リバーシの評価関数の最適化

Seal Software

1　はじめに

今から 10 年ほど前、LOGISTELLO というオセロプログラムがリバーシプログラミング界を大きく変えました。LOGISTELLO はとにかく強く、1997 年には人間のチャンピオンであった村上氏を完膚無きまでに叩きのめしました。

LOGISTELLO の卓越した評価関数は論文で詳しく解説され、それを元に実装された強いプログラムもたくさん公開されています。しかしながら、その原理と実装について日本語で解説した文書は私の知る限りこれまで存在しませんでした。そこで、私が理解している範囲で、LOGISTELLO[1]で用いられている「パターンの線形和による評価関数」の作り方について、原論文及び筆者の実装「Thell 3.0[2]」を元に解説します。記述の誤り、不足等に対する指摘は大歓迎です。

なおこの文書は勢いで書いたものですので、必要に応じて追記・削除・訂正される可能性があります。

2　リバーシの評価関数について

2.1　線形和による古典評価関数

おそらく LOGISTELLO 以前から、「ある特徴の値に重みをかけたものの線形和を評価値とする」という形の評価関数は広く使われていました。すなわち、特徴 (feature) を f、重みを w としたときの評価値 e は

$$e = \sum_{i=1}^{N} w_i f_i \tag{1}$$

と表されます。

特徴 (feature) とは例えば

- 着手可能手数
- 確定石の数
- 隅の石の数
- 開放度

といったものです。こうした「オセロの戦略本に載っていそうな経験的知識」に基づいて構成された評価関数を、ここでは便宜的に「古典評価関数」と呼ぶことにします。

[1] 正確には LOGISTELLO2
[2] http://fujitake.dip.jp/sealsoft/thell/

古典評価関数の重み係数は、従来人の手によって経験的・実験的に決められていました。それでも、適切な特徴を適当に重みづけしてやることで、それなりに強い (初心者にはまず負けない) 思考ルーチンを作成することが可能です。

筆者の実装においても、Thell 2.x はこの古典評価関数を利用しています。Thell 2.x は 1〜2 級の実力であると評されています[3]。

2.2 線形和による統計評価関数

LOGISTELLO2 が導入した評価関数は、盤面全てをカバーする「パターン」に全て得点をつけ、それらを線形和によって合計するというものです。重み係数は回帰分析によって最適化され、極めて正確な盤面評価をすることができます。

LOGISTELLO2 が用いるパターンは以下の通りです。それぞれのパターンがどのような形かは論文を参照してください。

- horizotal 2 × 2
- horizotal 3 × 2
- horizotal 4 × 2
- vertical 2 × 2
- vertical 3 × 2
- vertical 4 × 2
- diagonal 4 × 4
- diagonal 5 × 4
- diagonal 6 × 4
- diagonal 7 × 4
- diagonal 8 × 2
- edge+2X × 4
- corner2x5 × 8
- corner3x3 × 4

これらのパターンによって、ボード全体の石の配置を高精度で得点化できます。

各パターンはインデックスで識別することができます。「パターンの得点」とは、例えば「edge+2X パターンのインデックス 4515 番は 2.6 石分」とか、「vertical 3 パターンのインデックス 1242 は-0.3 石分」といった数値を指します[4]。

[3] http://uguisu.skr.jp/othello/
[4] 一応ことわっておきますが、値は適当です。

LOGISTELLO2 が導入した評価関数を古典評価関数と対比する意味で「統計評価関数」と呼ぶことにします。LOGISTELLO2 の評価関数は、feature としてパターンを選んでいるということです[5]。

統計評価関数は次のような形をしています。

$$\begin{aligned} e &= \sum_{i=1}^{6561} w_{horz2,i} f_{horz2,i} \\ &\quad + \sum_{i=1}^{6561} w_{horz3,i} f_{horz3,i} \\ &\quad + \sum_{i=1}^{6561} w_{horz4,i} f_{horz4,i} \\ &\quad + \cdots \end{aligned} \tag{2}$$

