

卒業論文

ゲーム「2048」のプレイヤーについて

08-152021 金澤望生

指導教員 山口和紀 教授

2018 年 1 月

東京大学教養学部学際科学科総合情報学コース

概要

インターネットブラウザやスマートフォン上で遊ぶことのできるパズルゲーム「2048」をプレイする AI の改良を行った．改良には盤面上で最も大きな数のタイルが隅にあることを重視する独自のヒューリスティック「corner bonus」を使用した．(仮)

キーワード ゲーム AI , 機械学習

目次

第 1 章	導入	1
1.1	モチベーション	1
1.2	2048 について	1
1.3	2048 のルール	3
第 2 章	先行研究の紹介	6
2.1	Szubert & Jaskowski (2014)	6
2.2	Wu et al. (2014)	9
2.3	Oka & Matsuzaki (2016)	12
2.4	Yeh et al. (2016)	15
2.5	Jaskowski (2017)	17
第 3 章	本研究のアイデア	22
3.1	base 方策	22
3.2	array 方策	23
3.3	maximum-in-corner 方策	23
第 4 章	提案と実装	25
4.1	maximum-in-corner 方策の導入	25
4.2	実装	26
第 5 章	実験	28
5.1	実験 1 : 常に $\rho = 1.2$ とする	28
5.2	実験 2 : 2 つの ρ の値を用いる	29
5.3	実験 3 : 学習率および学習ゲーム数を変更する	31
第 6 章	考察と結論	33
	謝辞	34
	参考文献	35

付録 A	37
------	----

第 1 章

導入

本章においては，2048 を本研究の題材として採用した動機付けを述べるとともに，2048 というゲームのあらましやルールについて説明し，次章以降の展開に対する準備を行う．

1.1 モチベーション

「2048」[1] は，インターネットブラウザおよびスマートフォンなどの端末でプレイすることができるパズルゲームである．非常にシンプルで誰もが直感的にルールを理解できるのに対し，ゲームを理解したり安定して勝ち続けたりすることは非常に難しい [4]2048 は，その絶妙なバランスが評価されて非常に多くの人々が遊ぶようになった．同時に 2048 はゲーム AI 研究の対象としても活発に取り上げられるようになり，近年では強化学習を用いて訓練した非常に強いプレイヤーが複数の研究者から発表されている．[4][5][6][7][8]

これらのプレイヤーの実装手法は汎用性が高く，他のゲーム（特に 2048 に類似したゲーム）や課題に対しても適用することが可能な手法が多く用いられている [7] 反面，人間が 2048 をプレイする際に用いている経験的な知識・方策は明示的に組み込まれていないことが多い．一部の先行研究においては特徴量として人間が注目する指標を組み込んでいる [7] ものの，これらの指標も強化学習によって訓練されるため，経験的な知識が学習によって獲得されるとは言い難い．

そこで，本研究においては，これまでに発表されてきた強化学習で訓練を行う 2048 プレイヤーに対して，人間による経験的な知識を計算機の強化学習とは独立にトップダウン型で組み込むことにより，計算機によってのみ訓練されたプレイヤーよりも強いプレイヤーを実装し，新たな側面から 2048 の自動プレイヤーに改良の余地があることを示すことを目的とする．

1.2 2048 について

2048 が Gabriele Cirulli によって GitHub 上に公開されたのは 2014 年 3 月のことである．[2]Cirulli は 2048 を開発していた当時，「1024」および「2048」*¹ というゲームに熱中してい

*¹ Cirulli 自身が発表した 2048 とは似ているが，ルールが異なる別のゲームである

2 第1章 導入

た．[3] これらのゲームはどちらも Asher Vollmer によるゲーム「Threes」にルールがよく似ている類似作品であるが，Cirulli は当時 Threes の存在については認識していなかった．Cirulli はこれらの Threes に端を発するゲームを改良するべく新たなゲームを開発し，ついに「2048」として公開することとなった．ここで挙げた 1024，Cirulli が参考にした 2048，そして Cirulli 自身が発表した 2048 は全て Threes の類似作品であるものの，これらの作品群の中では Cirulli による 2048 が最も人気を集めており，全世界で 2300 万人以上の人々が 2048^{*2}で遊んだとされる．[3]

2048 はシンプルで直感的にルールを理解できる一方で，マスターすることが非常に難しいゲームであることが特徴となっており，このために多くのプレイヤーを熱中させていると同時にゲーム AI 研究の題材としても多く取り上げられていると考えられる．また，ゲームとしての 2048 について着目すると，2048 については以下のような特徴を挙げることができる：

完全情報ゲームである

2048 は単一の状態から始まり，1 人のプレイヤーが自分の手番で何かしらの行動を起こすことによって状態が遷移し，さらにプレイヤーはゲームの状態を全て把握することができる．したがって，2048 は完全情報ゲームであると言える．

非決定論的ゲームである

2048 は 1 人のプレイヤーによってのみプレイされるゲームであるが，状態の変化の際には乱数によって決定される要素があるため，プレイヤーは次の状態を知ることはできない．したがって，2048 は非決定論的 (nondeterministic) ゲームである．ただし，プレイヤーは次の状態がいくつあり，その状態にどれぐらいの確率で遷移するのか自分で計算することはできる．

明確な終局が予め設定されていない

チェスや将棋といったゲームについては，どのような状態に到達するとゲームが終了するのか規定されており，プレイヤーはお互いに相手を終局に向かわせることを念頭にゲームをプレイする．一方で，2048 には次の状態に遷移することが不可能になったらゲームが終了するというルールはあるものの，その状態に到達するまでは無限にゲームを進めることができる．すなわち，プレイヤーが熟達すればするほど長い時間ゲームを進めることができ，その分プレイヤーの優秀さを示す指標も良くなる．この特徴により，2048 は強化学習やゲーム AI のベンチマークとしてよく機能していると考えられる．

^{*2} なお，ここから単に 2048 と表記した場合，それは Cirulli が開発した 2048 のことを指す

1.3 2048 のルール

1.3.1 用語定義

盤面

2048 を遊ぶのに用いる大きな土台のことで、 $4 \times 4 = 16$ 個のマスが正方形をなすように並べられている。1 つのマスは、空であるか、1 個のタイルが入っているかのどちらかである。1 個のタイルには $2^n (n > 0)$ の数 1 つが書かれている。また、盤面を囲んでいる 4 辺のことを盤面の終端と呼ぶことにする。

初期盤面 (*initial state*)

プレイヤーが 2048 のゲームを始める時には、必ず初期盤面を与えられる。初期盤面とは、全てのマスにタイルが入っていない状態の盤面に対して、2 個のランダムタイルが与えられた盤面のことである。ランダムタイルの発生の仕方はその度に違うため、プレイヤーはゲームごとに異なる初期盤面を与えられ、なおかつその初期盤面を予想・決定することはできない。

終末盤面 (*terminal state*)

プレイヤーが 2048 のゲームを進めた結果、空のマスが 1 つもなく、なおかつ後述するアクション選択が行えない盤面に到達したなら、ゲームを終了する。この時の盤面を終末盤面とする。

ν -タイル

ν という数が書かれたタイルのことを ν -タイルと呼ぶことにする。

ランダムタイル

2048 のゲームを遊ぶ中で、ランダムタイルと呼ばれる新しいタイルが盤面に追加されることがある。ランダムタイルは盤面上の空のマスのどこかに追加される。なお、空のマスが存在しない盤面に対してランダムタイルが追加されるような状況は、2048 のゲーム中に存在しない。ランダムタイルは 2-タイルもしくは 4-タイルであり、2-タイルが追加される確率は 0.9、4-タイルが追加される確率は 0.1 と定められている。ランダムタイルが追加されるマスの位置はランダムに決定される。

報酬 (*reward*) とスコア

2048 における報酬とは、プレイ後の報酬加算の手順で加算される数値のことを指す。また、ゲームが終了するまでに加算された報酬の総和をスコアと呼ぶ。

プレイ

本研究において、プレイという用語を以下のように定義する：

- プレイヤは、自分の手番 1 回につき、1 回のプレイを行うことができる。
- 1 回のプレイは、アクション選択、遷移、ランダムタイル生成の 3 つの手順からなる。
- プレイヤは、1 回のプレイの間、アクション選択を行うことで盤面を操作することができる。なお、プレイヤはアクション選択の結果行われる遷移の後どのような盤面になるのか計算して求めることはできるが、ランダムタイル生成後にどのような盤面になるかを知ることはできない（確率的に予想することはできる）。

学習ゲーム

学習器を訓練するために行うゲームのことで、価値関数などのパラメータの更新を行いながらゲームを進める。ゲームを開始してから終末盤面に到達するまでを 1 ゲームと数え、1 ゲームが終了したら次の学習ゲームに進む。本研究および先行研究においては、1 回の実験あたり 100,000 ゲーム、500,000 ゲーム、もしくは 1,000,000 ゲームの学習ゲームを行うようになり、 n ゲーム学習することに検証ゲームを行うことになっている。本研究では $n = 5000$ としている。一般的に学習ゲームを多く行えば行うほど学習器の性能は良くなるが、特定の状況に特化して学習を行う過学習を引き起こす可能性もある。[7]

検証ゲーム

訓練された学習器を用いてゲームを行う。この際、価値関数などのパラメータは参照するだけで更新は行わない。学習中に検証ゲームを行うインターバルと検証ゲームを行う回数 m は指定されており、本研究では $m = 1000$ としている。

1.3.2 ゲームとプレイヤ評価の流れ

2048 において、以下の手順を「1 ゲーム」と数える。

1. ゲームを開始する。プレイヤは初期盤面を与えられる。ゲーム開始時のスコアは 0 である。
2. プレイを行う。プレイは次の 3 つの手順からなる：
 - (a) アクション選択：盤面上の全てのタイルを動かす方向を、上下左右のうちいずれかから選ぶ。ただし、次の手順で遷移を行っても盤面に変化が生じないようなアクションを選ぶことはできない。
 - (b) 遷移：盤面上の全てのタイルを、選択したアクションの方向へ、他のタイルもしくは終端に接触するまで動かす。もし任意の ν -タイル 2 つが接触したならば、マージを行う。マージを行ってもなお、アクションの方向へ動かすことができるタイルがあるならば、さらに他のタイルが終端に接触するまで動かす。

- マージ：アクションの方向に向かって前方のタイルを 2ν -タイルに変更し，後方のタイルを削除する．値が変更された 2ν -タイルはこの遷移中二度とマージを行わない．同一列において複数組のマージできるタイルがあるならば，マージは前方のタイルを優先して順番に行う．
- 報酬加算：プレイ中にマージが発生した結果，新たに盤面に 2ν -タイルが発生したならば， 2ν を報酬としてスコアに加算する．報酬加算はマージを行った結果新たに発生した全てのタイルについて行う．

