ちのうのつくりかた：強化学習入門　草稿 ver 0.2

吉田尚人

2016 年 1 月 19 日

第 1 章

はじめに

「知能」という言葉を聞いて，みなさんはどのようなものを思い浮かべますか？

人間の言語能力や，ものを作り出す力を思い浮かべる方もいるかもしれません．犬や猫，その他の動物の，ある程度複雑な問題を解くことができる能力を指す人もいるかもしれませんね．もっと詳しい人だと，コンピュータで “使う” 知能としての Google 検索や Yahoo!の路線検索のような実務的な物を思い浮かべる方もいるかもしれません．もしかしたら，HAL やドラえもん，鉄腕アトムやマルチといった SF の中の人工知能を連想する人もいるかもしれないですね．

本書で説明する「**強化学習**」は，今の 3 番目の例で見た実務的なものとしてはまだ遠い段階にある技術と言えます．強化学習はもともと，むしろ犬や猫，あるいはネズミたちが，置かれた環境で適応していく能力を説明するための神経科学のでの理論として生まれてきました．ある生き物が “**試行錯誤**” を通して，成功と失敗を経験しながら，“**最も良い選択**” を選びとっていく学習プロセスを表現する方法，これが強化学習の出発点でした．そういう意味で，強化学習は実際の「いきものの知能」に最も沿って考えられてきた知能と言えるかもしれません．実際，近年の神経科学では強化学習理論に基いて作られた脳のモデルを用いた研究も頻繁に行われています．

この本は，「強化学習ってなんだろう？」とか，「ちょっとやってみたい」という人への入門編として，極力，理論の深みにハマって行かないコンパクトな文献を目標として書きました．読者の数学的背景は，高校生位を想定しています[*1]．そのくらいの内容の文献であったとしても，現在は強化学習を扱った日本語文献が極端に少ないと思ったからです．また本の最後にはサンプルコードを付録して，基本的にはこれを写すだけで Octave（または Matlab）環境中で強化学習を試すことができるようにしました．Octave のインストール方法や詳しい扱い方は web や他所の文献で非常に多くあるので，この書籍では割愛します．触ったことのない方も，この機会にいじってみてはいかがでしょうか？（いろいろと問題もありますが）とても簡潔な言語なので，プログラミングが始めての方でもオススメできます．

[*1] 少し難しいかも・・・．その時はとりあえずプログラム動かして後から内容を読むようにしてください．

強化学習はもともと上述のように理論神経科学から生まれてきたものですが，現在，強化学習には “マルコフ決定過程” と “確率的推定手法” の深遠な数学理論が土台にあります．この本でこの 2 つの世界を渡り歩くことは入門書としては行き過ぎなので，「そういった理論を通して，このアルゴリズムがあるんだよ」というくらいの気持ちでいてください．本書はアルゴリズムの説明のため，最低限の数式を使用する以外には極力数式を用いず直感的な説明を心がけることとしています．それには賛否両論あるかもしれませんが，あくまで「初めて強化学習と接する，ふつうの高校生がわかる（わかった気になれる）」ものを目標とするため，了承してもらえることを願います．間違いや感想等，改善のためのフィードバックを貰えるとやる気が出ます．

さあ，それでは知能をつくりはじめましょうか．

第 2 章

強化学習とは？

まず「強化学習」とは，どういうものなのか？何をしようとしているのか？をざっと知ることとしましょう．強化学習の目的を理解するためには，まずは強化学習がどういう状況を想定しているのかという “枠組み”を知る必要があります．図 2.1 を見てください．

2.1　強化学習の枠組み：エージェントと環境

強化学習はまず，「**エージェント**」と「**環境**」と呼ばれる 2 つの部分が “**相互作用**” する状況を考えます．これだけではあまりに抽象的ですね．詳しく説明していきましょう．

「エージェント」は，私達がこれから学習システム・アルゴリズムを組み込んでいく “もの” を指します．具体的には，ロボットやパソコンの（知的にしたい）ソフトウェアが当たります．神経科学ではマウスやラットといった動物が当たります．エージェントは環境から “現在の**状態**” と “**報酬**” を受け取り，また現在の状態に応じて “**行動**” を選択して環境に返します．

このとき，エージェントは行動を，現在の状態だけに依存して決めるものとします．つまり，2 つ前の状態や 3 つ前の状態まで考えるのではなく，現在見えている状態から得られる情報のみで行動を選択します．こんなもので大丈夫なのか？という声が聞こえて来そうですが，強化学習の枠組みでは十分であることが保証されています．

「**環境**」は，その言葉からなんとなく連想するものがあると思います．しかし強化学習のなかで定義される環境は，日常で使われることばでの「環境」より，もう少し広い意味を持っていて “エージェント以外のもの” を指します．環境はエージェントから “行動” を受け取ります．また現在の状態と行動から，次の状態と報酬をエージェントに渡します．

まだ少し抽象的でうまく飲み込めないかもしれません．そこで，ここではサーカスで調教師から訓練を受けているライオンという例で説明することにしましょう．サーカスのライオンは色々な芸をする必要があります．今回は「調教師が赤い旗をあげた時，ライオン

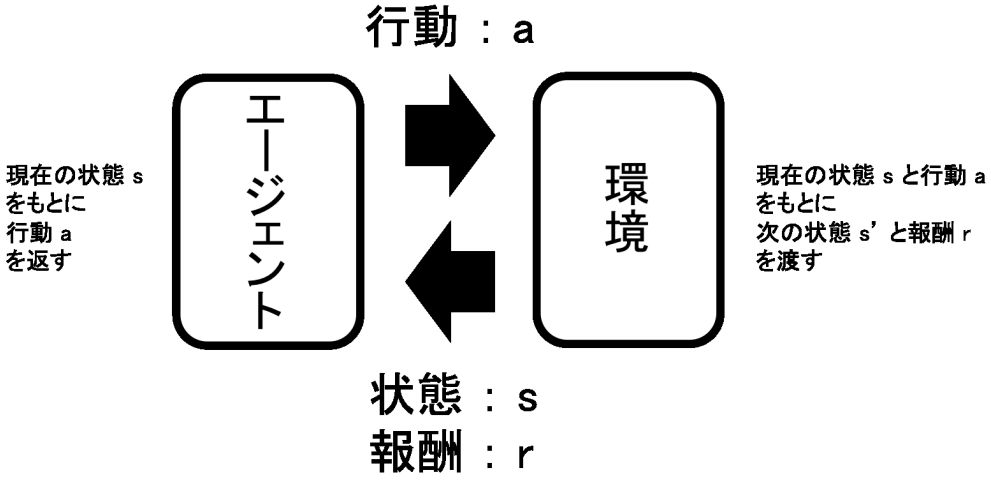

図 2.1　強化学習の枠組み

に特定の芸させる」ことを考えることにします．ライオンは人間の言葉を理解することはできません．そんなライオンに芸を教えるにはどうすればよいでしょうか？

まず，ライオンはお肉が好きで，ムチで打たれるのが嫌いです（そういうことにしましょう）．そこで調教師が赤い旗をあげた時（状態：赤い旗）に，ライオンが目的の芸に近い行動を取った場合は美味しいお肉をあげることにします（正の報酬）．一方，ライオンが全然関係ない行動をとった時は，ムチで叩くことにします（負の報酬）．そのうちライオンはお肉につられて，そのうち調教師が思う芸に沿った行動を覚えることでしょう．

