愛知工業大学研究報告
第 54 号 平成 31 年

深層学習とプレイアウトに基づく囲碁アルゴリズム

Go Algorithm Based on Deep Learning and Playout

伊藤 雅 † 伊藤 有人 ††
Masaru ITOH Arito ITOH

Abstract: This paper describes a go algorithm based on deep learning and playout. The algorithm runs on a small resource environment which consists of one CPU and one GPU. The best next move can be obtained by using a Value-Monte-Carlo tree search method. It is one of the best-first search methods. The proposed method omits the process of tree policy which has been proposed by AlphaGo. Instead of tree policy, the method adds the top 20 candidates with the highest probability in synchronization with SL policy network as leaves of the node when expanding a leaf node. The win/loss function according to the rollout policy advocated by AlphaGo is substituted by playout, which is commonly used in ordinary Monte-Carlo tree search. As a node evaluation value, not an ordinary UCB1 value but an action value advocated by AlphaGo is adopted. Numerical experiments confirmed the statistical significance of the proposed method and clarified both the best mixing parameter value and the node expansion threshold.

1. はじめに

2016 年に登場したアルファ碁 [1] とその翌年に発表されたアルファ碁ゼロ [2] は従来のモンテカルロ木探索（Monte-Carlo Tree Search: MCTS）[3] に基づく囲碁思考ルーチンの開発に衝撃を与えた。アルファ碁以前はモンテカルロ木探索の改良が囲碁アルゴリズムの主流であった。木探索部を改善するかプレイアウト部を改善するかの 2 つである。プレイアウトとは、ある盤面から合法手を適当に生成して終局までランダムにシミュレーションすることである。

木探索の改善で顕著な功績を挙げたのが RAVE（Rapid Action Value Estimation）[4, 5] である。自分が勝ったプレイアウトの手を全部「最初に打った」と見做して他の葉ノードの勝数に加算する手法である。プレイアウト数が早く閾値に達し、葉ノードの展開を早める効果がある。

プレイアウトの改善では、LGRF（Last Good Reply with Forget）[6] や 3×3 パターン [7] を活用したプレイアウトの精度向上がある。LGRF はプレイアウト中のある局面で勝った手のみを記憶し、負けた手は忘却するという手法である。同じ局面に遭遇したとき、勝ったときの手を打つことでプレイアウトを効率化する。局面タブーリスト [8] を導入してプレイアウトの多様性を確保する手法もある。

このように木探索とプレイアウトの改善でモンテカルロ木探索は様々に発展してきた。しかし、2016 年まで囲碁ソフトはプロ棋士に互先で全く勝てなかった。それがアルファ碁の登場から僅か 2 年足らずで囲碁ソフトの実力はプロ 9 段を凌駕するまでになった。

アルファ碁の特徴は、教師付学習の SL Policy Network、強化学習の RL Policy Network、盤面評価関数の Value Network、これら 3 つの深層学習を駆使する点にある。従来のプレイアウトの代わりとなる Rollout Policy の導入や Tree Policy の採用といった提案も同時になされている。Tree Policy は APV-MCTS（Asynchronous Policy and Value-MCTS）のノード展開時に威力を発揮する。

アルファ碁の再現を目指すオープンソースプロジェクトのひとつに RocAlphaGo[*1] がある。GitHub を介した Python 言語による開発がその中心である。

アルファ碁のロールアウトやモンテカルロ木探索のプレイアウトはどちらもある局面から仮想対局を行い勝敗を決定する。本研究ではオープンソース囲碁である Ray[*2] のプレイアウトでロールアウトを代用する。Ray はプレイアウトに非決定論的なヒューリスティックを取り入れた優れた囲碁思考ルーチンである。C++言語で開発されている。2016 年に BSD ライセンスで公開され、2017 年第 10 回 UEC 杯コンピュータ囲碁大会で第 3 位の実績がある。棋力は KGS で 2 段程度とされている。Ray のプレイアウトと RocAlphaGo の深層学習を組み合わせるためにラッパークラスを Cython[9] で記述し、Ray の初期化処理とプレイアウトの実行を Python で制御できるようにする。

2. Value Network とモンテカルロ木探索の融合

1202 個の CPU と 176 個の GPU で構成される分散型アルファ碁は APV-MCTS というマルチスレッドに対応した非同期方策と Value Network を内包した MCTS で次の一手を求めている。一方の提案法は、Value-MCTS の部分こそ使うが、アルファ碁のような分散非同期型ではなく、1CPU・1GPU で構成されるマルチプロセスを用いた単体非同期型の少資源環境で動作する囲碁 AI である。

† 愛知工業大学 情報科学部 情報科学科（豊田市）
†† Hamee 株式会社 開発部（神奈川県小田原市）

*1 RocAlphaGo https://github.com/Rochester-NRT/RocAlphaGo/
*2 Ray http://computer-go-ray.com/

通常の MCTS では多腕バンディット問題の解決で有効な UCB 方策[10] が使われることが多い。提案法ではアルファ碁と同様、この部分に Value Network から得られる盤面評価値とプレイアウトから得られる勝敗情報を組み合わせて UCB1 値の代わりとする。これについては後述する。

提案法とアルファ碁の違いは大別すれば次の 2 点である。ひとつ目の相違点はノード展開時の処理である。アルファ碁では SL Policy Network が非同期に実行され、同期が取れるまで Tree Policy からの事前着手確率に基づいてノードを展開処理する。対象となる手は全空点である。一方の提案法では、Tree Policy の処理部分を割愛し、ノード展開時に SL Policy Network と同期させて、着手確率が高い有望手の上位 20 手のみをノード展開時に追加する。これが相違点のひとつ目である。

