MCTS-UN's prediction will be more accurate. Therefore, we can use a larger threshold for a larger checkpoint, as long as most ($> 96\%$) of the states that need to keep searching will be identified. This can be verified by evaluating the threshold on the self-play record first.

Related Work

Resource allocation (Hyatt 1984; Šolak and Vučković 2009; Donninger 1994; Markovitch and Sella 1996) has a long history in game playing since the total thinking time is normally fixed (sudden death). However, it is not intensively studied in MCTS (Huang, Coulom, and Lin 2010; Baier and Winands 2015), which is very different from the alpha-beta search because MCTS is an any-time algorithm. PV-MCTS trained by AlphaZero is even more different from traditional searching since the action provided by PV-NN is already decent, and our model needs to evaluate the probability of other actions being better.

Resource allocation schemes can be divided into three categories: static, semi-static, and dynamic strategies based on decision time (Markovitch and Sella 1996). Static strategies decide at the beginning of the game, and semi-static strategies decide before each search is started. These strategies usually focus on reserving time for the critical game stages. One of the semi-static strategies proposed by Kocsis, Uiterwijk, and van den Herik (2000) uses a neural network to predict the search depth of the alpha-beta search. The search depth will follow the network's prediction. When the time runs out, the search depth will always be 1. Unlike state-UN, it is trained on a fixed total thinking time; hence we need to retrain the model when the total thinking time is changed. Also, since it is semi-static strategies, once the model predicts a search depth, it cannot change it according to the current searching result.

Our method belongs to dynamic strategies that allocate resources during the search. One of the most popular dynamic strategy "STOP" (Baier and Winands 2015) terminates the search when the best move will not change even if the remaining simulations all select the current second-best move. Although "STOP" guarantees the same best move, it can only terminate the search after $N_{max}/2$ of simulations. Prob-Cut (Buro 1995) can also be considered as dynamic strategies on a higher level. It speeds up alpha-beta search by pruning the search tree according to shallow subtree searching results. However, there is no obvious way to use it on MCTS, since MCTS has already use UCT (Kocsis and Szepesvári 2006) or PUCT (Rosin 2011) algorithm to handle subtrees that are not promising.

Beyond tree searching, many methods have been proposed to quantify the uncertainty. Some methods utilize the original network's input and output (Guo et al. 2017; Hendrycks and Gimpel 2016; Liang, Li, and Srikant 2017), some methods ensemble randomized models (Gal and Ghahramani 2016; Lakshminarayanan, Pritzel, and Blundell 2017), some methods query model's understanding of the environment (Pathak et al. 2017; Burda et al. 2018), and some use Bayesian methods (Guo et al. 2017; Azizzadenesheli, Brunskill, and Anandkumar 2018). However, non of the above methods can utilize the information provided by the current tree search, since they can only provide uncertainty quantification for a static model, while tree search is a dynamic process.
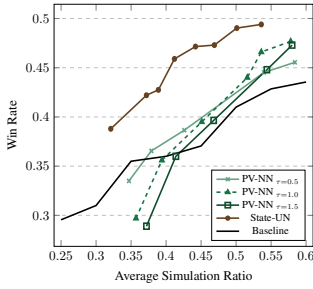
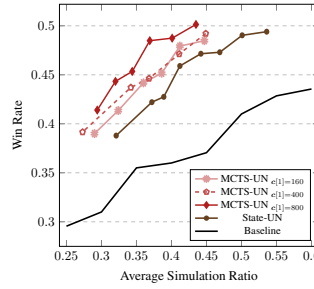Figure 3: The results of using the PV-NN and State-UN.

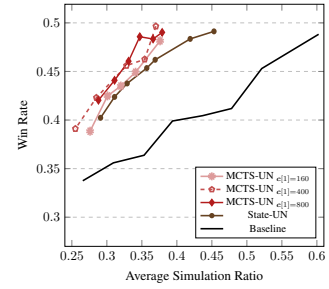Figure 4: The MCTS-UN results using different check point $c[1]$.

Figure 5: Applying "STOP" algorithm to MCTS-UN, State-UN, and Baseline.

Experiment

In this section, we empirically investigate our method mainly on 9x9 NoGo and some on 9x9 Go.

Experiment setting for NoGo   NoGo is one of the tournament items in the Computer Olympiad (Müller 2015). We choose NoGo in our experiments since the searching space is relatively low compared with 19x19 Go while maintaining many similar characteristics (Chou, Teytaud, and Yen 2011).