(c) ランダムタイル生成：1 個のランダムタイルを盤面上に生成する．

3. 1 回のプレイが終末盤面に到達することなく終了したならば，次のプレイを行う．もし遷移もしくはランダムタイル生成の手順において終末盤面に到達したならば，ゲームを終了する．

1.3.3 2048 で重要な指標

maxtile

ゲームが終末盤面に到達した時点で，盤面上で最も大きな数のタイルを *maxtile* と呼ぶ．近年の 2048 プレイヤの研究の多くは 16384-タイルの作成に成功しており，32768-タイルの作成に成功したプレイヤもある．[8]32768 を達成することが 2048 のプレイヤ実装において大きなメルクマールとなることから，32768-タイルを作成できた割合を指標とすることもある．[8]

勝率

2048 において， $maxtile \geq 2048$ ならば，プレイヤーの勝利であると定義されている．多くの研究が 90% 以上の勝率を達成しているが，ゲーム木探索を組み合わせたプレイヤは勝率は 100% を達成しており，勝率については問題にしないこともある．

スコア

2048 において最も重要な指標がスコアである．プレイヤの学習が終わった後にその性能を検証するゲームを何度か行った際，その検証中のスコアの算術平均を取った平均スコアと，その検証中に最も高かったスコアである最高スコアの 2 つが，プレイヤの性能を示す指標として用いられることが多い．

第 2 章

先行研究の紹介

この章では、2048 プレイヤを実装した先行研究について、その研究が用いた手法と実装されたプレイヤの成績について紹介する。(編注・提出原稿では以下は削除)なお、Jaskowski は Jaśkowski が正しい表記ですが、差し当たり Jaskowski と表記し、最後にまとめて正しい表記に置換します。

2.1 Szubert & Jaskowski (2014)

Szubert & Jaskowski は、TD 学習を用いたプレイヤの訓練と n タプルネットワークを用いた価値関数の表現を組み合わせることによって、人間の知識やゲーム木探索を使用しないで十分強い 2048 プレイヤを実装することに成功した。

2.1.1 TD 学習

TD 学習の「TD」とは *temporal difference* の略であり、すなわち状態間における価値の差分を学習することによって学習器の訓練を行う手法である。TD 学習は Tesauro によるバックギャモンへの適用 [10] でよく知られるようになり、碁 [11][12] やオセロ [13][14]、チェス [15] におけるゲーム AI の方策決定の手法として用いられるようになった。2048 にあてはめると、ある盤面 s の評価値と、その盤面の 1 プレイ後の盤面 s'' の評価値の差分を取り、これを現状定まっている s の評価値に足し込んでいくことで価値関数の更新を行うことになる。2048 プレイヤに対して TD 学習を適用する方法はさまざまなものがあるが、Szubert & Jaskowski が検討した適用の方法は以下の通りである：

アクションの評価 (Q 学習)

平易な価値関数の実装として最初に考えられるのが、アクション a を選択した時の盤面 s の価値を評価する関数 $Q(s, a)$ を学習する Q 学習である。Q 学習において、行動評価値 $Q(s, a)$ は以下のように更新される：

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \max_{a' \in A(s'')} Q(s'', a') - Q(s, a))$$

ここで、 r は盤面 s においてアクション a を選択したことにより得られた報酬、 s'' は盤面 s においてプレイを行った後の盤面、 $A(s)$ は盤面 s において選択可能なアクションの集合である。この Q 学習においては、プレイ後の盤面 s'' における最も高い行動評価値と報酬の和と、現在の盤面 s においてアクション a を選択することの行動評価値の差を取り、学習率 α を掛けて $Q(s, a)$ に足し込むという形で、行動価値関数の更新を行っている。

なお 2048 に Q 学習を適用する場合は、どの盤面に対しても有り得るアクションは高々 4 つしかないで、 s と a の組み合わせにより値を与える 1 つの関数を使用するのではなく、4 通りのアクションごとに価値関数 $V_a(s)$ ($a \in \text{UP, DOWN, RIGHT, LEFT}$) を使用することができる [4]。この場合、アクション a に対する行動評価値 $V_a(s)$ は以下のように更新される：

$$V_a(s) \leftarrow V_a(s) + \alpha(r + \max_{a' \in A(s'')} V_{a'}(s'') - V_a(s))$$

盤面の評価 (TD(0))

次に、TD(0) を利用して盤面 s の価値を評価する手法が挙げられる。TD(0) は最もシンプルな TD 学習のアルゴリズムであり、TD(0) における価値関数 $V(s)$ は次の式のように更新される：

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

一般に、TD(0) によって制御されるエージェントは $r + V(s'')$ を目標とする [9]。したがって、盤面の評価を行う際にはこの値を最大化するようなアクションを選択することになるが、このモデルにおいては $r + V(s'')$ の期待値を最大化するようなアクションを選択する。すなわち、ある盤面 s の価値は以下のように評価され、この値を最大化するアクションを選択することになる：

$$R(s, a) + \sum_{s'' \in S''} P(s, a, s'') V(s'')$$

ここで、 $R(s, a)$ は盤面 s においてアクション a を選択した際に獲得する報酬を与える関数、 S'' は盤面 s においてプレイを行った後に有り得る盤面の集合を、 $P(s, a, s'')$ は盤面 s においてアクション a を選択した後に盤面 s'' が得られる確率を与える関数を表す。

盤面の評価モデルは価値関数として単一の $V(s)$ のみを保持していればよいので、4 つの価値関数を保持する必要があるアクションの評価モデルに対して優位性がある一歩で、1 つのアクションを評価する際にそのアクションの結果得られる全てのプレイ後の盤面を求める必要がある。2048 では最大で 15 個の空のマスが存在し、これらのマスに生成されるランダムタイルは 2 種類で、さらに 4 方向へのアクションを選択できることから、1 回のアクション選択のために最大で 120 個の盤面を求めなければならない。このため、盤面の評価モデルはアクションの評価モデルと比べて実行速度が非常に遅いプログラムとなってしまう。

遷移後盤面の評価 (TD(0))

最後に提案されたのが、アクションの評価モデルと盤面の評価モデルの組み合わせとも言える遷移後盤面の評価モデルである。遷移後盤面の評価モデルにおいては、単一の価値関数 V を用いるが、この価値関数は盤面 s に対して適用するのではなく、 s の遷移後の盤面 s' (afterstate, すなわち遷移を行った後でランダムタイルを生成する前の盤面) に対して適用する。ランダムタイル生成後ではなく遷移後の盤面ならば1つのアクションに対して高々1個しか存在しないため、計算量を抑えることが可能になる。このモデルにおいては、以下の値を最大化するアクションを選択する。

$$R(s, a) + V(T(s, a))$$

ここで、 $T(s, a)$ は盤面 s においてアクション a を選択した際に遷移した後の盤面、すなわち s' を与える関数である。また、遷移後盤面の評価モデルにおいては、 $V(s)$ の更新を連続した2つの遷移後盤面の評価値を用いて行い、以下のように更新されることになる。

$$V(s') \leftarrow V(s') + \alpha(r_{next} + V(s'_{next}) - V(s'))$$

ここで、 s'_{next} とは s の1プレイ後の盤面 s'' の遷移後盤面として最も評価値が高い盤面を、 r_{next} は s'_{next} への遷移によって獲得した報酬を表す。

Szubert & Jaskowski による実験の結果、以上の3つのモデルのうち最も性能が良いのは遷移後盤面の評価を行うモデルであった。以降、本研究および先行研究において単に TD 学習と表記した場合、それは Szubert & Jaskowski が実装した TD(0) による遷移後盤面の評価モデルのことを指すこととする。

2.1.2 n タプルネットワーク

TD 学習によって盤面の評価とその学習を行うことができるが、盤面と評価値をどのように結びつけるかが問題になる。まず、2048 で有り得るすべての盤面に対して評価値を与える1対1対応のルックアップテーブル (LUT) を作成することを考えると、2048 で有り得る盤面の数は $(4 \times 4)^{18} \approx 4.7 \times 10^{21}$ と膨大な数になり、このような LUT を計算機上で実装することは現実的に不可能である。

そこで、一部のマスの組み合わせによる「タプル」というクラスターを作成し、さらに複数のタプルを組み合わせることで盤面を表現する手法「n タプルネットワーク」を 2048 に導入することが、Szubert & Jaskowski によって提案された。たとえば、図 2.1 のような n タプルネットワークを実装した場合、1つのゲーム内で保持すべき重みの数は 860625 であり、全ての有り得る盤面に対する LUT を保持するのに比べて非常に少なく済む。

n タプルネットワークは Bledsoe & Browning によりパターン認識に用いられたのが最初の採用例である [16]。ゲーム AI の分野では、Lucas によってオセロに適用されたことをきっかけに、より広く知られるようになった [17]。

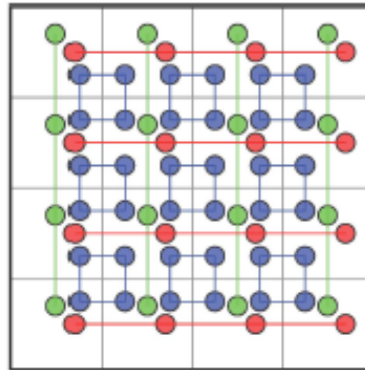


図 2.1. n タプルネットワークの例

2.1.3 結果と課題

本手法をもとに行った実験のうち，最も良い勝率を達成したプレイヤーを用いて 10 万ゲーム中の成績を検証したところ，勝率は 0.9781 であり，平均スコアは 100,178 であった．1 ゲーム中に達成されたスコアで最も良かったのは 261,526 であった．Szubert & Jaskowski による新たな手法は探索ベースの手法よりも大幅に高速で，かつ成績が良かった [4]．

しかしながら，この手法では「常に 2048-タイルを生成すること」よりも「時々 16384-タイルを生成すること」を重視しているため，勝率は必ずしも 100% を達成できていない．また，人間の知識を一切導入していないため，最も大きな数のタイルが盤面上の端に配置されないなど，人間の直感的な戦略とは反しているといったデメリットがあった．

