

1 Ray について

OSS 版の Ray[1] に以下のような改良を施してあります.

1. 木探索の着手評価に Factorization Machines + Bradley-Terry モデルを利用
2. 木探索部とシミュレーション部に特徴を追加
3. 木探索部, シミュレーション部ともに細かなパラメータ調整
4. 深層強化学習

2 Deep Learning 対応

2.1 利用する深層学習フレームワーク

ニューラルネットワークの学習には Pytorch を利用しています.

2.2 ニューラルネットワークアーキテクチャ (去年から変更あり)

2.2.1 Bottleneck 型 Residual Block

ニューラルネットワークの共通部は

1. Convolution layer (N 3x3 filters).
2. M Residual Blocks.
3. Batch normalization layer.
4. Swish activation layer.

となっており, 8 の倍数番目の Residual Block は図 1, それ以外の Residual Block は図 2 で構成されています. これは Gumbel MuZero で提案されていた Residual Block を少し組み替えたものになります.

2.2.2 Nested-Bottleneck 型 Residual Block + SE Block (新規対応部)

途中から図 3 に示す KataGo[3] 方式の Nested-Bottleneck Residual Block に Squeeze-and-Excitation module をつけた Residual Block に, 6 の倍数番目の Residual Block を図 1 に置き換えたものを使った Residual Network をニューラルネットワークの共通部に使っています.

2.2.3 ニューラルネットワークの出力

出力は以下の 5 つありますが, 対局時には Player's Policy, Value, Ownership, Score の 4 つを利用します.

- Player's Policy : 次の自分の着手の予測
- Opponent's Policy : 自分が着手した後の相手の着手の予測
- Value : 最終的な勝敗の予測
- Ownership : 各交点の占有率
- Score : 最終的な目数差の予測

