強化学習入門

京都大学 理学部 3 回生

スイス連邦工科大学ローザンヌ校

小南 佑介

1　本稿について

今回、私は「クラウドファンディングのお返し」という目的から本稿を執筆するに至った。

私は 2018 年 8 月 16 日から約 1 年間スイス連邦工科大学ローザンヌ校に留学する。それにあたって幾らかの留学資金が必要となるのだが、自力を集めることは難しかったためにクラウドファンディングに頼ることとなった。クラウドファンディングに頼った結果、想像以上にたくさんの人から支援していただき、十分以上とも言える留学資金が集まった。非常にありがいことであり、感謝してもしきれないことである。そうした支援・投資のお返しとして私に何かできることがないか検討した結果、私にできることで皆の役に立つものを提供しようということで私自身の専門である機械学習についての勉強資料を作成するという結論に至った。私自身まだまだ未熟で機械学習について全てを熟知しているか言われれば首を縦に振ることはできないだろう。論文のサーベイも十分かと言えばそうでもない。しかし、それでもまだまだ強化学習に関する勉強資料が少ない状況で、私が四苦八苦しながら得られた強化学習の知見をこうした形でシェアすることで強化学習という分野の今後の発展に貢献できるのではないかと考えている。

読者の皆がこれを読んでいるとき、きっと平成が終わりかけているか、もしくは終わって新たな年号となっているのだろう。これを執筆している私はまだ次の年号を知らない。しかし、未知の年号になっても本稿が読まれ続け、本稿で得られたことが誰かの役に立っていれば幸いである。

1.1 構成

本稿の構成として、基本的には数式を用いながら理論的に話を展開していく。機械学習に限らず、数値解析など様々なアルゴリズムを勉強するにあたって実装することなく理解することはできない。ましてや実装なしで理解しようとするというのは頭でっかちになってしまい、良質な理解とは言えないだろう。さながら現場を知らずに妙な閣僚決定を下す官僚のようである。そのため、理論と実装は常に表裏一体であるべく、アルゴリズムを紹介した際に擬似コードを同時に付した。擬似コードは理論と実装の橋渡しを担う素晴らしいツールである。実際に読み進めていくにあたって、擬似コードから実際のコードへと実装するのは読者への課題としたい。

1.2 対象の読者

- 強化学習について知らない人
- 機械学習に興味がある人
- 生物学の分野出身でモデル生物の行動に興味がある人
- ゲームを AI に解かせてみたい人

基本的には強化学習について平易に解説したつもりである。多少は理論的な方へ偏っているが、豊富な具体例を用いながらイメージがしやすいようになるべく簡単に解説し、難しいと思われる箇所については詳細な解説を加えている。強化学習は生物学における学習を模倣したもののため、生物を数理モデル化してシミュレーションを行うという人にも有用な内容となるだろう。

1.3　対象でない読者

- 超最新の強化学習理論を知りたい人
- 厳密な数学の議論を行いたい人

一方で、超最新の内容についてはカバーしきれていない。中には一部の分野で state-of-the-art なアルゴリズムも存在するが、本稿で紹介するアルゴリズムは 2017 年までのものとなっており、2018 年以降の研究についてはカバーしていない。今や機械学習はかなり注目を浴びている分野で、新たなアルゴリズムの開発は日々行われている。故にアキレスと亀のように、こうした資料を執筆してもすぐに新たな論文が出てくるので追いつかないのである。しかし、本稿では最新の論文が読めるようベースとなる強化学習の思考法を与えるため、本稿で学んだ内容を理解しておけばある程度の論文は訓み下せると思われる。

本稿ではある程度、数式を用いた理論の紹介を行う。しかし、それらは数式をいじるだけであり、そこに厳密な数学的議論はない。よって数学科出身というような人は本稿を読むと蕁麻疹が出てくるような気持ちになるかもしれない。私自身、数学についてそこまで深い教養があるわけではないので、時にトンチンカンな内容を書くかもしれないが、そうした面については「コイツはまだ未熟だから」と大目に見ていただきたい。

目 次

2　はじめに

「強化学習」という単語を聞いて、どのようなものをイメージするだろう？私自身、最初にこの分野を聞いたとき、「なんか強くするんかなぁ」といったよくわからないイメージを持っていた。実際のところ、そうしたイメージは遠くはないものであった。きっとまだ強化学習を知らない読者は「強化学習がどんなものか」というのを知ったら目からウロコが出るだろう。それだけ強化学習は面白い分野であるので、面白さに関しては筆者が保障しよう。

本章では強化学習の数式的な解説はせず、強化学習とはどういうものか、機械学習の中での強化学習の位置付け、生物学とのつながりなど定性的な解説を行う。

2.1　強化学習とは

ここでは簡潔に強化学習 (Reinforcement Learning) の定義を紹介しようと思う。しかし、これは機械学習全般に言えることなのだが、以下で紹介する定義が数学的に厳密というわけではないので、読者自身の理解を含めて定義については自分の中で加筆・修正を行っていただきたい。

定義：強化学習

強化学習とは、ある環境におけるエージェントが現在の状態を観測して行動を選択、それに応じて得られた報酬をもとに最適な行動を学習するアルゴリズムである。

具体例として犬がおすわりを学習する際、飼い主に「おすわり！」と言われ、そのタイミングで座るとご褒美としてオヤツがもらえる。これを強化学習の言葉で表現すると、「犬というエージェント」が「飼い主から『おすわり！』と言われた現在状態」に対して「座るという行動」を選択し、そによって「飼い主からもらえるオヤツという報酬」によって「座るという最適な行動を学習する」となる。これを図として表すと以下の通りである。

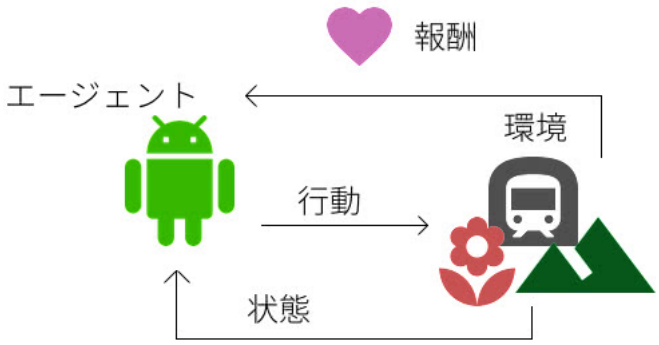

図 1: 強化学習のモデル

2.2 機械学習の中の強化学習

機械学習というものは非常に範囲が広い。言ってしまえば線形回帰のような単純なアルゴリズムも機械学習の 1 つである。説明を簡明するべく、機械学習の分類を図に示す。

図にある「教師あり学習 (Supervised Learning)」とは、エージェントの出力に対して正解のデータと照らし合わせ、その正誤判定を行い、正解のデータとの誤差がなくなるように学習を行うアルゴリズムである。具体例として、

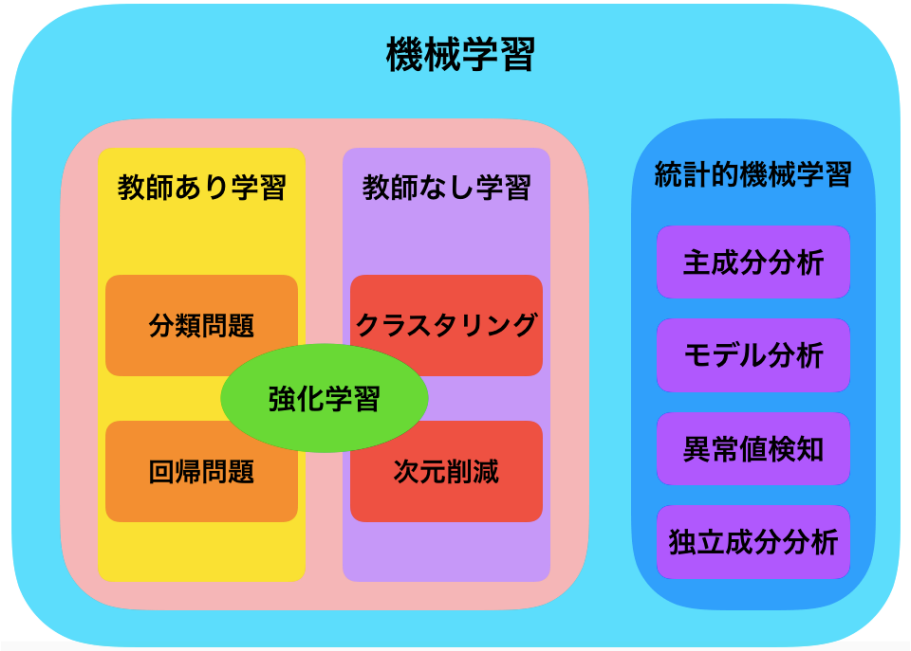

図 2: 機械学習の中の強化学習

昨今話題となっている画像認識などが挙げられる。一方で、「教師なし学習 (Unsupervised Learning)」とは、多量にあるデータから傾向などを導き出していくアルゴリズムである。これの具体例としてはデータ群に対する最適線形関数の発見などが挙げられる。

強化学習とは教師あり学習のように絶対的に正しい行動が存在しているわけではなく、かといって教師なし学習のように過去の行動の履歴だけから行動選択を行うわけでもないので、強化学習とは教師あり学習と教師なし学習のちょうど中間のようなアルゴリズムであると言うことができる。よって、図での強化学習の位置付けはこのようになるのである。

強化学習とは第 1 節の定義で示した通り、エージェントが報酬に基づいて

より良い行動を探索する最適化問題であると言うことができ、第 3 章で述べることであるが、こうした強化学習の行動選択と状態遷移は定式化することができる。

強化学習の定義について、本稿では強化学習は報酬に基づく最適出力なので教師あり学習に近しいものであると定めるが、人によっては「定式化された強化学習も報酬に基づく最適出力の学習だ！」という主張のもと、強化学習を教師あり学習として分類していることがある。こうした主張は厳密な定義がないことから必ずしも間違っているというわけでもないので頭ごなしにそうした主張を否定することはできない。しかし、研究者によって定義がまちまちであるのは事実でもある。このように定義が異なったりするのは機械学習がまだ発展段階の分野である顕著な証拠でもあるのだが、最先端の論文でも定義が異なっている場合があるので、そうした微細な違いについてあまり固執しないようにするのが良い関わり方だと言えよう。

2.3 生物学との関わり

第 1 節にて述べたことであるが、強化学習はまさに生物の学習形式を数理モデル化して実装したものと言える。この強化学習は生物の学習過程を理論的に数理モデル化したものであるが、実際に生物は強化学習的に学習していることが神経科学的アプローチによって証明されている。[1] この事実から「強化学習と実際の生物の学習過程は 1 対 1 に対応したものである」という主張が導ける。この主張に基づいた生物学的に非常に興味深い研究がいくつかあ

[1] Schultz, W Dayanm, P Montague; 1997; ”A neural substrate of prediction and reward”; Science. 275 (5306); 15931599.

るため、ここで紹介しておこうと思う。

1 つ目は強化学習を用いた動物の行動戦略の解読[2]である。この研究では動物の行動データから報酬に基づく行動戦略を明らかにする逆強化学習という手法を開発、そしてこの手法を線虫の行動へと応用することでその有効性を示したものである。通常の強化学習は報酬を前提として最適行動を学習するものであるが、この逆強化学習は先ほどの「強化学習と実際の生物の学習過程は 1 対 1 に対応したものである」という主張に基づき、すでに最適行動を行うエージェントからどのような報酬を獲得しているかを学習するアルゴリズムである。本手法によって、従来の行動が制限された行動実験系から解放され、より自然な状況において自由に振る舞う動物の行動戦略の研究が進むことが期待される。

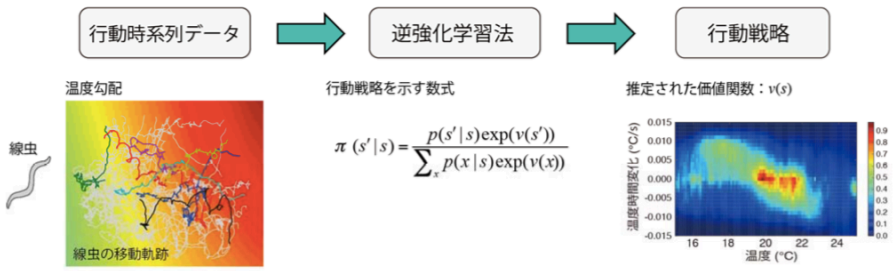

図 3: 逆強化学習による報酬推定

2 つ目は群強化学習を用いた複数の生物モデルの迷路学習[3]である。これは Google の発表した論文で、複数のエージェントを同じ環境下に置いて相互の

[2]S Yamaguchi et al; "Identification of animal behavioral strategies by inverse reinforcement learning"

[3]J Perolat et al; "A multi-agent reinforcement learning model of common-pool resource"

情報交換を効率的に行うことでエージェントが単体である場合よりも効率的に学習をすることができる、といったことを提案したアルゴリズムである。こうした内容は実際にはちが花の蜜のありかを集団に情報伝達していることや、アリがエサなど情報を集団へ情報伝達を行なっていることなど、実際の生物集団にも確認できることで、これも群衆生態学的な視点から行けば非常に興味深い内容と言えるだろう。

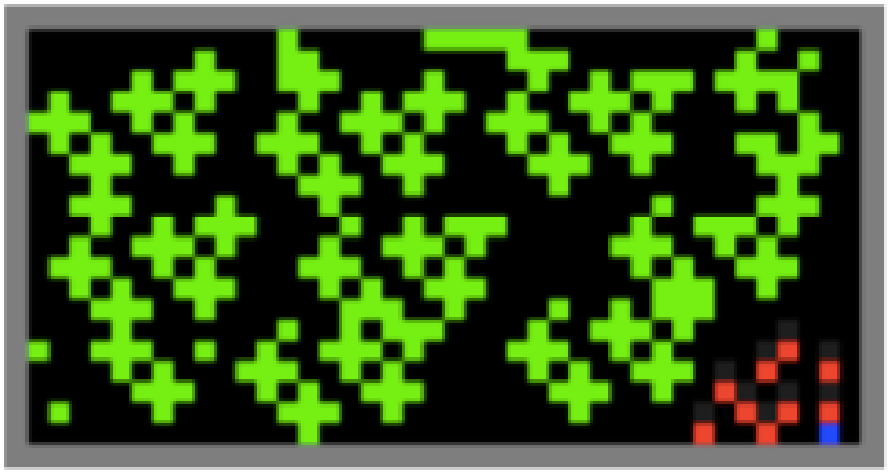

図 4: 群強化学習による情報伝達

以上より、強化学習は学習に報酬を用いている事実から生物学と非常に密接に関わっており、相互的に新たな知見を提供し合っていると言える。今後、生態学を専攻する場合や線虫などのモデル生物を研究する上で強化学習は非常に有用な道具となっていくだろう。

3　強化学習の定式化

第 2 章までは強化学習について定性的な話を展開してきた。本章では今まで定性的に論じてきていた強化学習のアルゴリズムについて具体的に定式化していく。

3.1　状況設定

まず最初に、強化学習の具体的なアルゴリズムについて確認しておこう。計算フローとしては以下の通りである。

強化学習の計算フロー

1. 現在の環境を観測
2. 現在の判断材料に基づき、最適行動を選択
3. 行動
4. 行動によって環境が変化
5. 報酬を得る
6. 最適行動を改めて検討、学習
7. 1. に戻る

登場人物としてはエージェントと環境の 2 種類で、このように状況を観測しては最適行動を選択する。以下ではこの状況設定に基づいて数理モデル化

していく。

3.2 定義

本節では定義を述べていく。最初に断っておくと、今回は簡単のために離散時間を想定して状態を記述していく。離散の場合の数式をグッとにらめば連続的な状況の定式化は容易だが、今回は離散時間をテーマに話を進めていくこととする。

今後使う文字の定義

$\mathcal{S} \subset \mathbb{Z}^d$：状態空間

$\mathcal{A} \subset \mathbb{Z}^1$：行動空間

$s_t \in \mathcal{S}$：時刻 t における環境の観測状態

$a_t \in \mathcal{A}$：時刻 t において選択した行動

$\pi(a_t|s_t) \in [0, 1]$：状態 s_t における方策 (行動選択の確率)

$r_t \in \mathbb{R}^1$：行動 a_t によって得られた報酬

$V^\pi(s_t) \in \mathbb{R}^{\not\vdash}$：方策 π で状態 s_t における価値

$Q^\pi(s_t, a_t) \in \mathbb{R}^{\not\vdash}$：方策 π で状態 s_t に置いて行動 a_t の価値

第 1 節で表した計算フローにて、パッと見で必要なものはだいたいこのようなものである。

時刻や観測状態については比較的理解しやすいものであると思われるが、状態空間と行動空間についてはあまりイメージがつかないものだと思われるので具体例を用いながら少し説明を付け加えよう。

シチュエーションとして「エージェントが迷路問題を解く」問題を考える。このとき、迷路は 20×20 の形でトータル 400 マスで構成される。

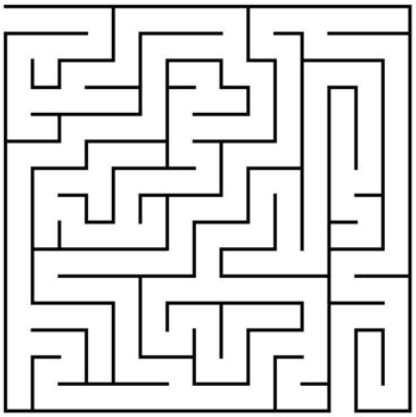

図 5: 20×20 の迷路

各マスはどの方面に壁があるのかで 14 通りである。このマップについて、左上を原点として水平方向に x 座標を、鉛直方向に y 座標をとった際、座標にとってエージェントのいる位置が $(2, 4)$ だとすれば。現在エージェントがいるマスは x 方向への移動しか許さないマスである (y 方向の両方に壁がある)。

このマップの状態空間 $\mathcal{S}$ は以下のように定式化される。

$$\mathcal{S} = \left\{ m \in \mathbb{Z}^{400} \mid m = 0, 1, \ldots, 13 \right\} \tag{1}$$

このとき、0 から 13 までの整数値はマスの種類で、どの方面に壁があるのかを表す。状態空間はこのように定義される。見方を変えれば、状態空間は

エージェントに行動を起こした際の報酬がどのようになるかのファクターであるという説明も可能である。実際に、迷路ゲームにおいてゴールの一歩手前のマスで、正しくゴールに到達できる行動をとれば報酬として +10 点が入ってくるし、逆にゴールに到達しない行動をとった場合はただスタミナを消費しただけで −0.1 点というようになるだろう。

次に行動空間について説明する。これは状態空間が理解できれば行動空間についての理解が簡単である。先に挙げた迷路ゲームでは上下左右の 4 パターンの移動が可能である (ここで「その場でも立ち止まることもできるぞ！」と思ったかもしれないが、このエージェントは歩き続けないと死んでしまう病気にかかっていると思っておいて欲しい)。このように行動空間は要素に行動をもつ集合と言える。実際に行動空間 $\mathcal{A}$ を定式化すれば以下の通りである。

$$\mathcal{A} = \{0, 1, 2, 3\} \tag{2}$$

中に入っている数字はそれぞれ上下左右に対応する。

状態空間と行動空間は一見難しそうな概念であるが、実際のところはそれぞれ取りうる状態や行動の選択肢の集合であり、そこまで難しいことはない。

次に状態価値 $V(s_t)$ と状態行動価値 $Q(s_t, a_t)$ について説明する。状態価値 $V(s_t)$ とは、「ある状態 s_t がその後どれだけ報酬を得られそうか」というのを表す指標で、例えば迷路ゲームで言えばゴールまで 1 歩手前の状態とゴールまで 30 歩手前の状態では前者の方が状態価値が高そうな気がするだろう。実際にどのように定義するかについては様々な議論があるので、それらについては第 5 節に譲りたいと思う。状態行動価値 $Q(s_t, a_t)$ についても同様である。