2.2 Wu et al. (2014)

Wu は，Szubert & Jaskowski の手法を改良し，木探索を用いた先読みと組み合わせることによってさらに良いプレイヤーを実装することに成功した．

2.2.1 n タプルネットワークの配置の改善

Wu は，Szubert & Jaskowski が考案した n タプルネットワークのうち，直線型で 4 タプルとして配置していたタブルを，図 2.2 (b) のように柄杓型の 6 タプルに変更した．これによって増える重みの数は約 2 倍程度であったが，この変更によって Szubert & Jaskowski のものよりも飛躍的に良い成績を得ることができた．なお，何故このようなタブルの配置が最善だと判断したのかについて，Wu は論文において言及していない．

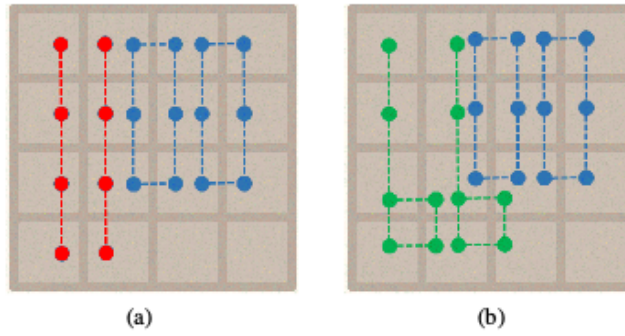


図 2.2. Szubert & Jaskowski (a) と Wu (b) が提唱した n タプルネットワーク

2.2.2 Multi-Stage TD 学習の導入

Multi-Stage TD 学習 (MS-TD 学習) とは, ゲームの局面に応じて異なる価値関数を保持することによって, よりそれぞれの局面に対して適切な重みを学習させることを目的とした手法である. Wu は学習のプロセスを 3 つのステージに分割し, ゲームプレイも同様に 3 つのステージに分割して行うようにした. Wu の提唱した 2048 における MS-TD 学習は, 以下のよう手順で学習を行う. なお, 「 T_{16k} 」とは「そのゲーム中で初めて 16384-タイルを生成することに成功した時」, 「 T_{16+8k} 」とは「そのゲーム中で初めて 16384-タイルを生成した後に, 初めて 8192-タイルを生成することに成功した時」のことを示す.

1. 第 1 ステージにおいては, 初期盤面からゲームを始めて, 価値関数が十分飽和するまで学習を行う. このステージで学習された価値関数の重みのことを「Stage-1 価値関数」と呼ぶことにする. また, 学習ゲーム中に T_{16k} を達成したなら, その時の盤面を全て保存しておく.
2. 第 2 ステージにおいては, 第 1 ステージで保存した盤面からゲームを始めて, TD 学習を行う. このステージで学習された価値関数の重みのことを「Stage-2 価値関数」と呼ぶことにする. また, 学習ゲーム中に T_{16+8k} を達成したなら, その時の盤面を全て保存しておく.
3. 第 3 ステージにおいては, 第 2 ステージで保存した盤面からゲームを始めて, TD 学習を行う. このステージで学習された価値関数の重みのことを「Stage-3 価値関数」と呼ぶことにする.

その後, 以下のような手順でゲームプレイを行う.

1. 盤面が T_{16k} を達成するまでは, Stage-1 価値関数を用いてゲームプレイを行う.
2. 盤面が T_{16k} を達成してから T_{16+8k} を達成するまでは, Stage-2 価値関数を用いてゲームプレイを行う.
3. 盤面が T_{16+8k} を達成してからは, Stage-3 価値関数を用いてゲームプレイを行う.

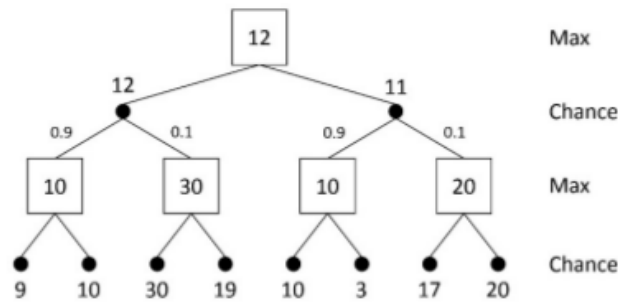


図 2.3. Expectimax 木の例 (Wu et al.)

2.2.3 Expectimax 木探索

Szubert & Jaskowski によって、木探索ベースの 2048 プレイヤは強化学習で訓練したプレイヤに劣ることが示されたが、Wu は強化学習によるプレイヤに対して Expectimax 木探索を補助的に組み合わせることによって、さらにプレイヤの性能を高めようと試みた。

Expectimax 木探索には、Max ノードと Chance ノードという 2 種類のノードがあり、それぞれのノードの値は子ノードから決定される。Max ノードの値は、子ノードのうち最も大きな値のノードの値となる。例えば図 2.3 の根ノードの値は 12 であるが、これは子ノードの「12」と「11」のうち最大の値である 12 を取ったものである。一方で Chance ノードの値は子ノードの期待値となる。例えば図 2.3 の根ノードの子ノードの 1 つである「12」というノードは、0.9 の確率で 10 となるノードと 0.1 の確率で 3 となるノードの期待値、すなわち $0.9 \times 10 + 0.1 \times 3 = 12$ によって 12 という値が決定する。

Wu の提案においては、Max ノードの値はプレイヤがアクションを選択して遷移を行った後の盤面、Chance ノードは遷移を行った後にランダムタイルを発生させた後の盤面が与える評価値となる。例えば、ある盤面 s (アクション選択と遷移が終わった直後の盤面とする) の評価値を深さ 3 の Expectimax 木探索を用いて求めたい時、図 hogehoge のような探索木が考えられる。 s の盤面が分かれば、その子ノードであるランダムタイル生成後の盤面、さらにその子ノードである遷移後の盤面を求めることができる。さらに、葉ノードにあたる Chance ノードの値は価値関数が与える評価値とすることで、各ノードの値を求めることができ、最終的に根ノード、すなわち評価値を求めたい盤面 s の評価値も求められるということになる。

(図 2.4. 何かいい感じの 2048 の探索木を自分で描画して貼る)

Expectimax 木探索を用いて先読みをすることで、将来の盤面を予想してより良いアクションを選択できるようになった。これはスコアの面で成績が良くなることに繋がる一方で、 $maxTile = 2048$ を達成する割合、すなわち勝率の向上にも大きな影響をもたらした。

2.2.4 結果と課題

Wu によるプレイヤは Szubert & Jaskowski のものに比べて著しく良い成績を達成した。まず Szubert & Jaskowski が達成できなかった 32768-タイルの生成に成功し、10.9% の確率で 32768-タイルを生成できるようになった。勝率に関しては 1 を達成、すなわち 2048-タイルは 100% の確率で生成できるようになり、平均スコアは 328,946、最大スコアは 605,752 を記録した。これは当時としては 1 つの例外^{*1}を除き、当時の計算機による 2048 プレイヤの中で最も優れた成績であった。

一方で、 n タプルネットワークの形状変更や MS-TD 学習のステージ分けの方法については、この研究で示された手法が最も良いということを説明する根拠が述べられておらず、依然として改良の余地は残していた。特に n タプルネットワークの形状変更については、次の Oka & Matsuzaki の研究で詳しく検討されることとなった。

2.3 Oka & Matsuzaki (2016)

Szubert & Jaskowski は当初図 2.1 のような n タプルネットワークを考案していたが、この n タプルネットワークによる学習器はあまり性能が高くなく、平均スコアは 10 万ゲーム学習した時点で 5 万～6 万程度にとどまっていた。そこで、図 2.2 (a) のような n タプルネットワークへの改良が行われ、その結果平均スコア・勝率ともに改善することができた。さらに Wu は Szubert & Jaskowski の考案した n タプルネットワークを改良し、成績を伸ばすことができた。

しかしながら、ここまでは「タプルのサイズを大きくすると学習器の成績も良くなる」という大まかな関係しかわかっておらず、成績を最善にする n タプルネットワークはどのような形状になるのか厳密には検討されていなかった。これを厳密に検討したのが Oka & Matsuzaki である。

2.3.1 n タプル単体の性能の評価

Oka & Matsuzaki は、まず n タプル単体の性能を網羅的に評価することを目指した。2048 の盤面上で n 個のタイルを選んで形成されうる n タプルの数は、表 2.1 の通りとなっている^{*2}。

Oka & Matsuzaki は Connected タプルのうち、十分性能が良く、なおかつ現実的に n タプルネットワークとして運用することができる $N = 6$ と $N = 7$ の場合のみ検討することとした。彼らは次に、形成されうる Connected な 6 タプル 68 個の中からランダムに 10 個のタプ

^{*1} Xiao による深深度先読みと人間による調整を行った評価関数を用いたプレイヤがこれにあたるが、Wu のものよりも 100 倍遅い：<https://www.youtube.com/watch?v=JQut67u8LIg>

^{*2} 「Connected」とは、タプル上の全てのタイルが 1 個以上の他のタプルと隣接していることを示す

表 2.1. 形成されうる n タブルの数

N	3	4	5	6	7	8	9	10	11	12	13
All	77	252	567	1051	1465	1674	1465	1051	567	252	77
Connected	8	17	33	68	119	195	261	300	257	169	66

















N	4 best tuples				4 worst tuples			
	$\langle 6-01 \rangle$	$\langle 6-02 \rangle$	$\langle 6-03 \rangle$	$\langle 6-04 \rangle$	$\langle 6-65 \rangle$	$\langle 6-66 \rangle$	$\langle 6-67 \rangle$	$\langle 6-68 \rangle$
6								
	31,161	24,530	22,207	20,576	9,644	9,642	9,563	9,052
7	$\langle 7-001 \rangle$	$\langle 7-002 \rangle$	$\langle 7-003 \rangle$	$\langle 7-004 \rangle$	$\langle 7-116 \rangle$	$\langle 7-117 \rangle$	$\langle 7-118 \rangle$	$\langle 7-119 \rangle$
								
	32,900	23,504	23,483	23,338	8,543	8,204	7,918	7,683

図 2.4. 最も優れている / 劣っている上位 4 つの 6 タブル・7 タブル (Oka & Matsuzaki)

ルを選んで 100 万ゲームの学習を行う実験を 680 回^{*3}繰り返し、各タブルの性能を比較可能な数値として算出した。7 タブルについても同様の実験を行った。

その結果、6 タブルと 7 タブルのうち最も優れた 4 つのタブルと最も劣った 4 つのタブルが、図 2.4 の通り明らかになった。

2.3.2 n タブルネットワークに組み込む n タブルの数の検討

6 タブルおよび 7 タブルの性能が判明したので、性能が良い任意の数のタブルを組み合わせることで n タブルネットワークを作成することが可能になった。Szubert & Jaskowski および Wu は 4 個のタブルを組み合わせることで n タブルネットワークを作成していたが、Oka & Matsuzaki は 5 個以上のタブルを組み合わせることを検討した。すなわち、 n タブルの数が多くなればなるほど n タブルネットワーク全体としての性能も上がると思われるが、最も性能が良くなるタブルの数は何個かを明らかにすることである。