統計評価関数では、評価値はただの数値ではなく、現実の何らかの数量を近似した値であると考えられます。現在統計的手法を用いているプログラムのほとんどは、以下のいずれかの指標を評価関数の値として算出します。

- 最終的な予想勝率
- 最終的な予想石差

勝率を推定する場合にはロジスティック回帰が、石差を推定する場合には線形回帰が用いられることが多いようです。LOGISTELLO2 では、後者の予想石差を線形回帰によって求めます。

2.3 その他の統計評価関数

LOGISTELLO2 以降、ほとんどのプログラムが線形和による統計評価関数を採用していますが、線形和以外の統計評価関数を採用しているプログラムとして、「vsOtha[6]」があります。

vsOtha では、LOGISTELLO と同じパターンを入力として利用していますが、予想石差の算出に 3 層パーセプトロンを用いている点が異なります。vsOtha の評価関数も、線形和による統計評価関数と同等の精度を発揮します。

2.4 評価関数の学習

以下では、パターンの線形和によって最終的な石差を予想する評価関数の作り方について述べます。

さて、評価関数の学習には大きく分けて 2 つのアプローチがあります。

- 教師つき学習
  いくつかの「入力」とその「正解」をあらかじめ与えることで、与えられた「入力」に対しなるべく「正解」に近い値を出力できるように重み係数を調整する手法です。容易に学習可能ですが、「入力」と「正解」のペアをたくさん用意しなければなりません。
- 教師なし学習 (強化学習)
  例えば「ゲームに勝った」「ゲームに負けた」といった「結果」からパラメタを増減し、重み係数を最適化していく手法です。

[5] より正確に言うと、LOGISTELLO2 はパターンの他に mobility(着手可能手数)、potential mobility(開放度のようなもの？)、parity(空きマスの偶奇？) といった要素を加えて評価値を算出しています。もちろんそれらの値も全て統計的に重みづけされます。

[6] http://homepage2.nifty.com/kazenohoyo/download.htm

リバーシの評価関数学習のためには、ほとんどの場合教師つき学習が用いられます。これは、人間やコンピュータがインターネット上などで戦ってきた棋譜が多量に入手でき、棋譜を教師信号として学習することが容易にできるためです。LOGISTELLO2 をはじめとする多くのプログラム (筆者の Thell も含みます) は、教師つき学習によって評価関数を最適化しています。

これに対して、強化学習のアプローチを選んだプログラムとしては Herakles[7]があります。(正確には、Herakles に搭載されている思考エンジンの 1 つ、”Charly”が強化学習のアプローチを採っています。)

以降この文書では、教師つき学習による評価関数の最適化について詳しく述べます。Herakles の強化学習については、論文”MOUSE(m): A Self-Teaching Algorithm that Achieved Master-Strength at Othello”を参照してください。

2.5 棋譜による学習

教師つき学習においては多数の「入力」とその「正解」を用いて学習を行う、と述べました。評価関数では、「入力」とはある局面、より正確にはその局面におけるそれぞれの特徴の値であり、「正解」とは評価関数が予想すべき数値 (例えば最終石差) のことを指します。

最終石差を予想する評価関数に的を絞って話を進めましょう。そもそも評価関数による評価が「正確」であるとはどういうことかというと、「その局面から双方が最善手を打った場合の結果にどれだけ近い値を出せるか」ということに他なりません。従って、評価関数をよりよくするための教師信号として与えられる「正解」は、「その局面から双方が最善手を打った場合の最終石差」である必要があります。

ですが、果たしてそのような理想的な「正解」を常に与えられるのでしょうか？局面が終盤付近であれば、そこから完全読み切りをすることで正確な最善手が得られます。しかしながら、局面が序盤であった場合等、最善手探索が事実上不可能な場合も多々あります。このような場合はどうすればよいのでしょうか。

実際のところ、棋譜がある程度多ければ、こうした問題はあまり考える必要がありません。棋譜が与える「正解」は本来の最善手とずれている可能性がありますが、棋譜が多ければ、同じような局面において逆方向にずれている局面が存在する確率も高く、誤差が打ち消し合うと考えられるためです。