ここで状態行動価値 $Q(s_t, a_t)$ について、状態 s_t と行動 a_t の 2 つを引数と

してとるので、状態行動価値については 1 列のベクトルで考えるよりもテーブルとして考えた方がイメージがつきやすいであろう。実際に、状態 s_t と行動 a_t に対応させた Q 値を表したテーブルを Q-table などと表現したりする。具体例を以下に載せておく。

	a_1	a_2	...	a_n
s_1	10.3	-810	...	-33.4
s_2	114.514	19.19	...	810
⋮	⋮	⋮	⋱	⋮
s_k	-93.1	53	...	12.3

3.3 離散か、連続か

状態空間と行動空間について、先ほどは迷路ゲームという状態空間と行動空間がともに離散な状況を想定した。しかし状況によっては状態空間か行動空間が連続な場合も考えられうるだろう。ここで、連続とは数学的に厳密に定義された連続のことではない。コンピュータが以下に優れたものであろうと、現在のアーキテクチャでは連続を定義することはできず、所詮は 0 か 1 のバイナリがその本質である。よって連続というものはコンピュータでは表現できない。その代わりにグリッドの目を細かくすることで連続を記述するのである。このような部分については数値解析などを経験したことがある人ならば非常に親しみ深い事実であろう。

図 6: 各空間が離散か連続かでの想定環境

状態空間と行動空間は集合として記述され、連続な場合は要素が連続となるだけで空間の記述自体は容易であるが、空間が連続なタスクについては離散の場合に比べて注意が必要である。それは何かというと、連続な場合は空間の要素数が増えて計算量が爆発してしまうという問題がある。

例えば先ほどの 20×20 の迷路ゲームでは状態数は高々400 であるが、これが 210×160 ピクセルで RGB の 3 色スケールだと状態数はかなり大きくなる。それらの状態数に対して最適行動選択の計算が収束するには非常に時間がかかる。このように、状態空間の要素数が増えると指数関数的に計算量が増大し、最終的には計算量が爆発して解が現実的な時間では収束しなくなる現象を「次元の呪い (Curse of Dimension)」という。

この「次元の呪い」は昔から非常に悩ましい問題であったが、現在この問題は Deep Learning の登場によって徐々に解決されつつある。詳細については第 6 章を参照されたい。ここでは説明を簡潔にするべく「次元の呪い」と

いう概念の紹介に留めておく。

3.4 マルコフ性

ここでマルコフ性について定義しておく。というのも以下に示していくベルマン方程式はマルコフ性に基づいた状態遷移方程式のためである。

定義：マルコフ性

マルコフ性とは時刻 $t+1$ の状態は時刻 t によってのみ左右される、という性質である。

このマルコフ性を仮定すると、現在の状態と行動から次の時刻の状態と報酬を予測することができる。さらに、繰り返し計算により、数学的帰納法によって今後将来の全ての状態と報酬を予測することができる。

マルコフ性を満足する強化学習はマルコフ決定過程 (MDP; Marcov Decision Process) と呼ばれる。有限 MDP では、任意の状態 s_t と行動 a_t が与えられることで次の時刻 s_{t+1} を記述することができるのである。

以下ではこのマルコフ性を満足するベルマン方程式を用いて強化学習を具体的に定式化していく。

3.5 ベルマン方程式

本節では、前節まで定義した文字を用いながら強化学習の基本公式とも言えるベルマン方程式を導出する。まず最初に状態価値関数 $V(s)$ を定式化してい

こう。ここで、状態価値の定式化において割引率というものを用いる。というのも状態価値は各時刻の報酬を無限時間まで足し合わせたものなのだが、単純に足し合わせただけでは状態価値の値は発散する。そこで割引率 $\gamma \in \mathbb{R}_{0<\gamma<1}$ を各時刻の補修に掛け合わせることで状態価値の発散を防ぐことができる。それに加え、割引率を導入することでどの程度の時刻の報酬を重要視するかをコントロールすることができ、一石二鳥なのである。

以上のもと、状態価値関数 $V(s)$ は期待値を用いて以下の通りに定義される。

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{n=t}^{\infty} \gamma^{n-t} r_n \ ; \ s_t = s\right] \tag{3}$$

また、状態行動価値関数 $Q(s,a)$ についても同様に定義される。

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{n=t}^{\infty} \gamma^{n-t} r_n \ ; \ s_t = s, a_t = a\right] \tag{4}$$

ここで $\mathbb{E}[\,\cdot \mid \cdot\,]$ は期待値を表す。これらの意味するところは、時刻 t で状態が s_t であった場合、それ以降の時刻で得られうる報酬の合計の期待値を価値として定義する、ということである。

この状態価値 $V(s)$ と状態行動価値 $Q(s,a)$ は以下のような関係にある。

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s_t) Q^{\pi}(s,a) \tag{5}$$

方策 $\pi(a|s)$ は状態 s において行動 a を選択する確率であるので、行動 a について和をとれば 1 であることに注意したい。

この価値と報酬に関する数値機を少しいじることでこれらの価値関数を時

間差分によって再帰的に定義できる。

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\left[\sum_{n=t}^{\infty}\gamma^{n-t}r_n \ ; \ s_t = s\right] && (6)\\
&= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \ ; \ s_t = s\right] && (7)\\
&= \mathbb{E}\left[r_t + \gamma \sum_{n=t+1}^{\infty}\gamma^{n-(t+1)}r_n \ ; \ s_t = s\right] && (8)
\end{aligned}
$$

期待値については方策 $\pi(a_t|s_t)$ と状態遷移確率 $P(s_{t+1}|s_t, a_t)$ を用いることで $\sum$ を表現できる。

$$
\begin{aligned}
V^{\pi}(s'_t) &= \mathbb{E}\left[r_t + \gamma \sum_{n=t+1}^{\infty}\gamma^{n-(t+1)}r_n \ ; \ s_t = s'_t\right] && (9)\\
&= \sum_{a\in\mathcal{A}}\sum_{s_{t+1}\in\mathcal{S}}\pi(a|s'_t)P(s_{t+1}|s'_t, a)(r_t + \gamma V^{\pi}(s'_{t+1})) && (10)
\end{aligned}
$$

以上から、状態価値を再帰的に定義した綺麗な式が求まる。

$$
V^{\pi}(s_t) = \sum_{a\in\mathcal{A}}\sum_{s_{t+1}\in\mathcal{S}}\pi(a|s_t)P(s_{t+1}|s_t, a)(r_t + \gamma V^{\pi}(s_{t+1})) \qquad (11)
$$

状態行動価値 $Q(s, a)$ についても同様に表現できて

$$
Q^{\pi}(s_t, a_t) = \sum_{s_{t+1}\in\mathcal{S}}P(s_{t+1}|s_t, a_t)(r_t + \gamma \sum_{a_{t+1}\in\mathcal{A}}\pi(a_{t+1}|s_{t+1})Q^{\pi}(s_{t+1}, a_{t+1})) \qquad (12)
$$

ここで状態価値 $V(s)$ は行動が未定であったが故に、期待値を考慮する際に方策を計算に含まなければならなかったが、状態行動価値 $Q(s, a)$ は行動をすでに条件に加えているので方策についての $\sum$ を取る必要はないのである。逆に 1 ステップ以後の状態行動価値 $Q(s_{t+1}, a_{t+1})$ は行動がまだ定まっていないので方策について $\sum$ を取らなければならない点に注意である。

これらの表現の意味するところは、「ある状態で行動をとったとき、その行動の価値は、その行動で得られた報酬と次の状態で (方策に従って) 行った行動の価値の和の期待値」ということになる。

これらの方程式はある方策 π のもとで得られる期待報酬を表しているが、今後はエージェントが最適な方策を獲得している場合について書き表してみよう。

まず最初に、普通の方策と区別するべく、最適方策を π^* と表す。また、最適方策における状態価値や状態行動価値をそれぞれ $V^*(s)$、$Q^*(s,a)$ と表す。

最適方策では行動の選択確率は決定的 (ある行動を取る確率が 1 で、それ以外は 0) なので、行動に関する $\sum$ を $\max_{a\in\mathcal{A}}$ で代用して方程式を表現することができる。

これを表すと以下の通りである。

$$V^*(s_t) = \max_{a\in\mathcal{A}} \left[\sum_{s_{t+1}\in\mathcal{S}} \pi(a|s'_t) P(s_{t+1}|s'_t, a) \left\{ r_t + \gamma V^*(s_{t+1}) \right\} \right] \tag{13}$$

もちろん、最適状態行動価値 $Q^*(s,a)$ についても同様である。

$$Q^*(s_t, a_t) = \sum_{s_{t+1}\in\mathcal{S}} P(s_{t+1}|s_t, a_t) \left\{ r_t + \gamma \max_{a_{t+1}\in\mathcal{A}} Q^*(s_{t+1}, a_{t+1}) \right\} \tag{14}$$

以上でベルマン方程式の導出は完了である。このような各時刻において最適行動を決定していくような手法を「動的計画法」という。

ついでにここでベルマンの最適性原理について紹介しておこう。

ベルマンの最適性原理 (Principle of Optimality)

最適な方策は、初期状態と初期決定がどんなものであれ、その結果得られる次の状態に関して、以降の決定が必ず最適方策になっているという性質を持つ。

これの意味するところは、「強化学習の状態遷移式のような動的決定問題は、より小さな部分問題に分割することが可能である」ということを主張している。実際に状態 s_t についての状態価値や状態行動価値について、数式を操作することによって s_{t+1} における状態価値や状態行動価値についての問題へ、小さな問題へ分割できている。このような分割可能な問題をコンピューターサイエンスの言葉で「部分構造最適性 (Optimal Substructure) を持つ」という言葉で表現したりする。

3.6 モンテカルロ法

ベルマン方程式の導出に際して、状態価値や状態行動価値は割引報酬和の期待値を用いて表し、状態遷移確率と方策を用いることで期待値の記号を $\sum$ を用いて表現し、そうして再帰的な定義式を導出した。

しかし現実の問題を考えた際、状態遷移確率が明にわかっているシチュエーションはそう多くない。実際に、将棋やチェスなどで、自分がコマをおいたあと次に自分のターンになったときどのような盤面になっているかは相手のコマの置き方次第であり、状態遷移については非自明であろう。

迷路ゲームのように状態遷移確率がわかっている状況ではベルマン方程式は簡単に解くことができるが、状態遷移確率がわからなければベルマン方程式は解くことができない。

ここで、環境のことを「モデル」と呼び、状態遷移確率がわかっている場合の問題解法を「モデルベース (Model-based)」、状態遷移確率がわかっていない場合の問題解法を「モデルフリー (Model-free)」という。ベルマン方程

式を愚直に解く手法がモデルベースに相当する。

では状態遷移確率がわからない状況ではどうするかというと、たくさん試行錯誤するのである。たくさん試行錯誤をしたとき、その状態の遷移は近似的に状態遷移確率に従う。何度も繰り返し実行することでそのときの状態遷移は「(知ることができない) 本当の状態遷移確率」に従うので、その後で得られる報酬の平均は期待値に収束、そうして状態価値や状態行動価値がわかるのである。

このように実際に何度も繰り返して得られた経験をもとに状態価値や状態行動価値を推定する手法を「モンテカルロ法」という。

モンテカルロ法の具体的なアルゴリズムについて見ていこう。状況として、M 回学習し、現在 m 回目の学習中だとする。また、時刻 $t = 0$ から試行を開始し、時刻 $t = T$ に試行が終了したとする。そのとき、状態列 $s_0, s_1, \ldots, s_T$ に対して報酬列 $r_0, r_1, \ldots, r_T$ を観測する。これらの報酬列に対して状態価値を推定していくのだが、「初回訪問モンテカルロ法」と「逐一訪問モンテカルロ法」の 2 つの手法があるのでここで紹介しておく。

初回訪問モンテカルロ法

状態列 $s_0, s_1, \ldots, s_T$ について、状態 s_t が初めての訪問なら収益 $R_m(s_t)$ について

$$R(s_t) \leftarrow R(s_t) + \sum_{t'=t+1}^{T} r_{t'} \tag{15}$$

として状態価値 $V(s_t)$ を

$$V(s_t) \leftarrow \frac{1}{m} R \tag{16}$$

とする。ここで初回訪問という条件を入れているのは、観測された状態列の中に同じ状態が複数回現れる可能性があ理、各状態の平等に評価するためである。

逐一訪問モンテカルロ法

初回訪問モンテカルロ法では「状態 s_t が初回訪問なら状態価値 $V(s_t)$ を評価する」としていたが、逐一訪問モンテカルロ法は任意の時刻の状態 s_t について収益 $R(s_t)$ を導き出し、状態価値 $V(s_t)$ を評価する。

ここでは状態価値を評価したが、モデルが不明であるので状態価値だけ算出してもどうしようもないのである。どういうことかというと、行動 a をとった際にモデルが不明なので状態がどのように遷移するか不明であり、その行動による収益の期待値が計算できないのである。よって状態価値がわかっていてもとるべき行動が決定できないのである。

そのため、モデルフリー手法では状態価値ではなく状態行動価値を評価する。状態行動価値の値がわかっていれば、最適行動は状態行動価値が最も大きい行動を選択することが最適行動選択となるのだ。

よって状態行動価値を算出するのだが、ここで一つ問題が存在する。それは「評価しようとしている状態行動価値 $Q(s,a)$ について、状態行動タプル (s,a) が観測されないとその状態行動価値 $Q(s,a)$ を評価することができない」という問題である。実際に状態 s' において行動 a' を選択する確率が 0 であると、その状態行動タプル (s',a') は永遠に観測されない。

この問題を解決するための手法が 3 つあるので、それらについて紹介していこうと思う。

3.6.1 開始点探査の仮定

「開始点探査 (Exploring Starts) の仮定」とは「任意の状態行動タプル (s, a) について $(s_0, a_0) = (s, a)$ となる確率が 0 ではない」という仮定である。

このような仮定をおいてしまえば決定論的な方策を用いていたとしても任意の状態行動タプル (s, a) が観測されないということは起こらない。このようなモンテカルロ法を「モンテカルロ-ES 法」という。

さて、実際のアルゴリズムとしては先ほど状態価値を評価したときと同様に、状態と行動のタプル列 $(s_0, a_0), (s_1, a_1), \ldots, (s_T, a_T)$ とそれらに対応する報酬列 $r_0, r_1, \ldots, r_T$ を保存しておいて、それで収益を評価するのである。

擬似コードとして表すと以下の通りである。ただし逐一訪問モンテカルロ法を採用している。

```
Q <- Initialize
R <- Initialize

## 学習回数 ##
for (episode : 0 ... M):

    ## 状態など初期化 ##
    state <- Initialize
    List(tuples) <- Initialize

    ## 状態行動タプル列の保存 ##
    for (step : 0 ... T):
        action <- pi(state)
        state_new, reward <- move(action)
        List.append(state,action,reward)
        state <- state_new

    ## 状態行動価値の評価 ##
    for (step : 0 ... T):

        ## 収益 ##
        R(state) <- sum(reward)

        ## 状態行動価値を更新 ##
        Q(state, action) <- R(state) / episode
```

3.6.2 方策オン型モンテカルロ制御

モンテカルロ-ES 法では開始点探査の仮定をおくことで「評価しようとしている状態行動価値 $Q(s,a)$ について、状態行動タプル (s,a) が観測されないとその状態行動価値 $Q(s,a)$ を評価することができない」という問題を回避した。

一方で、方策オン型モンテカルロ法ではこの問題に対して「確率的な方策を用いる」というアプローチによってこの問題を解決する。どういうことかというと、決定論的な方策を用いていると $\pi(s,a)=0$ というような場合で状態行動タプル (s,a) を一切観測することはない。しかし、確率的な方策を用いることで任意の状態行動タプル (s,a) について $\pi(s,a)>0$ を保証するのである。この手法によって開始点探査の仮定を外すことができる。

確率的な方策は主に ϵ-greedy 法と softmax 法が使われる。ここで ϵ-greedy 法とは以下のような方策である。

定義：ϵ-greedy 法

ϵ を小さな値とする。状態 s_t において行動 a_t を選択する確率は以下の通りに定義される。

$$\pi(a_t|s_t)=\begin{cases}1-\epsilon & (a_t=\arg\max_{a\in\mathcal{A}} Q(s_t,a)) \\ \epsilon & (otherwise)\end{cases} \tag{17}$$

ここで ϵ は学習回数に応じて減衰させていくことで学習の収束が保証される。

また、softmax 法は以下のような方策である。

softmax 法

温度パラメータと呼ばれるパラメータ $\tau \in \mathbb{R}_{>0}$ を用いて、状態 s_t において行動 a_t をとる確率は以下の通りに定義される。

$$\pi(a_t|s_t) = \frac{\exp\left[Q(s_t, a_t)/\tau\right]}{\sum_{b \in \mathcal{A}} \exp\left[Q(s_t, b)/\tau\right]} \tag{18}$$

ここで τ は学習率をコントロールするパラメータで、$\tau \to 0$ で ϵ-greedy 法と一致する。

これらを用いて状態行動価値を学習していく。アルゴリズムとしてはモンテカルロ-ES 法と同様である。

擬似コードを以下に示す。ただし、逐一モンテカルロ法を採用している。

```
Q <- Initialize
R <- Initialize

## 学習回数 ##
for (episode : 0 ... M):

    ## 状態など初期化 ##
    state <- Initialize
    List(tuples) <- Initialize

    ## 状態行動タプル列の保存 ##
    for (step : 0 ... T):
        action <- pi(state)
        state_new, reward <- move(action)
        List.append(state,action,reward)
        state <- state_new

    ## 方策評価 ##
    for (step : 0 ... T):

        ## 収益 ##
```

```
        R(state, action) <- sum(reward)

        ## 状態行動価値を更新 ##
        Q(state, action) <- R(state, action) / episode
```

ここで、今回は方策を既に持っている Q 関数を参照する関数として表記したが、方策を確率を要素として持った配列のような形で実装することも可能で、その場合は別途で方策改善部分を実装する必要があろう。

3.6.3 方策オフ型モンテカルロ制御

方策オン型モンテカルロ制御では方策として確率的なものを用いることによって開始点探査の仮定を外したが、方策オフ型モンテカルロ法では評価・改善される方策 π と状態行動タプル列を作るための方策 π' という 2 つの方策を用いることで開始点探査の仮定を外す。評価・改善される方策 π を「推定方策」といい、実際の行動する方策を「挙動方策」という。方策オン型モンテカルロ制御ではこの推定方策と挙動方策が同一ものであったと言える。

さて、方策オフ型モンテカルロ制御では挙動方策を用いることで推定方策を改善するのであるが、これには 1 つ問題が存在する。それは「方策が変わってしまえばそこで得られた収益も挙動方策での収益であり、推定方策でも同じ収益が得られるとは限らない」という問題である。

この問題の原因は「推定方策と挙動方策では収益が異なってしまう」「片方だけ改善したら、当然その収益は改善した方だけ良くなる」といったことが挙げられる。つまり、改善した際の方策ごとの差がなくなればこの問題は解決できよう。

そこで実際に観測された状態行動タプル列 $(s_0, a_0), (s_1, a_1), \ldots, (s_T, a_T)$ に

ついて、それぞれの方策で観測される確率を比較し、それを収益を評価するときの重みとして利用する、という手法を考える。

実際に、推定方策 π において状態行動タプル列 $(s_0, a_0), (s_1, a_1), \ldots, (s_T, a_T)$ と収益 R が観測される確率を p、挙動方策 π' においてこの状態行動タプル列と収益が観測される確率を p' とする。このとき、推定方策 π においてこの状態行動行動タプル列と収益を確率的に $\frac{p}{p'}$ 回観測すると言える。

このようにすることで推定方策と挙動方策が同一ならば $\frac{p}{p'} = 1$ で方策オン型モンテカルロ制御と一致する。また、ある状態行動タプル列を観測する確率が推定方策の方が大きい場合、この状態行動タプル列と収益を $\frac{p}{p'} > 1$ 回観測していることになり、その収益が推定方策では強い意味を持つということになって挙動方策もそれに応じて改善される。