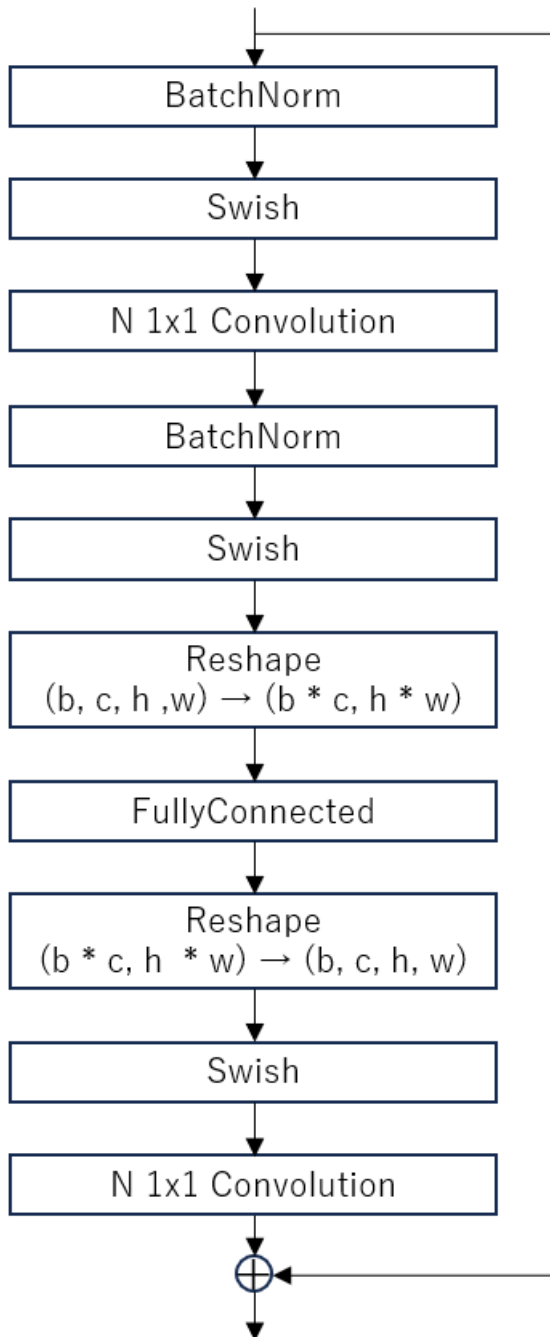


图 1 Broadcast Residual Block

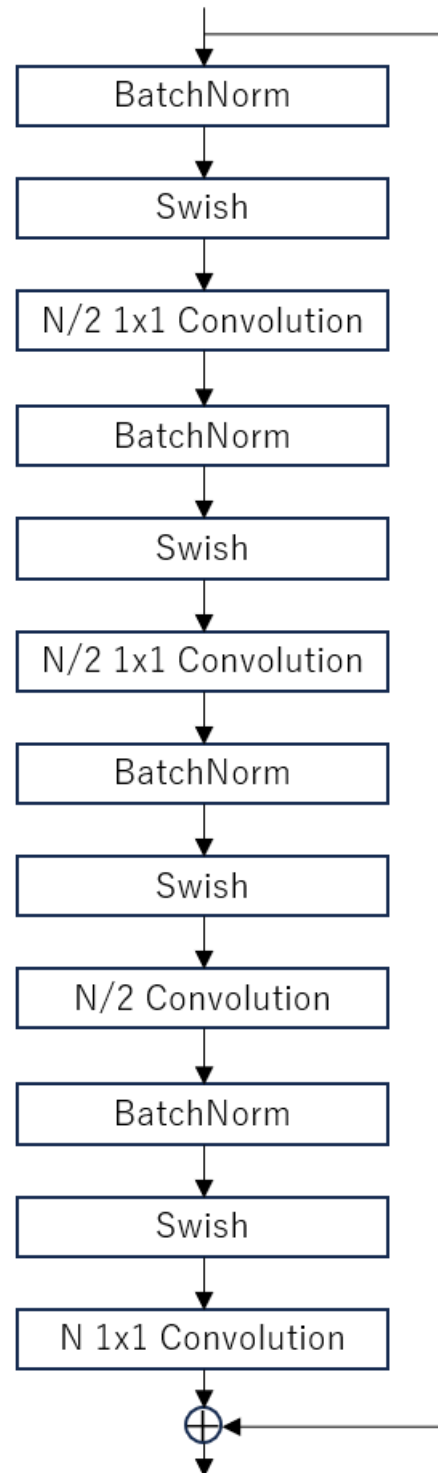


图 2 Bottleneck Residual Block

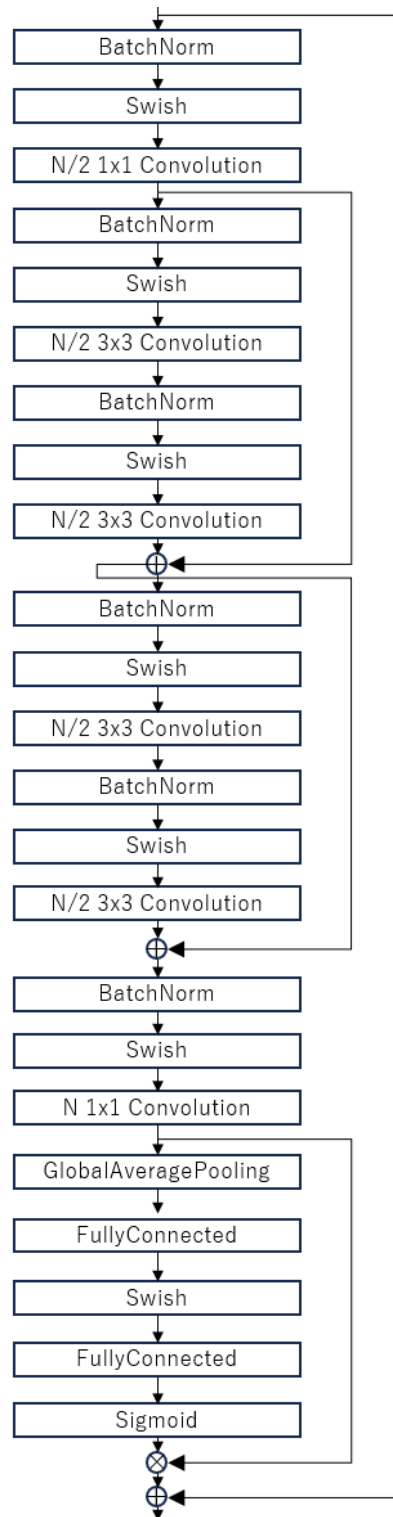


图 3 Broadcast Residual Block

入力には石の配置や呼吸点数, 着手の履歴以外に

- 対局ルール：日本ルールか中国ルールか Pass-Incentive ルールか (中国ルール + 先にパスした方に 0.5 目加算)
- コミの値：コミの値の 0.05 倍
- 持碁の有無：持碁ありの時は 1, 持碁なしの時は 1
- パスした回数：自分がした場合は 1 回あたり 0.1, 相手がした場合は -0.1 加算

を入れることで, いろいろな棋譜を混ぜて学習できるようにしています. 入力特徴は合計で 60 面あります.