このため、IOS[8]の棋譜のように相当ひどい打ち間違いを多く含む棋譜で学習させても、かなりの強さの評価関数を得ることができます。

2.5.1 よりよい棋譜

とはいえ、棋譜中に誤りがあるよりはない棋譜の方がよいことは言うまでもありません。「よい棋譜データベース」の条件について考えてみましょう。「よい棋譜データベース」とは…

- 数が多い
  そもそも質がよくても量が少なければ、評価関数を最適化するだけの局面が集まりません。
- 様々な局面を含んでいる
  双方が最善に近い手を打った場合だけでなく、片方に極端に有利になっている状態の局面も

[7]http://www.herakles.tournavitis.de/
[8]Internet Othello Server

必要です。棋譜中に全く出現しなかったパターンについては評価のしようがないので、とにかく様々な局面を集めることが重要です。

- 最善もしくはそれに近い結果がわかる
  強いプログラム同士に対局を行わせると、完全読み切りに入っていなくともそれなりに最善に近い手を打つことができます。これを利用して、ある局面以降をプログラムに打たせて最善手ないしはその近似値を求めておくことで、より正確な棋譜データを学習できます。実際、IOS の棋譜そのものを使うのに対して、訂正した棋譜を用いると評価関数精度が向上することが筆者やなるめる氏 (vsOtha 作者) によって確かめられています。

3 線形回帰分析

(略; あとでちゃんと書く？)

4 反復法による最小二乗化

実は、線形回帰式の最小二乗化は直接的に解けることが知られています[9]。ところが、リバーシの評価関数に登場する回帰分析は説明変数が数十万と非常に多く、データの数 (局面数) も膨大であり、かつ非常に疎であるという特徴があり、メモリ上に全てのデータを格納して直接解法を試みるのは困難なサイズである場合がほとんどです。

そこで、ここでは反復法によって段階的に重み係数を最適化していくという手法について解説します。Buro の論文で述べられていることはまさにこの手法に他なりません。

4.1 定式化

回帰分析、つまり最小二乗化とは、次のような最適化問題に他なりません。

- 目的変数を y_i、N 個の説明変数からなるベクトルを $\mathbf{x_i}$、N 個の重み係数のベクトルを $\mathbf{w_i}$、M を全データの個数とする。$\mathbf{x_i} = (x_{i1}, x_{i2}, \ldots, x_{iN})$ 及び $\mathbf{w_i} = (w_1, w_2, \ldots, w_N)$ である。回帰式は $y = \mathbf{w} \cdot \mathbf{x}$ となる。
- 関数 f を $f(\mathbf{w}) = \sum_{i=1}^{M} \left(y_i - \sum_{k=1}^{N} w_k \cdot x_{ik} \right)^2$ と定義する。関数 f は回帰式の誤差の二乗を表している。
- 最小二乗化とは、f を最小化するベクトル $\mathbf{w}$ を求めることである。

4.2 最急降下法

反復法による関数の最適化手法として最も単純な最急降下法について説明します。

最急降下法とは、ある点での関数の勾配方向 $-\nabla f$ を計算し、$x^{(k+1)} := x^{(k)} - \alpha \nabla f(x^{(k)})$ とすることで段階的に解 x を真の解に近づけていく手法です。α は学習係数、ステップサイズなどと呼ばれる係数で、1 回の反復でどれだけ値を変化させるかを示すパラメタです。

[9] 詳しくは「QR 分解」などの用語を調べてみるとよいでしょう。

この文書は最適化法の教科書ではないので、最急降下法の収束性の証明などは省略します。興味のある人は参考文献をあたってみるとよいでしょう。

さて、最急降下法の最小二乗化問題への応用を考えます。最小二乗化問題とは、

$$f(\mathbf{w}) = \sum_{i=1}^{M} \left(y_i - \sum_{k=1}^{N} w_k \cdot x_{ik} \right)^2 \tag{3}$$