このときエージェントは現在の状態（旗）をもとに行動を選択するライオン．環境は「ライオン以外のもの」ここでは主に報酬（肉，ムチ）と状態を与える調教師がそれに当たります．ここで強化学習のアルゴリズムは，ライオンの脳内にあると考えることもできるでしょう．そのアルゴリズムは，どのようなものと考えることができるでしょうか？

ここでのライオンの学習を説明するひとつの考え方として，「ライオンは今から将来にわたって受けるムチ打ちの数を**最小化**し，将来にわたってもらうお肉の数を**最大化**する」という解釈があります．負の報酬を受ける回数の最小化は報酬の総和を最大化するということでまとめられるので，結局 “**将来にわたって受け取る報酬の総和を最大化する**” という言葉でまとめられます．

強化学習のアルゴリズムはこの “将来にわたって得られる報酬の総和の最大化” という形で定式化された**最適化問題**を解くという形で定式化されます．

強化学習での表記

表記を簡単にするため次からは，行動を a，状態を s，報酬は r で表します．特に報酬は状態 s と行動 a で決められるため $r(s,a)$ とも表記され，実数（$-1.01, 3.5, 10.99$ 等の

小数点を含む正負の数）の値を取ります[*1]．s や a は数に限らず，状況によって様々なものが入ります（s =「赤」，a =「前へ進む」等)[*2]．詳しい感覚は後の例等を通して感じてください．

エージェントの行動は何らかの関数を用いて選択されます．強化学習ではその選択方法を “**方策**” と呼び，$\pi(a|s)$ で表します[*3]．多くの強化学習アルゴリズムにおいて方策は確率で表現されますが，方策 $\pi(a|s)$ の具体的な形は後の章で解説することにします．

強化学習で環境は，次の状態 s' を現在の状態 s とエージェントから受け取った行動 a に基いて変化させます．この変化は “**状態遷移**” と呼ばれ，状態は規則 $P(s'|s,a)$ に基いて変化するものとします．この $P(s'|s,a)$ を**状態遷移確率**（あるいは，確率でなく関数で与えられる場合は状態遷移関数）と呼びます[*4]．

2.2 強化学習の目的

強化学習の枠組みはこれまでのように，エージェントと環境，そして状態・行動・報酬で構成されます．さらに方策関数，状態遷移関数で全体のダイナミクスが変化していく事になります．つまり，最初の状態を s_0 として

$$(\text{環境}：s_0) \to (\text{エージェント}：a_1) \to (\text{環境}：s_1, r_1) \to (\text{エージェント}：a_2) \to (\text{環境}：s_2, r_2) \to \dots$$

という繰り返しで時間が進んでいきます．ここで現れた $s_0, s_1, s_2, \dots$ などの右下の添字はエージェント・環境が何回目のターンを迎えたかという時間刻みを表します．

ここでは先ほどライオンの例を通して考えたアルゴリズムを，もっと具体的に考えてみましょう．そのために，まず最大化（最適化）すべき量を具体的に数値として表現することにします．

強化関数で最大化する量

強化学習の目的は，エージェントに状態遷移確率と報酬関数の事前知識を与えること無しに，このダイナミクスに沿ってあらゆる現在の状態 s_t からの報酬の和

$$\begin{aligned} R_t &= \sum_{i=0}^{\infty} \gamma^i r_{i+t+1} \\ &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \gamma^4 r_{t+5} + \gamma^5 r_{t+6} + \gamma^6 r_{t+7} + \dots \end{aligned}$$

[*1] 正確に言うと報酬は有界の実数値である必要があります．

[*2] これもまた詳しくは，s，a は状態集合 $\mathcal{S}$ と行動集合 $\mathcal{A}$ を構成する要素であればなんでも良い．

[*3] $\pi(a|s)$ は円周率ではないので注意！ policy（方策）の頭文字 ‘p’ のギリシャ文字は ‘π’ なのでそのためです．

[*4] ここでは単に規則と書きましたが，専門的には $P(s'|s,a)$ は「s, a が与えられた時の s' についての条件付き確率」と呼ばれる確率を表します．また，条件付き確率で今の状態遷移関数のように “次の時刻の値が現時刻の値のみに依存する性質” は「**マルコフ性**」と呼ばれます．

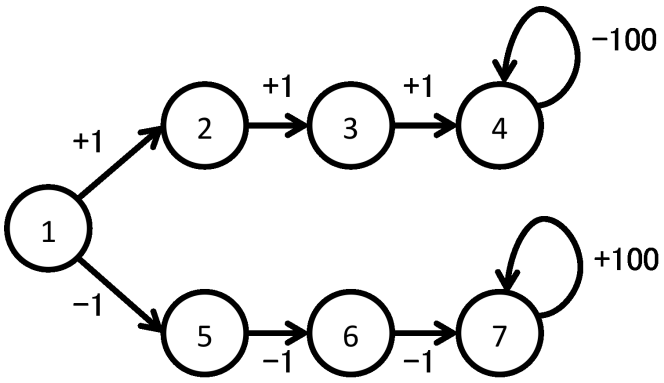

図 2.2　割引率の影響

を最大化させる方策関数 π を見つけることです．ここで γ は 0 以上 1 未満のパラメータで「**割引率**」と呼ばれます．割引率には通常 0.95 や 0.99 といった 1 に近い値を設定します．従ってエージェントは直近の報酬だけでなく，未来に受け取る報酬を含めて行動を選択しなければなりません．

γ の性質を直感的に理解するために，図 2.2 を見てください．図は 7 個の状態（◯）を持つ環境を表しています．各状態から出ている矢印はエージェントの選ぶことのできる行動と，それに伴う状態遷移（2 → 3 など）を表しています．また矢印上の数（+1，-1 など）は状態遷移によってエージェントが受け取る報酬を表しています．エージェントは行動を選択して別の状態に移ることができます．図 2.2 の環境では状態 1 のみ 2 つの行動があり，残りの状態での行動は全て一択になっています．そのため状態 1 以外は行動は 1 つだけなので，状態遷移は決まってしまっています．