この重みを用いた収益評価は、今まで推定方策で収益 R が 1 回観測されたとき

- $R_{sum} \leftarrow R_{sum} + R$
- $k \leftarrow k + 1$
- $Q^{\pi}(s, a) \leftarrow \frac{1}{k} R_{sum}$

となっていたところが重みつき収益評価によって

- $R_{sum} \leftarrow R_{sum} + \frac{p}{p'} R$
- $k \leftarrow k + \frac{p}{p'}$
- $Q^{\pi}(s, a) \leftarrow \frac{1}{k} R_{sum}$

となる。ここで状態行動タプル列と収益の観測確率 p について、ある時刻 t において状態行動タプル (s_t, a_t) を観測し、その後で状態行動タプル列 $(s_{t+1}, a_{t+1}), (s_{t+2}, a_{t+2}), \ldots$ を観測する確率を $p^\pi(s_t, a_t)$ とすると、状態遷移確率 $P(s_{t+1}|s_t, a_t)$ と方策 $\pi(a_t|s_t)$ を用いて

$$p^\pi(s_t, a_t) = P(s_{t+1}|s_t, a_t) \prod_{k=t+1}^{\infty} \pi(a_k|s_k) P(s_{k+1}|s_k, a_k) \tag{19}$$

と表されるので、挙動方策 π' での収益 R の評価に対しての推定方策 π での収益の評価の重みを $w(s_t, a_t)$ とすると、これは

$$\begin{aligned} w(s_t, a_t) &= \frac{p^\pi(s_t, a_t)}{p^{\pi'}(s_t, a_t)} && (20) \\ &= \frac{P(s_{t+1}|s_t, a_t) \prod_{k=t+1}^{\infty} \pi(a_k|s_k) P(s_{k+1}|s_k, a_k)}{P(s_{t+1}|s_t, a_t) \prod_{k=t+1}^{\infty} \pi'(a_k|s_k) P(s_{k+1}|s_k, a_k)} && (21) \\ &= \prod_{k=t+1}^{\infty} \frac{\pi(a_k|s_k)}{\pi'(a_k|s_k)} && (22) \end{aligned}$$

となって重み $w(s_t, a_t)$ は状態遷移確率に依存しない値なので方策のみで計算可能である。

以下では推定方策が決定論的なものである場合を考える。このとき重み $w(s_t, a_t)$ は

$$w(s_t, a_t) = \begin{cases} \frac{1}{\pi'(s_t, a_t)} & (a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a)) \\ 0 & (otherwise) \end{cases} \tag{23}$$

となり、選択行動が最適行動選択でなかった場合は重み $w(s_t, a_t)$ が 0 となる。試行を時刻 $t = 0$ から時刻 $t = T$ まで行ったとき、時刻 $t < \tau$ で選択行動が全て最適行動であったが時刻 $t = \tau$ で初めて最適行動でなくなったときは、先ほどの重みの計算式によりその後の時刻の重みは全て 0 となる。よって時刻 τ より前の時刻については学習しても意味がないのである。

これを踏まえた上での方策オフ型モンテカルロ制御の擬似コードは以下の通りである。ただし逐一モンテカルロ法を採用している。

```
Q <- Initialize
R <- Initialize

## 学習回数 ##
for (episode : 0 ... M):

    ## 状態など初期化 ##
    state <- Initialize
    List(tuples) <- Initialize

    ## 状態行動タプル列の保存 ##
    for (step : 0 ... T):
        action <- pi(state)
        state_new, reward <- move(action)
        List.append(state,action,reward)
        state <- state_new

    ## 最適行動でなくなる時刻判定 ##
    tau <- check(List)

    ## 方策評価 ##
    for (step : 0 ... T):

        ## 重み評価 ##
        w <- prod(1/pi_soft)

        ## 収益 ##
        R(state, action) <- sum(reward)

        ## 重みつき収益を計算 ##
        update = w * R(state, action)
        R_sum(state,action) <- R_sum(state,action) + update

        ## カウンタ ##
        k <- k + w

        ## 状態行動価値を更新 ##
        Q(state, action) <- R_sum(state,action) / w
```

3.7 TD 学習

ベルマン方程式はモデルがわかっている場合、つまり状態遷移確率がわかっている場合に有効な手法であり、モデルが不明な場合についてはモンテカルロ法を使えば良いということであった。しかし、モンテカルロ法は試行が終了したときに方策を改善するので試行が終了するまで方策の更新ができないという欠点があった。こうした欠点を回避したのが TD 学習 (Temporal Difference Learning) である。

TD 学習には基本となる「1 ステップ TD 法」があり、それを拡張した「k ステップ TD 法」、さらに拡張した「TD(λ) 法」がある。以下ではそれら一つ一つを解説していく。

3.7.1 1 ステップ TD 法

TD 学習の基本思想はモンテカルロ法と同様で、試行錯誤をすることで状態価値や状態行動価値を求める。TD 学習は、モンテカルロ法の「試行が終了するまで方策が改善されない」という問題を解決し、1 ステップごとに方策を改善する。

ベルマン方程式を導出する際、状態価値を割引報酬和の期待値で定義し、その中に現れる $\sum$ を部分的に展開することで状態価値に関する再帰的な定義式を導出した。その中で、状態価値 $V(s_t)$ は $r_t + \gamma V(s_{t+1})$ を方策 π と状態遷移確率 $P(s_{t+1}|s_t, a_t)$ について $\sum$ をとったものと等しくなるとあった。ここで、天下り式ではあるのだが、ベルマン方程式は $r_t + \gamma V(s_{t+1})$ の部分を $V(s_t)$ とすると両辺の等号が成立する。実際に、行動についてとった $\sum$ は

よってベルマン方程式の解の一つは

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{24}$$

であると言える。これは状態行動価値 $Q(s, a)$ についてのベルマン方程式も同様に

$$Q(s_t, a_t) = r_t + \gamma Q(s_{t+1}, a_{t+1}) \tag{25}$$

これが解の 1 つとなる。最適方策を獲得しているとき、方策は決定的となるので

$$\sum_{a \in \mathcal{A}} \pi(a|s) Q(s, a) = Q(s, a) \tag{26}$$

が成立することに注意したい。

この等号が成立するとき、ベルマン方程式が成立していることから状態価値 $V(s_t)$ は割引報酬和の期待値を表現できていることとなり、それに応じて最適な方策を獲得できていると言えることは明白であろう。

しかし、実際のところ学習が収束していない (=状態価値や状態行動価値が最適な値に収束していない) 場合はこの等号が成立しておらず、左辺と右辺でズレが生じる。よって TD 学習の目的はこの左辺と右辺のズレを是正することが目的となる。

さて、TD 学習を行うのだが、先ほどのベルマン方程式の解についての議論から、$V(s_t)$ が $r_t + \gamma V(s_{t+1})$ に近くなるように $V(s_t)$ を更新していけばいいとわかる。このときの状態価値 $V(s_t)$ の更新方法は単純に

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_t + \gamma V(s_{t+1}) - V(s_t) \right] \tag{27}$$

として、これを何度も繰り返せば状態価値 $V(s_t)$ は最適な値に収束する。ここで α は学習率というパラメータで、学習速度をコントロールする。実際に、状態 $V(s_t)$ について $r_t + \gamma V(s_{t+1}) > V(s_t)$ であるとこの更新式は状態価値 $V(s_t)$ の値を少し大きくすることになり、$r_t + \gamma V(s_{t+1})$ と $V(s_t)$ の差を小さくすることになる。

以上では状態価値 $V(s)$ についての式展開をしたが、状態行動価値 $Q(s, a)$ についても同様の更新式が考えられる。

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{28}$$

この状態行動価値の更新式について、この更新材料となる $r_t + \gamma Q(s_{t+1}, a_{t+1})$ の行動 a_{t+1} の選択方法については様々な議論がある。行動 a_{t+1} の選択方法は様々な提案手法があり、それらについては第 5 章を参照されたい。

ここで学習率パラメータ α の注意について述べておこう。一般に学習率 α には小さな正数を採用するのだが、ここで大きすぎる値を採用すると最適解に収束せず最適解の周辺をウロウロするようになる。かといって学習率に小さすぎる値を採用すると学習の進みが非常に遅くなり現実的な時間では学習が収束しなくなる可能性がある。よって学習率の選択は強化学習のパラメータチューニングでは非常に重要な作業となる。

3.7.2 k ステップ TD 法

モンテカルロ法は「試行が完全に終了したとき」価値関数の更新を行っており、TD 法は「1 ステップごとに」価値関数の更新を行っていた。ではこれ

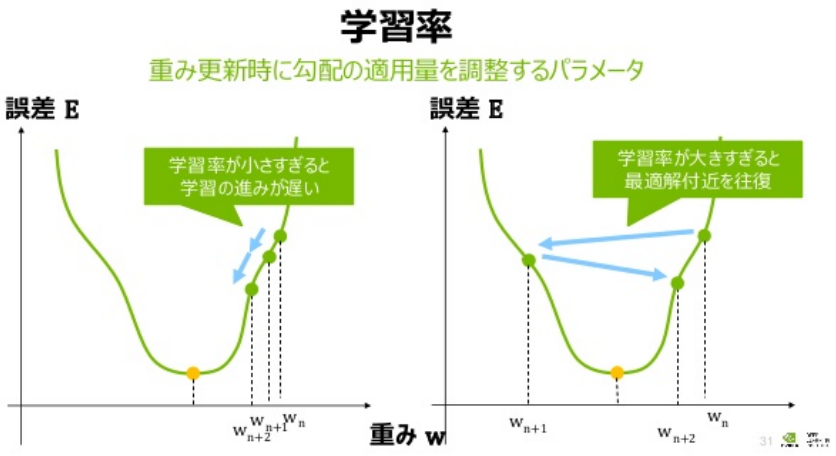

図 7: 学習率の選択方法

らのモンテカルロ法と TD 法の折衷案のようなアイデアも存在しても問題ないだろう、という発想のもとに考案されたのが k ステップ TD 法である。

1 ステップの TD 学習を実現するにあたって、価値関数に関する定義式において $\sum$ を部分的にほどく作業を通して式変形することでマルコフ性を満足するベルマン方程式を導出し、そうして TD 学習を表現した。しかし、この $\sum$ を部分的にほどく作業について、マルコフ性を棄却して k ステップ分の報酬を考慮したベルマン方程式を導出することはできないだろうか？

状態価値の定義式の式変形について、通常のベルマン方程式では

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\left[\sum_{n=t}^{\infty}\gamma^{n-t}r_n \ ; \ s_t = s\right] && (29)\\
&= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \ ; \ s_t = s\right] && (30)\\
&= \mathbb{E}\left[r_t + \gamma \sum_{n=t+1}^{\infty}\gamma^{n-(t+1)}r_n \ ; \ s_t = s\right] && (31)
\end{aligned}
$$

としたが、この過程で確かに k ステップへと拡張する余地がある。これを

拡張して k ステップの場合へ一般化すると

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}\left[\sum_{n=t}^{\infty}\gamma^{n-t}r_n \ ;\ s_t = s\right] && (32)\\
&= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ ;\ s_t = s\right] && (33)\\
&= \mathbb{E}\left[r_t + \gamma r_{t+1} + \cdots + \gamma^k \sum_{n=t+k}^{\infty}\gamma^{n-(t+k)}r_n \ ;\ s_t = s\right] && (34)\\
&= \mathbb{E}\left[\sum_{n=0}^{k-1}\gamma^n r_{t+n} + \gamma^k \sum_{n=t+k}^{\infty}\gamma^{n-(t+k)}r_n \ ;\ s_t = s\right] && (35)
\end{aligned}
$$

となる。これを TD 学習の更新方法に持ち込むと

$$
V(s_t) \leftarrow V(s_t) + \alpha\left[\sum_{n=0}^{k-1}\gamma^n r_{t+n} + \gamma^k V(s_{t+k}) - V(s_t)\right] \qquad (36)
$$

となり、「k ステップ TD 法」を実現できる。

ここで k ステップ TD 法の部分で、$k = 1$ とすることで 1 ステップ TD 法と一致し、$k \to \infty$ とすることでモンテカルロ法と一致する。

k ステップ TD 法の嬉しいポイントとして、未来の報酬まで考慮するので局所解にハマりにくいというメリットがある。例えば、迷路ゲームにおいて 1 歩進むごとに体力を消耗するので -0.1 点、あるチェックポイントを通過すると $+3$ 点、ゴールに到着すると $+20$ 点もらえるような形式だとしよう。このとき、1 ステップ TD 学習では 1 歩ごとの行動価値を勘案するので、チェックポイントの周りをウロウロしたりするのである。これが「局所解にハマる」というもので、大域解はゴールに到達することであるのに対して局所解にハマって大域解に到達できないのである。

これが k ステップ TD 法では、k ステップ先の報酬まで先読みした上で行動評価を行うので、局所解に比較的ハマりやすいタスクであっても大域解に到達しうるのである。

しかしこの k ステップ TD 法はデメリットも存在し、それは毎回 k ステップの先読みを行うので学習効率が悪いのである。この辺に関しては「大域解に到達しやすい」というメリットを得た代わりに「学習効率を犠牲にする」というデメリットがあり、ある種のトレードオフが働いていると言えるだろう。

実装に際して、この報酬の先読み部分での行動は既存の方策に従って行動するものとする。

3.7.3 TD(λ) 法

k ステップ TD 法はマルコフ性を棄却してモンテカルロ法のアイデアを利用することで 1 ステップ TD 法を拡張し、そうして更新式を構築した。TD(λ) 法は k ステップ TD 法をさらに拡張する。

ここで時刻 t における k ステップ収益 $R_t^{(k)}$ を以下のように定義する。

$$R_t^{(k)} = \sum_{n=t+k}^{\infty} \gamma^{n-(t+k)} r_n \tag{37}$$

このとき状態行動価値 $Q(s,a)$ の定義は時刻 t における k ステップ収益 $R_t^{(k)}$ を用いて

$$Q^{\pi}(s,a) = \mathbb{E}\left[R_t^{(0)} \ ; \ s_t = s, a_t = a\right] \tag{38}$$

と表せる。

TD(λ) 法では以下で定義される収益平均 R_t^{avg} を考える。

$$R_t^{avg} = \sum_{n=0}^{\infty} w_n R_t^{(n)} \tag{39}$$

ここで重み w_i は以下の拘束条件に従う。

$$\sum_{i=0}^{\infty} w_i = 1 \ \ ; \ \ w_i > 0 \ (\forall i \in \mathbb{N}) \tag{40}$$

TD(λ) 法はこの k ステップ収益を平均化する方法で、各収益を λ^{n-1} ; $(0 \leq \lambda \leq 1)$ に比例して重み付けをする。こうすることで各ステップ収益の比が以下のようになる。

$$R_t^{(0)} : R_t^{(1)} : R_t^{(2)} : \cdots = 1 : \lambda^1 : \lambda^2 : \cdots \tag{41}$$

このようにすることで近いステップの収益に比重を置きつつ遠くのステップの収益も参考にするようにできる。また

$$1 + \lambda + \lambda^2 + \cdots = \frac{1}{1-\lambda} \tag{42}$$

となるので、重みの総和が 1 となるように、各重みに $(1-\lambda)$ をかけたものが実際の重みとなる。つまり収益平均 R_t^{avg} の重み w_i について

$$w_i = (1-\lambda)\lambda^i \tag{43}$$

が成立していると見ることができる。この重みを採用している場合の平均収益 R_t^{avg} を λ 収益 R_t^{λ} と呼ぶ。

これより、λ 収益 R_t^{λ} は以下の通りに定義される。

$$R_t^{\lambda} = (1-\lambda)\sum_{n=0}^{\infty} \lambda^n R_t^{(n)} \tag{44}$$

しかし実際のところ無限時間ではなく終端ステップ T までの平均収益を考慮するので

$$R_t^{\lambda} = (1-\lambda)R_t^{(0)} + (1-\lambda)\sum_{n=1}^{T-1} \lambda^n R_t^{(n)} + \lambda^T R_t^{(T)} \tag{45}$$

ここで T ステップまでの重みの総和は

$$(1-\lambda)(1 + \lambda + \lambda^2 + \cdots + \lambda^{T-1}) = 1 - \lambda^T \tag{46}$$

となり、重みの総和は 1 とする規約が存在するので、$R_t^{(T)}$ の係数は λ^T となる。

ここで $\lambda = 0$ とすると $R_t^{\lambda} = R_t^{(0)}$ となって 1 ステップ TD 法と一致する。また、$\lambda = 1$ とすると $R_t^{\lambda} = R_t^{(T)}$ となってモンテカルロ法と一致する。

ここで TD(λ) 法を可視化すると以下の通りである。

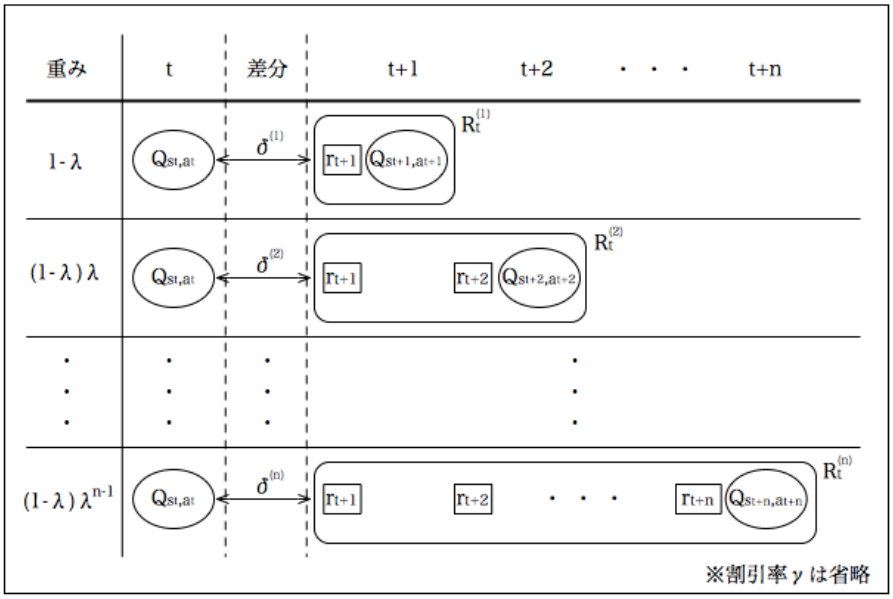

図 8: TD(λ) 法のイメージ

k ステップ TD 法も TD(λ) 法も鍵となるパラメータを操作すれば、一方に 1 ステップ TD 法があって他方にモンテカルロ法があるので混同しうる。

違いとして、k ステップ TD 法は各 k の値ごとにアルゴリズムが存在し、実際に使うのは k を固定したものを扱う。一方で TD(λ) 法は各 k の値ごとの k ステップ TD 法を同時に行っているイメージで、それぞれの結果をパラメータ λ によって重み付けをしていることになる。

基本的なアーキテクチャとしてはこれで問題ないのだが、ただ、このままだと「k ステップ先にならないと現在の状態価値を更新できない」という問題が存在する。よって更新されるまでしばらく待たなければならない。

今までは「時間ステップ t から時間ステップ $t+k$ での推定価値との差分を時間ステップ t での価値の推定に使おうとしている」という見地にあった。

ここで視点を逆転させ、「時間ステップ $t+k$ から見てその推定価値との差分が時間ステップ t の時点の価値の推定にどれくらい影響を与えているのか」という見方をする。