を最小化するベクトル $\mathbf{w}$ を求めることでした。f を最急降下法で最適化してみましょう。まず ∇f は

$$\nabla f(\mathbf{w}) = \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_N} \right) f(\mathbf{w}) \tag{4}$$

と表せます。j 番目の重み w_j に関する f の偏微分を計算すると、

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{M} 2 \left(y_i - \sum_{k=1}^{N} w_k \cdot x_{ik} \right) (-x_{ij}) \tag{5}$$

となります。ここで、真ん中の項 $\left(y_i - \sum_{k=1}^{N} w_k \cdot x_{ik} \right)$ は i 番目のデータに対する残差に他なりません。そこで、式を見やすくするために

$$r_i = y_i - \sum_{k=1}^{N} w_k \cdot x_{ik} \tag{6}$$

と置きます。r_i を用いて再び式 (5) を書き直すと、

$$\frac{\partial f}{\partial w_k} = -2 \sum_{i=1}^{M} r_i x_{ij} \tag{7}$$

となり、

$$\nabla f(\mathbf{w}) = \left(-2 \sum_{i=1}^{M} r_i x_{i1}, -2 \sum_{i=1}^{M} r_i x_{i2}, \ldots, -2 \sum_{i=1}^{M} r_i x_{iN} \right) = -2 \sum_{i=1}^{M} r_i \mathbf{x_i} \tag{8}$$

が得られます。最急降下法の更新式としては、$\mathbf{w^{(k+1)}} = \mathbf{w^{(k)}} - \alpha \nabla f(\mathbf{w}^{(k)})$ に従って

$$w_j^{(k+1)} = w_j^{(k)} + 2\alpha \sum_{i=1}^{M} r_i x_{ij} \tag{9}$$

としてやればよいことになります。

4.3 その他の反復改良法

最急降下法は最も単純ですが、収束も遅いという欠点があります。最急降下法以外の反復改良法としては以下のような手法が広く知られています。

- ニュートン法
- 共役勾配法

これらの手法は収束においては最急降下法よりも高速ですが、そのかわり一回の反復により多くの計算を要します。評価関数の最適化のためには最急降下法でも十分実用的なため、ここでは解説しません。

5　評価関数への応用

前節で説明した最急降下法による最小二乗化を用いて、リバーシの評価関数を最適化する手法について説明します。

i 番目 $(1 \leq i \leq N)$ の局面データを p_i、その局面からの最終石差 (教師信号) を e_i とします。一方、最適化途中の評価関数によって予測される石差を $\bar{e_i}$ とします。

$$\bar{e_i}(\mathbf{w}) = \sum_j w_j f_j(p_i) \tag{10}$$

ただし、j は (feature, パターン No.) の組を表し、w_j はその (feature, パターン) の組に対する重み (得点)、$f_j(p_i)$ は局面 p_i において特徴 j の出現する回数を表します。次に、二乗誤差 E を以下のように定義します。

$$E = \frac{1}{N} \sum_{i=1}^{N} (e_i - \bar{e_i}(\mathbf{w}))^2$$

E は重み係数ベクトル $\mathbf{w}$ の関数と見ることができます。従って式 (9) から、

$$w_j^{(k+1)} = w_j^{(k)} + 2\alpha \sum_{i=1}^{N} (e_i - \bar{e_i}(\mathbf{w}^{(k)})) f_j(p_i) \tag{11}$$

として重み係数を最適化すればよいことになります。

5.1　α の取り方

原理的には式 (11) を用いて評価関数の最適化ができるわけですが、実際に学習プログラムを書くためにはいくつかの注意点があります。

まず、学習係数 α の取り方ですが、式 (11) を見ての通り N(=局面データの個数) が多くなるほど 1 回の更新幅が大きくなっていきます。このため、

$$\alpha = \frac{\beta}{N} \tag{12}$$

として定数 β を与えることで、局面数に依存しない幅で重みを更新するようにします。

5.2　計算の高速化

式 (11) の通りに

1. for j
   (a) $\sum_{i=1}^{N} (e_i - \bar{e_i}(\mathbf{w}^{(k)})) f_j(p_i)$ を計算
   (b) w_j を更新