このとき，状態 1 で R_t を最大にする行動は γ の値によって変わってしまいます．γ が小さい値，極端な例で $\gamma = 0$ の時はどうでしょうか．上方向に向かう場合は

$$\begin{aligned} R^{\text{上}} &= +1 + 0 + 0 + \dots \\ &= +1 \end{aligned}$$

となります．一方，下方向に向かう場合は

$$\begin{aligned} R^{\text{下}} &= -1 + 0 + 0 + \dots \\ &= -1 \end{aligned}$$

なので，$R^{\text{上}} > R^{\text{下}}$ です．従って $\gamma = 0$ と設定されたエージェントは将来 -100 を受け続ける悲劇的な結末を迎えてしまいます．これは γ が小さいため，目先の報酬に釣られてしまったと言えますね．

逆に γ が大きい値，例えば $\gamma = 0.9$ の場合はどうなるでしょうか？この時は

$$\begin{aligned} R^{\text{上}} &= +1 + 0.9 + 0.9^2 + 0.9^3 \times (-100) + 0.9^4 \times (-100) + 0.9^5 \times (-100) + \dots \\ &\approx -997.29 \end{aligned}$$

となります．一方，下方向に向かう場合は

$$\begin{aligned} R^{下} &= -1 - 0.9 - 0.9^2 + 0.9^3 \times 100 + 0.9^4 \times 100 + 0.9^5 \times 100 + \ldots \\ &\approx 997.29 \end{aligned}$$

となるため $R^{上} < R^{下}$．めでたくエージェントは $+100$ を受け続ける桃源郷に足を運ぶ事になります．γ を大きくすることで，エージェントは遠くまで見据えて最終的に将来の報酬が大きくなる行動を選択できるようになります[*5]．

従って，繰り返すようですが，このように割引率 γ はどれだけ遠くの未来を含めて行動を選択するかを決める重要な数と言えます．

R_t は現在の状態から “未来に貰う報酬の和” を表します．強化学習の理論は，「これまでの “過去の経験” から “未来の報酬和” の最大化」を可能とする技術ということができます．先程の例では環境を見ながら R_t を計算して方策を比較してきました．次章以降では，この方策をエージェントが過去の経験をもとにして**自動的に**計算する手続きを説明していきます．

[*5] ただし，γ が大きすぎると今度は学習が遅くなってしまうという弱点があります．さきのことを考えすぎて困ったことになるのは強化学習でもおこってしまうのです．

第 3 章

Q 学習と SARSA

前章では強化学習の枠組みと目的を紹介しました．強化学習のアルゴリズムは環境との相互作用を通して “経験” を積み，未来の報酬の和 R_t を最大化する方策関数 $\pi(a|s)$ を探します．

アルゴリズムの込み入った問題に立ち入る前に，先に強化学習の代表的なアルゴリズムである Q 学習（きゅーがくしゅう）と SARSA（さるさ）アルゴリズムの紹介をすることにしましょう．少し数学的な話も出てきてしまうので，とりあえずアルゴリズムを先に使って試してからよく知りたいという方はアルゴリズムの紹介だけ読んで次の章に進んで戻ってくるのも良いかもしれません．

3.1 Q 学習と SARSA

アルゴリズムの説明では，$y \leftarrow x$ といった代入操作をよく用います．この表記は “y に x を代入する”という操作を表すものです．プログラミングの経験がある方には馴染みやすい表記だと思います．不慣れな方は，まあここでは x という箱に入っているものを別の箱 y の中に入れるんだなという感覚でいてください[*1]．Q 学習と SARSA ではどちらも「行動価値関数」と呼ばれる関数 $Q(s, a)$ を用いて学習が進んでいきます．行動価値関数についての少し詳しい説明は 3.2 節でするものとして，この節ではひとまずこの行動価値関数 $Q(s, a)$ という ‘箱’，もう少しプログラミング的に言うと ‘変数’ を使ってアルゴリズムが書かれると思っていてください．この箱は (状態の種類数) × (行動の種類数) というたくさんの区画があり，例えば状態 s_1 と行動 a_3 に関する区画は $Q(s_1, a_3)$ で表されることになります．

[*1] もう少し正確には，「x の中に入っているものをコピーして，y の中に入っているものを捨ててからコピーしたものを y に入れる」といった操作になります．

3.1.1　Q 学習

Q 学習は強化学習において最も単純なアルゴリズムで，しかも必ず目的の強化学習問題で最適な方策（R_t を最大にする方策）を学習することが知られている有用なアルゴリズムです．Q 学習のアルゴリズムは以下のように表されます．なお，全ての $Q(s,a)$ は学習の開始前に任意の値に初期化されているとします．

1. 状態 s で行動 a を方策 $\pi(a|s)$ から選択する．
2. 環境から次の状態 s' と報酬 r を受け取る．
3. TD 誤差 $\delta = r + \gamma \max_b Q(s',b) - Q(s,a)$ を計算する．
4. $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$，$s \leftarrow s'$ として 1 に戻る．以下，繰り返し．

ここで γ は前述した割引率，α は「学習率」と呼ばれるパラメータです．学習率 α には正の小さな値，例えば 0.1 や 0.01 といった値が使われます．$\max_b Q(s',b)$ は「状態 s' が固定された時，$Q(s',b)$ の最大値を返せ」という事を表しています．$Q(s,a)$ は状態と行動の関数なので，状態 s を固定されると後は a を動かして $Q(s,a)$ の最大値を探さなければなりません．

3.1.2　SARSA

SARSA もまた Q 学習と同様，強化学習において最もポピュラーなアルゴリズムの一つです．Q 学習と非常によく似ていますが，TD 誤差の計算に行動価値関数の最大値ではなく，実際にエージェントが選択した行動を用いるところが異なります．

1. 状態 s で行動 a を方策 $\pi(a|s)$ から選択する．
2. 環境から次の状態 s' と報酬 r を受け取る．
3. 状態 s' で行動 a' を方策 $\pi(a'|s')$ から選択する．
4. TD 誤差 $\delta = r + \gamma Q(s',a') - Q(s,a)$ を計算する．
5. $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$，$s \leftarrow s'$，$a \leftarrow a'$ として 2 に戻る．以下，繰り返し．

SARSA は Q 学習に比べ，より学習が安定していることが知られています．そのため神経科学においての脳のモデルや，非常に状態数や行動数の大きな強化学習問題を様々な方法を組み合わせて解こうとする場合に用いられる傾向があります．

3.1.3　方策関数

いまのような Q 学習や SARSA はいずれも，方策関数 $\pi(a|s)$ を使って状態 s から行動 a を選択します．ここでは，計算した行動価値関数 $Q(s,a)$ から方策関数 $\pi(a|s)$ を実際に組み立てる方法を説明します．一般的に広く使われている確率的な行動選択規則，ϵ(いぷしろん)-貪欲行動選択とソフトマックス行動選択について紹介しましょう．