実験においては、6 タブルの場合最大で上位 45 個、7 タブルの場合最大で上位 10 個のタブルから n タブルネットワークを作成することとした。各 n タブルネットワークに対して、600 万ゲーム学習を行わせ、10,000 ゲームごとに平均スコアと最大スコアを記録した。その結果が図 2.5 である。なお、このグラフにおける m とは、実験を行うにあたって、全ての n タブルの中から最も性能が良い m 個のタブルを採用して n タブルネットワークを生成したことを示す。

上記の通り、実験群の中では 10 個の 7 タブルを組み合わせたタブルネットワークが最も良

^{*3} $680 \times 10 = 6800$ より、全てのタブルが 100 回は選ばれるようにするため

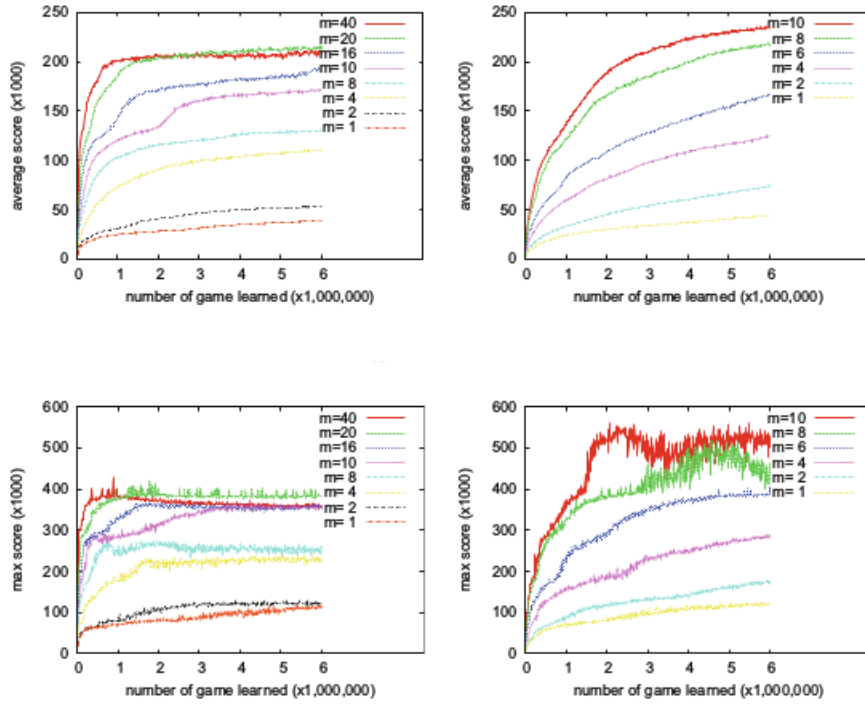


図 2.5. 成績上位タプル m 個を組み合わせたタプルネットワークの実験 (Oka & Matsuzaki)

い成績を収めることがわかった．なお，グラフからも読み取れるが，7 タプルによるネットワークに関しては 600 万ゲーム学習しても平均スコア・最高スコアが収束しない．そのためさらに組み合わせるタプルを増やすとさらに成績が良くなることが考えられるが，技術上の制約により $m = 11$ 以上は実現できなかった^{*4}．

2.3.3 結果と課題

Oka & Matsuzaki の研究における最も良い n タプルネットワークは，勝率が 0.9850，平均スコアが 234,136，最高スコアが 504,660 という成績を残した．これはゲーム木探索を用いない計算機を用いた 2048 プレイヤとしては最も良い成績であった．しかしながら，ゲーム木探索を用いなかったことによりゲーム序盤では安定性を欠いてしまい， W_u が達成した勝率 1 を割り込む形となってしまったし，単純に 2048 プレイヤとしての成績を比べると W_u のものよりも平均スコア・最大スコアともに低くなっている．一方で，ゲーム木探索を採用しなかったことはプログラムの高速化という点では利があり，1 秒あたり 88000 手の遷移を行うことができるプログラムとなった．これは W_u の研究と比べて約 290 倍高速である．

^{*4} 7 タプルを組み合わせるタプルネットワークを作成する場合，重みを保持するために 1 個のタプルにつき 1GB のメモリを消費しなければならない．Oka & Matsuzaki が実験を行った環境はメモリが 12GB しか確保できなかったため， $m = 11$ 以上は実験を行うことが困難だったのではないかと考えられる．また，この研究においてゲーム木探索を組み込むことができなかったのもメモリの制約が原因ではないかと考えられる

2.4 Yeh et al. (2016)

Yeh は Wu et al. (2014) の研究にも参画していた共同研究者で, Wu の研究の流れを引き継ぐ形で 2048 プレイヤの改良を行った. Wu の研究で用いた MS-TD 学習の考え方, 改良された n タプルネットワーク, Expectimax 木探索は引き続き用いられているため, この研究で新たに加わった要素のみを述べる.

2.4.1 新たな特徴量の追加

Yeh は, n タプルネットワークによる学習の他に, 盤面に表れる種々の特徴量を合わせて価値関数として用いることとした. 盤面の評価値には従来の n タプルネットワークによる評価値と新たに採用した特徴量による価値関数の和を用い, これを TD 学習で訓練した. なお, 採用した特徴は以下のつである.

1. 大きな数が書かれたタイルの個数
2. 何も書かれていないタイルの個数
3. 異なる数が書かれた (distinct) タイルの個数
4. マージすることができるタイルのペアの組数
5. 書かれているタイルの数が $(\nu, 2\nu)$ の形になっているタイルのペアの個数

2.4.2 MS-TD 学習のステージ分けの改良

Wu は MS-TD 学習を行うにあたり 3 ステージに分けて訓練・ゲームプレイを行っていたが, このステージの分け方をさらに細かくした. Yeh によってテストされたステージ分けは以下の通りである. なお, T_{x+y+z} という表記は, 「盤面に初めて $1000x, 1000y, 1000z$ のタイルが同時に現れた時」のことを指す. 各タイルの数は百の位で切り捨てられて表現されている.

- 戦略 1 : $T_{8k}, T_{16k}, T_{16+8k}$ を境界として 4 つにステージを分ける
- 戦略 2 : $T_{16k}, T_{16+8k}, T_{16+8+4k}$ を境界として 4 つにステージを分ける
- 戦略 3 : $T_{16k}, T_{16+8k}, T_{16+8+4k}, T_{16+8+4+2k}$ を境界として 5 つにステージを分ける
- 戦略 4 : $T_{16k}, T_{16+8k}, T_{16+8+4k}, T_{16+8+4+2k}, T_{16+8+4+2+1k}$ を境界として 6 つにステージを分ける

まず戦略 1 をテストした際, T_{8k} を境界としてステージを分けることは意味がないことがわかったため, 戦略 2 以降では T_{16k} 以降のステージ分けのみを考えることとした. 戦略 2 から戦略 4 をテストした結果, 平均スコアでは戦略 4 が最も優秀であることがわかったが, 最高スコアと 32678-タイルの到達率では戦略 3 が最も優秀であった. 各戦略ごとの成績は表 2.2 の通りである. 各学習器は 3 手先読みによるゲームプレイを行っており, 最高スコアを除く各指標は 10,000 ゲームの検証ゲームの結果の平均である.

表 2.2. 戦略 1～4 のステージ分けによる MS-TD 学習で訓練した学習器の成績 (Yeh et al.)

	戦略 1	戦略 2	戦略 3	戦略 4
16384-タイル作成率	87.23% ($\pm 0.65\%$)	87.68% ($\pm 0.64\%$)	87.68% ($\pm 0.64\%$)	87.68% ($\pm 0.64\%$)
32768-タイル作成率	14.15% ($\pm 0.68\%$)	15.30% ($\pm 0.71\%$)	18.31% ($\pm 0.76\%$)	16.82% ($\pm 0.73\%$)
最高スコア	594,716 ($\pm 51,886$)	579,996 ($\pm 40,583$)	614,204 ($\pm 10,053$)	610,011 ($\pm 13,720$)
平均スコア	349,996 ($\pm 3,433$)	350,150 ($\pm 9,097$)	361,395 ($\pm 9,892$)	364,438 ($\pm 9,448$)
速度 (手数/秒)	1,152 (± 93)	1,165 (± 114)	1,106 (± 106)	1,078 (± 104)

以上の結果より, Yeh は戦略 3 が最も優れたステージ分けだと判断し, これ以降は戦略 3 を用いて実験を行った.

2.4.3 学習率の調整

TD 学習をより正確に行うために, まず学習率 $\alpha = 0.0025$ で TD 学習を行った後, 価値関数に大きな改善が見られなくなったと判断したら, 学習率 $\alpha = 0.00025$ に変更して学習を続行させるようにした.

2.4.4 TD(λ) の使用

Szubert & Jaskowski および Wu は TD 学習において TD(0) を使用していたが, Yeh は TD(0.5) を用いて 5 ステップの報酬を平均化することとした. すなわち,

(ここに数式を書きます)

で表現される R_t^λ を用いて TD 学習を行うようにした.

2.4.5 結果と課題

上記の改良を重ねた結果, Yeh は平均スコア 443,526, 最大スコア 793,835 と非常に良い成績の学習器を訓練することに成功した. なお, 32678-タイルへの到達率は 31.75%, 1 秒あたりの遷移速度は 500 手であった. さらに, 検証ゲーム中のある 1 ゲームにおいて 65536-タイルを生成することにも成功した. これは強化学習ベースの 2048 プレイヤでは当時として最も強く, Xiao のよる深深度探索ベースのものよりも平均スコアでは優っており, Yeh の方が 125 倍速いプログラムであった.

