elmo アピール文書

瀧澤 誠　twitter: @mktakizawa

開発動機

今まで評価関数ばかり取り組んでいたが、昨今は評価関数の改善が見込めなくなった。そこで(大会直前に急遽作成した定跡が効果的に機能していたこともあり)定跡作成に注力することにした。ディープラーニングを利用したソフトが台頭する中、NNUE でもディープラーニングでもなく、定跡が勝敗に決定的な影響を与えるような“ゲームチェンジ”が見込めるのではないかと考えた。結果は理想とは程遠かったが、優勝することが出来たため成功したと見做して良いのではないかと今では考えている。

開発過程

2020 年

KristalWeizen/水匠 2/elmo の評価関数を利用し、定跡を用いた対局を繰り返し、定跡の改善と新規定跡の作成を繰り返した。

2021 年

電竜戦の実施に伴い、多くの評価関数が公開された。中でも水匠 3 改の評価関数は強く、定跡の見直しが必要となった。評価関数も改善が必要と認識し、1 手 300 万局面探索で 9200 万局面(※1)を生成した(※2)。その他、学習方法についても色々試したが、残念ながらほぼ改善しなかった。何らかアプローチを変える必要があると認識している。

(※1) 私の開発環境で約 2 ヶ月、電気代は 4 万円ほどで、試行錯誤する時間が必要であることや、計算量を増やしてもあまり変わらない印象もあり、この辺が限界
(※2) 昨年の評価関数があまり改善しなかった原因として、教師となるデータが少なかったのではないかと考えていた

開発内容

定跡の生成

以下探索ノード数を設定し、対局する。

定跡生成側: 4 億局面／手、生成中の定跡を利用。 評価関数は elmo を利用
対抗側:　4 千万局面／手、定跡無し、多様な評価関数を利用

上記対局結果を利用し、以下の通り定跡を採用する。

1. 対局全体を通じて、(評価値を基準として)定跡生成側が不利を受けなかった場合、定跡生成側のすべての手を定跡として採用する。
2. 定跡生成側が不利を受けた場合、不利を受ける前に遡り、検討モードで代替の手を検討し、それを定跡として採用する。この時、既存の定跡の訂正も行う。

ponder 手が定跡に存在した場合にも思考

やねうら王では、ponder 手が定跡にヒットしてしまうと探索を行わない。定跡を抜けた直後は殆ど探索を行えていない状況で指し手を決めてしまうため、不利を受けやすい。この対策として、ponder 手が定跡に存在する場合にも思考するように変更した。

角／飛不成の定跡手

やねうら王では定跡に設定しても(角等の)不成を生成するオプションを指定しない限り、これを指すことが出来ない(このオプションを利用するとやや弱くなるため elmo を含め利用されていない)。従って、定跡手として指せるようにすることで相手の定跡を外し、思考時間を奪い有利を得ることが期待出来る。採用方法としては、角／飛車が成って次の手で取られる定跡については不成でも同様の指し手となる場合が多いため、この手を検出し実際同様の指し手になる場合に不成と置き換えた。ただし、不成による有利は 1 手分の思考時間に留まる一方、通常のやねうら王では逆に利用できない定跡となってしまい使い勝手が悪くなることから、簡単な角交換に関しては全て不成を止め戻すことにした。未だ不成の定跡は多数残っているが、定跡中の不成の手を検知する適切な方法が思いつかず、残念ながら戻せない状況である。

教師データの生成効率化

やねうら王で教師データを生成する場合、gensfen 関数を用いて生成するが、これを用いず通常の対局結果から教師データを生成することで高速化した。具体的には、bestmove とともに評価値を出力するように細工し、自作の対局ツールで対局、教師データを生成した。対局時にはやねうら王と同様、1 つの思考エンジン(1 つの置換表)で生成することとした。厳密な比較は出来ていないが、2 つ思考エンジンを用いて対局するよりも探索が深くなり、良い教師を生成するようである。既存の gensfen 関数と比較し、約 50%時間当たりの生成局面数が増加した。

実験結果

まず、評価関数の違いでは 1500 万ノード／手の条件では水匠 3 改に微差ながら勝ち越すことが出来た(132-17-126;elmo から見て勝-引分-負;平手局面開始)が、3000 万ノード／手の条件で 600 局程行ったところ、勝率は 48%程度であり、改善したとは言い難い(※)。以下では簡単のため評価関数は水匠 3 改で固定し、定跡と定跡進行中の ponder の効果を示す。
(※)move accuracy が最も良かったものを採用したが適切ではなかったかもしれない。

結果

324-66-113(勝-引分-負)　勝率(74.1%)※勝/(勝+負)で算出。
定跡有り側の先手勝率 87.6%に対し、後手勝率が 59.7%に留まっており、後手側の定跡に改善の余地があると考えている。

対局条件

- 各々4 コア(Xeon E5-2696 v4 2CPU 44 コア)利用、Hash 4096MB　※約 300 万ノード/秒
- 探索部はやねうら王(4/29 時点での最新版から開発内容の改善を行ったもの) ※定跡を利用しない場合は既存同様のため同じバイナリを用いる
- 定跡有、定跡無で比較
- ponder 有
- 投了値 -3000
- 320 手で引き分け
- 持ち時間 5 分、秒読み 5 秒/手
- ShougiGUI 利用(4 並列で実施)

追試可能性について

可能です。

WCSC31 詳細アピール文書（PAL）

山口祐

1　開発動機

近年、AlphaGoZero[*1]を発端とする深層強化学習ベースの将棋ソフトは dlshogi[*2]をはじめ開発が加速しており、推論デバイスである GPU の性能向上もあり飛躍的に強化されつつある。

一方で、初期段階から学習手法が一貫した学習の報告は、AlphaZero[*3]の追試を主目的とする AobaZero[*4]のみであり、これとは異なるニューラルネットワークの構造や教師あり学習を組み合わせた場合の学習過程については明らかになっていない点が多い。

そこで PAL（以下、本ソフト）では Squeeze-and-Excitation Networks (SENet) など画像分類タスクで高い性能が報告されているモデル構造やチェス・囲碁ソフト開発における手法などを採用し、深層強化学習を行った。

2　開発過程

ニューラルネットワーク構造として、20 ブロック・256 フィルターの ResNet に Policy/Value 出力用の畳み込み層を持つ AlphaZero 型をベースとした。本ソフトではこれに加えて、ResNet の各畳み込みブロックに SENet ブロック (ボトルネック係数 =8) を挿入し、活性化関数も Relu から Swish[*5]に変更した（表 1)。

また、入力特徴としては先後の駒の位置 (28)、先後の駒の利き (28)、先後の駒の利き数（6)、持ち駒の数（14)、手番 (1) の合計 77 特徴平面の情報を採用した。各数値は持ち駒の数のみその実数、それ以外は 0 または 1 が 9x9 マス分保持される。