The NoGo agent we used has a 98% win rate against Ha-HaNoGo (Lan 2016), which won the champion on the 2016 TAAI competition. The PV-NN is a ResNet (He et al. 2016) with ten blocks and 128 filters. State-UN and MCTS-UN have ten blocks and 196 filters and are trained by 20,000 self-play games with $N_{max} = 1600$. MCTS-UN is trained by $n = 160$, where n is acquired by Equation 7. Each experiment played at least 2,048 games; therefore, the standard deviation was less than 1.1%. The default opponent is PV-MCTS with $opp\_sim = 1600$ simulations on each move. We do not other programs because their are much weaker ($< 10\%$ win rate) than our agent that is trained by AlphaZero. The default N_{max} for DS-MCTS method is also 1600.

Performance of original PV-NN and State-UN   First, we investigated the performance of DS-MCTS that only checks at the beginning of the search using PV-NN or State-UN. The results are shown in Fig. 3. We use average simulation ratios as the x-axis to show the acceleration, namely, if the average simulation is $E[n]$, then the average simulation ratio is $E[n]/opp\_sim$. The baseline (black line in Fig. 3) is drawn by PV-MCTS with different simulations. We have also tested stopping the searching randomly (Wu 2019). The results show that it is much weaker than our baseline.

We experiment using Equ. 4 with three different temperatures ($\tau = \{0.5, 1, 1.5\}$). Since using different thresholds $thr[0]$ will result in different average simulations, we properly select the set of thresholds (see appendix) for each temperature so that they can be compared under similar average simulations. The result is shown as the green lines in Fig. 3, where each data point is generated by a threshold $thr[0]$ in the set. The results show that PV-NN can achieve more than 45% win rate against the original PV-MCTS with only using less than 60% of time. Temperature $\tau = 1$ even achieves a 48.6% win rate with an average of 925 simulations, where the original PV-MCTS (black) can only achieve about 42% win

rate using the same simulations. However, as the threshold $thr[0]$ increase (average simulation decrease), the win rates drop quickly. It even performs worse than the baseline when the average simulation ratio is smaller than 0.4. This implies that some critic states that require searching are misclassified, and the game will be unsavable even if searching longer on other states.

Same as original PV-NN, we also use a set of $thr[0] = 0.05, 0.1, 0.2, ..., 0.7$ to show how the win rate decrease when more states play without search according to the State-UN. The brown line in Fig. 3 shows that the uncertainty prediction u provided by State-UN is much better than using the original PV-NN. Especially when using less than 40% of the resource, State-UN can still have a 43% win rate. This also indicates that our definition of uncertainty $U(s, n)$ for MCTS tree search serves properly on training State-UN. Since $thr[0] = 0.1$ achieve a 49% win rate with only 801 average simulations, we will use it for following experiments.

Performance of MCTS-UN   Although our MCTS-UN is only trained with tree information $T(s, 160)$ at 160 simulations, we want to show that it can be used on other simulations such as 400 and 800. Hence, in these experiments, we add only a checkpoint on different simulations ($c[1] = 160, 400, 800$) of the searching and use a set of thresholds ($thr[1]$) to show the trade-off. The State-UN is used in the beginning with a threshold $thr[0] = 0.1$. Fig. 4 shows that adding a single checkpoint $c[1]$ during the search can achieve a higher performance than only using State-UN, even if the checkpoint is different from the simulations MCTS-UN is trained. For example, with $c[1] = 160, thr[1] = 0.025$ we have 49.2% win rate with 715 average simulations and with setting of $c[1] = 800, thr[1] = 0.2$ we have 48.5% win rate with 591 average simulations. Note that $c = 800$ has a higher win rate when having lower average simulations. This shows that even if MCTS-UN does not check and stop the searching until 800 simulations, the quality of MCTS-UN with the tree information $T(s, 800)$ allows us to stop the searching of more states (using a higher $thr[1]$) without dropping the win rate. Hence, using only one checkpoint $n = 160$, which was selected by Equ. 7, is an efficient way to train MCTS-UN.