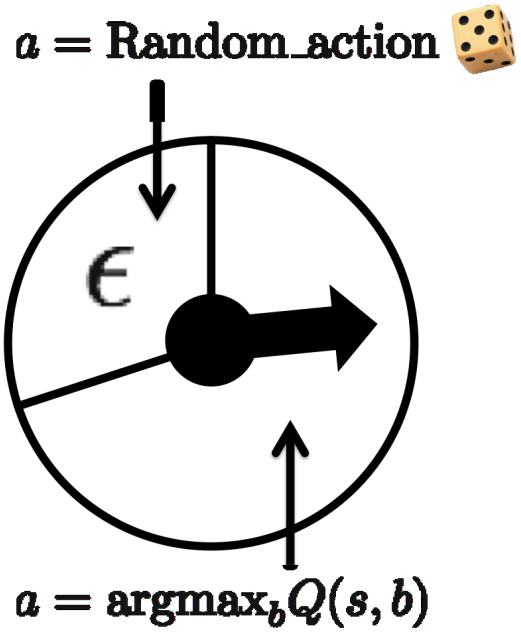

図 3.1 ϵ-貪欲行動選択：確率 ϵ でランダム行動が選ばれる．図の中の $\mathrm{argmax}_b Q(s,b)$ は行動 b のなかで $Q(s,b)$ を最大にするようなものを表している．行動選択ではさらにそれが a に入れられることを表している．

ϵ-貪欲行動決定はとても単純な方法で，以下の方法で行動が選択されます．エージェントがある状態 s にあるとき，

1. 小さな確率 $0 < \epsilon < 1$ でランダムな行動 a を選択する．
2. そうでない場合，つまり確率 $1 - \epsilon$ で Q(s,a) を最大にする行動 a を選択する．

ϵ は 0.1 や 0.01 といった値が使われます．ここで出てきた確率的な選択を実際にプログラムで行う場合は，図 3.1 に示すようなルーレット選択で選ばれます．Octave では **rand** 関数が用意されているので，この関数を使って 0 から 1 の間の乱数を作り出し，この値が ϵ より小さければランダムな行動を選びます．整数値の乱数を作るには専用の **randi** 関数が用意されています．この関数は **randi**(N) と入力することで 1 から N までの整数を等確率で返してくれるので，N に全行動の数 N_a を入力すれば簡単にランダムな行動を選ぶことができます．

ソフトマックス行動選択はもう少し複雑な形状をしています．少し難しくなりますが，ソフトマックス行動選択では状態 s で以下の確率に基いて行動 a を選択します．

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_b^{N_a} e^{Q(s,b)/\tau}}$$

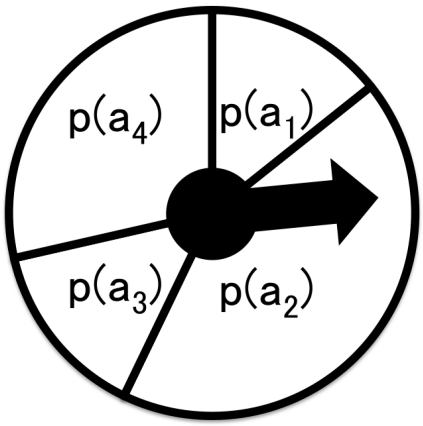

図 3.2　ソフトマックス行動選択：行動は計算された行動選択確率に従うルーレット選択で決まる

ここで e はネイピア数で $e = 2.718\ldots$ の定数です．τ は「温度」と呼ばれる，行動選択のランダムさを調整するパラメータです．τ を無限大にまで大きくするとどうなるでしょうか？

このときはすべての a で $Q(s,a)/\tau$ がゼロになってしまうため，全て $e^0 = 1$ になってしまいます．なので $\pi(a|s) = 1/N_a$ となって，完全にランダムな方策になります．逆に τ がゼロに近づく場合は，ソフトマックス行動選択は $Q(s,a)$ の値が一番大きくなる行動だけが確率 1 となります．温度の代わりに $\beta = 1/\tau$ で定義した「逆温度」もよく用いられます．

ソフトマックス行動選択でもルーレット選択が用いられます．具体的には，

1. $e^{Q(s,a)/\tau}$ を各行動 a について計算する（分子の計算）．
2. 計算した $e^{Q(s,a)/\tau}$ 全ての和を Z とする（分母の計算）．
3. 各行動 a の $e^{Q(s,a)/\tau}$ を Z で割る．この値を $p(a)$ とする（選択確率を求める）．
4. $i = 1$ として $P = p(a_i)$ とする（a_1 は 1 つめの行動）．
5. $0 \sim 1$ の範囲の実数から乱数 rnd を 1 つ生成する．
6. rnd が P 以下の場合，行動 a_i を返す．
7. rnd が P 以下でない場合 $i \leftarrow i + 1$ とした後 $P \leftarrow P + p(a_i)$ として，6 に戻る．

の手順で行動が選択されることになります（図 3.2）．プログラミングの経験がない人には複雑な操作に見えるかもしれませんが，付録のサンプルコードにプログラミング例を載せているので参考にしてください．

ϵ-貪欲行動選択では ϵ，ソフトマックス行動選択では τ（あるいは β）が行動のランダムさを決めるパラメータでした．これらパラメータによるランダムさ十分ゆっくり小さくしていくことで，すなわち ϵ と τ を学習が進むに連れゆっくりとゼロに近づけてゆくことで，SARSA は最も良い方策が必ず学習されることが証明されています．

なぜ，方策は確率的な形で表されるのでしょうか？実は毎回 $Q(s,a)$ が最大となる行動を選択する方策では，ある種の問題でいつまでも同じところをぐるぐる回り続けるといった事が発生することが知られています．これでは目的の方策をいつまでたっても学習できません．どのような問題でも学習が進むためには，ほどよい乱雑さを加えて “気まぐれな” 行動を取らせる必要があるのです．

3.2 なぜ Q 学習は働くのか？

先程は行動価値関数 $Q(s,a)$ をいきなり導入してアルゴリズムの紹介をしました．ところでなぜ，Q 学習のような単純なアルゴリズムで強化学習問題は解くことができるのでしょうか？

ここではもう少しアルゴリズムについての説明をしようと思います．理論の概要を説明するに留めますが，ここの内容は少し数学的な表現や専門的な部分に入っていくことになります．なので難しい内容は飛ばして，とりあえずアルゴリズムを試したいという方は次の章に進まれても構いません．

前節の強化学習の枠組みの説明では，環境は状態遷移 $P(s'|s,a)$ と報酬関数 $r(s,a)$ で表され，エージェントとの相互作用で学習が進むという説明をしました．“学習” という制約を外して「$P(s'|s,a)$ と $r(s,a)$ が予めわかっている場合には，環境はどのような性質を持つか？」を考えるのが，強化学習の基盤となる「マルコフ決定過程（Markov Decision Process, MDP）」という数学的枠組みです．