Policy の出力特徴は、手番側からみた駒の (移動元, 移動先) のマス（駒打ちの場合は移動先と駒の種類）および成・不成のの組み合わせのうち、合法手として出現しうる 3781 通りをベクトル表現とした softmax 後の方策分布が出力される。また、Value は手番側から見た評価値として [-1.0, 1.0] のスカラー値が出力される。

表1: ニューラルネットワーク構造の比較

	AlphaZero	PAL
入力次元	362 x 81	77 x 81
ブロック数	20	20
フィルター数	256	256
SE ブロック	なし	あり
活性化関数	Relu	Swish
出力次元	1 + 11259	1 + 3781

初期のランダムパラメータからの学習においては、WCSC29 版の PAL (NNUE 評価関数) による評価値および指し手の教師データを別途 5000 万局面生成した。プロ棋士の公式戦棋譜約 15000 からランダムに 32 手目までを選択し、depth12, multipv5 で探索を行った他、探索値の差が最善手から 200 cp 以下の手については一定確率でランダムに選択するように実装し、生成局面の分散化を図った。

得られた上位 5 位までの cp 評価値 q_i に対して

$$v = \frac{1}{1 + \exp(-q_0/k_v)} \tag{1}$$

[*1] https://www.nature.com/articles/nature24270
[*2] https://github.com/TadaoYamaoka/DeepLearningShogi
[*3] https://science.sciencemag.org/content/362/6419/1140
[*4] https://github.com/kobanium/aobazero
[*5] https://arxiv.org/abs/1710.05941v1

$$p_i = \frac{\exp(-q_i/k_p)}{\sum \exp(-q_j/k_p)} \tag{2}$$

ここで $k_v = 600$, $k_p = 100$ とし、初期学習用の状態価値 v, 方策確率 p_i $(i < 5)$ をそれぞれ設定した。また、勝敗価値 z は初期学習では常に v と同じとした。

学習の損失関数は AlphaZero と同様、方策分布の交差エントロピー、価値の二乗誤差、重みの L2 正則化項から構成される。学習用ライブラリには Tensorflow を採用し、最適化アルゴリズムとして Adam を学習率 2.8e-4 から epoch ごとに半減させた。NVIDIA Tesla V100 8 基を用いてバッチサイズ 1024 で 4epoch（2 億局面）分の学習を行い、強化学習の第 0 世代とした。

強化学習では技巧[*6]に囲碁ソフト AQ[*7]の探索部、学習部を移植する形で実装した。1GPU ごとに 256 の異なる対局を割り当て、各対局で 1 手あたり 800 回探索を行うとともに、探索で得られた評価すべき 256 の末端局面をまとめて GPU で推論し、局面生成効率の向上を図った。

強化学習の開始局面はプロ棋士の公式戦棋譜約 15000 の 32 手目までの局面、および floodgate[*8]の 2017 年から 2020 年までの対局のうち、双方レーティング 3000 以上の棋譜で評価値の絶対値が 500 以下、33 手目以降の局面からランダムに選択した。

終局判定については全体の 10%は詰みまたは引き分けまで実施し、90%については 10%の対局の評価値推移を調べ、評価値 x を 5 手連続で下回った場合にその後実際には負けなかった対局が 5%未満になるように、動的に投了閾値を調整した。

局面の生成には平均で Tesla V100 35 基程度を使用した。生成した棋譜は 20000 対局 (約 200 万局面) ごとに 1 世代とし、世代あたりの学習時間は約 2 時間 20 分程度であった。パラメータの更新には Replay buffer として直近 50 世代分の局面データからランダムに 400 万局面を選択した。Value の損失関数としては、局面の探索評価値 v とその対局の結果報酬 z の事情誤差の平均とし、Policy の損失関数は探索開始局面の探索訪問回数を方策分布とみなして交差エントロピーを求めた。0-100 世代目までを学習率 1e-4, 101-150 世代目までを 5e-4、それ以降では 2.5e-5 とし、バッチサイズ 128 で Tesla V100 1 基で 220 世代まで学習を行った。

対局用の探索部については PV-MCTS を採用し、LeelaChessZero[*9]を参考に、ノード構造の省メモリ化、バックアップ完了前の探索回数の管理、複数スレッドによる非同期探索等を実装した。また、詰みルーチンについてはやねうら王[*10]の dfpn 実装を参考にルート局面のみ探索をするようにした。（探索の末端局面については離れ駒以外の簡易 3 手詰判定のみを採用）

探索の推論モデルは TensorRT（7.2.3, CUDA11.1）を用いて GPU ごとに最適化したものを使用した。平手初期局面において GeForce 2080Ti では探索速度が 8600nps だったのに対し、Ampere A100 では 32000nps と約 3.7 倍に向上した（表 2)。また、A100 を 8 基並列に使用した場合でも 230000nps と 90%程度の効率でスケーリングできている。

表2: 平手初期局面の探索速度の比較

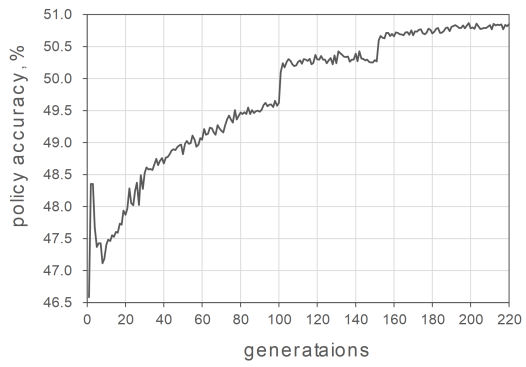
推論 GPU	探索速度 (nps)
GeForce RTX2080Ti	8,600
Ampere A100 x1	32,000
Ampere A100 x8	230,000

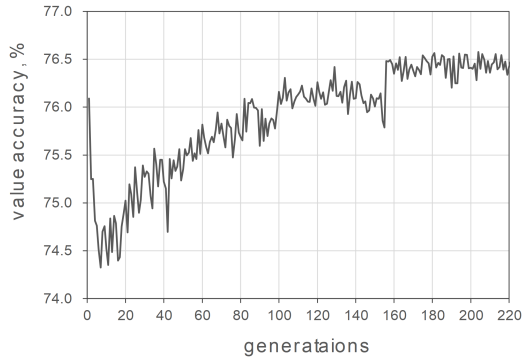
3 結果・考察

学習過程のモデルの評価として、floodgate の 2017 年から 2020 年までのレーティング 3500 以上同士の対局のうちランダムに選択した棋譜から重複を除いた 52 万局面程度を抽出し、テスト用データセットとした。テスト用データセットに対して、各モデルの policy accuracy（一手一致率)、value