つまり、上図では今まで横で見ていたのが、視点を変えて縦で見てみるということである。上図の各ブロックを変形する。

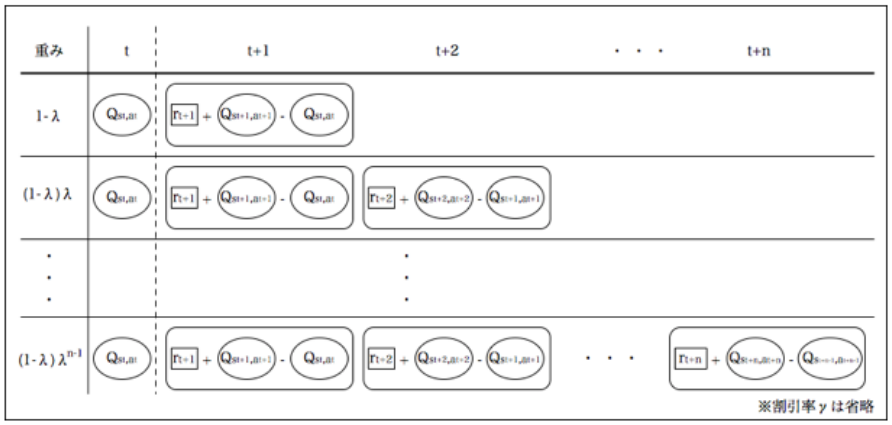

図 9: TD(λ) 法のイメージ

これを縦として見るので図は以下の通りになる。

つまり、時刻 $t+1$ の箇所については、係数和が

$$(1-\lambda)(1+\lambda+\lambda^2+\cdots+\lambda^{n-1}) = 1-\lambda^n \approx 1 \tag{47}$$

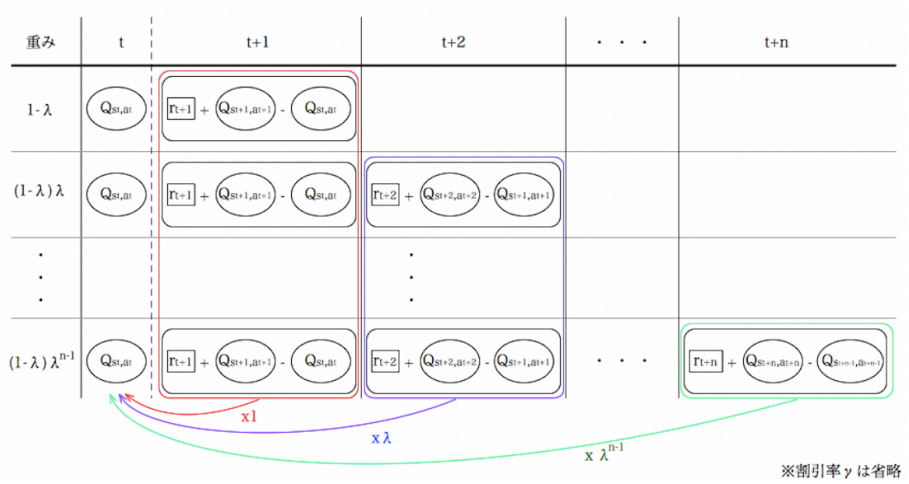

図 10: TD(λ) 法のイメージ

より 1 倍の影響を与えている。

また、時刻 $t+2$ の箇所については、係数和が

$$(1-\lambda)(\lambda+\lambda^2+\lambda^3+\cdots+\lambda^{n-1}) = \lambda(1-\lambda^n) \approx \lambda \tag{48}$$

より λ 倍の影響を与えている。

以上より、数学的帰納法から時刻 $t+i$ の要素は現在時刻 t に対して λ^{i-1} の重みで影響を与えていると言える。

そこで次のような時刻 t での適格度トレース $e^{(t)}(s,a)$ を考える。

$$e^{(t)}(s,a) = \begin{cases} 0 & (t=0) \\ \gamma\lambda e^{(t-1)}(s,a)+1 & (s=s_t, a=a_t) \\ \gamma\lambda e^{(t-1)}(s,a) & (otherwise) \end{cases} \tag{49}$$

この適格度トレースは、時刻 t で次の時刻での状態行動タプル (s_{t+1}, a_{t+1}) を観測したとき、その価値の差分をどれくらいの重みで各状態行動タプル

(s, a) の価値に反映させるのか、という意味になる。

実際に、最初は 0 なので状態行動タプル (s, a) が観測されるまでは価値 $Q(s, a)$ に各時間ステップでの価値の差分は反映させていない。また、時刻 t で状態行動タプル (s, a) が観測されたら状態行動価値 $Q(s, a)$ に時刻 $t + 1$ での価値の差分を追加して反映させるようにしている。そして時間ステップが離れていくごとに、観測された状態行動タプル (s, a) の状態行動価値 $Q(s, a)$ に反映させる各時間ステップでの価値の差分は $\gamma\lambda, \gamma^2\lambda^2, \ldots$ と減衰していく。

これによって 1 つ前の状態行動タプルの価値には、価値の差分の 1 倍、2 つ前の状態行動タプルの価値には差分の $\gamma\lambda$ といったように価値の差分の反映を行うことができるようになる。

具体的な価値関数の更新として、時刻 t の TD 誤差を $\delta^{(t)}(s_t, a_t)$ を

$$\delta^{(t)}(s_t, a_t) = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{50}$$

としたとき、$\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$ について

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta^{(t)}(s, a)e^{(t)}(s, a) \tag{51}$$

となる。

4　動的計画法

本章では第 3 章にて導出したベルマン方程式を解く手法について解説する。一般に、動的計画法はモデルがわかっている場合 (状態遷移確率がわかっている場合) に有効な手法である。このようなモデルがわかっている場合の手法を「モデルベース (Model-based)」という。

強化学習のエージェントの目標は「報酬の最大化」で、その手段として「方策の最適化」がある。その「方策の最適化」として、方策を何によってパラメタライズするかという問題が動的計画法において浮かび上がってくるのだ。

一般に、ベルマン方程式を解く手法として「反復法」と「勾配法」がある。

本章ではこの反復法と勾配法をそれぞれ擬似コードを交えながら解説していく。

4.1　反復法

ベルマン方程式は線型方程式なので反復法による数値解法が可能であり、それをアルゴリズムとして実装したのがこの反復法である。

反復法は方策の良さを状態価値 $V(s)$ や状態行動価値 $Q(s,a)$ を用いて評価する。いわば方策を価値関数を用いてパラメタライズしているのである。そうして価値関数自体を更新していき、方策をだんだんと最適化していくのである。

反復法はヤコビ法に基づく解法で、解の収束性については数学的に証明されているが、本稿では紙面の都合から証明は割愛する (要するにめんどくさいのである)。興味のある読者は数値解析系の参考書を参照してもらいたい。

反復法では「価値反復法 (Value Iteration Method)」と「方策反復法 (Policy Iteration Method)」の 2 種類があり、以下でそれぞれについて解説していく。

4.1.1 価値反復法

価値反復法では、方策として ϵ-greedy のような最適方策を用意する。

定義：ϵ-greedy 法

ϵ を小さな値とする。状態 s_t において行動 a_t を選択する確率は以下の通りに定義される。

$$\pi(a_t|s_t) = \begin{cases} 1-\epsilon & (a_t = \arg\max_{a\in\mathcal{A}} Q(s_t, a)) \\ \epsilon & (otherwise) \end{cases} \tag{52}$$

ここで ϵ は学習回数に応じて減衰させていくことで学習の収束が保証される。

k 回目の学習での状態 s における状態価値を $V^{(k)}(s)$ として表せば、

$$V^{(k+1)}(s) \leftarrow \sum_{s'\in\mathcal{S}} \sum_{a\in\mathcal{A}} \pi(a|s) P(s'|s,a) \left[r_t + \gamma V^{(k)}(s') \right] \tag{53}$$

として状態価値 $V^{(k)}(s)$ を更新していく。

これは状態行動価値についても同様で、

$$Q^{(k+1)}(s,a) \leftarrow \sum_{s'\in\mathcal{S}} P(s'|s,a) \left[r_t + \gamma \sum_{a'\in\mathcal{A}} \pi(a'|s') Q^{(k+1)}(s',a') \right] \tag{54}$$

このように非常に単純で明快なアルゴリズムである。

これを擬似コードとして表すと以下の通りである。

```
Q <- Initialize

## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 終了判定初期化 ##
    done <- False

    ## 状態の初期化 ##
    state_old <- Initialize

    ## ゲームが終わるまで ##
    while not done:

        ## 行動選択 ##
        action_old <- pi(state_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 終了判定 ##
        done <- check(state_new)

        ## 新しい行動選択 ##
        action_new <- pi(state_new)

        ## 価値更新 ##
        update <- reward + gamma * Q(state_new,action_new)
        Q(state_old,action_old) <- update
```

4.1.2 方策反復法

価値反復法では方策を最初に ϵ-greedy としていたが、方策反復法では最初に方策を任意に初期化する。

まず最初にその方策に基づいて行動を行って状態行動価値 $Q(s,a)$ を求め、得られた状態行動価値を用いて greedy 方策を構築する。

具体的に、k 回目の学習での状態 s と行動 a における状態行動価値を $Q^{(k)}(s,a)$ とし、方策を $\pi^{(k)}(a|s)$ として表せば、

$$Q^{(k+1)}(s,a) \leftarrow \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[r_t + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s'_t) Q^{(k+1)}(s',a') \right] \tag{55}$$

として最初に状態行動価値を求める。

その後、方策を

$$\pi^{(k+1)}(a|s) = \begin{cases} 1 & (a = \arg\max_{a' \in \mathcal{A}} Q(s,a')) \\ 0 & (otherwise) \end{cases} \tag{56}$$

として更新する。

価値反復法との違いとして、価値反復法は方策を ϵ-greedy で固定してしまって価値の更新だけを行っていたが、方策反復法では価値を更新したあとに方策も更新する。

方策勾配法では最初に全パターンの状態 s と行動 a について状態行動価値を求めるので非常にコストが高いが、確実に全パターンの状態行動タプルを調べるので収束の確実性が保証されている。

これを擬似コードとして表すと以下の通りである。

```
Q <- Initialize
pi <- Initialize

## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 終了判定初期化 ##
    done <- False

    ## 状態の初期化 ##
    state_old <- Initialize

    ## 全パターンの状態行動タプル ##
    for (state_old, action_old : all pattern):

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 新しい行動選択 ##
        action_new <- pi(state_new)

        ## 更新量を計算 ##
        update <- reward + gamma * Q(state_new,action_new)

        ## 価値更新 ##
        Q(state,action) <- update


    ## 全パターンの状態行動タプル ##
    for (state_old, action_old : all pattern):

        ## 方策更新 ##
        pi <- greedy(Q)
```

4.2 勾配法

反復法では状態価値や状態行動価値を求めることによって最適方策を獲得していた。

一方で、勾配法は方策を価値ではない別のパラメータを用いてパラメタライズし、その勾配を考えることで方策の最適化を図る。

勾配法のことを特に「方策勾配法」と呼ぶ。

具体的なアルゴリズムとして、方策は価値ではない別のパラメータ θ によって表現され、方策の良さを表す評価関数 $J(\theta)$ で方策を評価していく。そして方策を改善するために評価関数の勾配 $\nabla_\theta J(\theta)$ によって θ を更新する。

これを数式と表すと以下の通りである。

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{57}$$

θ でパラメタライズされた方策としては softmax 法やガウス分布などが考えられる。

softmax 法

行動空間が離散であるとする。温度パラメータと呼ばれるパラメータ $\tau \in \mathbb{R}_{>0}$ を用いて、状態 s_t において行動 a_t をとる確率は以下の通りに定義される。

$$\pi(a_t|s_t) = \frac{\exp\left[\theta(s_t, a_t)/\tau\right]}{\sum_{b \in \mathcal{A}} \exp\left[\theta(s_t, b)/\tau\right]} \tag{58}$$

ここで τ は学習率をコントロールするパラメータで、$\tau \to 0$ で ϵ-greedy 法と一致する。

ガウス分布

行動空間が連続であるとする。温度パラメータ $Z \in \mathbb{R}_{>0}$ と分散共分散行列 C を用いて、状態 s_t において行動 a_t をとる確率は以下の通りに定義される。

$$\pi(a_t|s_t) = \frac{1}{Z}\left[-\frac{1}{2}(a_t - Ws_t)^T C^{-1}(a_t - Ws_t)\right] \tag{59}$$

方策勾配法は方策を状態行動価値を用いず θ でパラメタライズしているので行動空間が連続でも学習をすることが可能なのが特徴である。

以上から方策の評価関数 $J(\theta)$ を定義することで評価関数の勾配 $\nabla J(\theta)$ を数式を用いて表現でき、方策勾配法を簡単に導出・実装できそうであるが、そう簡単にはうまくいかない。というのも、方策を更新する前に「方策の良さ」を定義しておく必要があるためである。価値反復法や方策反復法では方策の良さが価値関数と紐づいていたために方策の良さは自明であったが、方策勾配法では方策を別のパラメータで表現しているので改めて「方策の良さ」を定義する必要があるのだ。

1 つ注意点として、価値反復法や方策反復法はあくまでベルマン方程式という線型方程式を反復解法によって解く手法で、状態価値や状態行動価値から出発して学習アルゴリズムを確立する。しかし、方策勾配法の出発点は方策をパラメータ θ を用いて表現して方策評価関数 $J(\theta)$ を定義することなので、状態価値や状態行動価値は評価関数 $J(\theta)$ に応じて自由に設計できる点に注意したい。

さて、評価関数 $J(\theta)$ を定義するのだが、定義の仕方は様々な方法がある。今回では $J(\theta)$ を「平均報酬」で定義する方法と「割引報酬和」で定義する方法の 2 通りについて検討する。

方策勾配法のゴールは評価関数の勾配 $\nabla J(\theta)$ を数式的に表現することであったが、先に答えを言ってしまうと評価関数を平均報酬で定義する場合と割引報酬和で定義する場合とで勾配 $J(\theta)$ は同じ形になるのである。勾配 $\nabla J(\theta)$ は状態行動価値 $Q(s_t, a_t)$ と方策 $\pi_\theta(a_t|s_t)$ を用いて以下の通りに表せる。

$$\nabla J(\theta) = \mathbb{E}\left[Q(s_t, a_t)\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] \tag{60}$$

以下では平均報酬と割引報酬和のそれぞれの場合について、この勾配を導出していく。

4.2.1 平均報酬による方策評価

評価関数を平均報酬で定義する。評価関数 $J(\theta)$ は以下の通りに定義される。

$$J(\theta) = \lim_{n\to\infty}\frac{1}{n}\mathbb{E}\left[\sum_{t=1}^{n} r_t \ ; \ \theta\right] \tag{61}$$

このとき状態行動価値関数 $Q(s, a)$ を以下のように定義する。

$$Q(s, a) = \sum_{k=0}^{\infty}\mathbb{E}\left[r_{t+k} - J(\theta) \ ; \ s_t = s, a_t = a, \theta\right] \tag{62}$$

状態行動価値関数を定義することができたので状態価値関数 $V(s)$ は以下の通りに定義できる。

$$V(s) = \sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q(s, a) \tag{63}$$

この状態価値関数を θ で微分する。

$$\frac{\partial V(s)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q(s,a) \tag{64}$$

$$= \sum_{a \in \mathcal{A}} \left[\pi_\theta(a|s) \frac{\partial Q(s,a)}{\partial \theta} + Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \right] \tag{65}$$

ここで状態行動価値関数 $Q(s,a)$ の定義式から以下のように式変形できる。

$$\begin{aligned} Q(s,a) &= \sum_{k=0}^{\infty} \mathbb{E}\left[r_{t+k} - j(\theta) \ ; \ s_t = s, a_t = a, \theta \right] \\ &= \mathbb{E}\left[r_t - J(\theta) \ ; \ s_t = s, a_t = a, \theta \right] \\ &\quad + \sum_{k=1}^{\infty} \mathbb{E}\left[r_{t+k} - J(\theta) \ ; \ s_t = s, a_t = a, \theta \right] \\ &= \mathbb{E}\left[r_t ; \theta \right] - J(\theta) + \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi_\theta(a'|s') Q(s',a') \end{aligned} \tag{66}$$

ここで行動 a_t は given なので報酬 r_t は θ に依存しない確定的な値より、$Q(s,a)$ を θ で微分すると報酬の期待値は消える。

$$\frac{\partial Q(s,a)}{\partial \theta} = \frac{\partial}{\partial \theta} \left[-J(\theta) + \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi_\theta(a'|s') Q(s',a') \right] \tag{67}$$

$$= -\frac{\partial J(\theta)}{\partial \theta} + \sum_{'\in \mathcal{S}} P(s'|s,a) \frac{\partial V(s)}{\partial \theta} \tag{68}$$

よって $J(\theta)$ の微分を含んだ状態行動価値関数の微分が得られたので、これを状態価値関数の微分の式に代入する。

$$\frac{\partial V(s)}{\partial \theta} = \sum_{a \in \mathcal{A}} \left[\pi_\theta(a|s) \left\{ -\frac{\partial J(\theta)}{\partial \theta} + \sum_{s' \in \mathcal{S}} P(s'|s,a) \frac{\partial V(s')}{\partial \theta} \right\} + Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \right] \tag{69}$$

これを $J(\theta)$ の微分についての式に変形する。

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{a \in \mathcal{A}} \left[Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} + \pi_\theta(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \frac{\partial V(s')}{\partial \theta} \right] - \frac{\partial V(s)}{\partial \theta} \tag{70}$$

ここで以下に示されるような、無限時間経ったときに状態が s である確率

を表す状態関数 $d(s)$ を考える。

$$d(s) = \lim_{t \to \infty} P(s_t = s | s_0, \theta) \tag{71}$$

この状態関数 $d(s)$ は定常分布であることに注意したい。

評価関数 $J(\theta)$ の微分の式の両辺に $d(s)$ を掛け、状態 s について $\sum$ をとる。

$$\begin{aligned} \sum_{s \in \mathcal{S}} d(s) \frac{\partial J(\theta)}{\partial \theta} = & \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \left[Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \right] \\ & + \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \left[\pi_\theta(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \frac{\partial V(s')}{\partial \theta} \right] \\ & - \sum_{s \in \mathcal{S}} d(s) \frac{\partial V(s)}{\partial \theta} \end{aligned} \tag{72}$$

ここで状態関数 $d(s)$ は定常分布なので、任意の関数 $F(s)$ について以下が成立。

$$\sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \sum_{s \in \mathcal{S}} d(s) \pi_\theta(a|s) P(s'|s,a) F(s') = \sum_{s' \in \mathcal{S}} d(s') F(s') \tag{73}$$

この事実から先ほどの $J(\theta)$ の微分の式の右辺第 2 項と第 3 項が相殺される。よって以下の式を得る。

$$\sum_{s \in \mathcal{S}} d(s) \frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \left[Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \right] \tag{74}$$

また、状態 s と行動 a の確率と期待値について、関数 $G(s,a)$ で以下の関係が成立する。

$$\mathbb{E}\left[G(s,a) \right] = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} d(s) \pi_\theta(a|s) G(s,a) \tag{75}$$

よって目的の式が得られる。

$$\nabla J(\theta) = \mathbb{E}\left[Q(s,a) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \right] \tag{76}$$

4.2.2 割引報酬和による方策評価

この場合、評価関数 $J(\theta)$ を割引報酬和で定義する。評価関数 $J(\theta)$ は以下の通りに定義される。

$$J(\theta) = \lim_{n\to\infty} \mathbb{E}\left[\sum_{t=1}^{n} \gamma^{t-1} r_t \ ; \ s_0, \theta\right] \tag{77}$$

このとき、状態行動価値関数 $Q(s,a)$ を以下のように定義する。

$$Q(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \ ; \ s_t = s, a_t = a, \theta\right] \tag{78}$$

平均報酬のときと同様に、状態価値関数は以下の通りに定義できる。

$$V(s) = \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q(s,a) \tag{79}$$

この状態価値関数を θ で微分する。

$$\begin{aligned} \frac{\partial V(s)}{\partial \theta} &= \frac{\partial}{\partial \theta} \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q(s,a) && (80) \\ &= \sum_{a\in\mathcal{A}} \left[\pi_\theta(a|s) \frac{\partial Q(s,a)}{\partial \theta} + Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta}\right] && (81) \end{aligned}$$

状態行動価値関数 $Q(s,a)$ について、式変形すれば以下が得られる。

$$Q(s,a) = \mathbb{E}\left[r_t; \theta\right] + \sum_{s'\in\mathcal{S}} \gamma P(s'|s,a) V(s') \tag{82}$$