Combining with the STOP strategy   Next, we apply the "STOP" algorithm (Baier and Winands 2015), as mention in the related work section, to both our methods and baseline

Table 1: The results of testing on larger simulations with $\boldsymbol{thr}[0] = 0.1$

	$\boldsymbol{c}[i]$	–	160	400	800	1600	800/1600	400/800/1600	160/400/800/1600
	$\boldsymbol{thr}[i]$	–	.025	.025	.05	.05	.05/.05	.025/.05/.05	.025/.025/.05/.05
4000 vs 1600	Avg Sim	2226	1922	1858	**1584**	1748	**1460**	1401	1351
	Win Rate	62.9%	59.7%	61.8%	**61.4%**	60.9%	**61.5%**	59.1%	54.8%
6400 vs 6400	Avg Sim	3914	3341	3273	2815	2983	2539	**2453**	2364
	Win Rate	48.0%	44.1%	46.9%	46.1%	46.5%	46.5%	**46.5%**	44.7%

programs. The results in Fig. 5 shows that using "STOP" can further accelerate the search since all lines (including baseline) move slightly to the left comparing to Fig. 4. For example, when $\boldsymbol{c}[1] = 800$, $\boldsymbol{thr}[1] = 0.2$ we achieves 48.5% win rate with only 555 simulations. The results also show that our methods still perform better than the baseline. For instance, to achieve more than 48% win rate, PV-MCTS still needs 964 simulations even with the "STOP" algorithm. Note that we can compare our method with "STOP" directly by comparing the brown line and red line in Fig. 4 and the black line in Fig. 5. Our method has a better result than "STOP", even if only using State-UN.

Testing on larger simulations In this experiment, we want to show that although our MCTS-UN is trained by the data generated with $N_{max} = 1600$, it can be used in a larger N'_{max}. Moreover, with more checkpoints, our method can perform even better.

In the first experiment (4000vs1600), we increase the N_{max} of DS-MCTS to 4000 and still fight against PV-MCTS with 1600 simulations per move. Besides showing our method can scale up, we also want to show that our method has a higher win rate under a similar average simulation count. Table 2 shows the results. The first column only uses State-UN with $\boldsymbol{thr}[0] = .1$. Columns from 6th to 8th show the results that use MCTS-UN more than one time. For example, the last column used checkpoints $\boldsymbol{c} = \{0, 160, 400, 800, 1600\}$ with thresholds $\boldsymbol{thr} = \{.1, .025, .025, .05, .05\}$ respectively. The result of $(800, .05)$ shows that we can have a 61.4% win rate against PV-MCTS that uses a similar simulation count. The result $(800/1600, .05/.05)$ shows that we can reduce the simulation count without decreasing the win rate by using MCTS-UN to check on both 800 and 1600 simulations.

In the second experiment (6400vs6400), we increase the N_{max} of DS-MCTS to 6400 and fight against PV-MCTS with 6400 simulations. We want to test if our method can scale up and fight against a much different and stronger agent. The last row of Table 2 shows we achieve more than a 46% win rate against 6400 simulations with most of the settings. Moreover, with the setting of $\boldsymbol{c} = \{0, 400, 800, 1600\}$, we can have a 46.5% win rate against 6400 simulations with only 2453 average simulations. Note that the win rate of (2400vs6400) for PV-MCTS is only 34%. Thus we show that MCTS-UN can be used when both N'_{max} is much larger, and the opponent is much stronger than training data.

Experiments on 9x9 Go We further apply our methods to 9x9 Go to demonstrate it is general to other games. The PV-NN of our AlphaZero 9x9 go agent is ResNet with 20 blocks and 256 filters. The training data is generated in the same way as NoGo.

There are two differences between the experiments of Go and NoGo. First, for each game, instead of starting from the empty position, we randomly select an opening from a public 9x9 Go opening book (Coulom 2017) as the starting position. Our State-UN and MCTS-UN are trained using self-play records. Using openings can help us verify whether our models can handle unseen positions in the training data. Second, for simplicity, we choose 0.1 as the threshold for every checkpoint.

The first experiment is DS-MCTS vs. PV-MCTS with both 1600 simulations. With only State-UN, the average simulation ratio can drop to 79% with about 50% win rate. Adding a checkpoint at $160, 400, 800$ separately can reduce the average simulation ratio to $67.6\%, 67.5\%, 71.1\%$ and remain more than 46.5% win rate. If we use three checkpoints together, the average simulation ratio can be further reduced to 59.4%. We then test this setting with $N_{max} = 3200$ against PV-NN with 3200 simulations per move and still get a 44.9% win rate. This shows that our method can scale up to a larger N_{max}, even if the position is unseen and the opponent is stronger. Other full results of 9x9 Go are shown in the appendix.

Conclusion and Future Work

In this paper, we first define the uncertainty for tree searching. We then train State-UN and MCTS-UN to predict the uncertainty. We also propose Dynamic Simulation MCTS to utilize those networks. Experimental results on NoGo and Go show that DS-MCTS can speed up the inference time and maintain the win rate. Moreover, although training on a small maximum simulation, MCTS-UN can scale up to a much larger one. After fine-tuning the thresholds on NoGo, DS-MCTS can speed up more than 2.5x and still maintain a 46.5% win rate under a much larger N_{max}.