2 つ目の相違点はノードが展開された後の処理である。まず、アルファ碁が提唱するロールアウトによる勝敗は使わず、この部分を通常のモンテカルロ木探索で使われるプレイアウトで代用する。ただし、ノードの評価値はモンテカルロ木探索の UCB1 値ではなく、アルファ碁が提案する加重平均値を採用する。

探索木において、ある第 t 手目の局面ノード s_t からひとつ下の局面ノードに降りる際の手 a_t の決定は式 (1) に従う。特に、加重平均とバイアス項の和 $Q(s,a)+u(s,a)$ はアクション値（action value）と呼ばれる。

$$a_t = \arg\max_a \{Q(s_t,a)+u(s_t,a)\} \quad (1)$$

$$Q(s,a) = (1-\lambda)\cdot\frac{W_v(s,a)}{N_v(s,a)} + \lambda\cdot\frac{W_r(s,a)}{N_r(s,a)} \quad (2)$$

$$u(s,a) = C_{puct}\cdot P(s,a)\cdot\frac{\sqrt{\sum_b N_r(s,b)}}{1+N_r(s,a)} \quad (3)$$

局面 s で手 a を打った場合を記号 (s,a) で表記すれば、式 (1)∼(3) を構成する記号の意味は次の通りである。

- Q: Value Network からの平均評価値とプレイアウトによる勝率の加重平均値
- u: バイアス項
- W_v: Value Network 出力 $[-1,1]$ の積算評価値
- N_v: Value Network の試行回数
- W_r: プレイアウトの勝敗 $\{\pm1,0\}$ の積算値
- N_r: プレイアウトの試行回数
- P: SL Policy Network による着手確率

ここで、式 (2) の $0\le\lambda\le1$ は Mixing パラメータ、式 (3) の C_{puct} は Exploration パラメータ（定数）である。

式 (2) を中心とする Value-MCTS の概念図を図 1 に示す。この図は実局面 X の黒手番の最善手を求めようとしている。簡単に説明する。まず、根ノード（root node）に実局面 X が登録される。ノードが局面、アークが着手候補手を示す。根ノード登録時には内部ノードは存在せず、根ノード直下に適当な黒の候補手が複数個追加される。この候補手追加に提案法では SL Policy Network を利用する。着手確率の高い有望な最大 20 手の候補手を追加する。

黒番の一手が打たれると局面がひとつ進む。ここからは実局面ではなく、仮想局面になる。図 1 では根ノード直下の白ノード v や s が白番局面を表している。ノードが追加されるとその局面からプレイアウトが 1 回試行される。ノード v から下に降りる波線がプレイアウトである。終局まで適当に打てば勝敗が決定する。

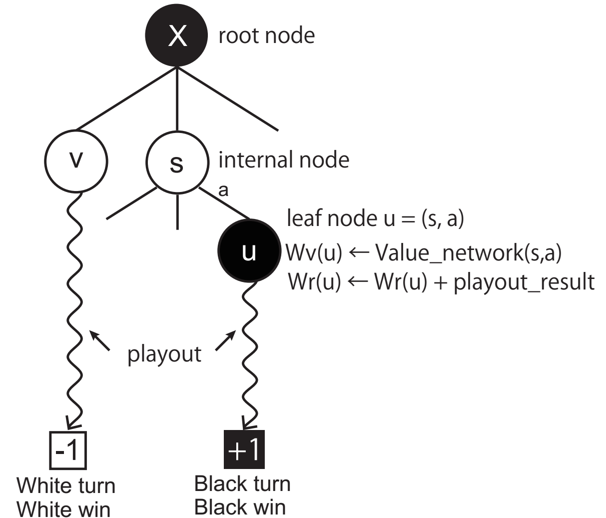

図 1 Value-MCTS の概念図

ノード v に着目すると、白手番ではじまるプレイアウトで白が勝利したので、この場合のプレイアウトの結果は -1 となる。現在の木探索は黒手番の探索である。白は相手手番である。自分手番で始まるプレイアウトで自分が勝てば $+1$、相手手番で始まるプレイアウトで相手が勝てば -1 がプレイアウトの勝敗結果となる。どちらの手番でも負けた場合は、そのプレイアウトの結果は 0 となる。よって、プレイアウトの勝敗結果は $\{\pm1,0\}$ である。プレイアウトの結果は $W_r(v)$ に反映される。

ノード v ではプレイアウトが実行されるだけではない。深層学習後の盤面評価関数 Value Network を起動して、その盤面を評価する。ノード v でいえば、$W_v(v)$ を計算することになる。RocAlphaGo が提供する Value Netrwork では、畳み込みニューラルネットの最終層が全結合ネットワークと双曲線正接関数（hyperbolic tangent function）で表現されている。よって、出力結果は $-1\le\tanh x\le1$ となる。出力結果が $[0,+1]$ の範囲では、$+1$ に近づくほど自分手番の勝利確率が高く、$+0$ に近いと自分手番の勝利確率は低いことを意味する。一方、出力結果が $[-1,0]$ の範囲では、-1 に近いほど相手手番の勝利確率が高く、-0 に近いと相手手番の勝利確率は低いことを意味する。つまり、±0 の近傍では対局が接戦状態にあることを示す。

ノードの評価が済めば、その結果を根ノードまで伝播する。これが木の更新である。更新が終了したら、根ノードからアクション値の大きい子ノードに降りることになる。葉ノードに到達すれば、そこでまたプレイアウトを 1 回試行し、その結果を根ノードまで返す。