ここで行動 a_t は given なので報酬 r_t は θ に依存しない確定的な値より、$Q(s,a)$ を θ で微分すると報酬の期待値は消える。

$$\frac{\partial Q(s,a)}{\partial \theta} = \sum_{s'\in\mathcal{S}} \gamma P(s'|s,a) \frac{\partial V(s')}{\partial \theta} \tag{83}$$

よって得られた状態行動価値関数 $Q(s,a)$ の微分を状態価値関数 $V(s)$ の微分の式に代入。

$$\frac{\partial V(s)}{\partial \theta} = \sum_{a\in\mathcal{A}} \left[Q(s,a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} + \pi_\theta(a|s) \sum_{s'\in\mathcal{S}} \gamma P(s'|s,a) \frac{\partial V(s')}{\partial \theta}\right] \tag{84}$$

すると状態価値関数 $V(s)$ の微分を状態価値関数 $V(s)$ の微分によってさいき的に表現できる。$V(s)$ の微分の式を右辺の微分の部分に再帰的に代入する、ということを無限回繰り返す。k ステップで状態 s から状態 x に遷移する確率を $P(s \to x, k)$ とすると以下の式が得られる。

$$\frac{\partial V(s)}{\partial \theta} = \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \gamma^k P(s \to x, k) \sum_{a \in \mathcal{A}} Q(s, a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \tag{85}$$

ここで評価関数 $J(\theta)$ は状態価値 $V(s_0)$ と同値なので

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \ ; \ s_0, \theta \right] & (86) \\ &= \frac{\partial V(s_0)}{\partial \theta} & (87) \\ &= \sum_{s \in \mathcal{S}} \sum_{k=0}^{\infty} \gamma^k P(s_0 \to s, k) \sum_{a \in \mathcal{A}} Q(s, a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} & (88) \end{aligned}$$

ここで状態に関する関数 $d(s)$ を以下の通りに定義する。

$$d(s) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s | s_0, \theta) \tag{89}$$

時刻 t での状態 s_t が s である確率 $P(s_t = s|s_0, \theta)$ と、状態 s_0 から k ステップで状態が s になる確率 $P(s_0 \to s, k)$ は同じなので、これを評価関数 $J(\theta)$ の微分の式に代入することで以下の式を得る。

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \left[Q(s, a) \frac{\partial \pi_\theta(a|s)}{\partial \theta} \right] \tag{90}$$

また、状態 s と行動 a の確率と期待値について、関数 $G(s, a)$ で以下の関係が成立する。

$$\mathbb{E}\left[G(s, a)\right] = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} d(s) \pi_\theta(a|s) G(s, a) \tag{91}$$

よって目的の式が得られる。

$$\nabla J(\theta) = \mathbb{E}\left[Q(s, a) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} \right] \tag{92}$$

4.2.3 方策勾配法の実装

以上、長い苦労を経てようやく評価関数の勾配 $\nabla J(\theta)$ を数式で表すことができた。勾配は以下の式で表される。

$$\nabla J(\theta) = \mathbb{E}\left[Q(s,a)\frac{\partial \log \pi_\theta(a|s)}{\partial \theta}\right] \tag{93}$$

しかし、この勾配の具体的な値を求めるにあたって状態行動価値 $Q(s,a)$ は未知であるので具体的な値を算出することができない。これを解決する方法として 2 つのアプローチを紹介する。

REINFORCE アルゴリズム

m 回目の試行における状態行動価値 $Q^{(m)}(s_t,a_t)$ を報酬 $R_t^{(m)}$ で近似し、モンテカルロ法で複数サンプリングをしてランダム推定を行う手法。T ステップの試行を M 回行って勾配を推定する。

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}\left[Q(s,a)\frac{\partial \log \pi_\theta(a|s)}{\partial \theta}\right] &(94)\\ &\approx \frac{1}{M}\sum_{m=1}^{M}\frac{1}{T}\sum_{t=1}^{T}(R_t^{(m)}-b)\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} &(95)\end{aligned}$$

ここで b は推定の分散を小さくするためのベースラインと呼ばれる定数で、ベースラインとして平均報酬がよく使われる。

このベースラインについて、定数 p を用いて

$$\begin{aligned}
\mathbb{E}\left[p\frac{\partial \log \pi_\theta(a|s)}{\partial \theta}\right] &= p\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} d(s)\pi_\theta(a|s)\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} && (96)\\
&= p\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} d(s)\pi_\theta(a|s)\frac{1}{\pi_\theta(a|s)}\frac{\partial \pi_\theta(a|s)}{\partial \theta} && (97)\\
&= p\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} d(s)\frac{\partial \pi_\theta(a|s)}{\partial \theta} && (98)\\
&= p\frac{\partial}{\partial \theta}\sum_{s\in\mathcal{S}} d(s)\sum_{a\in\mathcal{A}}\pi_\theta(a|s) && (99)\\
&= p\frac{\partial}{\partial \theta}\sum_{s\in\mathcal{S}} d(s) && (100)\\
&= p\frac{\partial}{\partial \theta}1 && (101)\\
&= 0 && (102)
\end{aligned}$$

という結果が得られるので、ベースラインを導入しても問題ないということがわかる。

自然勾配法

ベースラインとして状態価値 $V(s)$ を採用する。自然勾配法は、状態行動価値 $Q(s,a)$ の代わりに状態 s における行動 a の相対的な良さを表すアドバンテージ関数 $A(s,a) = Q(s,a) - V(s)$ を用いる。これを重み w による $w^T \log \pi_\theta(a|s)$ で近似し、重み w を TD 学習などで求める手法である。

これまでの勾配はパラメータのユークリッド距離を用いてきたが、確率分布の間の距離 (Kullback-Leibler) はこれと一致しない。そういう曲がった空間で計量テンソルという行列を用いて距離を扱うが、この逆行列に勾配をかけたものを自然勾配という。

数学的な証明は長くなるため割愛するが、証明の方針として、$Q(s,a)$ をアドバンテージ関数 $A(s,a) = Q(s,a) - V(s)$ とし、それを $w^T \log \pi_\theta(a|s)$ で代用すると、勾配はフィッシャー情報行列 $F(\theta)$ を用いて $\nabla J(\theta) = F(\theta)w$ となる。フィッシャー情報行列は KL-divergence を 2 次近似して計量テンソルを作ることで求められる。

重み w の求め方について、

$$\sum_{t=0}^{N-1} \gamma^t A(s_t, a_t) = V(s_0) + \sum_{t=0}^{N-1} \gamma^t R_t - \gamma^N V(s_N) \tag{103}$$

ここで N がエピソード終了なら第 3 項は 0 になるので

$$\sum_{t=0}^{N-1} \gamma^t \nabla_\theta \left\{ \log \pi_\theta(a|s)^T w \right\} = V(s_0) + \sum_{t=0}^{N-1} \gamma^t R_t \tag{104}$$

よって線形回帰によって重み w は求めることができる。

自然勾配を用いて方策を更新した方が学習が効率的であるという研究もある。[4]

[4] Amari, 1998; "Natural Gradient Works Efficiency in Learning"

5　TD 学習の基本手法

第 3 章ではベルマン方程式を導出し、そこから TD 学習の基本的な概念を紹介、k ステップ TD 法や TD(λ) 法を導出した。

第 3 章で軽く触れたことであるが、例えば状態行動価値について学習を進めるとき

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{105}$$

というように更新式を展開すると説明した。一見これで問題なさそうであるが、この更新部分の $r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ で行動 a_{t+1} をどのように選択するか次第でアルゴリズムとしての挙動が大きく変わってくるのである。

このように行動 a_{t+1} の選択方法についての議論や、または状態行動価値ではなく状態価値を更新することにスポットを当てた手法もある。

第 3 章ではざっと TD 法についての紹介にとどめたが、本章では実際にどのように学習を進めていくかという点にスポットを当て、擬似コードを交えつつ紹介していこうと思う。

本章ではあくまで 1 ステップ TD 法によるアルゴリズムの紹介をするが、もちろん以下で紹介するアルゴリズムを k ステップ TD 法や TD(λ) 法へと拡張した形で実装することも可能である。興味のある読者は試してみてほしい。

5.1 SARSA

SARSA は状態行動価値 $Q(s,a)$ についての学習アルゴリズムで、方策としては softmax 法を採用する (もちろん ϵ-greedy 法のような他のソフト方策でも可能である)。Q-table に基づいた知識利用と探査のバランスをとることで最適方策を獲得していく手法である。

ある状態 s_t において方策 $\pi(a_t|s)_t$ に基づいて行動 a_t を選択し、行動する。それによって報酬 r_t を得て、状態 s_{t+1} となる。そこで現在持っている方策 $\pi(a_{t+1}|s_{t+1})$ に基づいて行動 a_{t+1} を選択する。

このように $s_t, a_t, r_t, st+1, a_{t+1}$ によって状態行動価値 $Q(s_t,a_t)$ を更新する。

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \tag{106}$$

SARSA は学習に使用する各文字 $s_t, a_t, r_t, st+1, a_{t+1}$ の頭文字をとって SARSA という名前となっている。

モンテカルロ法の頁にて方策オン型モンテカルロ法は「推定方策」と「挙動方策」が同一のものであるとしたが、SARSA も同様に「推定方策」と「挙動方策」が一致したものである、という見方も可能である。

価値反復法に基づいたアルゴリズムで、非常にシンプルながらもいい成績を出すことが知られている。

擬似コードは以下の通りである。

```
##### SARSA ######

Q <- Initialize

## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 状態と行動を初期化 ##
    state_new, state_old <- Initialize
    action_new, action_old <- Initialize

    ## 終了判定を初期化 ##
    done <- False

    ## 試行が終わるまで ##
    while not done:

        ## 行動選択 ##
        action_old <- pi(state_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 新しい行動選択 ##
        action_new <- pi(state_new)

        ## 更新 ##
        value <- reward + gamma * Q(state_new,action_new)
        update <- alpha * (value - Q(state_old,action_old))
        Q(state_old,action_old) <- Q(state_old,action_old) \
                                        + update
```

5.2 Q 学習

Q 学習は SARSA と同様に状態行動価値 $Q(s,a)$ に基づく学習アルゴリズムであるが、SARSA とは異なり、「推定方策」と「挙動方策」を分離する。Q 学習では推定方策には確率的なものを採用し、挙動方策として greedy なものを採用する。

これを具体的に数式として表すと以下の通りである。

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1},a) - Q(s_t,a_t) \right] \qquad (107)$$

SARSA と同様にシンプルなアルゴリズムとなっている。Q 学習 (Q-Learning) で、なぜ Q を用いているかは諸説あるが、Quality の Q をとって Q 学習とした説が有力である。

SARSA では推定方策と挙動方策が一致しているので現在の方策 (softmax 法) によって選ばれた次の行動の状態行動価値を参照するが、Q 学習では推定方策と挙動方策が別で、推定方策 (greedy 法) に従った現時点で最適と思われる次の行動の価値を参照する。

SARSA と Q 学習のどちらがいいかということは特になく、SARSA では学習中の平均報酬は Q 学習よりも良いが、学習スピードが遅い。一方で Q 学習では学習スピードが早いが SARSA よりも劣る。SARSA だと推定方策と挙動方策が同じなので推定方策の改善が挙動方策の改善にもつながり、その挙動による報酬も向上するのに対し、Q 学習では推定方策と挙動方策が異なるので推定方策が改善されても挙動方策に悪い挙動が紛れたままになる可能性があるためである。しかし、Q 学習は推定方策と挙動方策が別になってい

るので理論的に解析しやすいというメリットがある。よって強化学習の研究ではよく Q 学習が用いられている。

擬似コードは以下の通りである。

```
##### Q-Learning ######

Q <- Initialize

## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 状態と行動を初期化 ##
    state_new, state_old <- Initialize
    action_new, action_old <- Initialize

    ## 終了判定を初期化 ##
    done <- False

    ## 試行が終わるまで ##
    while not done:

        ## 行動選択 ##
        action_old <- pi(state_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 新しい行動選択 ##
        action_new <- argmax(Q(state_new))

        ## 更新 ##
        value <- reward + gamma * Q(state_new,action_new)
        update <- alpha * (value - Q(state_old,action_old))
        Q(state_old,action_old) <- Q(state_old,action_old) \
                                        + update
```

5.3 Actor-Critic

Actor-Critic は方策勾配法に基づいたアルゴリズムで、価値関数とは独立に方策を表現する構造を持っている。Actor-Critic は方策勾配法と同様に行動選択のための方策と価値関数を予測する部分で分離しており、方策は行動を規定するための actor(演者)、価値関数の予測部分を actor によって選ばれた行動を批判するための critic(批評家) の役目を果たしている。

Actor-Critic は方策を別のパラメータでパラメタライズするので、状態行動価値関数を相手にしない。代わりに状態価値を相手にしていく。

状態価値の更新式は以下の通りである。

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_t + \gamma V(s_{t+1}) - V(s_t) \right] \tag{108}$$

TD 誤差 $r_t + \gamma V(s_{t+1}) - V(s_t)$ は行った行動を評価するために用いられる。TD 誤差が正の値ならば行った行動は価値を高めたことになるので、より頻繁に選択されるようにする。逆に TD 誤差が負の値ならば選択されないようにする方が良い。

ここで、TD 学習はモデルが不明な場合のモデルフリー的手法で、複数回の試行錯誤によって行動による遷移過程は本来の状態遷移確率に統計的に従う。よってこの場合は状態価値 $V(s)$ を用いた学習式の更新でも状態行動価値 $Q(s, a)$ を更新する場合と同様の結果が期待できるのである。

$p(s, a)$ を actor が状態 s における行動 a の良さを表すパラメータだとすると、行動選択のための方策は softmax 法に基づく。

$$\pi(a_t | s_t) = \frac{\exp\left[p(s_t, a_t)/\tau\right]}{\sum_{b \in \mathcal{A}} \exp\left[p(s_t, b)/\tau\right]} \tag{109}$$

このパラメータ $p(s,a)$ を更新するには TD 学習を利用する。

$$p(s_t,a_t) \leftarrow p(s_t,a_t) + \alpha\left[r_t + \gamma V(s_{t+1}) - V(s_t)\right] \tag{110}$$

ここで更新式に現れる状態 s_{t+1} は、状態 s_t において行動 a_t を選択した際に観測される次の状態である。

Actor-Critic は状態行動価値を使わず方策を別のパラメータでパラメタライズするので、行動空間が連続な系でも問題ないのが Actor-Critic の特長と言える。

実際に、行動空間が連続な場合については方策を

$$\pi(a_t|s_t) \approx N(\mu(s_t),\sigma^2(s_t)) \tag{111}$$

のような正規分布の確率密度関数として扱えば対応可能である。

連続な空間での強化学習は収束の発散性について様々な議論があり、細かい説明については第 6 章の譲りたい。

擬似コードは以下の通りである。

```
##### Actor-Critic ######

V, p <- Initialize

## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 状態と行動を初期化 ##
    state_new, state_old <- Initialize
    action <- Initialize

    ## 終了判定を初期化 ##
    done <- False

    ## 試行が終わるまで ##
    while not done:

        ## 行動選択 ##
        action <- pi(state_old)

        ## 行動 ##
        state_new, reward <- move(action)

        ## 更新 ##
        TD_error <- reward + gamma * V(state_new) - V(
            state_old)
        update <- alpha * TD_error
        p(state_old,action) <- p(state_old,action) + update
```

6　連続な空間での強化学習

今まで強化学習の具体的な手法について、状態空間や行動空間が離散な系を前提として動的計画法やモンテカルロ法、TD 学習を紹介してきた。本章では連続な系についての解説を進めていく。

6.1　連続な系の問題

ここで「次元の呪い」について思い出して欲しい。「次元の呪い」とは、「状態空間が膨らむと指数関数的に計算量が増大する」というものであった。よって連続な系をまともに相手すると計算量が爆発し、現実的な時間では解が収束しない、という大きな問題があるのだ。

実装に際して「連続な系ではまずグリッドで離散な系に近似する」という離散状態表現近似の手法を使うと、粗い近似だと非マルコフ性が生じることから最適方策が得られず、細かい近似だと次元の呪いにぶつかる。

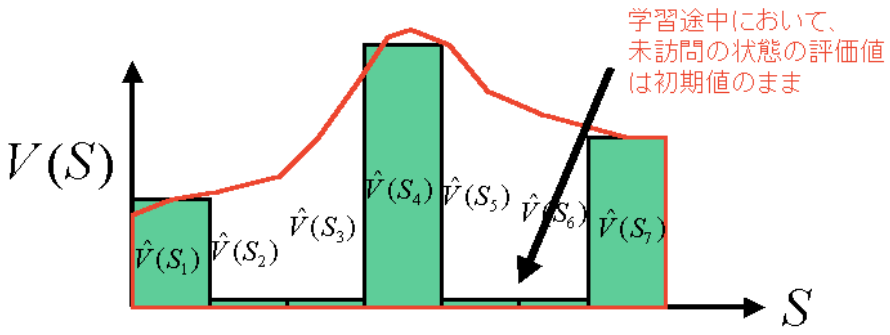

図 11: 離散状態表現近似による連続空間の表現

よってちょうどいいグリッドサイズでの近似をすればちょうど折衷案とし

てそれぞれのいいとこ取りができるのだが、良さそうな近似区間は環境によってバラバラであり、しかもそれらは非自明である。

こうした問題から離散状態表現近似 (=縦に割る感じの近似) ではない手法を模索するのが賢明であると察するだろう。

6.2 線形アーキテクチャ

前節で示した通り、離散状態表現近似はいくらかの問題を孕んでいることがわかった。

こうした問題の解決手法の一つが線形アーキテクチャによる近似である。

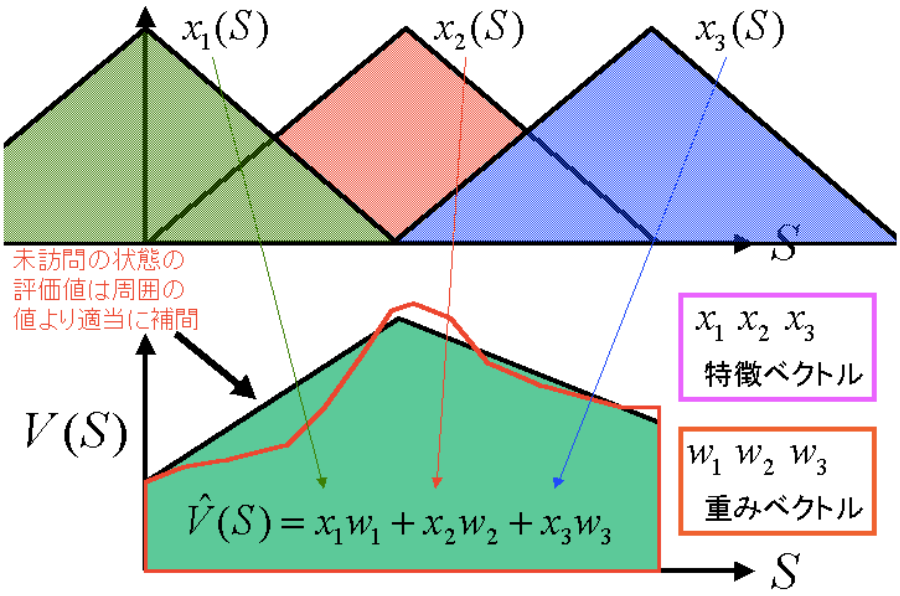

図 12: 線形アーキテクチャによる汎化と関数近似

先ほど示した離散状態表現近似ではグリッドによって未訪問となって情報が欠落してしまうが、線形アーキテクチャによる近似では情報が未知の部分

について適当に補完が効くので汎化性能がある。

6.3 RBF による線形近似

先ほどの単純な線形アーキテクチャによる近似では状態価値を特徴ベクトル x_i と重み w_i を用いて

$$\hat{V}(s) = \sum_{n=1}^{N} w_i x_i \tag{112}$$

として近似した。

ここで RBF(Radial Basis Function) による線形近似[5]について紹介する。

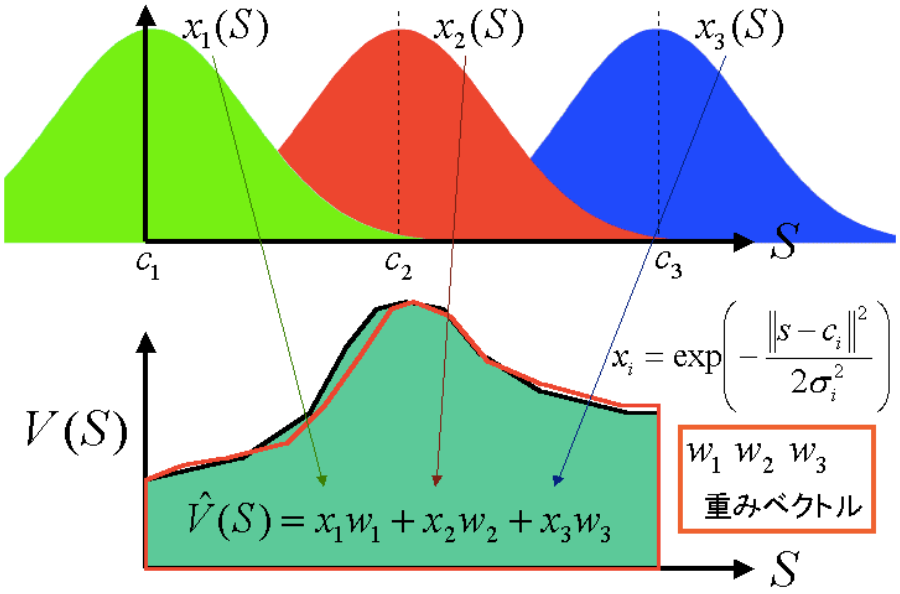

図 13: 線形アーキテクチャによる汎化と関数近似

[5]Santamaria, J. C., Sutton, R. S. and Ram, A.; Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces, Adaptive Behavior 6 (2), pp.163–218 (1998).