2.5 Jaskowski (2017)

Jaskowski は、Yeh までの 2048 プレイヤの改良の流れを受け継ぎながら、これまで用いられてきた TD 学習や訓練・検証のステージ分けなどに対して改良を加えつつ、新たな手法を加えることでさらに優秀なプレイヤの実装を目指した。

2.5.1 自動的な学習率決定アルゴリズムの適用

Jaskowski は $TD(\lambda)$ を使用する点は Yeh の研究を継承したが、TD 学習で最も重要なパラメタである学習率を決定するにあたり、新たな手法を提案した。すなわち、Yeh の研究では手動で学習率が設定されていたのに対して、Jaskowski はこれまでに提案されてきたオンライン適応学習による学習率決定のアルゴリズムを 2048 にも適用することを試みた。

Bagheri et al. による Connect 4 への適用の研究において、これらの学習率決定アルゴリズムはいずれも標準的な TD 学習よりも良い成績を残すことが明らかとなったが、一方で長期的に見るとアルゴリズム間の差はそこまで大きいものにはならなかった。そこで、Jaskowski は最もシンプルな Temporal Coherence ($TC(\lambda)$) と、より先進的でチューニングの必要がない Autostep を選択して実験することにした。

$TC(\lambda)$

これまでの $TD(\lambda)$ では実験者が学習率 α を手動で決定していたが、 $TC(\lambda)$ においては、手動で決定される α の代わりに以下のパラメータを用いて学習率 γ を決定する。

- メタ学習率 β : 実験者が手動で設定するパラメータ
- 累積誤差関数 $E(s)$: これまで TD 学習で価値関数 $V(s)$ の更新を行う際に用いてきた誤差 $\delta_t = r_{t+1} + V(s_{t+1}) - V(s_t)$ を、 $V(s)$ の更新を行う際に $E(s)$ に対して加算する。盤面 s は価値関数と同様に、 n タプルネットワークを用いて近似される
- 累積絶対誤差関数 $A(s)$: $V(s)$ の更新を行う際に、 δ_t の絶対値を $A(s)$ に対して加算する。盤面 s は価値関数と同様に、 n タプルネットワークを用いて近似される
- 適応学習率 α : $E(s)$ と $A(s)$ の値によって自動的に決定されるパラメータ

適応学習率 α は以下の式で求められる。

$$\alpha = \begin{cases} \frac{|E_i[s'_k]|}{A_i[s'_k]}, & \text{if } A_i[s'_k] \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

また、学習率 γ は以下の式で求められる。

$$\gamma = \frac{\alpha \times \beta}{m} \quad (m \text{ は } n \text{ タプルネットワークに格納されているタプルの数})$$

このような学習率の求め方をすることによって，累積誤差と累積絶対誤差の値が近い場合は学習率が大きく，逆に累積誤差と累積絶対誤差の差が大きくなると学習率は小さくなる．ある盤面 s に対して累積誤差 $E(s)$ と累積絶対誤差 $A(s)$ の差が大きいということは， s に対して負の値の δ_t が加算されることが多くなっているということであり，すなわちそれはこの盤面に対しての価値関数の評価が上手く行っていないということであるから，学習率を低くして価値関数の更新を穏やかに行うのは理にかなっていると考えられる．

Autostep

Autostep は Mahmood によって提案されたアルゴリズムで，Sutton による IDBD の拡張にあたる．Autostep では，ステップサイズの上限を定め，もしステップサイズが大きくなりすぎていることが検知されたらステップサイズ値を小さくするという方法で，IDBD を改良したものである．Autostep では 3 種類のパラメータを設定する必要があり，Jaskowski の研究では，このうち初期ステップサイズ $\alpha_{init} = 1.0$ ，割引率 $\tau = 0.0001$ は固定され，メタ学習率 μ の値をいくつか選択して実験が行われた．

結果

Jaskowski は，前述の 2 つの自動的な学習率決定アルゴリズムに加え，従来の TD 学習^{*5}を採用した合計 3 種類の手法で，1 手先読み（すなわち，先読みを行わない）と 3 手先読みの 2 つの条件で実験を行った．その結果，全ての学習率決定手法のうち，1 手先読みの場合は $TC(\lambda)$ で $\beta = 1.0$ の時に平均スコアが最も良くなり，3 手先読みの場合は $TC(\lambda)$ で $\beta = 0.5$ の時に平均スコアが最も良くなった．1 手先読み・3 手先読みの場合ともに， $TD(\lambda)$ および Autostep はどのように学習率・メタ学習率を設定しても $TC(\lambda)$ に優る平均スコアを達成することはできなかった．

Jaskowski は，Autostep は定常的な教師データが存在する作業に特化しているため，訓練データが定常的ではない 2048 にはあまり適していなかったと考えている．また， $TC(\lambda)$ および Autostep 両者に共通する欠点として，何かしらの近似した盤面に対応する重みを保持する必要があることから， $TD(\lambda)$ よりも多くの n タプルネットワークを保持しなければならないため，計算時間が長くなってしまうことを指摘している．

2.5.2 Multi-Stage Weight Promotion

Jaskowski は Wu が提唱した MS-TD 学習の考え方を概ね継承したが，ステージ分けと重みの選択において細かい改良を加えた．

^{*5} なお， $TC(\lambda)$ および $TD(\lambda)$ においては， $\lambda = 0.5$ とした

新しいステージ分け

Wu では 3 ステージ, Yeh では 5 ステージに分けることが最善であるとされていたステージ分けについて, Jaskowski は 2^g ステージに分け (g は正の整数), また各ステージの「長さ」 l を $l = 2^{15+1-g}$ と定義した. 例えば $g = 4$ の時, 長さ $l = 2^{12} = 4096$ である. この場合, 以下のようなステージ分けがなされる.

1. ステージ 1: 初期盤面から T_{4096} まで
2. ステージ 2: T_{4096} から T_{8192} まで
3. ステージ 3: T_{8192} から $T_{8192+4096}$ まで
4. ステージ 4: $T_{8192+4096}$ から T_{16384} まで
5. ...
6. ステージ 16: $T_{32768+16384+8192+4096}$ から T_{65536} まで

なお, $TC(\lambda)$ で導入した累積誤差関数 E と累積絶対誤差関数 A についても同様にステージ分けを行って重みを保持する.

Weight Promotion

Wu および Yeh のステージ分けにおいては, 新しいステージの学習を始める際に再び何も学習していない状態から n タプルネットワークの訓練を行わなければならなかったため, 汎化性能が著しく損なわれていた. この汎化性能の弱体化はステージ数が多くなればなるほど進むため, Yeh はステージを多くすれば多くするほど学習器の性能は悪くなることを指摘しており, その結果 5 ステージに分けるのが最も良いと考えたのである. この問題を「ステージ数を多くしない」以外の方法で解決するために, Jaskowski は各ステージの初期状態の評価値を, 前のステージの同じ盤面から引き継ぐことにした. Jaskowski はこの方策を Weight Promotion と呼び, 上記の多段ステージ分けとともに提案した.

結果

Jaskowski は, 以下の 6 つの条件について, 1 手先読みと 3 手先読みの 2 つの条件を掛けあわせて合計 12 の実験を行った. なお, WP は Weight Promotion を適用したことを示し, 6. 以外の 5 つの実験においては全て共通の 5 つの 6 タプルによる実験を行った.

1. ステージ分けを行わない (純然たる $TC(\lambda)$)
2. $g = 4$
3. $g = 3, WP$
4. $g = 4, WP$
5. $g = 5, WP$
6. 5 つの 7 タプルによる $TC(\lambda)$ で, ステージ分けを行わない

この結果, 1 手先読みでは 6. の 7 タブルによる $TC(\lambda)$ が最も良い成績を収め, 1. に関して 2. と 5. の実験には成績が優るなど, ステージ分けに対しては顕著な効果が見られなかった. 一方 3 手先読みでは 4. の実験が最も良い成績を収め, ステージ分けを行わない 1. と 6. の実験よりも高い平均スコアを得た. また学習プロセスに注目すると, ステージ分けを行う実験では平均スコアが単調増加していたのに対し, ステージ分けを行わない場合は平均スコアが逆に悪くなっていることが確認された.

以上の結果より, Jaskowski はステージ分けを行わない学習器は 1 手先読みの条件に対して過学習が起きてしまうが, ステージ分けを行う学習器は過学習に耐性があり, 成績の悪化は起こらなくなると結論付けた. なお, Weight Promotion については 1 手先読みに対しても 3 手先読みに対しても適用することでより良い成績が得られることが確認された.

2.5.3 Carousel Shaping と余剰タブルの追加

Carousel Shaping

前節のステージ分けにおける改良を行った際, 従来の学習器は 1 手先読みの条件の下で過学習を行ってしまい, 3 手先読みでは学習を重ねるに従って成績を落としてしまうという状況が確認された. これは従来の Multi-Stage 学習ではゲームで重要になる終盤のステージでの学習時間が少なくなってしまうことが原因であると考えた Jaskowski は, 終盤のステージにおける学習時間を確保するため, Carousel Shaping という新たな Multi-Stage 学習の手法を提案した. Carousel Shaping の擬似コードは以下の通りである.

Algorithm 1 Carousel Shaping

```

1: function CAROUSELSHAPING
2:    $stage \leftarrow 1$ 
3:    $initstates[x] = \emptyset$  for  $x \in \{1 \dots 2^g\}$ 
4:   while not enough learning do
5:     if  $stage = 1$  then
6:        $s \leftarrow \text{INITIALSTATE}()$ 
7:     else
8:        $s \leftarrow \text{RANDOMCHOICE}(initstates[stage])$ 
9:      $\text{LEARNFROMEPISODE}(s) \triangleright \text{Updates } initstates$ 
10:     $stage \leftarrow stage + 1$ 
11:    if  $stage > 2^g$  or  $initstates[stage] = \emptyset$  then
12:       $stage \leftarrow 1$ 
13: end function

```

Carousel Shaping (CS) では, 明確に学習を行うステージを分けるのではなく, 1 回のゲームの中で複数のステージにまたがった学習を可能にする. ステージ 1 の初期盤面から学習を始め, もし途中でステージ分けの境界にあたる盤面に到達したならその都度 $initstates$ に登録し

つつ^{*6}学習を行う。学習が 1 ゲーム分終わったら次のステージに進み、そのステージの初期盤面を取得し、学習を行う。もしステージが 2^9 を超えるか、そのステージの初期盤面がまだ 1 つも記録されていなかった場合、ステージ 1 に戻って学習を行う。このように、CS はその名 (Carousel=回転木馬) の通り学習するステージをローテーションさせながら学習を行う。

余剰タブルの追加

Jaskowski が導入した最後の手法が余剰タブルの追加 (Redundant Encoding) である。これまで使用してきた 5 個の 6 タブルに加え、より小さい 2 個の 3 タブルと 5 個の 4 タブルをタブルネットワークに加えて学習を行う。ここで追加したタブルは既に使用しているタブルの形状の中に含まれているため、一見すると無意味なタブルではあるが、追加することでより速く汎化することが確認された。ただし、サイズが小さいとはいえ 7 個のタブルを追加するため、学習時間は非常に長くなる。