[*6] https://github.com/gikou-official/Gikou
[*7] https://github.com/ymgaq/AQ
[*8] http://wdoor.c.u-tokyo.ac.jp/shogi/
[*9] https://github.com/LeelaChessZero/lc0
[*10] https://github.com/yaneurao/YaneuraOu

accuracy (局面評価値の符号と結果の符号の一致率)、value mse（局面評価値と結果の平均二乗誤差）について記録した（図 1)。





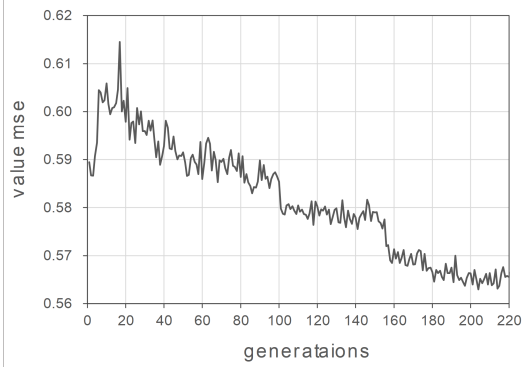


図1: テスト局面の policy/value 一致率の推移

最終的なモデルの評価指標では policy accuracy で 50.8%、value accuracy で 76.5%となり、第 1 回電竜戦バージョンの dlshogi および GCT[*11]の学習モデルをいずれも上回った（表 3)。

表3: テスト用局面の評価指標の比較

	dlshogi	GCT	PAL
policy accuracy	47.2%	47.3%	50.8%
value accuracy	74.9%	75.0%	76.5%
value mse	0.587	0.600	0.565

本ソフトの強化学習に使用した計算資源は Tesla V100 換算で延べ 20000GPU 時間程度であり、GeForce RTX3090 など一般向けの GPU であっても 4 基で 7 ヶ月程度あれば十分再現できると考えられる。

[*11] https://gist.github.com/lvisdd/9b49ab88600fa242f2138fad4eb06caf

Ryfamate アピール文書

Ryfamate チーム[1]　水無瀬 香澄

協力　杉村 達也, 磯崎 元洋

■ 開発動機

数年前に病気で倒れ入退院を繰り返していたが、自力で外の世界を歩けるまでに回復した時には世の中は変わり、機械学習の世界も大きく進歩していた。

そこで、機械学習の最近の動向を勉強するため、本大会に参加した。

■ 開発過程

近年、コンピュータ将棋の世界では、標準的な HalfKP 型 NNUE の評価関数(以下、NNUE)の成長が限定的となっている一方、Deep Learning 系の評価関数(以下、DL) [2]が目覚ましい成長を遂げている。

本来であれば、DL に関する文献やソースコードを読み込み理解したいところだが、初心者であり寝込んでいることが多い筆者には準備期間が短すぎた。また、寝込んでいる間に、手元の PC で教師局面の生成と学習を行わせていたが、特に教師局面の数を用意することが難しく過学習(overfit)を起こすことなどから、十分な成果が得られなかった。

一方で、手元の PC で様々な計測を行う中で、DL は NNUE に比べて勝率は低いが序盤で有利になる頻度が高いことや、棋力が伯仲している NNUE の中でも、序盤が得意な評価関数と、終盤、特に入玉系の得意な評価関数があることが分かった。加えて、自分の作成した評価関数も、特定の条件下では既存のものより僅かに強い可能性があることが分かった。

これらの経緯により、それぞれの評価関数の長所を活かすべく、合議制を採用した。

■ 実験結果

まず、手元の PC で DL と NNUE を対局させると NNUE が大幅に強く、2021 年 1 月時点で公開されている評価関数を用いた場合、DL 側に思考時間を 2.5 倍程度与えることで勝率が釣り合うことが確認できた(Table (1)-(2))。これは、DL と NNUE のエンジンを単純に手数で切り替えることの難しさを示している。過去に、序盤に強い NNUE と終盤に強い NNUE を手数で切り替える実験結果はあるが[3]、今回の実験では同じ持ち時間では DL が弱いため、DL と NNUE を組み合わせて有意に強くなる条件（手数）は見つけられなかった(Table (3)-(5))。

---

[1] https://twitter.com/komafont

[2] 話の単純化のため、本文では NNUE はやねうら王(YaneuraOu)による alpha-beta pruning search で用い、DL は DeepLearningShogi(dlshogi)による Monte Carlo tree search で用いることを前提とする。
https://github.com/yaneurao/YaneuraOu
https://github.com/TadaoYamaoka/DeepLearningShogi

[3] http://qhapaq.hatenablog.com/entry/2019/07/23/221409

次に、単純な多数決について検討した。2 つの異なる NNUE に 4 スレッドずつと 1 つの DL に 2 スレッドを割り当てた多数決エンジンを、4 スレッドの NNUE 単独エンジンと対局させた場合、多数決エンジンが僅かながら有意に強いことを確認した(Table (6)-(8))。

もちろんこれは使用したスレッド数に 10(=4+4+2)と 4 の違いがありフェアな比較ではないが、多数決エンジンの 2 つの NNUE は単独エンジンと比べて NPS が 2 割ほど低下しており[4]、多数決の有効性が確認できた。強いソフトや評価関数同士の多数決は前例があるが、同じ条件（時間）のもとでは弱い評価関数を多数決に加えても効果が見られたことに、本実験の意義があると考えた。

その上で、最初に思考するプロセスを DL と NNUE それぞれ 1 つずつとし、最善手が異なる場合のみ第 3 のプロセスに意見を聞くことを原則とするエンジンを開発した。定跡やその定跡に特化するよう学習した評価関数を用いて計測したところ、序盤と最終盤を除き 6-7 割の局面で、DL と NNUE の最善手が一致することを確認した。これらの結果をもとに、プロセスの優先順位や時間管理など各種調整を行い、今回の出場に至った。

なお、本文では公開された探索部と評価関数を用いた実験結果を掲載しているため、これらについては第三者による追試が可能である。手数による切り替えや単純な多数決については、tttak さん公開の合議将棋(GougiShogi)[5]を用いて検証できる。NNUE 同士の自己対局に比べて環境依存性が高く、CPU と GPU のバランスはもちろん、バックグランドで動く他のプログラムによる影響も受けやすいため注意が必要である。[6]

(Table)

	Player 1			Player2			Result				
Initial	Eval	Threads	Time	Eval	Threads	Time	Games	Win	Lose	Draw	WinRatio
(1) Hirate	GCT	4	0.6s	S3K	32	0.2s	256	143	109	4	56.7%
(2) fg20_r4000_agr16	GCT	4	1.0s	S3K	32	0.4s	288	139	141	8	49.6%
(3) fg20_r4000_agr16	GCT/S3K (ply: 32)	4/32/32	1.0s	S3K	32	1.0s	160	68	79	13	46.3%
(4) Hirate	GCT/S3K/BB (ply: 24,60)	4/32/32	1.0s	S3K	32	1.0s	138	49	63	26	43.8%
(5) fg20_r4000_agr16	GCT/S3K/BB (ply: 32,72)	4/32/32	1.0s	S3K	32	1.0s	148	58	83	7	41.1%
(6) fg20_r4000_agr16	GCT/S3K/BB (voting)	2+4+4	1.0s	S3K	4	1.0s	320	161	138	21	53.8%
(7) Hirate	GCT/S3K/BB (voting)	2+4+4	1m+1sf	S3K	4	1m+1sf	256	126	107	23	54.1%
(8) Hirate	GCT/S3K/BB (voting)	2+4+4	1m+2sf	S3K	4	1m+2sf	103	54	38	11	58.7%

fg20_r4000_agr16: floodgate 2020/Jan/01-2021/Jan/31, Rate>=4000, ply=16, Aigakari Opening