このアイデアは、入力信号の特徴ベクトルをガウス分布に変形させてから線形アーキテクチャを構成する手法で、状態価値を

$$\hat{V}(s) = \sum_{n=1}^{N} w_i \exp\left[-\frac{||s - c_i||^2}{2\sigma_i^2}\right] \tag{113}$$

として近似する。

6.4 線形アーキテクチャでの TD 学習

TD 学習は以下のように価値を更新するものであった。

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_t + \gamma V(s_{t+1}) - V(s_t)\right] \tag{114}$$

線形アーキテクチャでは同様に近似状態価値を以下のように更新する。

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha\left[r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)\right] \tag{115}$$

近似状態価値の更新に際して、重み w_i を以下のように近似する。

$$w_i \leftarrow w_i + \alpha\left[r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)\right] \underbrace{\frac{\partial \hat{V}(s_t)}{\partial w_i}}_{=\ x_i} \tag{116}$$

ここで $\alpha \to 0$ で、この線形アーキテクチャの更新法では真の状態価値 $V(s)$ に対して二乗誤差最小の近似へ収束することが示されている[6]。

今回は状態価値 $V(s)$ について更新式を示したが、状態行動価値 $Q(s,a)$ についても同様に

$$w_i \leftarrow w_i + \alpha\left[r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t)\right] \underbrace{\frac{\partial \hat{Q}(s_t, a_t)}{\partial w_i}}_{=\ x_i} \tag{117}$$

として重みの更新を図ることが出来る。

[6] Bertsekas, D.P. and Tsitsiklis, J.N.; Neuro-Dynamic Programming, Athena Scientific (1996)

6.5 特徴ベクトルへの要請

線形アーキテクチャの構成にあたって特徴ベクトルを入力信号とした。特徴ベクトルへの要請はいくつかあり、それらを述べていく。

まず最初に、特徴ベクトルは望ましい性質として「絶対ノルムが 1 である」が挙げられる。ただしこれは収束の必要条件ではないが、特徴ベクトルは基底の意味合いを持つので重みを正規化するという意味においても絶対ノルムが 1 であるというのは望ましい性質として挙げられるのだ。

また、特徴ベクトルの数について、特徴ベクトルを増やすとそれだけ近似精度が上がる。しかしこれの注意点として、特徴ベクトルを増やしすぎると過学習する危険性もあり、ある程度の数に収めておくことが必要である。

線形アーキテクチャの汎化 (補完) 性能を上げるために特徴ベクトルのカバー範囲を広げることも挙げられる。例えば RBF において分散を大きくすることでカバー範囲を広げるということは可能であろう。

より良い線形アーキテクチャの構成のために、こうした条件があるので注意が必要である。

ここで特徴ベクトルの生成方法について述べておこう。特徴ベクトルの作り方はいくつかあるが、ベーシックな手法としては離散状態表現によって作る方法がある。これはタイルコーディングとも呼ばれ、状態関数 $V(s)$ を状態 s について各グリッドごとに取り出し、その離散化された状態価値を特徴ベクトルとする手法である。

また、別の特徴ベクトルの生成方法として、タイルコーディングを複数組

み合わせる CMAC[7]という手法もある。

これ以外にも、適応的状態分割によって自動的に特徴ベクトルを獲得していく方法もある。

6.6 線形近似と非線形近似

ところで今までずっと線形アーキテクチャによる関数近似を行ってきたが、非線形の場合についても考えることが可能である。この手法が昨今話題となっている「ニューラルネットワーク」であるわけなのだが、特に工夫することなくニューラルネットワークによる関数近似を行うと学習が発散するという例がある。[8]

よって普通にニューラルネットワークは解の収束性が保証されていないのである。ニューラルネットワークは非線形関数の近似が行える点で非常に有用なのであるが、解が発散しないようにするために様々な工夫が必要となる。

解を収束させるためのニューラルネットワークへの工夫については第 7 章に譲るとするが、ひとまず現時点ではただのニューラルネットワークは解が発散しうるということを頭に入れておいてほしい。

今やディープラーニングが世間で大ブームとなっており、機械学習について何も知識のない人間が思考停止で「ディープラーニングだ！」とわめき立てているシチュエーションを定期的に目にするが、こうした事実を知ってお

[7]Sutton, R. S. and Barto, A.; Reinforcement Learning: An Introduction, A Bradford Book, The MIT Press (1998).

[8]Baird, L.; Residual Algorithms: Reinforcement Learning with Function Approximation; Proceedings of the 12th International Conference on Machine Learning,

けば「ディープラーニングを使う前に解の収束性を示してからものを言ってくれ」と思うだろう。

6.7 連続な行動空間への応用

今まで状態空間が連続な場合についての対処法を説明してきたが、行動空間が連続の場合についても同様に線形アーキテクチャで対処が可能である。

6.7.1 Actor-Critic

Actor-Critic では、行動空間が連続な場合について方策は

$$\pi(a_t|s_t) \approx N(\mu(s_t), \sigma^2(s_t)) \tag{118}$$

$$= \frac{1}{\sigma(s_t)\sqrt{2\pi}} \exp\left[-\frac{\{a_t - \mu(s_t)\}^2}{2\sigma_i^2}\right] \tag{119}$$

とするとした。

このとき、平均 $\mu(s)$ と分散 $\sigma^2(s)$ は

$$\mu(s) = \sum_{n=1}^{N} \theta_i x_i \tag{120}$$

$$\sigma^2(s) = \frac{1}{1+\exp(-\theta_{N+1})} \tag{121}$$

として表せば、状態価値のときと同様に重みの更新をする。平均 $\mu(s)$ については

$$\theta_i \leftarrow \theta_i + \alpha\left[r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)\right](a_t - \mu(s_t)) \tag{122}$$

として行動 a_t と平均 $\mu(s)$ が近くなる。分散 $\sigma^2(s)$ は

$$\theta_{N+1} \leftarrow \theta_{N+1} + \alpha\left[r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)\right]\left[\{a_t - \mu(s_t)\}^2 - \sigma^2(s)\right] \tag{123}$$

とすることで行動 a_t が $\sigma(s_t)$ の内側なら $\sigma(s_t)$ を小さくし、行動 a_t が $\sigma(s_t)$ の外側なら $\sigma(s_t)$ を大きくすることとなる。

6.7.2 Q 学習

Q 関数の近似には線形アーキテクチャを用いる。そして行動選択において行動空間を適当に離散状態表現近似し (one-step search)[9]、softmax 法や ϵ-greedy 法によって行動選択を行う。

連続な行動空間の Q 学習において ϵ-greedy 法を用いた場合、最大の Q 値をとる行動の探索に Newton 法を用いるという方法もある。

[9]Santamaria, J. C., Sutton, R. S. and Ram, A.; Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces, Adaptive Behavior 6 (2), pp.163–218 (1998).

7　深層強化学習

これまでに述べてきた通り、状態空間が連続な系は離散の系に比べて計算量や近似精度などに問題があり、非常に難しいタスクであったと言える。連続な系の対策として、第 6 章では線形和による近似をすることで収束を保証しつつ連続な系でのパフォーマンスを行えることを示した。

しかし、場合によっては非線形な近似も可能であろう。その非線形近似を可能とするのが「ニューラルネットワーク」である。

線形アーキテクチャにおいては状態 s から離散状態表現 (タイルコーディング) によって特徴ベクトルを取得し、重み付けをして足し合わせることで状態価値 $V(s)$ を近似したが、ニューラルネットワークにおいては状態 s においてコンボリューションニューラルネットワーク (CNN) によって畳み込み演算をすることで特徴ベクトルを用いることなく直接的に状態価値 $V(s)$ を近似する。

実を言うとコンボリューションニューラルネットワークにおける畳み込み演算とは数学的に定義された畳み込みのことではないので、ここでいう畳み込み演算は $\mathcal{S} \to \mathbb{R}$ のように「畳み込まれているような」イメージに過ぎないものであることに注意したい。

さて、ニューラルネットワークによる非線形近似を行うわけなのだが、第 6 章で一度触れた通り、ニューラルネットワークによる非線形近似を単純に行った際、解の収束の保障がされていない。

よってニューラルネットワークを使う際にはいくつかの工夫が必要なのだが、それらを実践した結果うまくいったのがこの深層強化学習である。そして最

初の深層強化学習の成功例が 2013 年に発表された DQN(Deep Q-Networks)である。

本章では DQN から始まって、DQN の改善をした DDQN や GORILA、Deling DQN、A3C などの深層強化学習アルゴリズムを紹介していく。

7.1 DQN; Deep Q-Networks

DQN[10]は深層強化学習の先駆けとも言える手法で、2013 年に発表[11]、2015 年に電子版 nature にて発表された。[12]

強化学習において、状態空間が大きくなると計算量が爆発してしまう「次元の呪い」はかなり根深い問題であった。この問題に対して、DQN は状態 s_t における各行動の状態行動価値 $Q(s_t, a_0), Q(s, a_1), \ldots, Q(s, a_n)$ を出力する Q 関数をニューラルネットワークによって近似する。

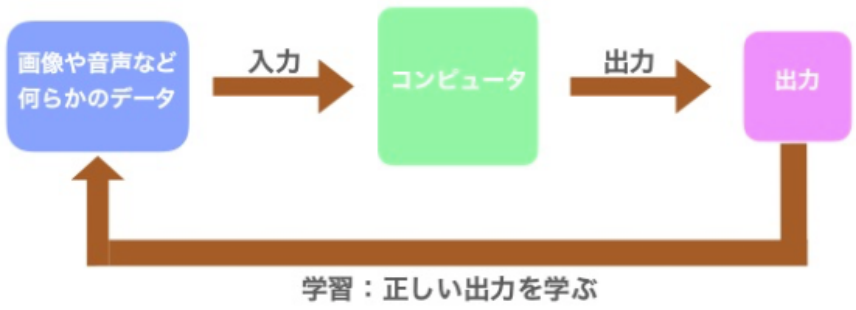

図 14: ニューラルネットのイメージ

アルゴリズムとしては非常に簡単で、状態 s をニューラルネットワークに投げた際に状態行動価値 $Q(s, a)$ を出力するようにネットワークを学習させれば良い。つまり DQN では入力としてゲーム画像の $w \times h$ のピクセルサイズと c パターンの色スケールの状態 $s \in \mathbb{R}^{w \times h \times c}$ をニューラルネットワークに入力し、出力として各行動についての状態行動価値 $Q(s, a)$ を出力する。

ここでニューラルネットワークによって近似された Q 関数はパラメータ θ

[10]DQN は「ドキュン」ではなく「ディーキューエヌ」と読む。
[11]V Mnih et al; "Playing Atari with Deep Reinforcement Learning"
[12]V Mnih et al; "Human-level control through deep reinforcement learning"

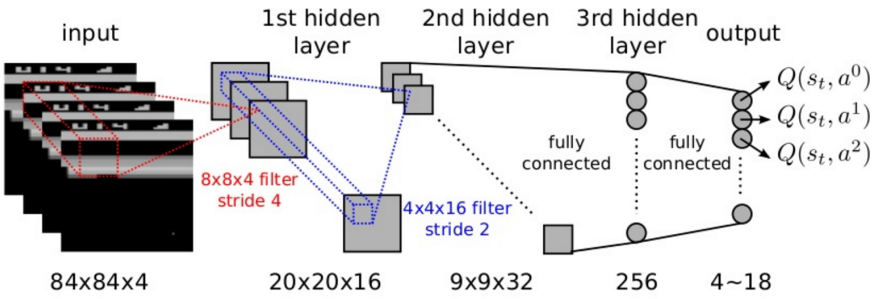

図 15: DQN のアーキテクチャ

によって表現されるので、以下では近似 Q 関数を $Q_\theta(s,a)$ と表す。ニューラルネットワークの実装に際して、$Q_\theta(s,a)$ の学習は Q 学習のアルゴリズムに従って $r_t + \gamma \max_{a\in\mathcal{A}} Q_\theta(s,a)$ に近づくようにすれば良い。

つまり DQN は、教師信号としての $r_t + \gamma_{a\in\mathcal{A}} Q_\theta(s,a)$ に対して $Q_\theta(s,a)$ を最適化すれば良い、という単純な回帰問題に帰着する。よって損失関数は

$$L(s) = \frac{1}{2}\left\{ r_t + \gamma \max_{a\in\mathcal{A}} Q_\theta(s,a) - Q_\theta(s,a) \right\}^2 \tag{124}$$

となる。

さて、DQN のニューラルネットワークについて先ほど説明した通り、ニューラルネットワークによる非線形関数近似は一般的に不安定が解が収束の保証はされていない。

これの原因として「データが時系列的に連続で、各データ間の総監が非常に大きいため」「Q 関数を更新することで方策が大きく変わって行動選択が変化するので、観測するデータの分布が多く変わってしまう」などが挙げられる。こうした原因を解決する手法があり、それらを以下に示す。

Exprience Replay

過去の遷移 (s_t, a_t, r_t, s_{t+1}) のセットを保存しておいて、そこからランダムサンプリングしてミニバッチとしてネットワークの重みの更新に利用する。これによって学習データ間の相関をバラバラにし、また、エージェントの行動の分布を過去に渡って平均化することができる。Experience Replay を利用することでパラメータが振動・発散するのを防ぐことができる。

Fixed Target Network

教師信号の Q 値を出力するための Target Network を作り、Q ネットワークのパラメータを定期的に Target Network にコピーし、次の更新まで固定する。つまり古いパラメータを使って教師信号を作る。これによって Q ネットワークの更新と Target Network の更新との間に時間差が生まれ、学習がうまく進むようになる。

Reward Clipping

報酬をある範囲内に制限することで報酬の大きさを制限する。例として

$$r_t = \begin{cases} -1 & (r_t < -1) \\ r_t & (-1 < r_t < 1) \\ 1 & (1 < r_t) \end{cases} \tag{125}$$

のように大きさを制限する。これによって状態行動価値 $Q_\theta(s, a)$ が急激に変化することを防ぎ、方策が大きく変わってしまうことを防ぐこととなる。また、範囲制限をかけることで学習内容によらずハイパーパラ

メータとして $Q_\theta(s,a)$ を最適化できるというメリットがある。

Applied Loss Function

Q 学習においての TD 誤差は

$$TDerror = r_t + \gamma \max_{a \in \mathcal{A}} Q_\theta(s,a) - Q_\theta(s,a) \tag{126}$$

であるが、一般的に二乗誤差を利用することで誤差伝播を行う。具体的に、一般的な誤差関数は

$$L(s) = \frac{1}{2} \left\{ r_t + \gamma \max_{a \in \mathcal{A}} Q_\theta(s,a) - Q_\theta(s,a) \right\}^2 \tag{127}$$

であるが、Applied Loss Function では TDerror の絶対値が 1 を超えた場合は二乗誤差ではなく絶対値誤差を利用する。よって誤差関数は Experience Replay によって得られた Q ネットワークの値を $\hat{Q}_\theta(s,a)$ として

$$TDerror = r_t + \gamma \max_{a \in \mathcal{A}} \hat{Q}_\theta(s,a) - Q_\theta(s,a) \tag{128}$$

の TDerror に対して

$$L(s) = \begin{cases} \left[r_t + \gamma \max_{a \in \mathcal{A}} \hat{Q}_\theta(s,a) - Q_\theta(s,a) \right]^2 & (|TDerror| \leq 1) \\ |r_t + \gamma \max_{a \in \mathcal{A}} \hat{Q}_\theta(s,a) - Q_\theta(s,a)| & (|TDerror| \geq 1) \end{cases} \tag{129}$$

となる。これによって学習は安定化する。

擬似コードは以下の通りである。

```
##### Deep Q-Networks #####

Experience Replay Memory <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        Q_old <- Q_network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi(Q_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_old, reward, state_new <- sample()

        ## 教師信号 ##
        if done:
            y <- reward

        else :
            Q_new <- Q-Network(state_new)
            y <- reward + gamma * max(Q_new)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)

        ## 定期的に教師信号リセット ##
        if episode % 5 == 0:
            reset(target)
```

7.2 DDQN; Double Deep Q-Networks

DDQN[13]は DQN の改良版として提案されたアルゴリズムである。状態空間が連続な系においてニューラルネットワークによる Q 関数の近似をする際、学習の不安定性が大きな壁となっていた。そこで DQN はこの学習の不安定性という問題に対して Experience Replay といった工夫によってこの学習の不安性を克服した。

しかし DQN には問題があり、Q ネットワークを更新する際、行動を選択するモデルと行動を評価するモデルが同じパラメータ θ で表現されているので教師信号となる $r_t + \gamma \max_{a \in \mathcal{A}} Q_\theta(s, a)$ によって行動を評価するときに行動を過大評価してしまうこととなる。

そもそも学習が不安定となる原因は学習データの相関の高さなどが原因となっていて、それを回避するために DQN では Experience Replay を利用して教師信号を作成していた。DDQN ではこの教師信号の作成方法を変え、さらにデータの相関を断ち切る手法を提案する。

具体的な手法として、DDQN では教師信号を以下のように変更する。

$$r_t + \gamma \max_{a \in \mathcal{A}} \hat{Q}_\theta(s, a) \to r_t + \gamma \hat{Q}_\theta(s, \arg\max_{a \in \mathcal{A}} Q(s, a)) \tag{130}$$

この工夫によって DQN での行動の過大評価を防ぐことができる。

[13]H.V.Hasselt et al; "Deep Reinforcement Learning with Double Q-Learning"

擬似コードは以下の通りである。

```
##### Double Deep Q-Networks #####

Experience Replay Memory <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        Q_old <- Q_network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi(Q_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_new, reward, state_new <- sample()

        ## 教師信号 ##
        if done:
            y <- reward

        else :
            Q_new <- Q-Network(state_new)
            action_old <- argmax(Q_old)
            y <- reward + gamma * Q_new(action)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)

        ## 定期的に教師信号リセット ##
        if episode % 5 == 0:
            reset(target)
```

7.3 GORILA; GOogle ReInforcement Learning Architecture

GORILA(GORILLA ではない) というネーミングについて、少々無理があるんじゃないかと思わざるを得ない。どうしてもキャッチーな名前にしたかったんだろうな、と想像がついてしまう。