葉ノードに到達する回数が一定回数以上になれば、葉ノードを展開して子ノードを追加し、木が 1 段深くなる。この事前に決定しておく一定回数のことをノード展開閾値

（node expansion threshold）という。図 1 は、白番葉ノード s が今展開されて、白石 a が打たれ、新たな子ノード局面 u が生成された状態を示している。葉ノード生成時には必ずプレイアウトが 1 回試行される。併せて Value Network で盤面評価値も計算する。図の葉ノード u では黒手番で始まるプレイアウトで黒勝ちなので $playout_result = +1$ である。$W_r(u)$ の初期値は零である。

葉ノード s が展開されると、このノードは内部ノードとなり、内部ノードではプレイアウトは実行されない。プレイアウトの試行は葉ノードに限定される。葉ノードから根ノードに至る経路上にある内部ノードでも葉ノード時に付与された W_v, N_v, W_r, N_r といった変数は逐次更新される。これを一定時間または一定回数の間繰り返す。提案法ではこの時間制御を事前に定めたプレイアウト回数で行う。

プレイアウトが上限回数に達したとき、Value-MCTS の木探索が終了する。根ノード直下の子ノードが保存する式 (1) のアクション値 $Q(X, a) + u(X, a)$ が最大となる手 a を、実局面 X の黒手番の最善手 a^* として採用する。

3. RocAlphaGo が提供する 3 つの深層学習

アルファ碁の再現プロジェクトである RocAlphaGo は GitHub で開発され、そこでは少なくとも 3 つの深層学習が提供されている。教師付学習の SL Policy Network、強化学習の RL Policy Network、そして盤面評価関数となる Value Network の 3 つである。主として Python 言語で開発されている。提案法では Branch: develop をベースに一部の Python スクリプトを Cython 言語で書き改める。

入力層から出力層まで全部で 16 層のニューラルネットワークで深層学習するが、必ずしもアルファ碁と完全に一致する訳ではない。ネットワーク構造とその評価については 6・2 節で述べる。SL Policy Network と RL Policy Network の 2 つはネットワーク構造が同じなので、SL Policy Network と Value Network についてのみ言及する。

アルファ碁では SL Policy Network への入力チャネルが 48、Value Network のそれが 49 である。手番情報となる Player color の 1 チャネル分が単純に追加されている。それに対して RocAlphaGo は両ネットワークとも 48 である。盤面（19×19）の 1 交点の特徴を 48 ビットで表現するが、RocAlphaGo では表 1 のようにチャネルを使う。最後の Zeros は文字通り 0 で埋めることを意味する。ここを RocAlphaGo では手番情報に切り替えて、入力を 48 チャネルにしている。よって、入力層での 1 盤面の構成には、19×19 の上下左右に 2 つのダミー交点を零パディングで追加して $23 \times 23 \times 48$ チャネル、ビット数でいえば、25,392 bits が必要となる。ダミー交点を追加するのは、あとで 5×5 フィルターで畳み込む際、盤端の特徴が消失しないようにするためである。

入力層の構成だけでなく、中間層となる畳み込み層の第 1 層から第 12 層も SL Policy Network と Value Network は全く同じ構成をとる。ともに第 1 層だけは 5×5 の 192 フィルターを使い、第 2 層から第 12 層までは 3×3 の 192 フィルターを使う。活性化関数には単純な $max(0, x)$ で記述される ReLU 関数（Rectified Linear Unit function）を利用する。第 13 層の畳み込みもやはり両者とも同じである。ただし、出力が 1×1 の 1 フィルタとなり、活性化関数にはやはり ReLU 関数を使う。第 14 層も両者とも同じ。ただし、Flatten 関数で 19×19 の 2 次元配列を 361 個の要素からなる 1 次元配列に整形している。

表 1 RocAlphaGo の盤面 1 交点の特徴表現

SL Policy Network	Value Network	bits
Stone color	Stone color	3
Ones	Ones	1
Turns since	Turns since	8
Liberties	Liberties	8
Capture size	Capture size	8
Self atari size	Self atari size	8
Liberties after move	Liberties after move	8
Ladder capture	Ladder capture	1
Ladder escape	Ladder escape	1
Sensibleness	Sensibleness	1
Zeros	Player color	1
	Total	48

第 15 層と第 16 層の構成は SL Policy Network と Value Network で完全に異なる。SL Policy Network が盤面交点の着手確率を出力するのに対し、Value Network は盤面の勝敗指数を $[-1, 1]$ の実数で出力するからである。両者の違いを表 2 に示す。表 2 の第 16 層で使用する softmax 関数 $f(x_i)$ と $\tanh x$ 関数とは次のような関数である。

$$f(x_i) = \frac{\exp(x_i + Bias)}{\sum_{n=1}^{N} \exp(x_n + Bias)}, \quad \sum_{i=1}^{N} f(x_i) = 1.$$

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad -1 \leq \tanh x \leq 1.$$

4. プレイアウトによるロールアウトの代用

アルファ碁の APV-MCTS では葉ノードを展開して新たな子ノードを生成した後、ロールアウトで勝敗を決定する。ロールアウトは SL policy Network が 1 手 3ms かかるところを 1 手 2μs で実行する。活性化関数を softmax 関数としたロジスティック回帰を使う。提案法ではこのロールアウト部分を通常のプレイアウトで代用する。プレイアウトの多くは乱数で終局まで打つことが多い。しかし、それではロールアウトの代わりとして使い難い。戦略や知識を利用していないからである。そこで、棋力が KGS で 2 段程度とされている Ray のプレイアウトを利用する。