GCT: GCT電竜(model-0000167), S3K: 水匠3改 (Suisho3Kai), BB: BURNING BRIDGES (BB_denryu-tsec1)

CPU: Ryzen Threadripper 2950X (15.7Mnps at 32 threads)
GPU: GeForce RTX 3090 (47Knps at 4 threads)

---

[4] 稼働コア数の増加に伴うクロック低下と、キャッシュメモリのヒット率低下によるものと思われる。

[5] https://github.com/tttak/GougiShogi

[6] 余談だが、計測中に動画を再生しただけでも計測結果に影響する。
間違っても対局中の PC で、盛り上がっている Zoom 対局室に参加してはならない。

WCSC31 Qugiy アピール文書 2

森　大慶

2021 年 5 月 18 日

1　はじめに

とても運が良いことに、Qugiy は今大会で決勝進出することができ、また独創賞まで頂いてしまいました。ビット演算で利きを求めるアレで賞を頂いたので、この文書では選手権前に提出したアピール文書では触れていなかった飛車、角の利きについて書こうと思います。

2　開発動機

正月、暇だったのでネットサーフィンをしていると AWAKE というコンピュータ将棋が題材の映画が取り上げられている記事を見つけました。数年前に一度将棋プログラムを作ろうとして途中でやめてしまったのを思い出し、今回はルール通り将棋が指せるところまで書ききるぞ！という気持ちで開発をはじめました。

3　飛車、角の利き

飛車、角の利き関数と、テーブルの初期化コードを掲載します。以下のソースコードは Qugiy のビットボード定義を想定しています（が、やねうら王でもそのままで動いた気がします）。Qugiy での定義は選手権前に提出したアピール文書を参照してください。PseudoAttacks の初期化も混じっていますが無視してください。

```
/*
    SQUARE_BIT[i]: マス i に対応するビットが立ったビットボード
    part<N>()    : N=0 ならビットボードの下位 64bit を，1 なら上位 64bit を取得するメソッド
*/

// テーブルの定義
alignas(32) uint64_t BISHOP_MASK[2][81][4];
alignas(16) uint64_t ROOK_MASK[2][81][2];

// 角の利き
inline Bitboard bishopAttacks(int i, const Bitboard& occupancy) {
    const __m256i shuffle = _mm256_set_epi8(
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
```

```
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
    );
    const __m256i bishop_mask_lo = _mm256_load_si256((__m256i*)BISHOP_MASK[0][i]);
    const __m256i bishop_mask_hi = _mm256_load_si256((__m256i*)BISHOP_MASK[1][i]);

    __m256i occ2 = _mm256_broadcastsi128_si256(occupancy.xmm());
    __m256i rocc2 = _mm256_shuffle_epi8(occ2, shuffle);
    __m256i hi = _mm256_unpackhi_epi64(occ2, rocc2);
    __m256i lo = _mm256_unpacklo_epi64(occ2, rocc2);
    hi = _mm256_and_si256(hi, bishop_mask_hi);
    lo = _mm256_and_si256(lo, bishop_mask_lo);
    __m256i t1 = _mm256_add_epi64(hi, _mm256_cmpeq_epi64(lo, _mm256_setzero_si256()));
    __m256i t0 = _mm256_add_epi64(lo, _mm256_set1_epi64x(-1ULL));
    t1 = _mm256_and_si256(_mm256_xor_si256(t1, hi), bishop_mask_hi);
    t0 = _mm256_and_si256(_mm256_xor_si256(t0, lo), bishop_mask_lo);
    __m256i a2 = _mm256_shuffle_epi8(_mm256_unpackhi_epi64(t0, t1), shuffle);
    a2 = _mm256_or_si256(a2, _mm256_unpacklo_epi64(t0, t1));
    return _mm_or_si128(_mm256_castsi256_si128(a2), _mm256_extracti128_si256(a2, 1));
}

// 飛車の利き
inline Bitboard rookAttacks(int i, const Bitboard& occupancy) {
    const __m128i shuffle = _mm_set_epi8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
    const __m128i rook_mask_lo = _mm_load_si128((__m128i*)ROOK_MASK[0][i]);
    const __m128i rook_mask_hi = _mm_load_si128((__m128i*)ROOK_MASK[1][i]);

    __m128i rocc = _mm_shuffle_epi8(occupancy.xmm(), shuffle);
    __m128i hi = _mm_unpackhi_epi64(occupancy.xmm(), rocc);
    __m128i lo = _mm_unpacklo_epi64(occupancy.xmm(), rocc);
    hi = _mm_and_si128(hi, rook_mask_hi);
    lo = _mm_and_si128(lo, rook_mask_lo);
    __m128i t1 = _mm_add_epi64(hi, _mm_cmpeq_epi64(lo, _mm_setzero_si128()));
    __m128i t0 = _mm_add_epi64(lo, _mm_set1_epi64x(-1ULL));
    t1 = _mm_xor_si128(t1, hi);
    t0 = _mm_xor_si128(t0, lo);
    t1 = _mm_and_si128(t1, rook_mask_hi);
    t0 = _mm_and_si128(t0, rook_mask_lo);
    __m128i updown = _mm_shuffle_epi8(_mm_unpackhi_epi64(t0, t1), shuffle);
    updown = _mm_or_si128(updown, _mm_unpacklo_epi64(t0, t1));
```

```
    return lanceAttacks<BLACK>(i, occupancy)
         | lanceAttacks<WHITE>(i, occupancy)
         | Bitboard(updown);
}

// 角のテーブル初期化
void init_BISHOP_MASK() {
    for (int i = 0; i < 9; ++i) {
        for (int j = 0; j < 9; ++j) {
            Bitboard rightup(_mm_setzero_si128());
            for (int k = 1; k <= std::min(i, 8 - j); ++k)
                rightup |= SQUARE_BIT[(i - k) + (j + k) * 9];
            Bitboard leftup(_mm_setzero_si128());
            for (int k = 1; k <= std::min(8 - i, 8 - j); ++k)
                leftup |= SQUARE_BIT[(i + k) + (j + k) * 9];
            Bitboard rightdown(_mm_setzero_si128());
            for (int k = 1; k <= std::min(i, j); ++k)
                rightdown |= SQUARE_BIT[(i - k) + (j - k) * 9];
            Bitboard leftdown(_mm_setzero_si128());
            for (int k = 1; k <= std::min(8 - i, j); ++k)
                leftdown |= SQUARE_BIT[(i + k) + (j - k) * 9];

            // ついでに PseudoAttacks も初期化
            BISHOP_PSEUDO_ATTACKS[i + j * 9] = leftup | rightup | rightdown | leftdown;

            rightdown = _mm_shuffle_epi8(rightdown.xmm(), _mm_set_epi8(
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15));
            leftdown  = _mm_shuffle_epi8(leftdown.xmm() , _mm_set_epi8(
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15));

            BISHOP_MASK[1][i + j * 9][0] = rightup  .part<1>();
            BISHOP_MASK[1][i + j * 9][1] = leftdown .part<1>();
            BISHOP_MASK[1][i + j * 9][2] = leftup   .part<1>();
            BISHOP_MASK[1][i + j * 9][3] = rightdown.part<1>();

            BISHOP_MASK[0][i + j * 9][0] = rightup  .part<0>();
            BISHOP_MASK[0][i + j * 9][1] = leftdown .part<0>();
            BISHOP_MASK[0][i + j * 9][2] = leftup   .part<0>();
            BISHOP_MASK[0][i + j * 9][3] = rightdown.part<0>();
        }
```

```
    }
}

// 飛車のテーブル初期化
void init_ROOK_MASK() {
    for (int i = 0; i < 9; ++i) {
        for (int j = 0; j < 9; ++j) {
            Bitboard up(_mm_setzero_si128());
            for (int k = 1; k <= 8 - j; ++k)
                up |= SQUARE_BIT[i + (j + k) * 9];

            Bitboard down(_mm_setzero_si128());
            for (int k = 1; k <= j; ++k)
                down |= SQUARE_BIT[i + (j - k) * 9];

            // ついでに PseudoAttacks も初期化
            Bitboard left(_mm_setzero_si128());
            for (int k = 1; k <= 8 - i; ++k)
                left |= SQUARE_BIT[(i + k) + j * 9];

            Bitboard right(_mm_setzero_si128());
            for (int k = 1; k <= i; ++k)
                right |= SQUARE_BIT[(i - k) + j * 9];

            ROOK_PSEUDO_ATTACKS [i + j * 9] = left | up | right | down;
            LANCE_PSEUDO_ATTACKS[i + j * 9][BLACK] = right;
            LANCE_PSEUDO_ATTACKS[i + j * 9][WHITE] = left;

            down = _mm_shuffle_epi8(down.xmm(), _mm_set_epi8(
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15));

            ROOK_MASK[1][i + j * 9][0] = up  .part<1>();
            ROOK_MASK[1][i + j * 9][1] = down.part<1>();

            ROOK_MASK[0][i + j * 9][0] = up  .part<0>();
            ROOK_MASK[0][i + j * 9][1] = down.part<0>();
        }
    }
}
```