GORILA[14]は、簡単に説明すると DQN を並列化したアルゴリズムである。GORILA では並列化したエージェントを並列して走らせ、非同期的にパラメータを更新していく。

もともと Q 関数の近似において学習データの相関の高さなどが原因となって学習の不安性を引き起こしており、それを回避するために DQN では Experience Replay を利用して教師信号を作成していた。Experience Replay は学習データ間の相関をバラバラにし、また、エージェントの行動の分布を過去に渡って平均化してパラメータが振動・発散するのを防ぐことであった。

この並列化による非同期的パラメータ更新はこの Experience Replay と同様の役割を果たすことができる。方策に基づく行動選択は確率的なものなので各エージェントはそれぞれ独立して様々な経験を蓄積する。その経験 (状態遷移) を非同期的にグローバルメモリに蓄積することでデータ相関は断ち切られることとなるのだ。

また、この並列化によって短時間でグローバルメモリに状態遷移が効率的に蓄積されるので、DQN の場合よりも学習速度が格段に早いのだ。

[14] A.Nair et al; "Massively Parallel Methods for Deep Reinforcement Learning"

擬似コードは以下の通りである。

```
##### GORILA #####

Global Memory <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 各エージェントについて終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        Q_old <- Q_network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi(Q_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_new, reward, state_new <- sample()

        ## 教師信号 ##
        if done:
            y <- reward

        else :
            Q_new <- Q-Network(state_new)
            y <- reward + gamma * max(Q_new)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)

        ## 定期的に教師信号リセット ##
        if episode % 5 == 0:
            reset(target)
```

7.4 Dueling DQN; Dueling Deep Q-Networks

Dueling DQN[15]は DQN の出力部分を改善したものである。

状態 s によって、その後どのような行動 a をとろうが試行が終了するような場合がある。このとき状態行動価値 $Q(s,a)$ は行動 a による価値よりも状態価値 $V(s)$ による価値の方が意味合いとしては大きいだろう。

つまり、深層強化学習においてネットワークの出力は状態行動価値 $Q(s,a)$ よりも状態価値 $V(s)$ と状態 s における行動 a の相対的な価値を表すアドバンテージ関数 $A(s,a)$ を出力した方が結果としては正確だろうということである。ここでアドバンテージ関数 $A(s,a)$ は以下のように定義される。

$$A(s,a) = Q(s,a) - V(s) \tag{131}$$

最終的な状態行動価値 $Q(s,a)$ はこのアドバンテージ関数の定義式から

$$Q(s,a) = A(s,a) + V(s) \tag{132}$$

によって求める。

Dueling DQN の利点は、行動の数が多い場合に状態価値 $V(s)$ が求められているので全行動をわざわざ探索する必要はなく、学習量の軽減が図れるのである。

[15]W.Wang et al; "Dueling Network Architectures for Deep Reinforcement Learning"

擬似コードは以下の通りである。

```
##### Dueling Deep Q-Networks #####

Experience <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        Q_old <- Q_network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi(Q_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_new, reward, state_new <- sample()

        ## 教師信号 ##
        if done:
            y <- reward

        else :
            V, A <- Q-Network(state_new)
            Q_new <- V + A
            y <- reward + gamma * max(Q_new)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)

        ## 定期的に教師信号リセット ##
        if episode % 5 == 0:
            reset(target)
```

7.5 Prioritized Experience Replay DQN

ニューラルネットワークによって Q 関数を近似したとき、学習が不安定となる原因は学習データの相関の高さなどが原因となっていて、それを回避するために DQN では Experience Replay を利用して教師信号を作成していた。そして DQN では Experience Replay としてストックした状態遷移群からランダムサンプリングして学習を行なっていた。

Prioritized Experience Replay DQN はこのランダムサンプリングをやめ、学習が進んでいない状態行動価値 $Q(s, a)$ を優先的に学習していく。優先順位は TD 誤差で順序づけを行なっていく。

DQN では Experience Replay 用にメモリを確保したが、この Prioritized Experience Replay DQN は Experience Replay 用のメモリと同時に各状態遷移に対応した TD 誤差をストックしておくメモリを用意しておく。ちょうどハッシュで対応づけておくイメージである。

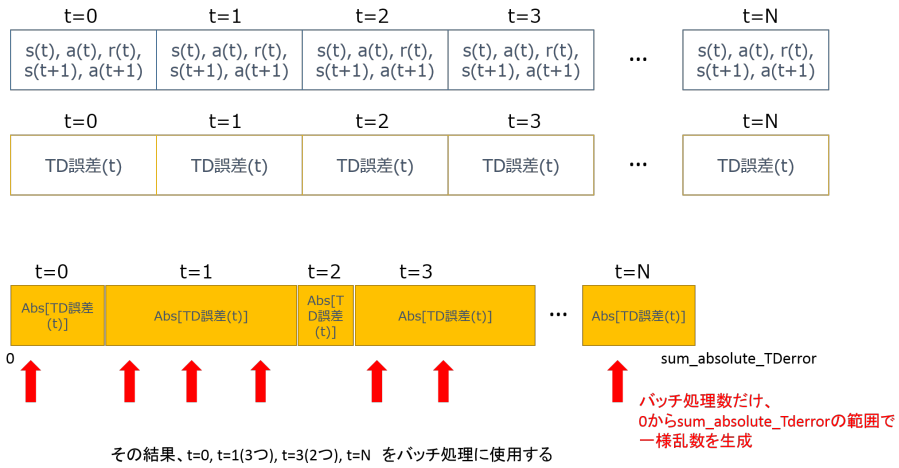

図 16: Prioritized Experience Replay のイメージ

実装に際して、各 TD 誤差について絶対値和を取り、一様乱数によって選択された状態遷移を選択する。これによって TD 誤差が大きい状態遷移ほど選択される確率が大きくなるわけである。この Prioritized Experience Replay DQN で用いられている、総和をとって一様乱数によって選択するアルゴリズムは「ルーレット選択」と呼ばれるものである。

ルーレット選択は重み付き確率選択でよく用いられるアルゴリズムで、Prioritized Experience Replay DQN は本質的に Experience Replay に対して確率的な選択を行なっているのだ。

擬似コードは以下の通りである。

```
##### Prioritized Experience Replay Deep Q-Networks #####

Experience Replay Memory <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        Q_old <- Q_network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi(Q_old)

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 誤差計算 ##
        Q <- max(Q-Network(state_new))
        TD <- reward + gamma * Q - Q_old

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new, TD)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_old, reward, state_new <- sample()

        ## 教師信号 ##
        if done:
            y <- reward

        else :
            Q_new <- Q-Network(state_new)
            y <- reward + gamma * max(Q_new)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)

        ## 定期的に教師信号リセット ##
        if episode % 5 == 0:
            reset(target)
```

7.6 A3C; Asynchronous Advantage Actor-Critic

2013 年に DQN が開発されてから深層強化学習の研究は進み、様々なアルゴリズムが開発された。そしてその中での深層強化学習の一つの到達点とも言えるのがこの A3C[16]である。

A3C は深層強化学習界隈にとって革命的なアルゴリズムで、かつての深層学習系のタスクは高級な GPU をフル稼働して初めて実行できるものであった。しかしこの A3C は、様々な工夫をこなすことで安価な CPU によって深層強化学習を実行することができるアルゴリズムなのである。

さて、A3C の具体的なアルゴリズムについてなのだが、A3C 自体の成績はものすごいものであるがアーキテクチャとしては非常に簡単である。

A3C とは Asynchronous Advantage Actor-Critic の略で、

Asynchronous

: GORILA に用いられた非同期的パラメータ更新のこと。並列化したエージェントを並列して走らせて非同期的にパラメータを更新し、学習の効率化と Experience Replay と同様の役割を果たす。

Advantage

: k ステップ TD 法のこと。k ステップ先まで見るので報酬の先読みができ、学習の高速化が図れる。

Actor-Critic

: Actor-Critic を採用し、ネットワークには状態価値 $V(s) \in \mathbb{R}$ だけを推定。

[16] V.Mnih et al; "Asynchronous Methods for Deep Reinforcement Learning"

A2C(Advantage Actor-Critic) というアルゴリズムがあるのだが、A3C はこの A2C に非同期性を加えたものである。A2C では方策の勾配の推定にベースラインとしてアドバンテージ関数を用いる。方策の評価関数 $J(\theta)$ の勾配は以下の通り。

$$\begin{aligned}
\nabla J(\theta) &= \mathbb{E}\left[Q(_st, a_t)\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] && (133)\\
&= \mathbb{E}\left[\{Q(_st, a_t) - V(s_t)\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] && (134)\\
&\approx \{R_t - V(s_t)\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} && (135)\\
&= \left\{\sum_{i=0}^{k-1}\gamma^i r_{t+i} - \gamma V(s_{t+k}) - V(s_t)\right\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} && (136)
\end{aligned}$$

このように方策の勾配を獲得し、$R_t - V(s_t)$ が大きい行動 a を選択するように更新し、現時点の方策より期待収益が大きくなるようにする。ここで、k ステップ先まで考慮しているので 1 ステップ先までしか見ない場合よりも学習が早く進むことが期待できる。

また、状態価値 $V(s)$ を算出するためのニューラルネットのパラメータについて

$$\nabla\theta_V = \frac{\partial(R_t - V(s_t;\theta))^2}{\partial\theta_V} \tag{137}$$

として、状態価値 $V(s)$ が小さくなるように更新し、方策 π についての期待収益の予測が正確になるようにする。

Actor はより素晴らしい行動をするように、Critic はより正確な批評をできるようにそれぞれのパラメータを更新していくのである。

擬似コードは以下の通りである。

```
##### Asynchronous Advantage Actor-Critic #####

Global Memory <- Initialize
Q-Network <- Initialize
Target-Network <- Initialize


## 学習回数 ##
for (episode : 0 ... max_episode):

    ## 初期状態を得る ##
    state_old <- Initialize

    ## 終了判定 ##
    done <- False

    ## 終わるまでゲームプレイ  ##
    while not done:

        ## 今の状態の状態行動価値を取得 ##
        V_old <- Network(state_old)

        ## 方策で行動選択 ##
        action_old <- pi()

        ## 行動 ##
        state_new, reward <- move(action_old)

        ## 遷移を保存 ##
        save(state_old, action_old, reward, state_new)

        ## 教師信号のための遷移をサンプリング ##
        state_old, action_old, reward, state_new <- sample()

        ## 数ステップ先の教師信号 ##
        for (step : 0 ... max_step):
            if done:
                y <- y + (gamma ** step) * reward
                break

            else :
                act <- pi(state)
                state, reward <- move(act)
                y <- y + (gamma ** step) * reward

            V_new <- Network(state_new)
            y <- y + (gamma ** step) * max(V_new)

        ## 損失関数を計算 ##
        loss <- (y - Q_old(action_old)) ** 2

        ## 誤差逆伝播 ##
        backward(loss)
```

7.7 TRPO; Trust Region Policy Optimization

TRPO[17]は ICML2015 で発表された深層強化学習アルゴリズムで、UC Berkeley のロボットチームの Schulman が発表した。2015 年は DeepMind の DQN が Nature に掲載された年であり、Nature 版 DQN と同じくらい古いアルゴリズムと言えるが、行動空間が連続の場合の深層強化学習では未だに state-of-the-art の 1 つである。

TRPO はロボットの強化学習において革新的な貢献をした手法である。というのも、DQN といった Q 学習は数千回数万回と何度も試行を行うことで最適な行動を獲得していくが、ロボットは何度も試行を行うには非常にコストがかかる。よって Q 学習のような手法は使えないということである。

ではどうするかというと、ロボット制御で必ずコストが低くなる (報酬が高くなる) 信頼区間で方策をロバストに改善していく。つまり、あまりにも急激に変な方向へ方策が変化しないようにする。

さて、具体的にこれを表すと、最適か計算における更新ステップの計算に KL ダイバージェンスによる制約を加える。一見、この手法を難しいことを述べているようだが、これは拘束条件として確率分布間の距離である KL ダイバージェンスを制限しているだけにすぎず、本質的には方策勾配法における自然勾配法と同じなのである。

θ でパラメタライズされた方策 π_θ については方策勾配は

$$\nabla J(\theta) \quad = \quad \mathbb{E}\left[Q(_st, a_t)\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] \tag{138}$$

[17]J.Schulman et al; "Trust Region Policy Optimization"

で、これにアドバンテージ関数を用いて近似する。

$$\begin{aligned}
\nabla J(\theta) &= \mathbb{E}\left[Q(_st, a_t)\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] && (139)\\
&= \mathbb{E}\left[\{Q(_st, a_t) - V(s_t)\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta}\right] && (140)\\
&\approx \{R_t - V(s_t)\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} && (141)\\
&= \left\{\sum_{i=0}^{k-1}\gamma^i r_{t+i} - \gamma V(s_{t+k}) - V(s_t)\right\}\frac{\partial \log \pi_\theta(a_t|s_t)}{\partial \theta} && (142)
\end{aligned}$$

これは以下の式で表される L_θ を θ で微分したものである。

$$L_\theta = \mathbb{E}\left[A(s_t, a_t)\pi_\theta(a_t|s_t)\right] \tag{143}$$

このとき、更新のステップを制限するために以下のように制約を課して最大化を行う。

$$\max_x L_\theta \ ; \ D_{KL}^{max}(\theta_{old}, \theta) \leq \delta \tag{144}$$

ここで θ_{old} は直前の方策のパラメータであり、$D_{KL}(\theta_{old}, \theta)$ は確率分布 π_θ と $\pi_{\theta_{old}}$ との KL ダイバージェンスで、$D_{KL}^{max}(\theta_{old}, \theta)$ は任意のパラメータの組み合わせに対して KL ダイバージェンスを計算したときの最大値を表す。

実用的には組み合わせが膨大になり、最大値を求めるのは難しいため、制約はヒューリスティックに以下のように平均値で代用する。

$$\max_x L_\theta \ ; \ \bar{D}_{KL}(\theta_{old}, \theta) \leq \delta \tag{145}$$

ここで、$\bar{D}_{KL}(\theta_1, \theta_2) = \mathbb{E}_{s\sim\rho}\left[D_{KL}(\pi_{\theta_1}(\cdot|s), \pi_{\theta_2}(\cdot|s))\right]$ である。

ただし、制約において問題を解くのは簡単ではないので、以下のようにソフト制約を使う方に書き下す。

$$\max_x \mathbb{E}_\pi\left[L_{\theta_{old}}(\theta) - \beta\bar{D}_{KL}(\theta_{old}, \theta)\right] \tag{146}$$

以上が TRPO の概要となる。

TRPO はロボット工学における強化学習に非常に貢献したものであるが、同時に強化学習の問題を制約付き最適化問題として表現したという理論的な貢献もあるので、昨今の強化学習界隈では非常に注目を集めている手法である。

8　最新の強化学習の研究

深層強化学習のアルゴリズムはまだ 3,4 年程度の歴史しかない。しかしその中でも深層強化学習はアルゴリズムは様々な広がりを見せている。

第 7 章では最初に DQN が発表され、それが DDQN や Dueling DQN など、DQN を改良した様々なアルゴリズムが提案されている。

2017 年以降、深層強化学習が適用可能な問題設定は多様になりつつある。第 6 章では主に単一環境のビデオゲームにおける深層強化学習を紹介したが、本章では多様な問題設定における深層強化学習のアルゴリズムを紹介していく。

8.1　行動選択の安定化

強化学習の今後の課題となりうるものの 1 つが「ロバストな行動選択」である。この課題の主な適用先としては、TRPO で見たように、ロボット工学である。もちろんロボット工学以外にもこの行動選択の安定化問題は発見されうるだろう。とにかく、試行にコストがかかる環境ではこうした問題は大きな壁となる。

TRPO はこの問題をうまく回避した、いい手法であったと言える。しかし TRPO は以下のような問題があった。

1. 確率分布の距離函数として KL ダイバージェンスを用いていたために実装が複雑
2. 方策勾配法を用いた動的計画法によって解いていたために Actor-Critic に対応できない

3. ドロップアウト手法が使えない

こうした問題を解決するために新たに提案されたのがPPO[18](Proximal Policy Optimization) である。

PPO は学習アルゴリズムとして A3C を採用し、方策の目標値をクリッピングすることでおおまかに方策の更新を制約する。TRPO では KL ダイバージェンスを制約条件として利用していたが、PPO では目的関数を以下の L^{clip} として勾配を求める。

$$L^{clip} = \hat{\mathbb{E}}_t \left[\min \left\{ r_t(\theta)\hat{A}_t, clip(r(\theta), 1-\epsilon, 1+\epsilon) \right\} \right] \tag{147}$$

ここで $r_t(\theta)$ は確率の比率で、

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{148}$$

また、$clip(r(\theta), 1-\epsilon, 1+\epsilon)$ は $r(\theta)$ が $1-\epsilon$ と $1+\epsilon$ の間に収まるようにした関数である。$\hat{A}_t$ は正則化項である。

PPO は簡潔なアルゴリズムであるにも関わらず高い学習性能を示すことが知られている。TRPO は長らくロボット工学における state-of-the-art であったが、この 2017 年に発表された PPO はそれを塗り替えたものであった。

しかし、完璧であるように思えるアルゴリズムでも弱点はあるものである。ひょっとしたらこの PPO も学習効率を引き上げたりと、まだまだ改善点があるかもしれない。そうしたアルゴリズムの改良については今後の課題となっていくだろう。

[18]J.Schulman et al; "Proximal Policy Optimization Algorithm"

8.2　汎化性能の向上

強化学習の課題、ないしは機械学習全体の課題としてあるのが「獲得知識による汎化」である。

人の目とコンピュータの目では明らかに見ているものが異なるし、そこから得ている情報も異なる。なので機械学習モデルでは絶対に人の場合じゃ考えられないことが起きたりする。実際に、機械学習モデルを構築する上で今まで与えていた情報と似ているが異なるものを与えると、モデルは今まで獲得していた知識に全く従わないでロクでもない結果を叩き出す。このような、すでに獲得している知識が使えないことを「破滅的忘却」と表現したりする。

この破滅的忘却の例として、DQN でブロック崩しを学習させたとき、新たな環境としてパドルを数マス上げる、ボールを増やす、ブロックに壁を用意する、ブロックの色を変える、といった操作を行うだけで DQN が今まで獲得してきた知識は容易に崩壊する。今までの環境で学習した知識が全く活かされないのである。

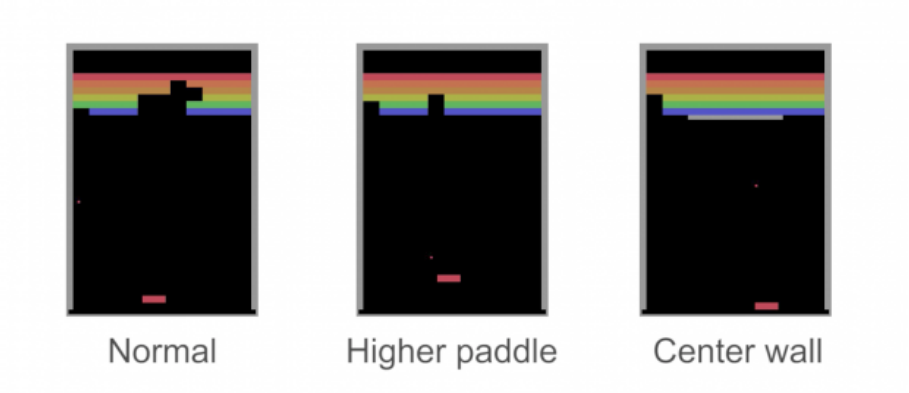

図 17: 汎化性能が試される環境

このように、汎化性能についての問題は非常に悩ましいものであった。汎化性能にチャレンジした研究は 2016 年あたりから活発化してきており、その中で汎化性能を示している手法がいくつかあるので紹介していく。