2.3 対局時に利用するニューラルネットワークの実装 (去年から変更あり)

対局プログラムは C++ で実装されているため, 前回までは CUDA, cuDNN, cublas の直接呼び出す方式でニューラルネットワークの計算を実行していましたが, 今回からはニューラルネットワークの推論を TensorRT で行うように変更しました.

2.3.1 圧縮した入力データの送信

入力特徴をそのまま GPU に送信すると

$$19 \times 19 \times 60 \text{planes} \times 4 \text{bytes} = 86640 \text{bytes} \quad (1)$$

となり, 1 バッチごとに約 85KB (16bit float で約 42KB) のデータ送信になり, データ転送量が思ったより多くなります.

入力特徴は

- One-Hot Encoding で表現する特徴 (56 個)
- 実数値で表現する特徴 (4 個)

のいずれかになっており, 実数値で表現する特徴は 2 つとも各交点に同じ値を代入するため, One-Hot Encoding で表現する特徴を 8byte の整数型のデータにビット列としてまとめ, 実数値の特徴は 2 つの単精度小数のデータとすることで

$$19 \times 19 \times 8 \text{bytes} + 2 \text{bytes} \times 4 = 2896 \text{bytes} \quad (2)$$

と元のデータサイズの約 3.3% 程度に圧縮して送信し, GPU のメモリ上で入力データを形成するようにしています.

GPU の計算力に依存しますが, この工夫によって 5~8% 程度ニューラルネットワークの計算が速くなる効果が有りました.

2.3.2 TensorRT 対応 (新規対応部)

ニューラルネットワークの推論を TensorRT で実行するようにしています. 具体的には下記順番で処理を行い, TensorRT を利用できるようにしています.

- PyTorch で学習したモデルファイルを ONNX 方式で出力
- ONNX 形式のファイルを読み込み, ニューラルネットワークをコンパイル
- コンパイルしたニューラルネットワークをキャッシュファイルとして出力

2 回目以降の起動はキャッシュファイルを読み込むことで高速に起動できるようにしています。

2.4 ニューラルネットワーク計算結果反映前の Policy (新規対応部)

ニューラルネットワークをミニバッチで推論する都合上、実際の探索時に Policy の反映が間に合わない状態が発生します。昨年度までのバージョンでは旧来の Factorization Machines + Bradley-Terry Model で学習した重い特徴を利用していたのですが、対局中盤になると主にシチョウ探索や攻め合い絡みで着手評価の計算の時間がかかり、ボトルネックになっていました。一方でこの部分の着手評価をある程度正確しておかないと、探索する手がバラバラになり、弱くなることを確認しています。

そこで Policy 反映までに使用する特徴をを軽量な特徴に置き換え、対局中盤でも探索速度を維持できるように変更しました。旧来の重い特徴を使ったときと同じ強さを保っていることを確認済みです。

3 深層強化学習

3.1 学習環境とマシンリソース

Jenkins を利用して学習と棋譜生成、更新したニューラルネットワークパラメータの配置等を自動で回すように環境を構築しました。たま～に熱暴走などでマシンがハングすることがありますが、自動的にサイクルが回るのでボタンひとつでゼロからの強化学習ができます。

マシンは個人所有の下記 3 台を利用しています。電気代が大体 40,000 円かかる構成です。

表 1 使用したマシン構成

マシン番号	OS	CPU	Memory	GPU	用途
1	Ubuntu 22.04	Core i7-6900K	16GB	GeForce RTX 5060 Ti	Jenkins Server & 自己対戦
2	Ubuntu 20.04	Ryzen 9 3900X	64GB	GeForce RTX 2080 Ti	自己対戦
3	Ubuntu 20.04	Core i9-10850K	64GB	GeForce RTX 3090	学習

表 2 使用したマシンのソフトウェア構成

マシン番号	CUDA	cuDNN	TensorRT
1	12.9	8.9.7	8.5.3.1
2	11.8	8.3.1	8.5.3.1
3	11.8	8.9.7	8.5.3.1