One of our future work is applying our method for generating self-play records during AlphaZero training. Wu (2019) shows that randomly reduce the simulation count on 75% of states is still able to train a strong AI. Hence, we believe that we can reduce simulation on even more states if we select the states properly instead of random. In our inner experiments of using PV-NN to estimate uncertainty, 19x19 Go is much easier to speed up than 9x9 Go.

Another future work is inspired by ProbCut (Buro 1995). We can stop not only the searching of the root state but also any state in the search tree. Also, to improve MCTS-UN, we need to input more information of the search tree. A promising way is using graph neural networks (Wu et al. 2020) to gather deeper information from the search tree.

Acknowledgments

References

Amodei, D.; Olah, C.; Steinhardt, J.; Christiano, P.; Schulman, J.; and Mané, D. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* .

Azizzadenesheli, K.; Brunskill, E.; and Anandkumar, A. 2018. Efficient exploration through bayesian deep q-networks. In *2018 Information Theory and Applications Workshop (ITA)*, 1–9. IEEE.

Baier, H.; and Winands, M. H. 2015. Time management for Monte Carlo tree search. *IEEE transactions on computational intelligence and AI in games* 8(3): 301–314.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1): 1–43.

Burda, Y.; Edwards, H.; Storkey, A.; and Klimov, O. 2018. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894* .

Buro, M. 1995. Probcut: An effective selective extension of the α-β algorithm. *ICGA Journal* 18(2): 71–76.

Chou, C.-W.; Teytaud, O.; and Yen, S.-J. 2011. Revisiting Monte-Carlo tree search on a normal form game: NoGo. In *European Conference on the Applications of Evolutionary Computation*, 73–82. Springer.

Clements, W. R.; Robaglia, B.-M.; Van Delft, B.; Slaoui, R. B.; and Toth, S. 2019. Estimating risk and uncertainty in deep reinforcement learning. *arXiv preprint arXiv:1905.09638* .

Coulom, R. 2017. Crazy Sensei 9x9 Go Opening Book. https://www.crazy-sensei.com/book/go_9x9.

Donninger, C. 1994. A la recherche du temps perdu:'That was easy'. *ICGA Journal* 17(1): 31–35.

Gal, Y.; and Ghahramani, Z. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, 1050–1059.

Gao, C.; Müller, M.; and Hayward, R. 2018. Three-Head Neural Network Architecture for Monte Carlo Tree Search. In *IJCAI*, 3762–3768.

Garcıa, J.; and Fernández, F. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research* 16(1): 1437–1480.

Guo, C.; Pleiss, G.; Sun, Y.; and Weinberger, K. Q. 2017. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1321–1330. JMLR. org.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hendrycks, D.; and Gimpel, K. 2016. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136* .

Huang, S.-C.; Coulom, R.; and Lin, S.-S. 2010. Time management for Monte-Carlo tree search applied to the game of Go. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, 462–466. IEEE.

Hyatt, R. M. 1984. Using time wisely. *ICGA Journal* 7(1): 4–9.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *European conference on machine learning*, 282–293. Springer.

Kocsis, L.; Uiterwijk, J.; and van den Herik, J. 2000. Learning time allocation using neural networks. In *International Conference on Computers and Games*, 170–185. Springer.

Lakshminarayanan, B.; Pritzel, A.; and Blundell, C. 2017. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in neural information processing systems*, 6402–6413.

Lan, L. 2016. HaHaNoGo: An open source NoGo program. https://github.com/lclan1024/HaHaNoGo.

Lan, L.-C.; Li, W.; Wei, T.-H.; Wu, I.; et al. 2019. Multiple policy value Monte Carlo tree search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 4704–4710.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature* 521(7553): 436–444.

Liang, S.; Li, Y.; and Srikant, R. 2017. Enhancing the reliability of out-of-distribution image detection in neural networks. *arXiv preprint arXiv:1706.02690* .

Liu, A.; Chen, J.; Yu, M.; Zhai, Y.; Zhou, X.; and Liu, J. 2020. Watch the Unobserved: A Simple Approach to Parallelizing Monte Carlo Tree Search. In *International Conference on Learning Representations*. URL https://openreview.net/forum?id=BJlQtJSKDB.

Markovitch, S.; and Sella, Y. 1996. Learning of resource allocation strategies for game playing. *Computational Intelligence* 12(1): 88–105.

Müller, M. 2015. NoGo History and Competitions. https://webdocs.cs.ualberta.ca/~mmueller/nogo/history.html.