4 実験結果

Qugiy には Magic Bitboard を実装していないため、やねうら王に実装して速度比較をしました。

■置換表 1GB, 1 **スレッド**, depth=19

	time_e1	nodes_e1	nps_e1	time_e2	nodes_e2	nps_e2	nps_diff
0	17970	22460409	1249883	17376	22460409	1292611	42728
1	17867	22460409	1257088	17425	22460409	1288976	31888
2	17883	22460409	1255964	17403	22460409	1290605	34641
3	17828	22460409	1259838	17392	22460409	1291421	31583
4	17890	22460409	1255472	17370	22460409	1293057	37585
5	17886	22460409	1255753	17320	22460409	1296790	41037
6	17862	22460409	1257440	17330	22460409	1296042	38602
7	17855	22460409	1257933	17302	22460409	1298139	40206
8	17895	22460409	1255122	17319	22460409	1296865	41743
9	17851	22460409	1258215	17357	22460409	1294025	35810

	time_e1	nodes_e1	nps_e1	time_e2	nodes_e2	nps_e2	nps_diff
count	10	10	10	10	10	10	10
mean	17879	22460409	1256271	17359	22460409	1293853	37582
std	38	0	2676	41	0	3043	3998
min	17828	22460409	1249883	17302	22460409	1288976	31583
25%	17857	22460409	1255542	17322	22460409	1291718	34933
50%	17875	22460409	1256526	17364	22460409	1293541	38094
75%	17889	22460409	1257810	17388	22460409	1296603	40829
max	17970	22460409	1259838	17425	22460409	1298139	42728

```
Result of 10 runs
==================
base           =    1256271 +/- 2676
test           =    1293853 +/- 3043
diff           =     +37582 +/- 3998

speedup        = +0.0299
P(speedup > 0) = 1.0000

Vendor ID         : AuthenticAMD
CPU Name          : AMD Ryzen 5 3600 6-Core Processor
Microarchitecture : x86_64
```

■**置換表** 1GB, 12 **スレッド**, depth=19

	time_e1	nodes_e1	nps_e1	time_e2	nodes_e2	nps_e2	nps_diff
0	6322	60220033	9525471	3721	37832468	10167285	641814
1	5275	51663910	9794106	4355	44842894	10296875	502769
2	4877	47339800	9706745	7180	70646510	9839346	132601
3	5178	51261782	9899919	5453	54992259	10084771	184852
4	6877	68141848	9908659	5109	52461532	10268454	359795
5	6803	65484524	9625830	6484	66316828	10227764	601934
6	5708	56561438	9909151	3791	38700044	10208399	299248
7	5754	55802825	9698092	5980	58420601	9769331	71239
8	4508	43599834	9671657	10472	103546544	9887943	216286
9	5503	54548347	9912474	7821	77349262	9889945	-22529

	time_e1	nodes_e1	nps_e1	time_e2	nodes_e2	nps_e2	nps_diff
count	10	10	10	10	10	10	10
mean	5680	55462434	9765210	6037	60510894	10064011	298801
std	787	7636000	139659	2076	20019892	198156	225419
min	4508	43599834	9525471	3721	37832468	9769331	-22529
25%	5202	51362314	9678266	4544	46747554	9888444	145664
50%	5606	55175586	9750426	5716	56706430	10126028	257767
75%	6180	59305384	9906474	7006	69564090	10222923	467026
max	6877	68141848	9912474	10472	103546544	10296875	641814

```
Result of 10 runs
==================
base           =    9765210 +/- 139659
test           =   10064011 +/- 198156
diff           =    +298801 +/- 225419

speedup        = +0.0306
P(speedup > 0) = 0.9994

Vendor ID         : AuthenticAMD
CPU Name          : AMD Ryzen 5 3600 6-Core Processor
Microarchitecture : x86_64
```

ちゃんと計ると 3% ぐらいの高速化みたいです。

W@nderER アピール文書(詳細)

2021/5/15

櫻井博光

・開発動機

以前参加した世界コンピュータ将棋選手権 （おそらく WCSC28 か 29? ） で決勝リーグを見物していた際、入玉して宣言勝ちが容易そうな展開から相手を寄せに行く将棋エンジンが多数みられ、一目散に宣言を目指せばいいのに…と思うことがあった。そこから詰みよりも宣言勝ちを目指すエンジンがあれば面白いのではないかという考えに至り W@nderER の開発を行うようになった。