主に手法としては大きく 3 つに分けられ、それぞれが独自の手法で興味深いものである。以下ではそれらの手法を示していく。

8.2.1 未来予知による価値伝搬

問題設定として迷路問題を考えよう。一度迷路をゴールした際、少しくらいなら迷路の道が変わっても人間なら解くことができるが、DQN は解けない。

未来予知法の基本的な主張は「現在のステップで立ち止まり、仮想敵に n ステップ先まで報酬を予測、それを評価した上で改めて現在の最適行動を選択する」というものである。

実際にどのように未来予知をどうするかについては様々な手法があるが、最もベーシックな手法といえば第 3 章にて紹介した k ステップ TD 法や TD(λ) 法のような価値伝搬法であろう。

これを実践したものとして DeepMind が発表した Predictron[19]がある。Predictron はまさに TD(λ) 法を実践したもので、報酬の先読みができている。

[19]D.Silver et al; "The Predictron: End-To-End Learning and Planning"

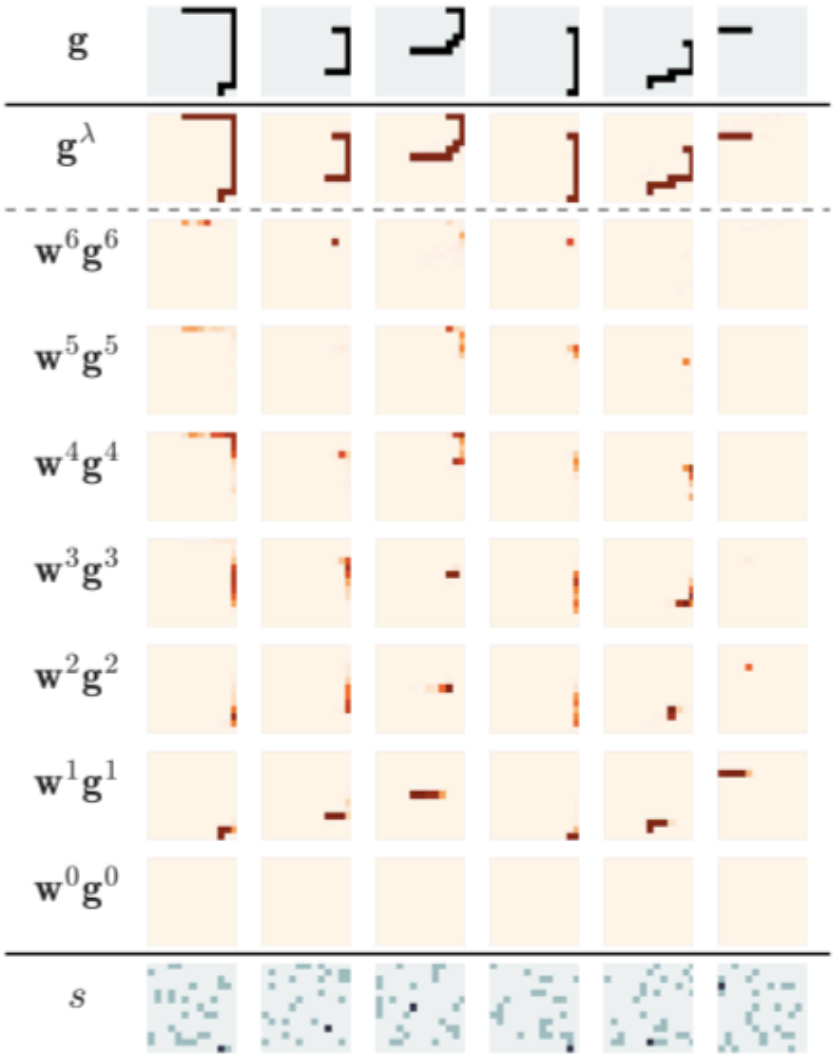

図 18: Predictron の価値伝搬

また、価値伝搬法についてkステップTD法に少し手を加えた手法がValue Iteration Networks[20]である。この論文はNIPSという機械学習のトップカンファレンスにてBest Paperにも選ばれた論文でもある。

Value Iteration Networksは報酬に基づいて行動価値を算出、それを用いて状態価値を算出する。そしてこの状態価値を用いて報酬を予測する。このようなループ構造によって再帰的に状態価値を算出する。

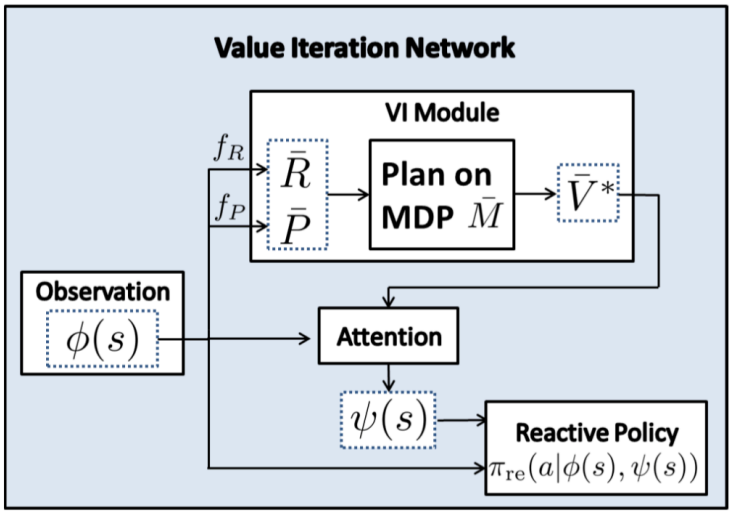

図 19: Value Iteration Networksのアーキテクチャ

実際に状態価値を可視化してみると、ゴール付近を中心に状態価値が高くなっているのがわかる。

Predictronや Value Iteration Networksはkステップ TD法からも着想を得た手法であるが、他にも異なる手法で未来予知を行なっている手法もある。

Predictronなどは状態の遷移確率などを気にせずに TD学習をするモデルフリー的学習であったが、ニューラルネットワーク用いて次の状態を予測する

[20] A. Tamar et al; "Value Iteration Networks"

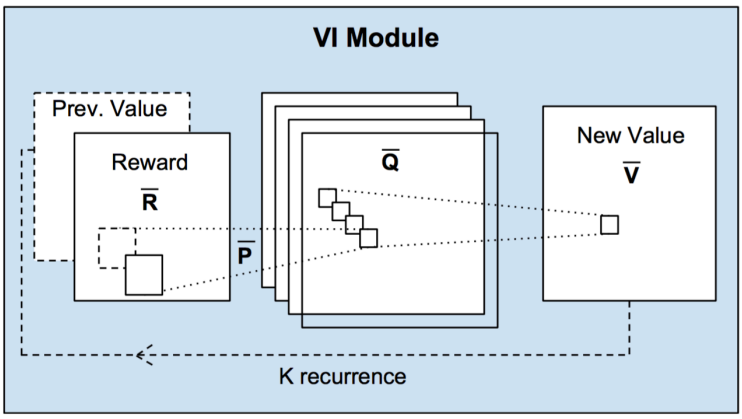

図 20: Value Iteration Networks のアーキテクチャ

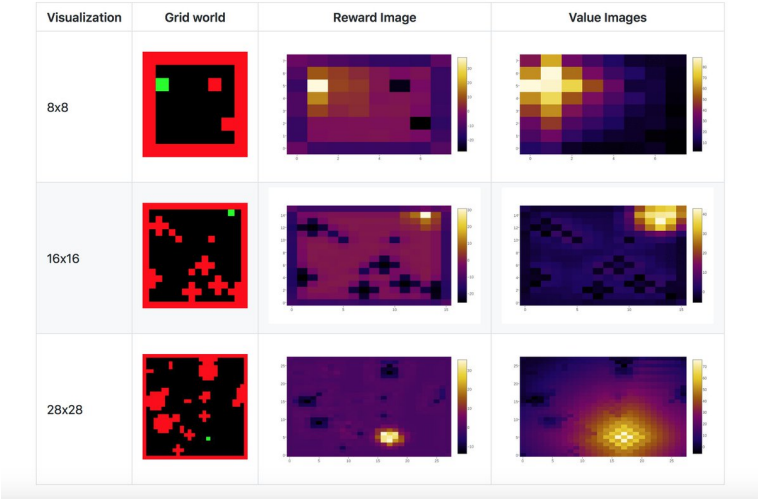

図 21: 価値伝搬を示す例

モデルを構築し、それによって先の未来の状態と報酬を予測するモデルベース的なアイデアもある。

これは DeepMind が提案した Imagination Augmented Agents[21](I2A) という手法もある。次の状態を予測するモデルを構築し、k ステップ刻んだときの報酬と状態のタプル列を時系列データとして LSTM で処理する。

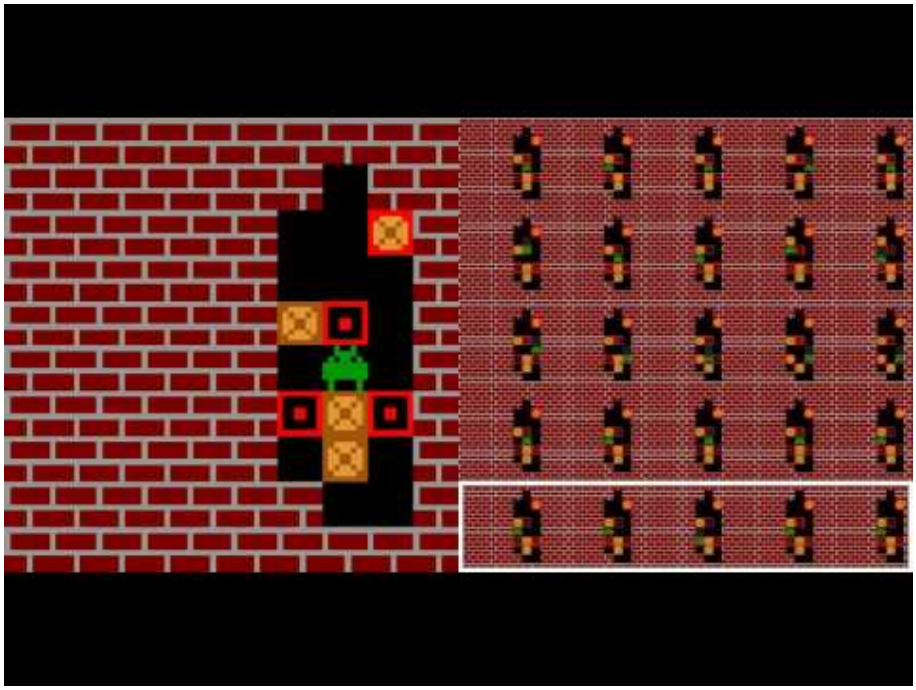

図 22: I2A の予測モデル

以上、未来予知によって汎化性能を持った強化学習エージェントの成功例を紹介した。未来予知による価値伝搬法は様々なアプローチがあるので、色々試行錯誤してみるのも面白いかもしれない。

[21]T.Weber et al; "Imagination-Augmented Agents for Deep Reinforcement Learning"

8.2.2 補助タスク

普通に DQN のように強化学習を行なった際、イレギュラーが発生した場合は既存の獲得知識ではどうすることもなく破滅的忘却が起きた。

そこでこの補助タスクによる手法はイレギュラーを処理する担当の補助タスク用エージェントを別途で用意することで汎化性能を引き上げる。

この手法を実践したのが DeepMind が発表した UNREAL[22]である。UNREAL は UNsupervised REinforcement and Auxiliary Learning(教師なしの強化および補助学習) の略称で、A3C のネットワークに直接の制御目的とは異なる教師なしエージェントを組み込んで、補助タスクによって学習を補助する。

実際にこの UNREAL ではチェックポイントつき 3D 迷路を環境として用いていて、補助タスクとして

Pixel Control

ピクセルで構成された画像が大きく変化する動き (=あちこち首を振る) を補助タスクとして学習させることでチェックポイントを発見しやすくし、ゴールまでの到達の学習を効率化する。

Prioritized Experience

報酬がもらえたときの状態を多く学習させ、現在の状態から将来の報酬を予測させる補助タスクを行う。

[22]V.Mnih et al; "Reinforcement leraning with unsupervised auxiliary tasks"

Shuffle

A3C では状態価値を学ぶ際に過去の経験をシャッフルしないが、シャッフルしたバージョンで状態価値を学ぶ補助タスクを行う

を用いている。これによりエージェントの学習効率を格段に引き上げている。汎化性能以外にも単一の迷路環境での学習スピードが非常に早く、学習スピードに関しては A3C を超える結果を出している。

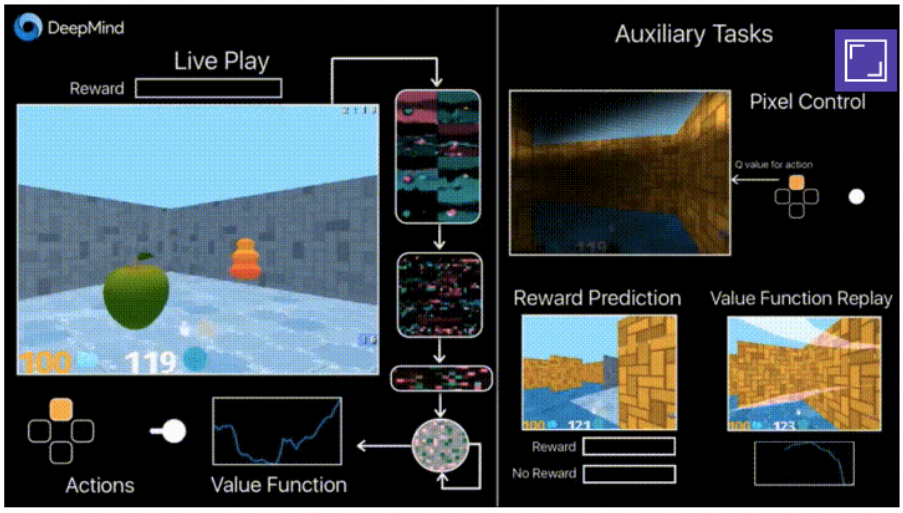

図 23: UNREAL の学習中の補助タスク

8.2.3 記号創発

汎化性能を引き上げるにあたって未来予知や補助タスクなど色々工夫があったが、記号創発はこれらの手法とは一線を画すものである。

何を行なっているかというと、環境中に出てくる物体などを物体として認識することで強化学習を行おうというものである。

DQN や A3C などの強化学習アルゴリズムは、入力としてゲーム画面を受け取り、出力として各行動の Q 値を出力していた。今まで DQN などが環境の変化に対応できていなかったのは入力の画面を一枚の絵としか認識できなかったためであるという見方ができる。ブロック崩しで言えば、どれがパドルでどれがボールか、どれがブロックか認識できていればパドルが数マス上がったとしても対応できる。

このように物体を記号として認識して意識的に操作を行えるかどうかの問題を「シンボル・グラウンディング問題」という。

ここで紹介するのは Deep Symbolic Reinforcement Learning[23]である。今までの DQN などは状態空間を一枚の絵として入力していたが、この手法では状態空間を各物体の座標関係で状態空間を構成する。

各物体についての座標関係を入力とした場合、ピクセルベースでの入力のときより処理するデータ量が非常に少なく、学習効率の向上が図れる。

このように状態空間に記号を用いることで汎化性能以外にも学習効率の向上など、非常に美味しいポイントが増える。記号創発が行えると汎用人工知能が作れつのではないかと言われているほど記号創発は夢のような技術であ

[23]M.Garnelo et al; "Towards Deep Symbolic Reinforcement Learning"

るが、まだ実現には至っていない。教師なし学習的に明示的に記号創発を行うアルゴリズムは多量のサンプル数を要求するので、記号創発自体が機械学習として非常に難しい問題である。

9　終わりに

本稿では強化学習の理論を基礎から構築し、最終的に最新の論文を解説するにまで至った。その中で様々な強化学習の枠組みに触れ、同時に他分野との関連性を見た。実際に、方策勾配法では確率分布や計量テンソルなどの情報幾何学の関わりが見え、TD 学習の学習方式は実際に生物の報酬系に基づく学習を数理モデル化したものであった。

現在の強化学習の研究は非常に長い歴史の中で培われ、そして計算機の発達に恵まれた結果として深層強化学習が花開いた。計算機の進化が既存の問題を解決し、機械学習の表現力を格段に高めたのである。そしてそれによって「これがあったら便利だな」と思えるような技術が実現され、社会実装されつつある。

しかし、一つ歩を進めればまた新たな問題が出てくる。実際に第 7 章では汎化性能や記号創発、学習効率など様々な問題が出てきた。最新の研究でもまだこうした問題は存在するのである。

今後の強化学習はどうなるのか、まだわからない。一部では強化学習を使わなくとも進化的アルゴリズムを用いれば強化学習と同等の計算をすることができ、DQN 程度ならばスコアを達成できるという報告もあり、強化学習の不要説すら唱える論文すら存在する。

いずれにせよ、強化学習の思考法は生物学や化学、ビジネスなど多様な分野で活かす余地があるだろう。本稿で学んだ知識が今後の様々な分野の発展につながれば筆者としてこの上ない幸いである。

10　付録

本稿では強化学習のアルゴリズムを紹介したが、それを実践するにあたってまず強化学習アルゴリズムよりもゲーム環境が用意できないと話にならない。

よって強化学習を試す環境を用意する必要があるのだが、自分でこれらを用意するのは非常に面倒な作業である。こうしたことが強化学習の厄介なポイントでもある。

こうした問題を解決するべく、OpenAI は OpenAIGym という多様なゲーム環境を扱うモジュールを開発した。

付録としてこの OpenAIGym の導入方法、使用方法を解説していく。

以下では macOS と Ubuntu の環境を想定して解説する。Windows 環境の読者は何らかの工夫をして Linux を導入するか、諦めるかを選んでほしい。

10.1　導入

Python のモジュール管理を pip で行なっているならば以下のコマンドで OpenAIGym を導入することが可能である。

```
pip install gym
```

もしくは conda や source を使っているならば以下でも導入ができる。

```
git clone https://github.com/openai/gym.git
cd gym
pip install -e .
```

ここで、ある程度の依存パッケージがあるので、それをインストールしておく。

macOS の場合、

```
brew install cmake boost boost-python sdl2 swig wget
```

Ubuntu 14.04d の場合、

```
apt-get install -y python-numpy python-dev cmake zlib1g-dev
    libjpeg-dev xvfb libav-tools xorg-dev python-opengl
    libboost-all-dev libsdl2-dev swig
```

10.2 使用方法

OpenAIGym は多様な環境が用意してある。公式サイトは以下の URL である。

https://gym.openai.com

その中の Environments のページを見ると、様々な環境が用意してあることがわかる。

https://gym.openai.com/envs/classic$_c$*ontrol*

使い方として、以下のように環境のコンストラクタを用意する。ここでは Breakout を用いる。

```
import gym
import numpy as np

env = gym.make("Breakout-v0")
```

これによって Breakout の環境を用意できた。状態空間や行動空間は以下で確認できる。

```
## 状態空間 ##
env.observation_space

## 行動空間 ##
env.action_space
```

OpenAIGym での状態は Breakout のようにピクセルベースで表現されているものや CartPole のように車の速度などで表現されているものがあるが、全て Numpy 配列によって構成されている。よって計算量を落とすための画面の前処理 (グレースケール化や画面のリサイズなど) について、Numpy での画像処理と同様に行える。Breakout では状態空間は 210×160 のピクセルが RGB の 3 色スケールで表されるので、画面は $210 \times 160 \times 3$ が $0 \sim 255$ の範囲の整数値をとる。

さて、実際に動かすときは以下のようにする。ここで簡便化のために行動は常にランダムをとるものとする。

```
import numpy as np
import gym


env = gym.make("Breakout-v0")

## 学習回数 ##
for episode in range(10):

    ## 状態の初期化 ##
    state = env.reset()

    ## 終了判定 ##
    done = False

    ## ゲームが終わるまで ##
    while not done:

        ## 行動選択 ##
        action = np.random.choice(4)

        ## 行動 ##
        state_new, reward, done, debug = env.step(action)

        ## 状態更新 ##
        state = state_new
```