マルコフ決定過程では，行動価値関数 $Q(s,a)$ を

$$\begin{aligned} Q(s,a) &= E_\pi\Big[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\Big|s_0=s,a_0=a\Big] \\ &= E_\pi\Big[r_1+\gamma r_2+\gamma^2 r_3+\gamma^3 r_4\ldots\Big|s_0=s,a_0=a\Big] \end{aligned}$$

と定義します．$E_\pi[\cdot|s,a]$ は「方策 π に沿った場合の期待値」を表しています．ごちゃごちゃしていて大変ですが，噛み砕いて言うならば $Q(s,a)$ は，「s で a を選択して，後は方策 $\pi(a|s)$ に沿った場合に予想される，エージェントが将来受け取る割引された報酬の和」ということになります．エージェントには現在の状態しかわからないので，前節で R_t で表したものを “期待値” というもう少し確率的なものに置き換えたと言えます．最適な方策を π^* で表すとすると，π^* はこの方策関数 1 つで $Q(s,\pi^*(s))$ を全てのsで最大化するような方策と定義されます．・・・と定義はしましたが，本当にそのような π^* は存在するのでしょうか？？

驚くべきことに，MDP のひとつの理論的帰結として，環境が上述のように定義されているならば必ず最適な方策は存在し，しかもその最適方策 π^* はエージェントの現在の状

態 s のみの関数 $a = f(s)$ として表すことができるということが知られています！

だから，強化学習ではあらかじめ方策関数を状態だけに依存する $\pi(a|s)$ で定義するのですね．

さらに MDP では最適方策 π^* の性質も明らかにしています．最適方策 π^* を用いた場合の行動価値関数 $Q^*(s,a)$ は以下の性質を持ち，しかもそのような行動価値関数 $Q^*(s,a)$ はただ 1 つしか存在しないことが知られています．

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_b Q^*(s',b)$$

この関係を満たす行動価値関数 $Q^*(s,a)$ を用いて，毎回 $Q^*(s,a)$ が最大となる行動を選択することで最適な方策を実現することができます．

さて一方，Q 学習（と SARSA）では学習の開始前に $Q(s,a)$ は任意の値に初期化されているので，当然今の関係は成り立ちません．そこで，上の関係を満たす様な $Q(s,a)$ に少しずつ修正していく戦略を取ります．そのためには，まず上の式の $Q^*(s,a)$ を $Q(s,a)$ で置き換えて，両辺の差 d を取ります．

$$d(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_b Q(s',b) - Q(s,a)$$

$d(s,a)$ は最適な $Q^*(s,a)$ では正確にゼロですが，そうでない場合は誤差を生じます．この差の分だけ，$Q(s,a)$ を修正してやる必要があるのです．しかしこの式の右辺第二項には環境の状態遷移 $P(s'|s,a)$ を使った式が含まれてしまっているので，強化学習のエージェントからはこれを計算することができません．そこで実際にアルゴリズムを動かしている時に得られた，次の時刻の状態 s' を使うことにして，そうしたものを δ と表しましょう．

$$\delta(s,a,s') = r(s,a) + \gamma \max_b Q(s',b) - Q(s,a)$$

見覚えのある式ですね．これは Q 学習のアルゴリズムで使った TD 誤差でした．ちなみに TD は Temporal Difference（即時差分）の略称です．Q 学習の更新式は

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta(s,a,s')$$

で，α は学習率と呼ばれるパラメータでした．これまでの説明から，Q 学習の TD 誤差は最適行動価値関数が満たすべき性質に対して現在の行動価値関数がどれだけズレているかを知らせるシグナルになっていると考えることができます．TD 誤差の定義をみてみると，現在の状態に関する行動価値関数 $Q(s,a)$ が大きすぎる場合は δ は負になり，小さすぎる場合は δ は正になります．Q 学習では経験から得られる，この誤差シグナルを使うことで行動価値関数を最適な値に学習することができるのです．

第 4 章

迷路問題

この章では 3 章で紹介した Q 学習を使って，実際に迷路問題というタスクを例にしてエージェントに学習させてみましょう．ここで使用した問題の Octave（または Matlab）用ソースコードは付録に記載しています．

4.1 Sutton の迷路

今回使う例は「強化学習」という研究分野を創始し，大きく発展させた人物の 1 人である Richard Sutton の考案した迷路（Sutton's Maze）を使うことにします．Sutton の迷路は 6 行 9 列の格子で表されていて，格子のマス目 1 つ 1 つが状態に当たります（図 4.1）．エージェントはこの格子世界の中で，現在の状態から上・下・右・左方向の移動に対応する各行動 4 種類の中から 1 つを選択していくことになります．実際にサンプルプログラムで学習が進行しているときの様子は図 4.2 の様になっています（Octave 使用時）．

今回の例ではエージェントがゴールに入るとそのスッテプのみゼロの報酬が与えられ，それ以外では常に -1 の報酬が与えられるものとします．エージェントがゴールに到着すると 1 回のトライアルが終わったとみなして，再びスタート地点からエージェントはスタートします．

この報酬設定のために将来報酬を最大化する最適方策は可能な限りゴールに早く到達する方策となっていて，Q 学習によって最終的にそのような方策が得られるはずです．

4.2 学習の手順

この迷路問題は，ゴールに到達した時点で学習がぶつ切りになっています．いいかえると，スタートからゴール到達までを 1 回の実験とみなして，それを繰り返しているというように考えることもできます．

強化学習ではこのように 'スタート' と 'ゴール' あるいは '終了' として表されるような明確な区切りを，「エピソード」と読んで 1 つのまとまりとして扱っています．ゲームな