とするのは非常に無駄が多い計算方法です。なぜなら

1. $\sum_{i=1}^{N} (e_i - \bar{e_i}(\mathbf{w}^{(k)}))$ の値を何度も計算する
2. $f_j(p_i)$ はほとんどの場合 0

だからです。かわりに次のように計算を行います。

1. $\forall j, d_j := 0$
2. for i
 (a) $e = \sum_{i=1}^{N}(e_i - \bar{e_i}(\mathbf{w}^{(k)}))$ を計算
 (b) 局面 p_i に出現する各特徴 f_j について $d_j := d_j + ef_j(p_i)$
3. 全ての特徴 j に対して、$w_j := w_j + \frac{2\beta}{N}d_j$
4. 1. に戻る

2.(b) において「局面 p_i に出現する」特徴のみに絞っていることが重要です。1 つの局面に出現する特徴は高々40 程度であり、全特徴 (10 数万) を処理するのに比べて圧倒的に高速になります。「評価関数の最適化に出現する最小二乗化問題は非常に疎である」とは、つまりこのことです。

5.3 収束を速める工夫

Buro 氏の論文では、さらに収束を速めるため、α の取り方を

$$\alpha_j = \min\left(\frac{\beta}{50}, \frac{\beta}{N_j}\right) \tag{13}$$

とすることが提案されています。j は各特徴、N_j は特徴 j が出現した回数を表します。つまり $alpha$ をそれぞれの特徴の出現頻度にあわせて正規化してやろうということです。50 で割ったものとの小さな方を採用しているのは、出現頻度の少なすぎるパターンの重みが急激に変化するのを防ぐためです。

筆者はこの式を

$$\alpha_j = \min\left(\frac{\beta}{100}, \frac{\beta}{N_j}\right) \tag{14}$$

に変えて用いています。

5.4 α の値と収束性

実は、最急降下法において最も効率よく収束を行わせるための α の値の取り方は決まっています。ある点 $\mathbf{x}$ における探索方向 $\mathbf{d} = -\nabla f$ が与えられたとき、

$$f(\mathbf{x} + \alpha\mathbf{d}) \tag{15}$$

を最小にする α を選んで次の点を決定してやればよいことが知られています。この問題 (15) を直線探索問題と言います。

これを解いて α を求めてもよいのですが、直線探索問題を解くためには 1 回の反復とほぼ同じ程度の計算が必要なため、α(上記の例では β) をある定数に固定して計算するということもよく行われます。

注意すべき点は、α(もしくは β) が大きすぎると、計算を反復するごとに解から離れていき、やがて発散してしまうことがあるという点です。実際の計算においては、直線探索問題を解いてステップサイズを正確に定めるか、もしくは反復のたびに誤差 (二乗誤差) を監視しておき、誤差が増加したらステップサイズを小さくしてやりなおす、などの工夫を施すとよいでしょう。

5.5 ステージ分け・スムージング・対称形・その他

(あとでちゃんと書く……というかかつて Web に書いた)

6　まとめ

線形和による統計評価関数の設計とその最適化について、ざっと解説を試みました。読者のリバーシプログラミングの参考になれば幸いです。

なお、評価関数についての別の側面からの解説、ボード構造や探索アルゴリズムについての話題、その他の参考文献などについては、Thell のアルゴリズム解説のページ

- http://fujitake.dip.jp/sealsoft/thell/algorithm.html

に記しています。

7　参考文献

1. M. Buro, *Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello* , Games in AI Research, H.J. van den Herik, H. Iida (ed.), ISBN: 90-621-6416-1, 2000
2. 田村 明久・村松 正和,「最適化法」, 初版, 共立出版 2002
3. 栗田 多喜夫,「サポートベクターマシン入門」, http://www.neurosci.aist.go.jp/ kurita/lecture/svm/svm.html
4. Gunnar Andersson, 奥原 俊彦訳, 「強いオセロプログラムについて」,http://www.amy.hi-ho.ne.jp/okuhara/howtoj.htm