Ray では終局までプレイアウトすることをシミュレーション [11] と呼んでいる。そのため、公開されているオープンソースの C++ファイル名と関数名は Simulation.cpp、void Simulation となっている。提案法では、関数 Simulation を呼び出す int Playout 関数を Playout.cpp として記述し、式 (2) に合致するよう変更した関数の戻り値をプレイアウトの結果として利用する。Playout 関数からはい

表 2 RocAlphaGo 深層学習の第 15 層と第 16 層の違い

	SL Policy Network	Value Network
第 15 層	361 個の交点に Bias 値を加算	361 個を全結合して 256 個のノード値を出力
第 16 層	361 個の交点から softmax 関数で着手確率を算出	256 個を全結合して tanh 関数で勝敗指数を算出

くつかの Ray の関数が呼び出されている。これらの関数を Python 言語から呼び出せるように Cython 言語 [9] を使ってラッパークラスを用意する。例えば、wrapper.pyx と wrapper.pxd を用意する。拡張子 "pyx" をもつファイルは Cython の実装ファイルであり、拡張子 "pxd" をもつファイルは Cython の定義ファイルである。

まず、wrapper.pyx を Cython コンパイラで処理して C++ソースコードの wrapper.cpp を自動生成する。次に、Ray が提供するすべての C++ソースコードと wrapper.cpp を C++コンパイラで処理してオブジェクトファイルを一括生成する。最後にリンカで適切な LDFLAGS を指定して拡張子 "so" からなる共有ライブラリを作成する。プラットフォーム Linux, アーキテクチャ x86_64, Python バージョン 3.6 の環境で setuptools パッケージの setup 関数と Cython.Build パッケージの cythonize 関数を使用した場合、wrapper.cython-36m-x86_64-linux-gnu.so という名前の共有ライブラリが生成される。このライブラリに含まれる Ray の Simulation 関数は Python スクリプトから直接利用することができる。

Ray のプレイアウトは文献 [12] によれば、次のような特徴がある。盤全体の着手確率テーブルを保持して、プレイアウト中の 1 回の着手ごとにその確率テーブルを再計算する。そして、確率テーブルを参照しながら、ランダムに次の 1 手を選択する非決定論的プレイアウトを実現する。着手ごとに次に該当する箇所の確率テーブルを再計算する。

- 直前の着手で配石パターンの及ぶ箇所
- 直前の着手で戦術的特徴が変化した箇所
- 直前の着手で石が取り除かれた箇所

これらの箇所を算出するために利用しているのが次の 3 つの特徴である。

1. 直前からの着手距離 $d = 2, 3, 4$
2. MD2 パターン
3. 戦術的特徴

1 つ目の特徴である着手距離 $d = 2, 3, 4$ は MD2 パターンの範囲内に収まることに注意されたい。

2 つ目の特徴である MD2 パターンについて簡単に説明する。今、交点 (x, y) からの碁盤 x 軸方向の偏差を dx、y 軸方向の偏差を dy とする。このとき点 (x, y) における $MD_n(x, y)$ のパターン領域の交点集合とは、マンハッタン距離が n 以下となる領域を指す。よって MD2 のパターン領域とは、式 (4) に $n = 2$ を代入すれば、図 2 の中央 × から菱形形状で広がる❶ ～ ❷までの領域となる。図 2 は MD1 から MD3 までの領域を同時に表示している。

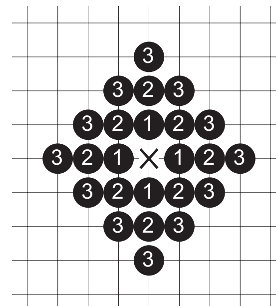

図 2 MD3 のパターン領域

$$MD_n(x, y) = \{(x, y) \mid 1 \le |dx| + |dy| \le n\} \quad (4)$$

3 つ目の特徴である戦術的特徴とは、次のようなヒューリスティックなものである。

- 直前の着手で呼吸点数が 1, 2, 3 になった自分の連に隣接する相手連を取る手
- 直前の着手で呼吸点数が 2, 3 になった自分の連に隣接する相手連をアタリにする手
- 直前の着手で呼吸点数が 1, 2, 3 になった自分の連の呼吸点に打つ手
- 直前の着手で呼吸点数が 3 になった自分の連に隣接する呼吸点数が 3 の相手連の呼吸点に打つ手
- 相手の連を取ってコウを解消する手
- 2 目の抜き跡を欠け眼にするホウリコミ
- 自己アタリ
- その他の石を取る手
- その他のアタリにする手

5. 提案法の概要

提案する少資源環境下で動作する単体非同期型囲碁アルゴリズムの概要について述べる。

まず、RocAlphaGo が提供する SL Policy Network を使って KGS サーバ上にある有段者 6 段以上の棋譜から熟練者の手を教師付学習する。次に、得られた SL Policy Network の重みを自己対戦により改善する。この部分が RL Policy Network の強化学習である。強化学習が済んだ後、Value Network 用のデータセットを作成する。初期盤面を学習済み SL Policy Network を使って作成し、そこから乱数で適当に 1 手打ち、学習盤面とする。学習盤面から学習済み RL Policy Network で交互に打ち合い勝敗

$\{1, -1\}$ を決定する。これを相当数だけ繰り返す。データセットが得られたら、最後に、Value Network を使って盤面の評価関数（出力値は $[-1.0, 1.0]$ の実数）が得られるように深層学習する。ここまですべてがオフライン作業である。実対戦で使うのは学習済みの SL Policy Network と学習済みの Value Network の 2 つだけである。

Value-MCTS の木探索を選択、評価、更新、展開の 4 つで記述する。提案法は RocAlphaGo の Python 版だけでなく Cython も利用するので、Cygo と称することにする。