2.5.4 結果

以上の改良を加え、さらに Expectimax 木探索による探索を先読みする手数ではなく 1 手あたり 1000 ミリ秒まで許容したところ、Jaskowski による 2048 プレイヤは平均スコア 609,104 を記録した。これは先行研究のいずれよりも優れた成績である。またこの研究における最も優秀なプレイヤは 1 秒に 1 手のペースでアクションを選択するため、1 秒あたり 500 手進めることができる Yeh のものより表面上は遅くなるが、仮に 1 秒に上限 1000 手のペースでアクションを選択するよう調整した場合は平均スコアが 527,099 となり、Yeh の 443,526 を上回る。すなわち、学習のみならず検証時のプレイにおいても Yeh のプレイヤよりも性能が高いということになる。

^{*6} initstates は各ステージの初期盤面を保持するデータ構造で、各ステージごとに最近到達された初期盤面 1000 個を保持する

第 3 章

本研究のアイデア

先行研究で挙げられてきた 2048 プレイヤは、強化学習によって有り得る 2048 の盤面の評価値を記録しておき、盤面に応じて最も価値が高いと思われる盤面につながるアクションを選択するという方策のもとにプレイを行ってきた。翻って人間が 2048 をプレイする場合、人間が盤面の価値を評価・記憶することは明らかに不可能である。そのため、人間が 2048 をプレイする際には、ある一定の方策をいくつか持ち、その方策に従う盤面を導き出すようにアクションを選択していると考えられる。ここで、人間が採用していると考えられる 2048 の方策をいくつか挙げる。

3.1 base 方策

人間が 2048 をプレイする時、一般的に大きな数のタイルを盤面のうちいずれか 1 辺に寄せようとする傾向がある。この方策を base 方策と名付けることとする。base 方策に基づいたプレイのフローの例は以下の通りである。

- 盤面の任意の辺 4 マスに「基地」(base) を作って、大きなタイルを蓄える場所と決めておく^{*1}
- 基地以外の残りの 9 マスを使って、新しく数の大きいタイルを作成する
- 基地にマージできるタイルを作成できたら、基地にあるタイルへマージする
- 基地内でマージすることができるタイルの組ができたら、それらをマージし、さらに大きな数のタイルにする

もし基地を定めない場合、大きなタイルが盤面のあちらこちらへ移動することとなる。2 ν -タイルを作成することが困難な ν -タイル 例え 2048-タイルなどが盤面を上下左右に移動することを考えると、例え 4-タイルや 8-タイルといった小さなタイルのマージを行う妨げとなり、盤面の自由度が極端に下がる。そのためこの方策は理に適っていると考えられ、先行研究によるプレイヤも強化学習の訓練を通じてこの方策を習得していると考えられる。

^{*1} プレイヤ自身の好みによって下や左などの辺が基地に選ばれるが、どの辺を選んだとしても対称な盤面を作ることができるので、どの辺を基地に選んでも実質的に変わらないと考えられる

3.2 array 方策

人間が base 方策に従って 2048 プレイする際、基地の中ではできる限りタイルが「 $2^k - 2^l - 2^m - 2^n$ ($0 < k < l < m < n$)」といった形へ、タイルの数が降順もしくは昇順になるように整列 (array) させる傾向がある。この方策を array 方策と名付けることとする。

この方策は base 方策との相乗効果が高い。例えば「32 - 64 - 128 - 256」という順列の基地を作った場合、基地の外で 32-タイルの作成に成功すれば、そのまま基地内でマージを行うことによって 512-タイルを作成することができる。あるいは、基地の外で 128-タイルの作成に成功した場合はマージを行うと「32 - 64 - 512」となるが、この状態でも array 方策は維持されており、ゲームを長く進めていても保持しやすい性質である。この方策についても、base 方策の習得と合わせて、先行研究によるプレイヤは自然と習得していると考えられる。

3.3 maximum-in-corner 方策

最後に、最もシンプルな方策として maximum-in-corner 方策を挙げる。この方策では、現状の盤面における *maxtile* が四隅のどれかのマスにあることを重視する。これは array 方策に強く関連している方策の拡張的な方策と考えられる。例えば「2 - 32 - 2048 - 2」という順列の基地を考えると、前の 3 つのタイルは array 方策に従っているものの、「2048 - 2」という並びは array 方策に反している。この状況を解決するためには、以下のいずれかの操作が必要になる。

1. 2 および 32 にタイルをマージし続け、2048-タイルを作成し、既に存在している 2048-タイルとマージする
2. 2048 に隣接している 2-タイルを基地の外へと移動させ、2048-タイルを順列の終端へと移動させる

このうち、1. を実現させることは難しいと考えられる。なぜなら、基地のうち 2 マスがマージすることができないタイルで占められているため、より少ないバッファでタイルをマージすることが必要になるからである。一方で、2. を実現させることも同様に難しい。横もしくは縦一列に並んでいるタイルの中から、1 つだけを選んで基地から外す操作をすることを、意図して行うのは難しい。したがって、一度 array 方策に反する順列を作ってしまった場合、array 方策に従うような順列に復帰させることは非常に難しいということになる。

このような困難な状況に陥ることを防ぐため、あらかじめ array 方策に反する順列を作らないような方策が必要になる。これが maximum-in-corner 方策であり、大きな数のタイルができる限り盤面の隅から離れないことで、より array 方策を長く実現できるようにしている。

base 方策および array 方策については、これに近いものが強化学習のプロセスの中で自然と習得されたことが確認されているが、maximum-in-corner 方策については明確に従っているとは言えず、maximum-in-corner 方策に反したアクションを選択した結果盤面の状態を非常

に悪くしているケースも確認できる．このことから，先行研究の強化学習を使用したプレイヤーは maximum-in-corner 方策を明確には習得していないと考え，本研究において方策を明示的に導入することで，強化学習の改良や探索木の導入とは独立に成績の改善が行えると考えた．

第 4 章

提案と実装

corner bonus を 2048 プレイヤへ導入することを提案します．具体的には評価値に CB を掛け合わせるとか

4.1 maximum-in-corner 方策の導入

強化学習を用いて盤面ごとに評価値を学習した 2048 プレイヤにおいて，maximum-in-corner 方策に従うこととゲームの展開の関係は，以下の 4 つの場合が考えられる．

- (a) maximum-in-corner 方策に従った結果，ゲームの展開が良くなる
- (b) maximum-in-corner 方策に従わなかった結果，ゲームの展開が良くなる
- (c) maximum-in-corner 方策に従った結果，ゲームの展開が悪くなる
- (d) maximum-in-corner 方策に従わなかった結果，ゲームの展開が悪くなる

このうち，(a) の場合については，maximum-in-corner 方策を改めて導入しなくても強化学習によって適切な学習がなされており，なおかつ学習器は既に maximum-in-corner 方策に従っているのだから，特に考慮する必要はない．逆に (d) については学習器が maximum-in-corner 方策を習得することに失敗しており，なおかつそのことによってゲームプレイに対しても悪影響を及ぼしているため，maximum-in-corner 方策を導入することによってプレイの内容を修正することが必要になる．

一方で，問題になるのが (b) および (c) の場合である．これらの場合はどちらも「maximum-in-corner 方策に従わないほうが良かった」という点が共通している．このような「誤って maximum-in-corner 方策に従ってしまったが，実際は従わなかった方が良かった」という状況のことを，corner 誤謬と呼ぶことにする．corner 誤謬の存在は，必ずしも maximum-in-corner 方策を導入することが良い結果に直結しないことを意味する．したがって，maximum-in-corner 方策を導入する場合，明らかに方策を適用した方が良い場合には maximum-in-corner 方策を適用しつつ，そうではない場合は maximum-in-corner 方策を適用しないという調整が必要になる．

このような調整を踏まえて maximum-in-corner の実装を実現するために、本研究においては以下のように maximum-in-corner 方策を適用することとした。

1. アクション選択の手順において、選択可能なアクションを a_n ($0 \leq n < 4$) とする。
2. 選択可能なアクションを元に、遷移を行った後の盤面 s_{a_n} を求める。
3. 求められた盤面に対する評価値 $V(s_{a_n})$ を求める。
 - この時、もし s_{a_n} の *maxtile* が盤面の四隅のいずれかのマスにあるならば、 $V(s_{a_n}) = \rho \times V(s_{a_n})$ とする。 ρ は実験者が設定する定数であり、この定数のことを **corner ratio** と呼ぶこととする。
4. 評価値を全ての有り得る盤面に対して求めた後、最も大きな $V(s_{a_n})$ を与えるアクション a_n を選択する。

このような適用の仕方によって、maximum-in-corner 方策に従っている盤面をより良く評価することができるため、(a) および (d) の場合に maximum-in-corner 方策に従う盤面を導くアクションを選択できる。また、評価値に対して maximum-in-corner 方策に従っているかどうかで ρ によって割増を行うという形式を取ることで、「評価値の割増を行った盤面よりも評価値が高く、なおかつ割増が行われていない（すなわち、maximum-in-corner 方策に従わない）盤面」を導くアクションを選択することが可能になる。すなわち、(b) および (c) の場合に maximum-in-corner 方策に従わない盤面を導くアクションを選択できるようになり、2048 のアクション選択の手順で考えられる全ての場合において、適切に maximum-in-corner 方策の適用を行うかどうかのスイッチングができると考えられる。

4.2 実装

まず、maximum-in-corner 方策を採用しない、通常の TD 学習の擬似コードを以下に示す。この擬似コードは Szubert & Jaskowski の実装に基づく。

本研究においては、上記のコードのうち、関数 `ChooseBestTransitionAfterstate` に数行を加えた下記の関数 `ChooseBestTransitionAfterstateCorner` を実装すると同時に、その盤面の *maxtile* が四隅のマスにあるかどうかを判断する関数 `isMaxtileInCorner` を実装する。

なお、`ChooseBestTransitionAfterstate` は、 n タプルネットワークを用いてプレイする^{*1}関数 `PlayByAfterstate` および n タプルネットワークを訓練する関数 `TDAfterstateLearn` に呼ばれている、 n タプルネットワークを参照して最も良い遷移を返す関数である。`ChooseBestTransitionAfterstate` と `ChooseBestTransitionAfterstateCorner` を両方実装しておいて、検証ゲームを行う際にどちらかの関数を呼び出すようにすることで、検証ゲームと学習ゲームのどちらか、もしくはそのどちらにもにおいて maximum-in-corner 方策を採用することができるようになる。

^{*1} ここでは検証ゲームを行うという意味である

Algorithm 2 Temporal Difference Learning (TD(0))

```

1: function CAROUSELSHAPING
2:    $stage \leftarrow 1$ 
3:    $initstates[x] = \emptyset$  for  $x \in \{1 \dots 2^g\}$ 
4:   while not enough learning do
5:     if  $stage = 1$  then
6:        $s \leftarrow \text{INITIALSTATE}()$ 
7:     else
8:        $s \leftarrow \text{RANDOMCHOICE}(initstates[stage])$ 
9:      $\text{LEARNFROMEPISODE}(s) \triangleright$  Updates  $initstates$ 
10:     $stage \leftarrow stage + 1$ 
11:    if  $stage > 2^g$  or  $initstates[stage] = \emptyset$  then
12:       $stage \leftarrow 1$ 
13: end function

```

Algorithm 3 Maximum-in-corner Policy

```

1: function CAROUSELSHAPING
2:    $stage \leftarrow 1$ 
3:    $initstates[x] = \emptyset$  for  $x \in \{1 \dots 2^g\}$ 
4:   while not enough learning do
5:     if  $stage = 1$  then
6:        $s \leftarrow \text{INITIALSTATE}()$ 
7:     else
8:        $s \leftarrow \text{RANDOMCHOICE}(initstates[stage])$ 
9:      $\text{LEARNFROMEPISODE}(s) \triangleright$  Updates  $initstates$ 
10:     $stage \leftarrow stage + 1$ 
11:    if  $stage > 2^g$  or  $initstates[stage] = \emptyset$  then
12:       $stage \leftarrow 1$ 
13: end function

```

第 5 章

実験

提案したアイデアの実験結果と既存研究の実験結果を比較します．比較対象は Szubert ?