・開発過程

WCSC30 (WCSOC2020) まで：

1. 入玉志向かつ一定の強さを担保する評価関数の作成

事前の調査段階では入玉ステップ数 [1]を入力特徴量に加えた NNUE 型評価関数の設計も考えたが、最終的には既に公開されていた NNUEkaiU[2] に追加学習することで目標とする評価関数を用意できたと判断した。

当時たややんさんにより公開された NNUEkaiU 評価関数は、序盤から積極的に中段玉陣形を志向し他の評価関数より入玉しやすいという特徴があったものの、平手では orqha1018[3] 等平手で強いと言われる評価関数に比べ弱くなる、入玉後に負ける展開の棋譜が目立つという側面もあった。

そこで Floodgate より宣言勝ち及び入玉後勝利した棋譜を収集し、入玉前後の局面を開始局面として、Apery の KPPT 評価関数 [4]と探索部に手を加えた複数種類のやねうら王にて教師データを depth8, 1000 万局面程度生成し、追加学習を行って _0322spv2NkU[5] を作成した。

2. NNUE 型評価関数のネットワーク改造、鵺としての重み行列のキメラ合成

1 で作成した評価関数をベースに自己対局からの強化学習を試みていたが、強くならない、強くなったら中段玉志向でなくなっているなど、その後の進展はなかった。

wcsoc2020 の直前に tttak さんによって様々な種類のネットワークが公開された [6]ため、そちらを参考に HalfKP と玉の段位置を入力特徴量としたネットワークの HalfKPKrank を作成した。その際、 9 倍になった特徴量変換の重み行列について、_0322spv2NkU を 9 倍せず、複数の標準 NNUE より重みを流用し配合することで、中段玉志向を維持したまま強くすることができた。（便宜上、標準 NNUE は最も種類の多い HalfKP 256x2-32-32-1 ネットワーク型とする。）

以下は流用したネットワークの構成である。

Bias	orqha1018
Weight	orqha1018, _0322spv2NkU, orqha1018, Kristallweizen_kaiV0.4[7], orqha1018, elmo2019[8], orqha1018, _0322spv2NkU, orqha1018
HiddenLayer, OutputLayer	orqha1018

WCSC31 まで：

人造棋士 18 号、白ビールのたまさんが公開されている標準 NNUE 用の走査スクリプト [9]を HalfKPKrank に対応させ、新たに配合を行った上でパラメターの平均化を行った。以下は元となったネットワークの構成である。

評価関数 A	Bias	水匠 3 改
	Weight	_0322spv2NkU, NNUEkaiU, 水匠 3 改 [10], Wandre20201222[11], elmo2019,

		orqha1018, 水匠 3 改, 水匠 U [12], _0322spv2NkU
	HiddenLayer, OutputLayer	水匠 3 改
評価関数 B	Bias	水匠 3 改と Wandre20201222 を 1:1 でマージしたもの
	Weight	水匠 3 改 × 3, Wandre20201222 × 3, 水匠 3 改 × 3
	HiddenLayer, OutputLayer	水匠 3 改

選手権では上記の評価関数 A と評価関数 B を 1:1 で配合した 2021-4 と 3:1 で配合した 2021-7 の 2 種類を使用した。

・開発工夫点

教師データ生成時の開始局面を入玉前後にすることで宣言回りを学習しやすくした。
重み行列をキメラ的に混ぜることで、いわゆる棋風的なものと強さの両立を行った。

・実験結果

各種評価関数の連続対局結果を以下に記す。

1． 各種入玉評価関数の vs 技巧 2 成績

※探索部はやねうら王 v4.89、平手開始、 6 スレッド 1 手 5 秒、投了評価値 -3000、Hash 1024 MB、対局打ち切り 320 手

NNUEkaiU	_0322spv2NkU	水匠 U
67 勝 33 敗	77 勝 23 敗	78 勝 22 敗

2. HalfKPKrank 評価関数の対局結果

2021-7, 2021-4 は、wcsoc1 版 (以下 2020) 及び水匠 3 改との対局で強さの評価を試みた。

2-1. 選手権前・選手権中の計測

※探索部はやねうら王 v6.00、投了評価値 -3000、Hash 1024 MB、対局打ち切り 320 手で固定

対局条件 1 ：平手開始

8 スレッド 1 手 4 秒	水匠 3 改	2021-4	2021-7	2020	6 スレッド 1 手 5 秒	水匠 3 改	2021-4	2021-7
水匠 3 改	-	62-10-28	61- 5-34	-	水匠 3 改	-	56-10-34	68-9-23
2021-4	28-10-62	-	58-18-24	59-8-33	2021-4	34-10-56	-	65-8-27
2021-7	34- 5-61	24-18-58	-	48-6-46	2021-7	23- 9-68	27-8-65	-
2020	-	33-8-59	46-6-48	-				

対局条件 2 ：たややん互角局面集 [13]の 36 手目より開始

8 スレッド 1 手 4 秒	水匠 3 改	2021-4	2021-7
水匠 3 改	-	46-10-44	55-10-35
2021-4	44-10-46	-	53-13-34
2021-7	35-10-55	34-13-53	-

2-2. 選手権後の計測

上記の結果より選手権では評価関数 2021-4 を使用する方針であったが、実際は開幕 2 局で千日手、負けと振るわなかったため、それ以降は中段玉志向の強い 2021-7 を使用した。
結果として 2021-7 を使用したことで決勝進出につながったが、事前計測との違いが気になり、手元の計算資源にて Hash が溢れない程度で選手権環境に近いノード数相当での対局を試みた。

対局条件：たややん互角局面集 36 手目開始、投了評価値 -1000、対局打ち切り 320 手

	6 スレッド 1 手 180 秒、Hash 4096 MB	8 スレッド 1 手 300 秒、Hash 8192 MB
2021-4 vs 水匠 3 改	3-2-5	2-4-4
2021-7 vs 水匠 3 改	5-1-4	5-0-5

行えた局数が少なすぎるため参考にならないが、高ノード条件下では、2021-7 に優位性があったのかもしれない

・追試可能性

_0322spv2NkU の学習の全くの再現は難しいと思われるが、キメラに用いた各評価関数及び W@nderER の評価関数は公開されている [14-15] ため、そちらの再現は可能だと考えられる。

[1] 滝瀬竜司, 田中哲朗, 入玉指向の将棋プログラムの作成 , 第 16 回ゲームプログラミングワークショップ 2011, pp. 25 – 31 (2011)

[2] https://twitter.com/tayayan_ts/status/1082436812369346560

[3] http://qhapaq.hatenablog.com/entry/2019/02/22/000101

[4] https://bitbucket.org/hiraoka64/apery-evaluation-binaries-2019-06-17/src/master/