選択： 現局面が登録される根ノードからアクション値の大きい子ノードを辿って木を降りる。

評価： 葉ノードに到達したら、プレイアウトを 1 回だけ試行する。学習済み Value Network は一度も実行されていない場合のみ非同期に起動する。

更新： 葉ノードのアクション値 $Q(s,a) + u(s,a)$ を構成する諸元値を更新したら、葉から根まで順次、アクション値を更新する。

展開： 葉ノードでのプレイアウト試行回数が事前に定めた閾値（n_{thr}）を超えた場合には、学習済み SL Policy Network を起動してその葉ノードを展開する。具体的には、SL Policy Network が出力する空点への着手確率上位 20 手を選出し、子ノードを生成する。同時に着手確率 $P(s,a)$ を子ノードに与え、プレイアウトを 1 回試行し、諸元値 W_v, N_v, W_r, N_r を初期設定する。

ノードの選択・評価・更新・展開で探索効率を高めるため、CPU の複数コアを使ったマルチプロセスによる並列化を施す。提案法は少資源環境を謳うためマルチ CPU やネットワークによる並列化は考慮しない。並列化には Python の threading モジュールにある Lock オブジェクトの acquire 関数や release 関数などを利用する。

マルチプロセスで使う共有メモリ空間には、探索木と 2 つの実行タスクキューを置く。SL Policy Network 用のタスクキューと Value Network 用のタスクキューである。

そして、GPU 実行用プロセスと探索プロセスの 2 種類を用意する。GPU 実行用プロセスは先の 2 つの実行タスクキューから GPU で実行する 2 つの深層学習のうちいずれかを選択し、タスクを順次 GPU 側に送る処理を担う。葉ノード展開時に使用する SL Policy Network の優先度は高い。そのため割り込み処理ができるようにする。

探索プロセスでは、Virtual Loss[13] という手法を併用した Tree 並列化を導入する。Virtual Loss とは、あるプロセス A が葉に到達してプレイアウトを試行して「勝った」とき、別のプロセス B はその葉でプロセス A は仮想的に「1 回負けた」と更新して木を降りる手法である。葉ノードから根ノードに戻る過程で「1 回負けた」情報を元に戻す。木の形状が異なる探索木を実現できるメリットがある。

共有メモリ空間に置いた探索木を使って複数の探索プロセスを並列で実行させ、ノードの選択・評価・更新を行う。このとき Value Network は非同期に起動され、ノードの諸元値 W_v, N_v, W_r, N_r を適宜更新する。葉ノードでプレイアウト回数がノード展開閾値（n_{thr}）を超えた場合、SL Policy Network が同期的に起動され、子ノードが追加されて、探索木の形状に変化が生じる。

SL Policy Network は探索木の形状に関わるため同期を取りつつ起動され、Value Network は並列実行されている複数の探索プロセスから非同期に起動される。これより提案法は単体非同期型のアルゴリズムといえる。

第 4 章で Ray の Simulation 関数を提案法のプレイアウトとして使えるようにする方法について述べた。そのとき利用したのが Cython である。Cython を使わざるを得なかった理由は、Python で開発された RocAlphaGo の Value Network からの出力と C++で開発された Ray のプレイアウトからの出力結果の 2 つを使って式 (2) の $Q(s,a)$ を算出しなければならなかったからである。

共有ライブラリを生成する Cython にはもうひとつ別の利点がある。Python スクリプトの高速化である。RocAlphaGo の深層学習で 7 つ、Value-MCTS の木探索で 4 つ、計 11 のスクリプトを Python でなく Cython に対応させる。Ray のプレイアウトと併せて全部で 12 の共有ライブラリを生成して提案法である Cygo の高速化を図る。

6. 数値実験

6・1 実験環境と事前の予備実験

数値実験の PC 環境は主として、OS Ubuntu 16.04 LTS, CPU Intel Core i7 3930K (6Core/3.20GHz), GPU NVIDIA GeForce GTX 1080Ti である。主なソフトウェアのメジャーバージョンは、Python 3.6、Cython 0.28、CUDA 9.0、Keras 2.1、tensorflow-gpu 1.12 などである。

事前の予備実験として 4 つの囲碁エンジンの棋力を 9 路盤総当たり戦にて先手 500 局、後手 500 局、計 1000 局で計測した。4 つの囲碁エンジンとは、Ray Ver.8, Fuego*1 Ver.1.1, Pachi*2 Ver.11.00, GNU Go*3 Ver.3.8 である。すべて公開されている定評のある囲碁エンジンである。連続対戦には GoGui*4 Ver.1.4.9 の gogui-twogtp.jar をシェルスクリプトから使用した。

9 路盤での勝敗と勝率は表 3 のようになり、結果は棋力の強い順に、1 位 Ray、2 位 Fuego、3 位 Pachi、4 位 GNU Go となった。勝率は先手からみた 500 局の勝数から求めている。Ray の勝数と勝率が際立っていることが判る。

19 路盤では対局時間の節約のためトーナメント方式を採用した。対局数は 9 路盤と同じ先手 500 局、後手 500 局、計 1000 局である。一手 30 秒、持ち時間 20 分、プレイアウト数 8000 回を起動オプションでそれぞれ指定した。コミは 6 目半である。ただし、GNU Go についてはモンテカルロ木探索が 19 路盤では使えないため、通常の思考ルーチンによる対局とした。

表 3 の 9 路盤での対局結果を参考に、1 回戦を Ray vs.