5.1 実験 1：常に $\rho = 1.2$ とする

まず最初に ,maximum-in-corner 方策を使用することに効果があるのか調べるため , $\rho = 1.2$ として実験を行った．詳細な実験のセッティングは以下の通りである．

- 学習ゲーム数：500,000
- 学習率：0.01
- n タプルネットワークの配置：Szubert & Jaskowski のスタイル
- ChooseBestTransitionAfterstateCorner を使用するゲーム：
 - 使用しない (ベースライン)
 - 検証ゲーム
- 使用する corner ratio： $\rho = 1.2$

上記の実験の結果のうち，各検証ゲームにおける平均スコアをグラフにプロットしたのが図 5.1 である．全体的な傾向としては，約 200,000 ゲームほどの学習を終えるまでの間は corner ratio あり，すなわち maximum-in-corner 方策を採用したほうが成績が良い．しかしながら，その後は maximum-in-corner 方策なしの方が全体的に成績が良くなり，一応平均スコアは伸び続けているものの maximum-in-corner 方策ありの方は成績が低調になる．

実験 1 の結果より，高い値の corner ratio を設定することは学習の序盤において効果があることがわかったが，一方で学習が進むと，かえって学習器の性能を弱めてしまうということがわかった．これらの事実に基づき，以下の仮説を立てた．

- 一般に，maximum-in-corner 方策を採用すること自体は 2048 プレイヤの性能を高める可能性がある
- 学習ゲームを重ねるにつれ TD 学習による学習器の性能は良くなるので，高い corner ratio の値は corner 誤謬を引き起こす

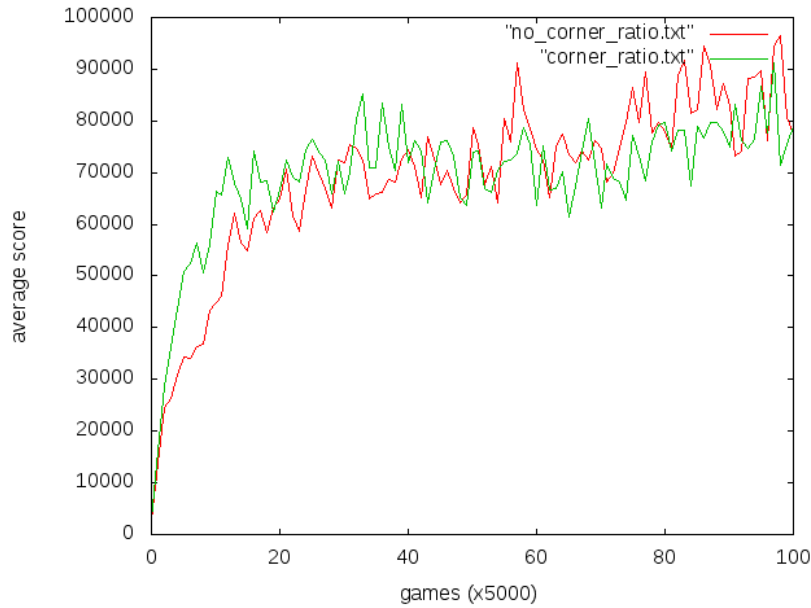


図 5.1. 実験 1 の結果

この仮説が正しいことを示すために，corner ratio の適用について様々な条件を用意する追加実験を行うことにした．

5.2 実験 2 : 2 つの ρ の値を用いる

実験 1 においては検証ゲーム中に一貫して $\rho = 1.2$ としていたが，学習ゲームが進んだ後もこの値を使い続けることはあまり良くないということがわかった．そこで，実験 2 においては，ゲームの状況に応じて異なる corner ratio の値を使用することとした．また，実験 1 では検証中のみに maximum-in-corner 方策を採用していたが，実験 2 においては学習中および検証中と学習中の両方に採用する条件も設定することとした．

なお，ゲームの進行状況を示す指標としては，スコアと *maxtile* を用いることができる．今回の実験では corner ratio を変更する指標として，スコアを参照する実験と *maxtile* を参照する実験の両方をそれぞれ行った．詳細な実験のセッティングは以下の通りである．

- 学習ゲーム数：500,000
- 学習率：0.01
- n タプルネットワークの配置：Wu のスタイル^{*1}
- ChooseBestTransitionAfterstateCorner を使用するゲーム：
 - － 使用しない（ベースライン）

^{*1} どのような n タプルネットワークに対しても maximum-in-corner 方策が適合することを示すために，Szubert & Jaskowski の実装よりも高級な Wu の実装による n タプルネットワークを用いた

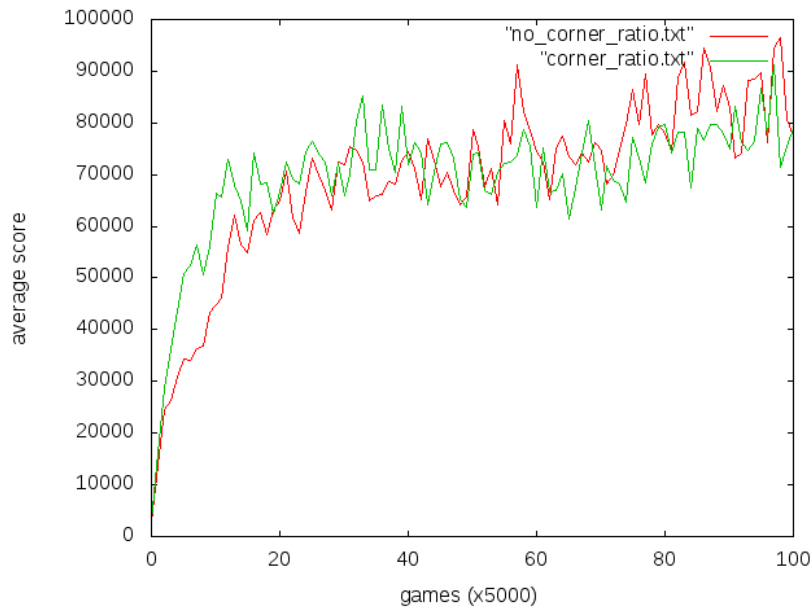


図 5.2. 実験 2 の結果 - 検証ゲームへの maximum-in-corner 方策の適用

- 検証ゲーム
- 学習ゲーム
- 検証ゲームと学習ゲームの両方
- 使用する corner ratio : $\rho = 1.1$ もしくは 1.2
- corner ratio を変更する条件 :
 - 最初は $\rho = 1.2$ とし , スコアが 70,000 に到達したら $\rho = 1.1$ とする
 - 最初は $\rho = 1.2$ とし , $maxtile = 4096$ が達成されたら $\rho = 1.1$ とする

この実験の結果のうち , 平均スコアの推移をプロットしたグラフを図 5.2 から図 5.4 に示す . 実験結果のうち , maximum-in-corner 方策を適用しているものについてはいずれもベースラインよりも平均スコアを上回っているが , 検証ゲームに適用したものと検証ゲーム・学習ゲームの両方に適用したものについては , 顕著にベースラインよりも平均スコアが高くなっている . また全体的な傾向としては , corner ratio を平均する条件にスコアを使用する方が平均スコアが良くなることが多い .

実験 2 の結果 , ゲームの進行に応じて corner ratio を変更することにより , TD 学習によって学習器の性能が良くなった後でも , maximum-in-corner 方策を適用することでさらなる成績の向上が行われることがわかった . それと同時に , ゲームの進行に応じて corner ratio を割り引くことで , corner 誤謬の発生を抑制できるということもわかった . 最後に , より適切な corner ratio の変更の条件を求めつつ , さらに 2048 プレイヤとしての性能を高めるため , 学習率と学習ゲーム数を変更する実験を行った .