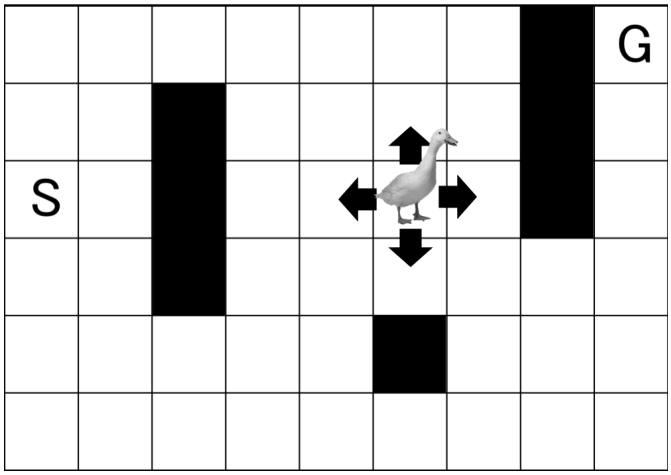

図 4.1　Sutton の迷路

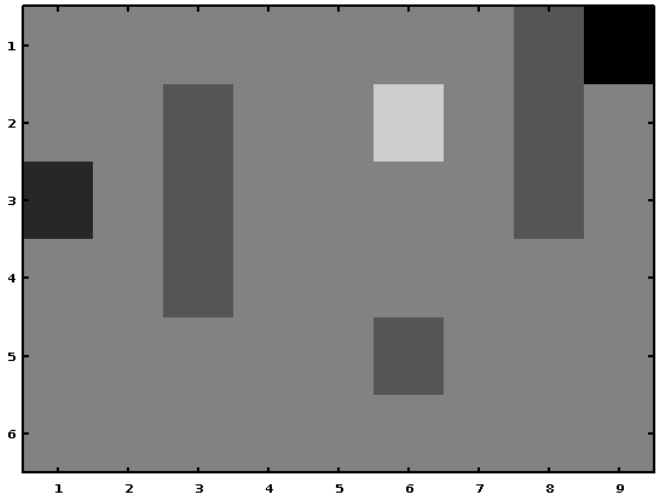

図 4.2　学習中の環境　黒：スタート，ゴール，灰色：壁，白：エージェント（サンプルコードではカラー）

ら，ゲーム開始からゲームオーバーまでが 1 エピソードと捉えることもできますね．マリオならスタートからゴール到着または死んでしまうまでが 1 エピソードとみなすことができます．これには明確な定義があるわけではないので，

論文著者：これを 1 エピソードとよぶ．
ぼく：そう思うんだな．

といったようにまあ納得できれば特に疑問は持たないで話が進みます．このようにエピソードが定義されるような問題は「エピソード的タスク」と呼ばれます．一方，エピソードを定義せず，いわばずーっと 1 エピソードが続くような問題は「連続タスク」と呼ばれ区別されます．

今回の迷路問題ではスタートからゴールまでを 1 エピソードと定義しています．エピソードが終了（エージェントがゴールに入った）ら，エージェントは報酬がずっとゼロで抜けられない状態に入るとみなして最後だけ TD 誤差を

$$\delta = r - Q(s, a)$$

として更新をします．これはゴールでの $\max_b Q(s_{goal}, b)$ が常にゼロとした更新で，この更新の後エージェントはスタート地点に戻されて学習はぶつ切りにされます．

強化学習でエージェントを学習させる手順の一つの方法として，以上のエピソードを何回繰り返すか予め決めてから学習を行わせるといった方法が取れます．エピソードの繰り返しを含めた全体のアルゴリズムの終了条件は今のエピソードの回数を制限する方法の他に，TD 誤差の絶対に関して時間平均をとり，これ十分減少したかを確認して終了するといった手法などがあります．エピソードのパフォーマンスが十分改善されていることを確認して打ち切るといった方法も考えられます．未知の問題で強化学習を試す場合はどれくらいの性能が見込まれるのか全くわからないので，常にパフォーマンスをチェックして必要なだけのエピソード数を学習させたかを確認することが重要です．

今回のサンプルプログラムでは単純に，エピソード数を予め決めて学習を打ち切っています．打ち切りについて興味が湧いた方は，このエピソード数を色々変えて学習された行動を確認してみてください．

4.3 実験の結果

これまでの方法を使って，実際に迷路問題を Q 学習で解いた結果の例を紹介します．強化学習は初期化や学習の進行が確率的に進行するので，皆さんが試した場合は多少異なる結果が得られるはずです．サンプルコードとここで紹介する結果は，ソフトマックス行動選択を使った Q 学習です．ここでは学習のパラメータは学習率 $\alpha = 0.1$，温度 $\tau = 1$ で，割引率は $\gamma = 0.95$ を使っています．皆さんが試す場合は，このパラメータを色々変えてみて違いを観察すると興味深いと思います．

図 4.3 はプログラムの途中経過と結果を表していて，エピソードの進行に沿った各状態での価値関数と呼ばれる関数 $V(s) = \max_a Q(s, a)$ の変化と，エピソードの進行に沿ったスタートからゴール到達までのステップ数の変化を表しています．図の左上は 1 エピソードが終了した後の価値関数の相対的な値を表しています．より明るい状態は価値関数が比較的大きいことを示していて，暗い状態は比較的小さいことを示しています．1 エピソード後はまだ価値関数はデコボコした形状をしていることがわかります．左下は 50 エピソード後の価値関数を表しています．全体はより滑らかになり，ゴールから遠い場所では価値が低くなっている様子が見られます．右上は最終的に得られた 300 エピソード後の価値関数を表しています．よりゴールから遠い状態にまでゴールの情報が伝播されてい

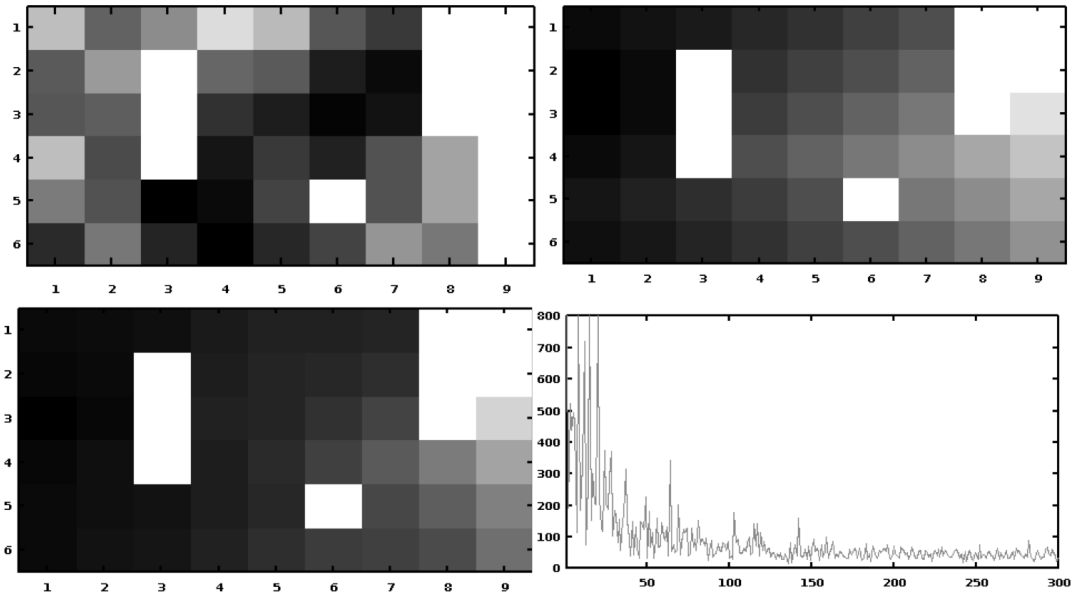

図 4.3　学習中の価値関数の変化とパフォーマンス曲線（サンプルコードではカラー）

ることがわかります．

右下の曲線はいわば環境中でのエージェントのパフォーマンスを表すカーブで，エピソードの進行に沿ってスタートからゴールまでのステップ数が減少していることが理解できます．曲線がギザギザしているのは強化学習が確率的な方法で更新を行っているからで，時々学習の進んでいない場所にハマり込んでパフォーマンスが下がっているところが観察されます．しかし概ね，パフォーマンスは向上していると言えますね．