*1 Fuego `https://sourceforge.net/projects/fuego/`
*2 Pachi `http://pachi.or.cz/`
*3 GNU Go `http://www.gnu.org/software/gnugo/`
*4 GoGui `https://sourceforge.net/projects/gogui/`

表 3 総当たり戦による囲碁エンジンの 9 路盤での棋力（先手 (B) からみた 500 局の勝敗と勝率）

先手 (B) ＼ 後手 (W)	Ray (W)	Fuego (W)	Pachi (W)	GNU Go (W)
Ray (B)	—	399 勝 101 敗 (79.8%)	383 勝 117 敗 (76.6%)	476 勝 24 敗 (95.2%)
Fuego (B)	244 勝 256 敗 (48.8%)	—	328 勝 172 敗 (65.6%)	486 勝 12 敗 (97.2%)
Pachi (B)	158 勝 342 敗 (31.6%)	246 勝 254 敗 (49.2%)	—	496 勝 4 敗 (99.2%)
GNU Go (B)	15 勝 485 敗 (3.0%)	24 勝 476 敗 (4.8%)	32 勝 468 敗 (6.4%)	—

表 4 トーナメント方式による囲碁エンジンの 19 路盤での棋力（先手 (B) からみた 500 局の勝敗と勝率）

先手 (B) ＼ 後手 (W)	Ray (W)	Fuego (W)	Pachi (W)	GNU Go (W)
Ray (B)	—	465 勝 35 敗 (93.0%)	486 勝 14 敗 (97.2%)	—
Fuego (B)	40 勝 460 敗 (8.0%)	—	—	415 勝 85 敗 (83.0%)
Pachi (B)	11 勝 489 敗 (2.2%)	—	—	437 勝 63 敗 (87.4%)
GNU Go (B)	—	81 勝 419 敗 (16.2%)	66 勝 434 敗 (13.2%)	—

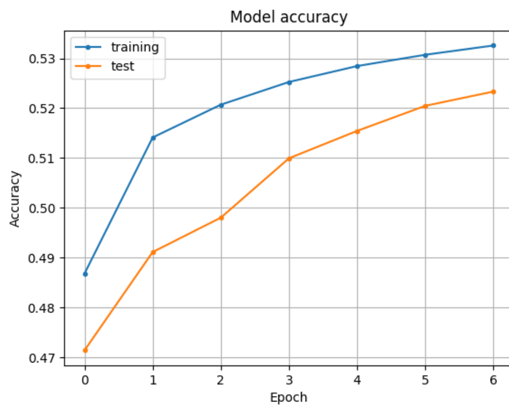

図 3 SL Policy Network の学習推移

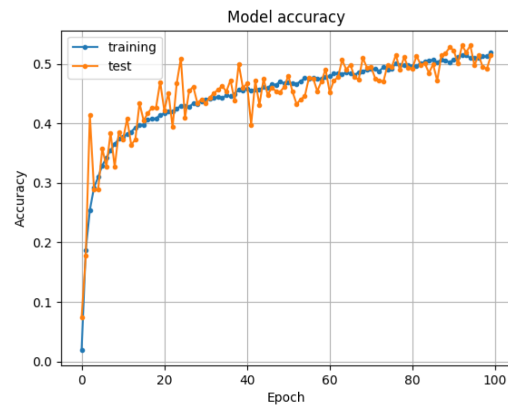

図 4 Value Network の学習推移

Pachi と Fuego vs. GNU Go で対戦させ、それぞれの勝者同士で Ray vs. Fuego を決勝戦とした。1 回戦の敗者同士で Pachi vs. GNU Go の 3 位決定戦も実施した。19 路盤で得られた勝敗と勝率を表 4 にまとめる。勝率は 9 路盤同様、先手からみた 500 局の勝敗数から計算している。19 路盤でも棋力は 9 路盤と同じで Ray > Fuego > Pachi > GNU Go の順となり、Ray の圧勝であった。

以上の結果から提案アルゴリズムである Cygo の棋力判定には対戦相手を Ray のみに限定した。Ray のみに限定した最大の理由は対局時間を節約するためである。

6・2 深層学習の実験結果

教師付学習の SL Policy Network で使用したデータ数やパラメータは、棋譜数 59,976、盤面数 94,731,144、エポック数 7、ミニバッチ 16 である。ネットワーク構造は conv2d_1_input, conv2d_1∼13, flatten_1, bias_1, activation_1 となっている。学習時間 10 日 3 時間 12 分を使って得られた学習推移を図 3 に示す。最終精度は 52.3%であった。有段者の手を約 52%の確率で模倣できるネットワークが構築できたことを意味する。

強化学習の RL Policy Network のネットワーク構造は SL Policy Network と同じである。使用したパラメータは、save-every=10, game-batch=10, iterations=6000, record-every=1 である。自己対戦の初期重みには、SL Policy Network を使って得られた最終エポック後の HDF5 フォーマット形式の weight を使用した。強化学習に要した時間は 20 日 10 時間 10 分であった。強化学習後の weight を使って Ray と先手 250 局、後手 250 局、計 500 局を対戦させた。結果は 142 勝 358 敗と Ray に負け越していた。

最後に盤面評価関数として機能する Value Network を CNN（Convolutional Neural Network：畳み込みニューラルネットワーク）で学習させた。CNN に与えるデータセットの作成には学習済み SL/RL Policy Network を使用した。作成したデータセット数は 194,122 個、盤面数で 1,552,976 個である。Value Network のネットワーク構造は conv2d_1_input, conv2d_1∼13, flatten_1, dense_1, dense_2 となっている。CNN のパラメータはバッチサイズ 16、ミニバッチ 16、エポック数 100 などである。エポック数を 100 まで伸ばせた理由のひとつは、SL Policy Network と比較して盤面数が約 60 分の 1 と少ないからで