3.2 AlphaZero と Gumbel AlphaZero の 2 つの学習を採用

AlphaZero 方式の強化学習と Gumbel AlphaZero[5] の 2 方式の強化学習を混ぜて行っています。2 つの手法の主な違いを下表に示します。

ここで重要なのが Policy の損失関数が異なる点ですが、どちらの損失関数を使用するかをフラグを付与することで、1 つのミニバッチの中に 2 手法のデータが混合しても問題なく学習できるようにしています。

ただしここ最近では以下の観点から Gumbel AlphaZero を使わなくなっています

表 3 AlphaZero と Gumbel AlphaZero の比較

項目	AlphaZero	Gumbel AlphaZero
探索手法	PUCT	SHOT
Policy のターゲット	探索回数の分布	Improved Policy
Policy の損失関数	Cross Entropy	Kullback-Leibler Divergence

- ニューラルネットワーク推論キャッシュの使用率が低い：自己対戦の棋譜生成速度が遅い
- 同じ時間探索させて対局すると PUCT より SHOT の方が弱い：Value のターゲットの質が悪い？

3.3 ニューラルネットワークの段階的な拡張 (去年から更新あり)

最初から大きなニューラルネットワークを使用すると学習のサイクルの回転速度が落ちるため、強化学習の効率化のために小さなネットワークから学習をめて、徐々にネットワークの大きさを大きくして学習させました。本手法については囲碁では LeelaZero や SAI, KataGo が利用しており、将棋では AobaZero も採用しています。

ネットワークの規模と生成した棋譜、探索回数については下表のとおりです。ネットワーク規模は Policy Head と Value Head の共通部の Residual Block の個数と Residual Block 内の Convolution Layer のフィルタの個数を表しています。

UEC 杯当日までに 5,000,000 棋譜生成できている見込みです。

表 4 ニューラルネットワークの規模と生成した棋譜数、探索回数の関係

ニューラルネットワークの構成	AZ と GAZ の混合比率	1 手あたりの探索回数 (AZ)	1 手あたり探索回数 (GAZ)	生成した棋譜の数	生成した棋譜の合計数
96 filters x 8 blocks	0 : 1	-	50	500,000	500,000
128 filters x 16 blocks	1 : 1	200	50	1,000,000	1,500,000
192 filters x 16 blocks	1 : 0	800	-	800,000	2,300,000
256 filters x 24 blocks	1 : 0	800	-	1,700,000	4,000,000
256 filters x 18 blocks	1 : 0	800	-	800,000	4,800,000

最後の 256filters x 18 blocks のみ Nested-Bottleneck 型 Residual Block + SE Module を使っており、それ以前のニューラルネットワークは Bottleneck 型 Residual Block を使っていました。去年の UEC 杯版は ELF Open Go v0 と互角程度でしたが、最新版は ELF Open Go v2 に約 70% 勝てるようになっています。

3.4 Stochastic Weight Averaging

1 世代学習するごとに 1 つ前の世代のネットワークパラメータとの重み平均を取るようにして移動平均を取るような Stochastic Weight Averaging を採用しています。これにより初期の学習の安定化に加えて、Score や Value の学習がよりよく進んでいるように見えています。本方式を使うにあたって、ネットワークパラメータの重み付き平均化の処理と Batch Normalization Layer の統計情報の再計算を実装しました。

3.5 CGOS 解析コマンド対応

CGF オープン 2023 で試しに実装した CGOS 解析コマンドの Ownership や PV の表示がおかしくなっていたのでそのバグを直して Web Viewer から見て楽しめるようにしています.

4 最後に

皆さんと現地でお会いできることを楽しみにしています.

参考文献

- [1] kobanium (Yuki Kobayashi), <https://github.com/kobanium/Ray>
- [2] kobanium (Yuki Kobayashi), <https://github.com/kobanium/TamaGo>
- [3] lightvector (David J. Wu), <https://github.com/lightvector/KataGo>
- [4] <http://entcog.c.ooco.jp/entcog/contents/lecture/igoAI2023.html>, コンピュータ囲碁講習会 2023
- [5] I. Danihelka, A. Guez, J. Schrittwieser, D. Silver, *Policy Improvement by planning with Gumbel*
<https://openreview.net/forum?id=bERaNdognO>