図 4.4 に描かれている矢印は，最終的に得られた行動価値関数 $Q(s,a)$ から各状態 s で $Q(s,a)$ を最大にする行動を表したものです．灰色部分は少々見えにくいですが，すべての状態でスタートからゴールに向かう最短経路の方向に矢印が向いている事が確認できます．

強化学習は成功しています！

4.4　迷路問題の後は・・？

サンプルコードはマップをいろいろ変えることで，さまざまに違った迷路問題を作ることができるようになっています．ゴールが２つ以上とか，もっと複雑な迷路とか，色々試してみても良いかもしれません．

ところで，一般的に迷路を解く有名なアルゴリズムに「右手法」と呼ばれる方法があります．べつに左手でもできるのですが，要は片手をずっと壁に付けてその壁を辿って行く

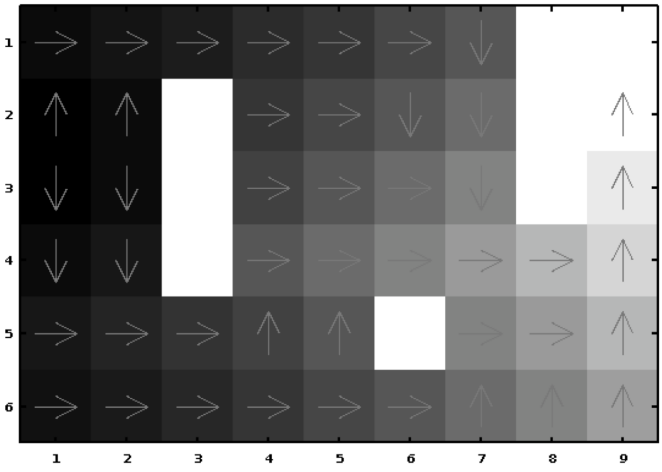

図 4.4　価値関数と最終的に得られた行動（矢印）（サンプルコードではカラー）

と必ず迷路を抜けことができるというものです．この迷路は壁がスッカスカなので，配置を変えてゴールが壁際に無ければ右手法では解けません．強化学習ではこのような問題でも解け，スタートからの経路を最適化することが可能です．右手法では解けない迷路を強化学習で解いてみるもの面白そうです．

本書では迷路問題を例に取りましたが，このような最短経路を探す問題はいたるところでみられます．どうしても環境の情報が限定的にしか得られない場合は，強化学習で解くといった手法を適用できるでしょう（いろいろと制限はありますが・・・）．

ここでは環境を私達が 1 から組み立てて状態遷移と報酬をエージェントに与えてきました．でも，より実際的な問題では環境はむしろ自然界や社会から与えられるものですよね？　強化学習は環境として「状態」と「報酬」をエージェントに与えてやれば勝手に将来の報酬和を最大化する方策を探してくれる一般的な方法です．今回の迷路の例はあくまで，パフォーマンスや学習された行動の経路，価値関数の全体像といったものを見やすくするために使ったとても「人工的な」環境です．

実際に強化学習をより “**ワイルドな**” 環境で試す場合には，このような可視化はとても難しいと思います．そういった場合には，この章で紹介した実験のデザインとエージェントだけをプログラミングすることになるでしょう．

第 5 章

おわりに＋あとがき

冒頭でも紹介しましたが，強化学習のとても単純な，入門的なものを書いてみたかったので今回はそのようなものにチャレンジしてみたのが本書です．できるだけ数学的なバックグラウンドを想定しないで，パソコンを持っている女子高生がちょっと強化学習を試してみようかというきもちになるようなものを目指しました（願望）．「嘘は書かない＋できるだけ数学的な素養のいらない説明[*1]＋サンプルプログラムを付録につける」とするとこれが最低限かな．という心持ちで書き始めた次第です．こういう形でのアウトプットは始めてですが，改めて学んだこともまた多かったと思います．

強化学習とそれを裏付けるマルコフ決定過程の深遠な世界．本書はその入門とするにはあまりに稚拙な内容です．本書で扱った Q 学習や SARSA の他にも，関数近似手法を本格的に用いて連続値で表される状態や莫大な数の状態を扱えるようになった近似手法による強化学習，連続値で表される行動を可能とするアクタークリティックアルゴリズムと方策勾配定理，状態観測に不備がある際の部分観測 MDP（POMDP）など，触れられなかった話題は多々あります．近年では深層学習とのコラボレーションによる，本格的に難しい問題への応用が進められてきています．

深層学習のみで全てが解決するほど，動物のように自然界（あるいは社会）で生き残るというタスクは人工知能のエージェントにとって甘くないというのが現状だと思います．しかしながら自分もまた，強化学習ひいては汎用人工知能に関心をもつものとして，楽しみな時代でもあるというのは紛れもない事実です．

強化学習は機械学習とよばれるコンピュータサイエンスの中でも，試行錯誤・行動を試しながら転んだり擦り剥いたりしながら学んでいく，とても生きものっぽい性質を持つ不思議な技術です（だから実用化がなかなか進まないのもありますが・・・）．本書を通して強化学習に少しでも興味を持っていただければ幸いです．

[*1] 自分の素養がどうだと言われると耳が痛いですが．

付録 A

サンプルコード

付録には第 4 章で紹介した，Q 学習で解く迷路課題の全ソースコードを記載します．ソースコードは Octave と MATLAB での動作確認を行いました．Octave の入手は https://www.gnu.org/software/octave/を参照してください．Octave のインストール方法は日本語での丁寧な解説がネット上にたくさんありますので，詳しくはそちらを参考にしてください．