ある。学習時間 1 日 14 時間 23 分を使って得られた学習推移を図 4 に示す。最終精度は 51.4%であった。これは盤面評価関数として概ね 51%信頼できることを意味している。

6・3　提案法の 19 路盤での対局結果

SL Policy Network と Value Network の学習済み CNN の重み係数が HDF5 形式で得られたので、これら 2 つの重みを使用した Cygo と Ray を自動対戦させた。コミは予備実験と同じ 6 目半である。式 (3) の定数 C_{puct} の値は 5 とした。先手 250 局、後手 250 局、計 500 局を gogui-twogtp で自動対戦させた。対局は公平性を保つため、探索木で実行するプレイアウト数をすべて 6000 に統一した。予備実験でのプレイアウト数は 8000 であったが、対局に要する実行時間節約のために本実験では 6000 を採用した。対局数を 1000 から 500 に下げたのも実行時間の節約が最大の理由である。提案法の Cygo が実行時間の制御まで考慮できていないため、Ray との対局ではプレイアウト数のみで公平性を確保することとした。Cygo が Cython を使って高速化を図っているとはいえ、Python ベースでの開発のため、C++で実装された Ray には実行速度で及ばない。プレイアウト数の上限で公平性を保つのが精一杯である。

Cygo は 1CPU & 1GPU からなる少資源環境下で動作するよう開発した。CPU プロセス数は複数個に対応させた。19 路盤の対局実験では、動作の安定性を考慮して CPU プロセス数を 2 とした。

実験で求めるべきパラメータは大別すれば 2 つである。ひとつは提案法の Cygo で使う最適な Mixing パラメータ λ^* の同定であり、もうひとつは葉ノードを展開する閾値の最適値 n^*_{thr} の推定である。これらを混合比、ノード展開閾値と呼ぶこともあるので、注意されたい。

ノード展開閾値 n_{thr} を 20 に固定して λ の値を横軸に 0.3∼0.8 の範囲で 0.1 ごとに変化させた場合の Cygo の勝数と勝率をプロットすると図 5 を得た。図 5 棒グラフが対局 500 回のうちの Cygo の勝数であり、これを左縦軸で示す。折れ線グラフは勝率であり、右縦軸で%表示する。$\lambda^* = 0.7$ のとき、Cygo の勝数が 343 で最大となり、そのときの勝率は 68.6%であった。

既存手法の Ray と提案法の Cygo の両者で棋力に差があるか否かを判定するために二項検定を行った。今、帰無仮説を「$\lambda = 0.3$ のとき、Ray と提案法の Cygo で棋力の差はない」とし、対立仮説を「$\lambda = 0.3$ のとき、2 つの手法で棋力の差はある」とする。帰無仮説が成立すると仮定して、有意水準 5% で検定を行う。p 値が有意水準 $\alpha(= 0.05)$ 未満ならば、帰無仮説は棄却され、対立仮説が採択される。つまり、19 路盤囲碁で $\lambda = 0.3$ のとき、既存手法 Ray の棋力と提案法 Cygo の棋力に差がある、といえる。λ の値が 0.4 ∼0.8 についても同様の二項検定を行った。

有意水準 5% $(\alpha = 0.05)$ で勝敗数を基に二項検定を行った結果が表 5 である。両者の棋力に「差があるとは言えない」という結果になったのは $\lambda = 0.5$ のときであった。Mixing パラメータの値が $0.6 \le \lambda \le 0.8$ であれば、有意水準 5% のもとで、Cygo の棋力と Ray の棋力で差がある、といえる。最も勝率が高かったのは $\lambda = 0.7$ のときで、そのときの勝率は 68.6%であった。残念ながら、$\lambda \le 0.5$ で Cygo の棋力の統計的有意性は消失し、0.4 以下では Ray に明らかに棋力で劣ることも統計的に明らかとなった。

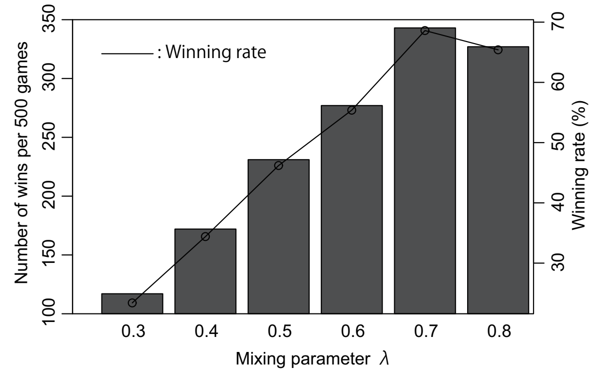

図 5 ノード展開閾値 n_{thr} を 20 に固定した Cygo の勝数と勝率

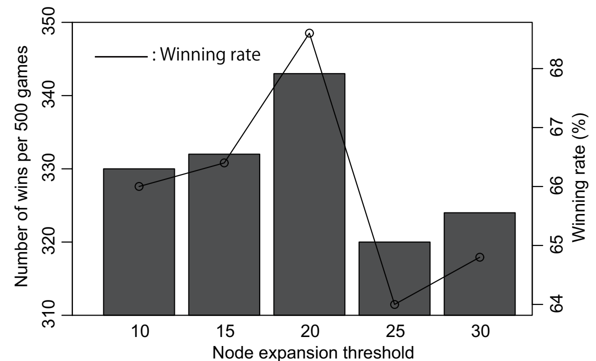

図 6 Mixing パラメータ λ を 0.7 に固定した Cygo の勝数と勝率

次に、Mixing パラメータを $\lambda^* = 0.7$ に固定して、ノード展開閾値 n_{thr} を横軸 $10 \sim 30$ の間で 5 刻みに変化させた場合の Cygo の勝数をプロットすると図 6 になった。有意水準 5%（$\alpha = 0.05$）で二項検定を行った結果が表 6 である。ノード展開閾値 n_{thr} に限れば、10∼30 の間であれば Cygo の統計的有意性が確認できたことになる。表 6 を勝率から判断すれば 68.6%が最大となり、そのときのノード展開閾値は 20 であった。