[5] https://drive.google.com/file/d/17Y_BeoJYsfCswtvuIHPAfKkOYDqJPPm0/view

[6] https://github.com/tttak/YaneuraOu/releases/tag/V4.89_NNUE-features_20200401

[7] https://github.com/Tama4649/Kristallweizen

[8] https://twitter.com/mktakizawa/status/1125312513422139392

[9] https://github.com/Tama4649/etc

[10] https://twitter.com/tayayan_ts/status/1351126961016479746

[11] https://twitter.com/ihme_vaeltaa/status/1342797331318493185

[12] https://twitter.com/tayayan_ts/status/1258702510564372481

[13] https://twitter.com/tayayan_ts/status/1326806629438824450

[14] https://twitter.com/ihme_vaeltaa/status/1258396166133174272

[15] https://twitter.com/ihme_vaeltaa/status/1391049506460897281

チーム Barrel house のアピール文書@第 31 回世界コンピュータ将棋選手権

Barrel house とは岡山の駅前にあるビアバーです．元々チームメイトを求めさすらって行きついたところです．マスターに許可頂いたので名前をお借りしました．その後，メンバーが変わって当初の方向性とは全く違って来ましたが，まぁ一度出した名前を変更するのもアレなのでそのままです．

プログラム名：白ビール
第 28 回の Hefeweizen が濁った白ビールでしたが，第 29 回では Kristallweizen とフィルターでろ過した透き通った白ビールでした．その後 Hefeweizen に戻すという形を取りました．しかしながら，欧州のチェスサイトでは命名由来から説明頂いているにも関わらず国内では白ビールとしか読んで頂けないので，もう白ビールでいいやってことです．

チームの特徴
フレッシュなチームを自認してますがどうやらおっさんチームのようです．本大会は体力勝負ではないので各方面に様々な知識や技術・勘などを働かせて今年も決勝に残れればいいかなぁと思っています。諸事情でメンバーが多忙なため具体的な策はこれから詳細を詰めていくところです．（前回も前々回もそんなアピール文だった気もしますが）

評価関数
昨年度の計測でもトップチームと劣らないものであることを確認しました．（詳しくは blog にて）今後の調整等は未定です．

使用マシン
今年もノートパソコンを中継にクラウドの力をお借りする予定です．一昨年と昨年は同じ仕様のものを用いましたが，クラウドの方も今年は若干様子が変わっているようでベンチマーク等も未だなので具体的なことは全く決まっていません．正直あまり予算がないのですが，マシンパワーだけで負けるのもアレなので予算内で一番速いマシンを借りようと考えています．

クラスタリングについて
第５回電王トーナメントで御披露目をした shotgun システムの進化版である Multi Ponder のクラスタリングを用います．制作メンバーが本年は外れておりますが，既に実装は文書化され多くの類似実装が選手権でも見られますので改良して用いるのは問題ないと考えます．

追記：
第 31 回世界コンピュータ将棋選手権 6 位 白ビール

１．参加にあたっての背景

昨年のオンライン大会ぐらいの時期より、「DL が NNUE を抜くのではないか？」と言われ始めていました。
それを裏付けるように昨年末に行われた電竜戦本戦では dlshogi ベースの GCT が電竜に輝くなど、着実に
力をつけていることが伺えました。これはいつまでも NNUE にしがみついていたら置いていかれるぞ！と
いう空気も開発者間には流れていたように思います。

そんな状況もあり、白ビールチームも DL 方面に舵を切るべく開発を進めていくことになり、まずは
芝さんが開発を進めていきました。
私のほうでも開発は進め始めていたのですが、いかんせん DL 方面に進出するには GPU を使える環境が
ないと動作速度の面でも進められません。私の手元には満足に GPU を動かせる環境がなかったため、
開発は遅々として進みませんでした。

そんなこんなで令和３年を迎え、芝さんから「今年どうします？」と DM が飛んできました。
芝先生は DL 勢として「二番絞り」を開発していましたが、私はそちらの開発には一切関わっていないため、
そこにチームとして顔を出すのは違うよなぁ……芝先生単独でどうぞという話をしておりました。
しかし、話していた当時手をつけたばかりで強いのかどうかもわからない「二番絞り」で一次予選免除は
どうなの？新しいエンジンで出るなら一次予選から戦いたいという芝さんの思いもあり、別チームで
新人として出場したいと思っていると仰いました。
そうなると白ビールはどうなっちゃうの？ということになり、せっかくのシード権はもったいないよねと
いうことで、こちらは私が引き継ぐ形で出場することになりました。

２．今回のコンセプト

白ビールは前回準優勝で NNUE 形式の評価関数最上位であるので、DL 勢を迎え撃つ壁とならなければなりません。
ですので「負けるにしても NNUE の旗頭としてしっかり立ち塞がり、しかと引導を渡されるという演出も必要
だよね？」ということで、探索部は最新のやねうら王 V6.02 としつつも評価関数はあえて第 29 回準優勝の
Kristallweizen でいくことにしました。

もし白ビールがこの構成で善戦してしまったとしたら、「俺たちの２年間はなんだったんだろう……」という
ネタも仕込まれていました。

３．ハードウェア構成

・USI 制御、マルチポンダー部　　Windows10 PC
・オーダー部　　？？？
・ポンダー部　　AWS C5.metal.24xlarge X 5 インスタンス

４．定跡ファイルについて

Floodgate の 2020/1/1～2021/5/2 の対局のうち、R4000 以上同士の対局から、先手、後手それぞれ勝った方の指し手のみを抽出し、そのまま定跡化しています。
この定跡化は 30 分おきに自動で取得～追加を行うようにしています。(現在も増加中です)

R4000 以上同士にしたのは、これよりも下のクラスであれば、定跡がなくとも勝てるという実績があったためです。実績がある評価関数を使っているからこその選択です。

また、DL 勢の読み筋が１本道になりやすいということがわかっていたため、わざと手を散らすような処置を施していない場合に、詰み局面までノータイムで誘導してしまうこともあり、なかなか侮れないものに仕上がりました。

５．参加してみての感想

２．で書いたようなコンセプトで出場したため、二次予選、まさかの８勝０敗１分で１位通過できるとは思いもしませんでした。
DL 勢と直接対決することを楽しみにしていたのですが、対局を望んでいたみざうら王 with お多福ラボ、dlshogi with GCT、そして同門？の二番絞りと対局できず、非常に残念でした。
DL 勢の中で対局できたのは準優勝した PAL でしたが、なんと２勝０敗であったこともビックリです。
そして負けたのが同じ NNUE 勢だったということで、「ん？負ける相手ちがくね？」というなんともビミョ～な心境です。

次回こそは「脱 NNUE」を目指して頑張りたいと思います！

以上

Daigorilla 開発文集

田中大吾

◆独自の工夫点

独自に工夫した点 DG 電竜 1 から水匠と DG.book を利用し約 5 億程度学習

●定跡

独自の作成したもの(自分で作成した）から約 30 万棋譜（DGVS 水匠）を生成し、DG の評価関数で

Depth28～34

Max62 手

Multipv 2～3

以上を条件にやねうら王で生成した。

それをまた再び DG にセットし棋譜生成をし,その棋譜を利用し再びテラショックを作成

以上を可能な限り loop した。（floodgate の棋譜もある程度利用）

これはやねうら王のテラショック定跡生成コマンドで定跡を使いながら loop すれば効率が良いと考えましたが、棋譜の系統が全く異なるため、自己対戦で生成した棋譜は DG で、Floodgate の棋譜は水匠 3kai でそれぞれ生成し merge した。