サンプルコードを試すには，ここで書かれている全てのソースファイル（.m ファイル）を同じディレクトリ（ファルダ）に入れて octave で該当のディレクトリに移動し，octave のコマンドラインで

octave:1> main_Q

と入力することで試すことができます．

A.1 メイン関数：main_Q.m

```
clear all; close all; clc
figure(1)
figure(2)

fprintf('-----------------------------------------------\n')
fprintf('RL learning algorithm in the Maze World!\n')
fprintf('Naoto Yoshida 09-11-2014\n')
fprintf('\n')
fprintf('Press any key to START! \n')
pause

% エピソード数の指定
maxEpisode = 300;


% 迷路の地図
% 0 : none
```

```
% 1 : wall
% 2 : start
% 3 : goal
WALL  = 1;
START = 2;
GOAL  = 3;
map = [ 0 0 0 0 0 0 0 1 3;
        0 0 1 0 0 0 0 1 0;
        2 0 1 0 0 0 0 1 0;
        0 0 1 0 0 0 0 0 0;
        0 0 0 0 0 1 0 0 0;
        0 0 0 0 0 0 0 0 0];

N_col    = size(map,2);
N_row    = size(map,1);
N_state  = N_col * N_row;
N_action = 4;


% Visualization variables
% 価値関数表示のための変数: V(s) = max_a (Q(s,a))
V = zeros(N_state, 1);
% 各エピソード終了時のステップ数を入れる変数
Steps_to_goal = zeros(maxEpisode, 1);


% エージェントのパラメータ
alpha    = 0.1;  % 学習率
gamma    = 0.95; % 割引率
% epsilon  = 0.1;  % epsilon貪欲行動選択のためのパラメータ-
tau      = 1;  % ソフトマックス行動選択のためのパラメータ


% 行動価値関数の初期化
Q = 0.0001 * randn(N_state, N_action);

% Main loop
for episode = 1:maxEpisode
    steps      = 0;
    goal_flag  = 0;
    [~, state] = max(map(:) == 2); % 初期状態はマップ上の「2」の場所


    while goal_flag == 0
        % 行動選択
        agent_policy
        action = action_dash;

        % 状態遷移・報酬の受け取り
        environment_update
```

```
        % Q 学習
        agent_learn_Q

        % 状態の更新
        steps  = steps + 1;
        state  = state_dash;


        % 各ステップの環境の表示
        % （各行の先頭に ‘%’ をつけてコメントアウトすることで実験を劇的に速く行えます）
        figure(1)
        a_pos = reshape((1:N_state == state_dash), N_row, N_col);
        visual = 0.6* a_pos - map/3;
        set(gca, 'YDir','normal')
        imagesc(visual)
        caxis([-1, 1])

        % MATLAB でプログラムを試す場合は次の行をコメントアウト！（% を先頭につける）
        fflush(stdout);

        % 1ステップごとゆっくり見る場合は次の行の ‘%’ を消すしてください
        % pause(0.2)

        drawnow
    end


    fprintf('Episode %d ::: steps to the GOAL : %d steps \n', episode, steps)

    % エピソード終了時の価値関数とパフォーマンス曲線の表示
    Steps_to_goal(episode) = steps;
    for s = 1:N_state
        V(s) = max(Q(s, :));
    end

    figure(2)
    subplot(2,1,1)
    imagesc(reshape(V, N_row, N_col)) % plot value function
    title('Value Function')

    subplot(2,1,2)
    plot(Steps_to_goal(1:episode)) % plot performance
    axis([1, maxEpisode, 0, 800])
    title('Performance')

    drawnow
end
```

```
% 最終的に得られた行動の表示
showAction


fprintf('Experiment Finished.\n')
```

A.2 環境のアップデート：environment_update.m

```
new_state = state;
switch action
    case 1 % UP
        if sum(state == 1:N_row:N_state) == 0
            new_state = state - 1;
        end
    case 2 % DOWN
        if sum(state == N_row:N_row:N_state) == 0
            new_state = state + 1;
        end
    case 3 % RIGHT
        if sum(state == (N_state-N_row):N_state) == 0
            new_state = state + N_row;
        end
    case 4 % LEFT
        if sum(state == 1:N_row) == 0
            new_state = state - N_row;
        end
end


% 報酬
Reward = -1;

map_vec = map(:);
if map_vec(new_state) == WALL % 行動が壁方向だった時の処理

    new_state = state;

elseif map_vec(new_state) == GOAL % ゴールに着いた時の処理（エピソード終了）

    goal_flag = 1;

    Reward = 0; % ゴールステップ時の報酬

end

state_dash = new_state;
```

A.3 行動選択：agent_policy.m

```
% イプシロン貪欲行動選択（試す場合はコメントアウトを外してください）
% rnd = rand;
% if rnd < epsilon
%     action_dash = randi(N_action); % Random action selection
% else
%    [~, action_dash] = max(Q(state,:)); % Greedy action selection
% end


%% ソフトマックス行動選択
p = Q(state, :)/tau;
p = exp(p - max(p));
Z = sum(p);
p = p/Z; % 行動選択確率

rnd = rand;
for tmp_action = 1:N_action
    if rnd < sum(p(1:tmp_action))
        action_dash = tmp_action;
        break
    end
end
```

A.4 強化学習（Q 学習）：agent_learn_Q.m

```
% TD 誤差
if goal_flag ~= 1
    delta = Reward + gamma * max(Q(state_dash, :)) - Q(state, action);
else
    % エピソード終了時の学習
    delta = Reward - Q(state, action);
end


% 行動価値関数の更新
Q(state, action) = Q(state, action) + alpha * delta;
```

A.5 最終結果の表示：showAction.m

```
greedyAction = zeros(N_state,1);
for s = 1:N_state
      [V(s), greedyAction(s)] = max(Q(s, :));
end

figure(3)
action_map = reshape(greedyAction, N_row, N_col);

imagesc(reshape(V, N_row, N_col))
hold on
```

```
for i=1:N_row
    for j=1:N_col
        if map(N_row * (j-1) + i) == GOAL || map(N_row * (j-1) + i) == WALL
           continue;
        end

        switch action_map(i,j)
            case 1
                dp = [0 -0.3];
            case 2
                dp = [0 +0.3];
            case 3
                dp = [0.3, 0];
            case 4
                dp = [-0.3, 0];
        end

        p = [j i];
        vectarrow(p-dp,p+dp)
    end
end
hold off
title('Obtained Action and VF')
```

A.6 矢印を表示する関数：vectarrow.m

※この関数は Rentian Xiong 氏が MATLAB CENTRAL で配布されている vectarrow.m 関数を改変したものです.

```
function vectarrow(p0,p1)

alpha = 0.5;  % Size of arrow head relative to the length of the vector
beta = 0.5;  % Width of the base of the arrow head relative to the length

x0 = p0(1);
y0 = p0(2);
x1 = p1(1);
y1 = p1(2);
plot([x0;x1],[y0;y1],'r');   % Draw a line between p0 and p1

p = p1-p0;

hu = [x1-alpha*(p(1)+beta*(p(2)+eps)); x1; x1-alpha*(p(1)-beta*(p(2)+eps))];
hv = [y1-alpha*(p(2)-beta*(p(1)+eps)); y1; y1-alpha*(p(2)+beta*(p(1)+eps))];

plot(hu(:),hv(:),'r')  % Plot arrow head

end
```