7.　おわりに

RocAlphaGo が提供する 3 つの深層学習と Ray のプレイアウトを組み合わせた囲碁アルゴリズムを提案した。

数値実験により提案法である Cygo の棋力を明らかにした。Cygo と Ray のプレイアウト数を共に 6000 とし、Cygo のノード展開閾値を 20、Mixing パラメータを 0.7 にすると、Cygo の Ray に対する勝率は 68.6%で最大となった。これはヒューリスティックを取り入れた Ray のプレイアウトが本論文で提案した Value-MCTS で有効に機能している証拠でもある。

提案法の Cygo には大きな問題点がある。実行に要する計算時間である。Mixing パラメータ $\lambda = 0.7$、ノード展

表 5 ノード展開閾値 n_{thr} を 20 に固定したときの Cygo の棋力

λ	win	lose	win-rate (%)	p-value	confidence interval	p-value $< \alpha$
0.3	117	383	23.4	ϵ	0.198 – 0.274	Yes
0.4	172	328	34.4	2.785e-12	0.302 – 0.387	Yes
0.5	231	269	46.2	0.098	0.418 – 0.507	No
0.6	277	223	55.4	0.018	0.509 – 0.598	Yes
0.7	343	157	68.6	ϵ	0.643 – 0.726	Yes
0.8	327	173	65.4	5.331e-12	0.610 – 0.696	Yes

ϵ is less than 2.200e-16.

表 6 Mixing パラメータ λ を 0.7 に固定したときの Cygo の棋力

n_{thr}	win	lose	win-rate (%)	p-value	confidence interval	p-value $< \alpha$
10	330	170	66.0	7.408e-13	0.617 – 0.701	Yes
15	332	168	66.4	1.902e-13	0.621 – 0.705	Yes
20	343	157	68.6	ϵ	0.643 – 0.726	Yes
25	320	180	64.0	3.940e-10	0.596 – 0.682	Yes
30	324	176	64.8	3.548e-11	0.604 – 0.690	Yes

ϵ is less than 2.200e-16.

開閾値 $n_{thr} = 20$、プレイアウト数 6000 という条件下で連続 150 回だけ Ray と対戦させると、1 局当りの平均で Cygo は 4038.0sec（約 1 時間 7 分 18 秒）を消費していた。一方の Ray は 267.8sec（約 4 分 28 秒）の計算時間しか消費していなかった。Cygo が使用する CPU プロセス数をデフォルトの 2 から 8 まで増やしても、同条件下で Cygo の消費時間の平均は 2719.1sec（約 45 分 19 秒）までしか短縮できなかった。一方、Ray の平均消費時間は 255.3sec と大差なかった。このことから 1 手の導出に要する計算時間の短縮こそが今後の最大の課題である。

謝　辞

本研究は文部科学省科研費基盤研究 (C) No. 16K00510 の助成を受けて達成された。ここに謝意を表する。

参考文献

1) D. Silver, A. Huang, et al.: “Mastering the game of Go with deep neural networks and tree search”, *Nature*, Vol. 529, pp. 484–489, 2016.
2) D. Silver, J. Schrittwieser, et al.: “Mastering the game of Go without human knowledge”, *Nature*, Vol. 550, pp. 354–359, 2017.
3) R. Coulom: “Computing Elo Ratings of Move Patterns in the Game of Go”, Computer Games Workshop 2007 (CGW 2007), 2007.
4) S. Gelly and D. Silver: “Combining online and offline knowledge in UCT”, Proceedings of the 24th International Conference on Machine Learning (ICML 2007), pp. 273–280, 2007.
5) S. Gelly and D. Silver: “Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go”, *Artificial Intelligence*, Vol. 175, No. 11, pp. 1856–1875, 2011.
6) H. Baier and P. D. Drake: “The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go”, *IEEE Trans. on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 303–309, 2010.
7) S. Gelly, Y. Wang, R. Munos, and O. Teytaud: “Modification of UCT with Patterns in Monte-Carlo Go”, INRIA, Technical Report, RR-6062, 2006.
8) 伊藤雅, 太田雄大: “局面タブーリストを内包したモンテカルロ木探索の 19 路盤囲碁への応用”, 愛知工業大学研究報告, Vol. 51, pp. 134–142, 2016.
9) K. W. Smith, 中田秀基 (監訳), 長尾高弘 (訳): Cython – C との融合による Python の高速化, オライリー・ジャパン, 2015.
10) P. Auer, N. Cesa-Bianchi, and P. Fischer: “Finite-time Analysis of the Multiarmed Bandit Problem”, *Machine Learning*, Vol. 47, Issues 2–3, pp. 235–256, 2002.
11) 村松正和: “コンピュータ囲碁の現状”, 情報処理, Vol. 53, No. 2, pp. 133–138, 2012.
12) 小林祐樹: モンテカルロ木探索を用いた強い囲碁プログラムの設計と開発, 電気通信大学大学院 情報理工学研究科 情報・通信工学専攻 修士論文, 2016.
13) G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik: “Parallel Monte-Carlo Tree Search”, in Proceedings of the 6th International Conference on Computers and Games (CG2008), Springer, *Computers and Games*, Vol. LNCS5131, pp. 60–71, 2008.

（受理　平成 31 年 3 月 9 日）