▶ 対局条件は

32threads

USI_Hash=8192MB

nodes=15000000

nobook/DG_test1.book/test2.book…

投了スコア 500　62 手まで　VS YO6.00/Suisho3kai

対局環境 E5 2698v4(2 ソケットを＊2 利用）

●評価関数

①教師生成

今回の評価関数も HKPE9 を選択した。理由としては、まだまだ余地があると考えている。学習パラメーターの限界が標準型の約 10 倍となるため、教師の質も十分に必要だが、定跡の作成に力を入れた後、定跡を浅い Depth(14)と深い Depth(26？)の教師生成時に上手く利用した。

②学習パラメーター

▶ HKPE9 は学習が非常に難しかった Eta や lambda の調整が非常に必要であった。

▶ eta=0.03　lambda=0.5 程度で行った。

余り eta を低くすると強くならなかった。

◆開発動機

最初に出場したのは、2 年前の WCSC29 の時であった。

開発動機としては大学の論文で人と AI の共存のなか評価関数を育成し､弱点などを観察し、どのように使用していくのが良いかと研究したのがきっかけであった。そもそも僕は理系ではないので、コンピュータ関連の扱いにとても苦戦したがある程度勉強もした。

後も趣味とて続けている。

◆開発過程・実験結果

今回は電竜戦に引き続き表現の限界がそろそろ迎えている標準型 NNUE を使用せず HKPE9 を質を上げたいと思った。

実験環境

- ▶ 投了値　500　 1 手 3.5 秒
- ▶ 対局数　100
- ▶ Threads 80　（E5-2698 v4 40C 2 ソケット）
- ▶ USI_HASH 8192MB
- ▶ ノード数平均（NPS）DG_hkpe9/yo.6.00 2500 万　 水匠/yo.6.00　3100 万

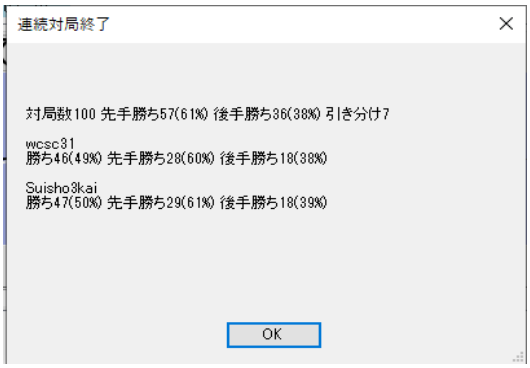
11 月　DG_denryu1 R4500 程度

DG_denryu1- 55 -1- 44 水匠 2

3 月　DG 210311 R4670 程度

DG_denryu1- 42 -0- 58 水匠 3

4 月　DGwcsc31 R4750 程度



◆追試試験　現状

現状新しい NNUE の特徴量での 0 からの開発をスタートしている。

玉の安全性も付与した特徴量で NPS は標準型の役 6 割。

今後もコンピューター将棋界を発展に貢献したいと思います。

よろしくお願いします。

大将軍 詳細アピール文書

横内健一・横内靖尚

概要

昨今のコンピュータ将棋において評価関数は劇的な変化を遂げてきている。Bonanza で始まった従来の 3 駒関係をベースとした評価関数から NNUE を経て、最新では Deep Learning を用いた評価関数も登場し、その精度の高さは家庭用 PC を用いた環境でも実用域に達していると考えられる。今回の大将軍は、そのようなトレンドのなか、評価関数については今更ながら従来の 3 駒関係（15 年前の技術）をベースに大会に参加することとした。

開発動機

大将軍は以前に N4 や N4S という名前で大会に参加していたことがあるが、4 駒関係を用いた評価関数を使用していた。その際の経験では、序盤はそこそこ戦えるものの、中盤以降、深い読みができず、敗戦するケースが多かった。評価関数の表現力を高め大局観の精度を高めることは大事ではあるものの、将棋の場合、最終的には詰む・詰まない、を正確に読み切ることが重要であり、具体的な手順を深く読む必要がある。そのため将棋では 4 駒関係のような重い評価関数は不向きとの考えがあった。
そこで、今回はあえて比較的軽い従来型の 3 駒関係の評価関数と最新の探索アルゴリズムを組み合わせた場合、どの程度大会で戦うことができるかをモチベーションに大会に参加することにした。

開発過程

今回の参加にあたっては、3 駒関係をベースにさらに新たな特徴を加えた 3 駒＋特徴 α で学習をテストした。いくつかのモデルを作成し、学習を行った結果、同じ NPS であれば、3 駒のみのモデルよりも強い評価関数を作成することはできた。
しかし、特徴の追加により評価値の計算コストが高くなるため、結果としては 3 駒のみのモデルよりも弱くなってしまったことから、従来の 3 駒のモデル（kkpt-kpp）で大会に参加することとした。
学習については、基本的には、以下の条件で実施した。

学習用の棋譜生成

主なパラメータ　探索深さ 6 or 10、 投了スコア 5000 or 詰み

正確なデータ取りはしていないが、探索深さは6よりも10、投了スコアは5000よりも詰みの値を設定し、詰みまでの棋譜を生成したほうが、勝率が高い評価関数が作成できるようである。また、学習時の学習率などは一致率をみながら適時調整した。

独自に工夫した点

3駒関係のモデルではkppt型（手番については玉とその他の2駒）が一般的であるが、大将軍では従来からkkpt-kpp型を採用している。これは手番に関する評価をkkp（自玉、敵玉、その他の駒）に対して評価するものであり、計算量が少ないことが利点といえる。また、これは過去からの工夫で、今では常識の技術ではあるが、評価関数の差分評価（動いた駒のみの評価値を更新）をいち早く取り入れて採用している（4駒関係の計算では、すべての駒の配置に対して評価値を計算すると組み合わせが多く膨大な計算量が必要となるため、差分計算を取り入れると3駒分の計算量で済む経験を利用）。

使用ライブラリとその選定理由

やねうら王の選定理由
　ソースコードがわかりやすく、ベースのエンジンとして使用
水匠2の選定理由
　学習の棋譜生成に使用（強いソフトで生成した棋譜を利用）

実験結果

対 水匠2　 4スレッド 4sec
対局数500（先手後手同数）平手局面
　対局結果
　　100勝　363敗　37引き分け（勝率0.216）

　3駒関係に対する学習では、NNUEベースである水匠2の棋譜を用いても、その優秀性に対抗することはできなかった。

追試可能か

学習は、教師局面をランダムにシャッフルしているため、同じ結果を得ることは難しい。

以上