Pathak, D.; Agrawal, P.; Efros, A. A.; and Darrell, T. 2017. Curiosity-driven exploration by self-supervised prediction.

In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 16–17.

Platt, J.; et al. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10(3): 61–74.

Rosin, C. D. 2011. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence* 61(3): 203–230.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2019. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265* .

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529(7587): 484.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550(7676): 354–359.

Šolak, R.; and Vučković, V. 2009. Time management during a chess game. *Icga Journal* 32(4): 206–220.

Wu, D. J. 2019. Accelerating Self-Play Learning in Go. *arXiv preprint arXiv:1902.10565* .

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Philip, S. Y. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* .

Appendix
Proof of Lemma 3.1

Lemma 2 *The problem* $\arg\min_n f(\alpha, n)$ *have the same solution for any* $\alpha \in [0, 1]$.

where $f(\alpha, n)$ is:

$$f(\alpha, n) = (n \times \alpha + N_{max} \times (1 - \alpha))p(M(s) < n) \\ + N_{max}p(M(s) \geq n) \quad (9)$$

Proof .1 *Assume* $f(1, n_1) < f(1, n_2)$, *we have*

$$n_1 \times p(M(s) < n_1) + N_{max}p(M(s) \geq n_1) \\ < n_2 \times p(M(s) < n_2) + N_{max}p(M(s) \geq n_2) \quad (10)$$

Times α *at both sides:*

$$\alpha \times n_1 \times p(M(s) < n_1) + \alpha \times N_{max}p(M(s) \geq n_1) \\ < \alpha \times n_2 \times p(M(s) < n_2) + \alpha \times N_{max}p(M(s) \geq n_2) \quad (11)$$

Add $(1 - \alpha) \times N_{max}$ *to both sides:*

$$(\alpha n_1 + N_{max}(1 - \alpha))p(M(s) < n_1) \\ + N_{max}p(M(s) \geq n_1) \\ < (\alpha n_2 + N_{max}(1 - \alpha))p(M(s) < n_2) \\ + N_{max}p(M(s) \geq n_2) \quad (12)$$

Hence we have $f(\alpha, n_1) < f(\alpha, n_2)$. *Therefore, for every* α, $\arg\min_n f(\alpha, n)$ *is same as* $\arg\min_l f(1, l)$.

Experiment Settings of NoGo

Although our goal seems simple, training MCTS-UN is still hard due to the data imbalance. Only less than 10% of states that are $M(s) > 160$. To train MCTS-UN, we first generate 20,000 self-play records using 800 simulations. We then re-evaluate the states in the records using $N_{max} = 1,600$ simulations to calculate the $M(s)$ of each state. We set $c_1 = 0.1$ for both loss functions. After training the State-UN, we use a threshold of 0.05 to prune out about $1/3$ of the data with $M(s) = 1$, since they are easy to be identified by the State-UN. During training MCTS-UN, we also increase the probability of sampling states that are $M(s) > 160$, in order to handle data imbalance problems. Each state sampled has 50% to be trained with both state and MCTS features; it has 25% to be trained with only state features, and the last 25% to be trained with only MCTS features. In this way, the model will not overly rely on one of the features to determine uncertainty.

In fig. 3, we use different sets of thresholds for different temperatures τ, so that their result can be compared under the same simulation ratio. We select the thresholds according to the distribution of training data. For $\tau = 0.5$, the thresholds are $\{0.094, 0.137, 0.23, 0.287, 0.332\}$. For $\tau = 1$, the thresholds are $\{0.35, 0.387, 0.495, 0.527, 0.563\}$. For $\tau = 1.5$, the thresholds are $\{0.567, 0.605, 0.655, 0.681, 0.706\}$.

Experiment of Go

Table 2 shows the full results of experiments on Go. All the thresholds are set to 0.1 without any fine-tuned. The results show that when we use a larger simulation, the are more room for us to speed up the agent and still remain almost the same win rate, especially for the larger checkpoints.

Table 2: The results of Go with $thr[0] = 0.1$

		$c[i]$	–	160	400	800	400/800	160/400/800
		$thr[i]$	–	.1	.1	.1	.1/.1	.1/.1/.1
1600 vs 1600	Avg Sim Ratio		79.2%	67.6%	67.5%	71.1%	64.0%	59.4%
	Win Rate		47.9%	46.4%	46.5%	48.0%	45.0%	45.3%
3200 vs 3200	Avg Sim Ratio		81.98%	67.1%	65.7%	65.5%	60.9%	58.0%
	Win Rate		47.3%	44.4%	44.8%	46.7%	44.7%	44.9%