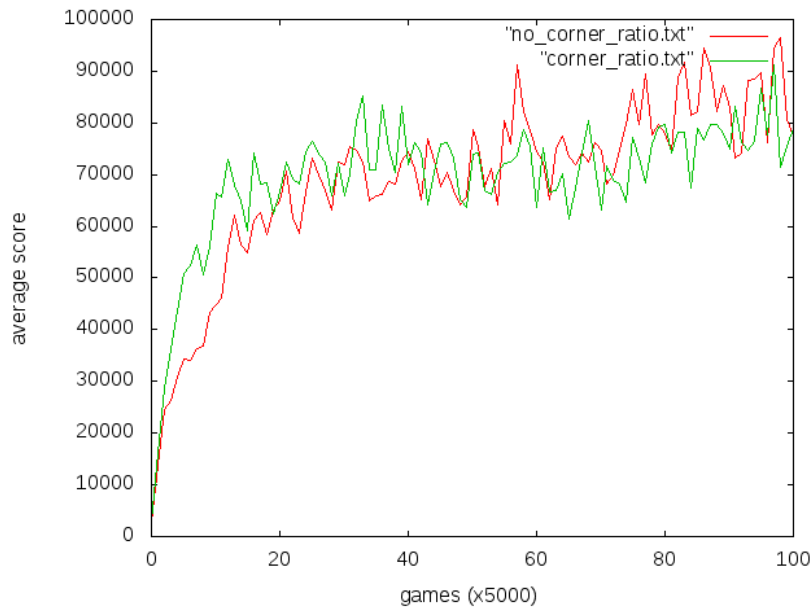


図 5.3. 実験 2 の結果 - 学習ゲームへの maximum-in-corner 方策の適用

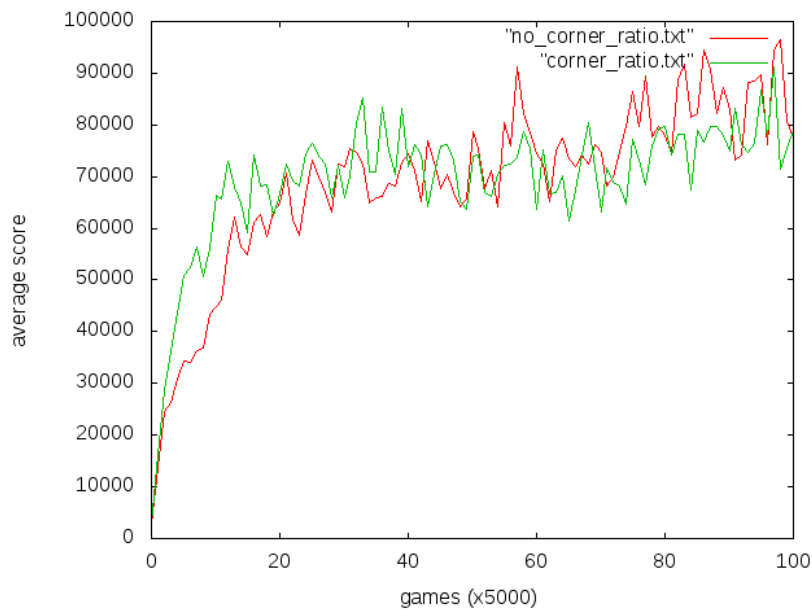


図 5.4. 実験 2 の結果 - 検証・学習ゲームへの maximum-in-corner 方策の適用

5.3 実験 3：学習率および学習ゲーム数を変更する

実験 3 においては、より 2048 プレイヤとしての性能を高めるために、学習率を 0.0025 に引き下げ、学習ゲーム数を従来の 2 倍の 1,000,000 ゲームに変更した。学習率を引き下げるこ

で価値関数 V の更新が穏やかになるため、学習の速さは若干遅くなるものの、その分精密な学習がなされて最終的な性能は高くなることが期待される。これに加えて、ゲーム中のスコアに応じて corner ratio を変更することが有効であると判明した実験 2 の結果を踏まえて、corner ratio を変更する条件として設定するスコアの境界値をさまざまな値に変動させて実験を行った。その他の条件も含めた実験のセッティングは以下の通り。

- 学習ゲーム数：1,000,000
- 学習率：0.0025
- n タプルネットワークの配置： W_u のスタイル
- ChooseBestTransitionAfterstateCorner を使用するゲーム^{*2}：
 - 使用しない（ベースライン）
 - 検証ゲーム
 - 検証ゲームと学習ゲームの両方
- 使用する corner ratio： $\rho = 1.1$ もしくは 1.2
- corner ratio を変更する条件：
 - 最初は $\rho = 1.2$ とし、スコアが 60,000 に到達したら $\rho = 1.1$ とする
 - 最初は $\rho = 1.2$ とし、スコアが 70,000 に到達したら $\rho = 1.1$ とする
 - 最初は $\rho = 1.2$ とし、スコアが 80,000 に到達したら $\rho = 1.1$ とする
- ランダムシード：3 種類

この実験の結果のうち、平均スコアの推移をプロットしたグラフを図 hoge から図 fuga に示す。ランダムシードによって結果に差があるが、全体的に見て以下の傾向を読み取ることができる。

（図 hoge ~ fuga がここに入ります）

- 学習率 0.0025 の条件の下においては、検証ゲームのみに maximum-in-corner 方策を適用しても、ベースラインと比べて顕著な差は現れない
- 約 300,000 ゲームほど学習するまでは、どの条件の間にも顕著な差は見られない
- 約 500,000 ゲームほど学習した後は、検証ゲームと学習ゲームの両方で maximum-in-corner 方策を適用したプレイヤーの性能が顕著に良くなる
- 局所的に他のプレイヤーが上回っていることもあるが、全体的に見ると「検証ゲームと学習ゲームの両方」で「最初は $\rho = 1.2$ とし、スコアが 80,000 に到達したら $\rho = 1.1$ とする」条件で corner ratio を変更するプレイヤーの成績が最も良い

^{*2} 実験 2 において学習ゲームのみに ChooseBestTransitionAfterstateCorner を使用することには効果が見られないことがわかったため、実験 3 の条件からは除外した

第 6 章

考察と結論

以上の実験により，maximum-in-corner 方策を適用することで，TD 学習を用いて訓練した 2048 プレイヤの成績をさらに改善できるということが示された．このことは「強化学習による訓練では maximum-in-corner 方策を習得することができない」という 2048 プレイヤの性質と表裏一体であり，強化学習とは別の観点から 2048 プレイヤの改良を行えることを示したとも言える．

一方で，maximum-in-corner 方策の一面的な導入が必ずしも良い結果をもたらすとは限らないことも示された．常に $\rho = 1.2$ としたプレイヤの成績は十分学習された TD 学習のみを用いたプレイヤに劣り，学習が進めば進むほど corner 誤謬が発生する確率も高まることが確認されている．この問題を解決するために corner ratio の値を変更することが有効であり，特にゲーム内のスコアに応じて corner ratio を割引することはプレイヤの成績の改善に大きく貢献した．

しかしながら，本研究において，corner bonus の割引は実験者による手動操作によってのみ行われており，より適切な corner ratio の選定・決定方法が存在する可能性もある．例えば，Jaskowski (2016) によって提案された $TC(\lambda)$ における学習率の自動的な適応のように，ゲーム内のパラメータや指標を参照して，自動的に corner ratio を決定するといった手法が考えられる．より精密な corner ratio の追究が不十分であることは，本研究が残した課題の 1 つである．

maximum-in-corner 方策の他にも，2048 において人間が経験的に習得しているが，強化学習によるプレイヤが習得できない方策が存在する可能性は少なくないと思われる．強化学習と人間の経験的知識の融合により，2048 を始めとするゲーム AI エージェントの改良が期待される．

謝辞

本研究を進めるにあたり，1 年間ご指導いただき，実験にあたって計算資源を提供していただいた，指導教員の山口和紀教授に感謝いたします．また，Y ゼミにて本研究や発表内容に対してアドバイスをくださった山口和紀研究室の皆様にも感謝いたします．最後になりますが，3 年次に履修した実習にて，課題として 2048 プレイヤの作成というテーマを課していただき，本研究の端緒を開いてくださった金子知適准教授に感謝の意を表させていただきます．

参考文献

- [1] 2048. <https://gabrielecirulli.github.io/2048/>. 2018 年 1 月 7 日閲覧 .
- [2] Megan Rose Dickey. Puzzle Game 2048 Will Make You Forget Flappy Bird Ever Existed. Business Insider, <https://www.businessinsider.com/2048-puzzle-game-2014-3>. 2018 年 1 月 7 日閲覧 .
- [3] Gabriele Cirulli. 2048, success and me. <https://medium.com/@gabrielecirulli/2048-success-and-me-7dc664f7a9bd>. 2018 年 1 月 7 日閲覧 .
- [4] Marcin Szubert & Wojciech Jaskowski. Temporal Difference Learning of N-Tuple Networks for the Game 2048. In IEEE Conference on Computational Intelligence and Games, pages 1-8, Dortmund, Aug 2014. IEEE.
- [5] I-Chen Wu, Kun-Hao Yeh, Chao-Chin Liang, Chia-Chuan Chang, Han Chiang. Multi-Stage Temporal Difference Learning for 2048. In Technologies and Applications of Artificial Intelligence, pp 366-378. Springer, 2014.
- [6] Kazuto Oka & Kiminori Matsuzaki. Systematic Selection of N-Tuple Networks for 2048. In International Conference on Computers and Games (CG 2016), Leiden, The Netherlands, 2016.
- [7] Kun-Hao Yeh, I-Chen Wu, C. H. Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multi-stage temporal difference learning for 2048-like games. In IEEE Transactions on Computational Intelligence and AI in Games, preprint. 2016.
- [8] Wojciech Jaskowski. Mastering 2048 with Delayed Temporal Coherence Learning, Multi-Stage Weight Promotion, Redundant Encoding and Carousel Shaping. In IEEE Transactions on Computational Intelligence and AI in Games, preprint. 2017.
- [9] R. Sutton & A. Barto. 『強化学習』(三上貞芳・皆川雅章訳) 森北出版, 2000.
- [10] G. Tesauro. Temporal Difference Learning and TD-Gammon. Communications of the ACM, vol. 38, no. 3, pp. 58-68, 1995.
- [11] T. P. Runarsson & S. M. Lucas. Co-evolution versus Self-play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go. IEEE Transactions on Evolutionary Computation, vol. 9, no. 6, pp. 628-640, 2005.
- [12] N. N. Schraudolph, P. Dayan, & T. J. Sejnowski. Learning to Evaluate Go Positions via Temporal Difference Methods. In Computational Intelligence in Games, ser. Studies

- in Fuzziness and Soft Computing, N. Baba and L. C. Jain, Eds. Springer Verlag, Berlin, 2001, vol. 62, ch. 4, pp. 77-98.
- [13] S. van den Dries & M. A. Wiering. "Neural-Fitted TD-Leaf Learning for Playing Othello With Structured Neural Networks. IEEE Transactions on Neural Networks and Learning Systems, vol. 23, no. 11, pp. 1701-1713, 2012.
- [14] M. G. Szubert, W. Jaskowski, & K. Krawiec. On Scalability, Generalization, and Hybridization of Coevolutionary Learning: A Case Study for Othello. IEEE Transactions on Computational Intelligence and AI in Games, vol. 5, no. 3, pp. 214-226, 2013.
- [15] J. Baxter, A. Tridgell, & L. Weaver. Learning to Play Chess Using Temporal Differences. Machine Learning, vol. 40, no. 3, pp. 243-263, 2000.
- [16] Woodrow Wilson Bledsoe & Iben Browning. Pattern recognition and reading by machine. In Proc. Eastern Joint Comput. Conf., pages 225-232, 1959.
- [17] Simon M. Lucas. Learning to play Othello with N-tuple systems. Australian Journal of Intelligent Information Processing Systems, Special Issue on Game Technology, 9(4):0120, 2007.

付録 A

表やプログラムリストの掲載が必要になったらここに掲載します。