モンテカルロ法による強化学習入門

田中章詞

(理化学研究所 AIP/iTHEMS)
https://github.com/AkinoriTanaka-phys

TOC

強化学習（RL）とは

強化学習（RL）とは

概要

環境 の中で プレイヤー がどのように動けば良いかを学習する

> プレイヤー の数による分類：
>
> | Single-Agent | Multi-Agent |
> | --- | --- |
> |  |  |
> | 一人でプレイ | 多人数でプレイ |
>
> 今回は **Single-Agent** に限定

❓ 環境やプレイヤーの数学的定義？ → ❗ **確率的に定義**

マルコフ決定過程（MDP）

■ 確率変数

- **状態**/state の集合　$\{s\} = S$
- **行動**/action の集合　$\{a\} = A$
- **報酬**/reward の集合　$\{r\} = \mathbb{R}$

環境 / Environment

- **状態**s で **行動**a をとった時の状態変化の確率

$$P_s(s_{\text{next}}|s, a)$$

- その際の**報酬**の確率

$$P_r(r_{\text{next}}|s, a, s_{\text{next}})$$

エージェント / Agent

- **状態**s で **行動**a を取る確率

$\pi(a|s)$　・・・**方策**と呼ばれる

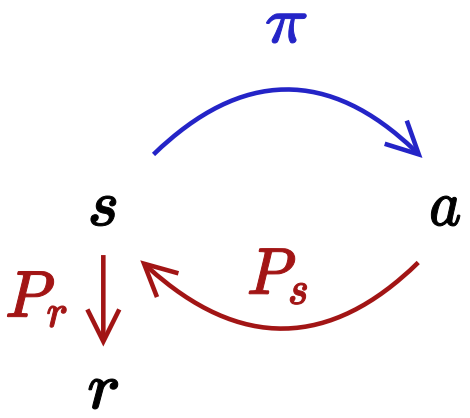
- その他の構造
  - Environment (P) のシミュレータ
  - 状態評価関数, ...etc

マルコフ決定過程（MDP）

■ 確率変数

- **状態/state の集合**　$\{s\} = S$
- **行動/action の集合**　$\{a\} = A$
- **報酬/reward の集合**　$\{r\} = \mathbb{R}$

■ 遷移の図



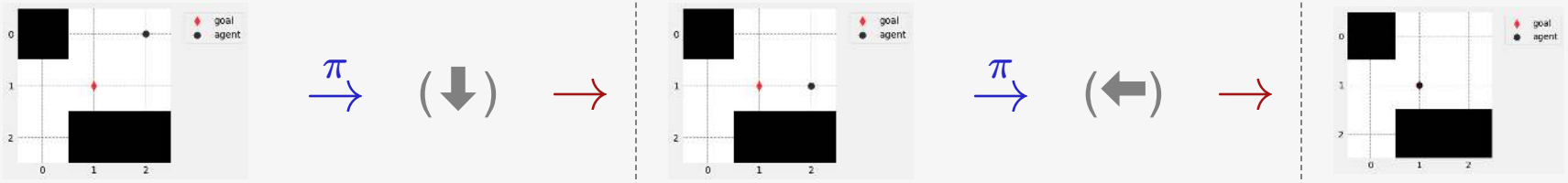
以後、P_s, P_r は略

マルコフ決定過程（MDP）

図を「開いて」書くと…（P_s, P_r は略）

$$s_0 \xrightarrow{\pi} a_0 \to \left| \begin{array}{l} s_1 \\ \downarrow \\ r_1 \end{array} \right. \xrightarrow{\pi} a_1 \to \left| \begin{array}{l} s_2 \\ \downarrow \\ r_2 \end{array} \right. \xrightarrow{\pi} a_2 \to \left| \cdots \right.$$

例：Grid world（二次元の迷路）



$\xrightarrow{\pi}$ (⬇) $\to$ ↓ 0 $\xrightarrow{\pi}$ (⬅) $\to$ ↓ +1

このように、ある環境下のエージェントの振る舞いをマルコフ的[1]として理論を構築する。

強化学習（RL）とは

学習の目標

大雑把には **報酬**r を大きくする **方策** $\pi(a|s)$ を見つけること

■ 本日の方針

> 1. 【**目的関数** ＝ r_t の重み付き和の期待値 】とする
>
> | | | | | | | | | | | | | |
> |---|---|---|---|---|---|---|---|---|---|---|---|---|
> | | s_0 | $\xrightarrow{\pi}$ | a_0 | $\rightarrow$ | s_1 | $\xrightarrow{\pi}$ | a_1 | $\rightarrow$ | s_2 | $\xrightarrow{\pi}$ | a_2 | $\rightarrow$ | s_3 | $\xrightarrow{\pi}$ | ... | |
					$\downarrow$					$\downarrow$				$\downarrow$			
					r_1					r_2				r_3		...	
$J(\pi) = \langle$					r_1		$+$			$\gamma \cdot r_2$		$+$		$\gamma^2 \cdot r_3$	$+$	...	$\rangle$
>
> 2. $J(\pi)$ が大きくなる π を探す

■ 内容

❓ どうやって 2 を実行？ → ❗ **今日は二種類**[2]
1. **方策勾配法**
2. **ε-貪欲法**

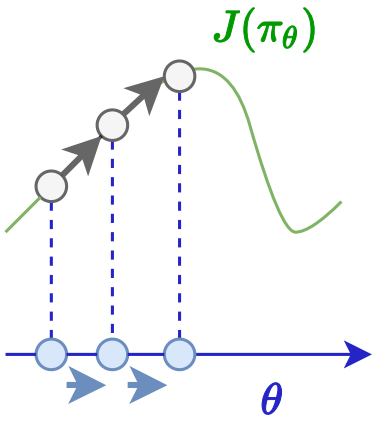
方策勾配法

方策勾配法

基本的なアイデア

方策 $\pi(a|s)$ をパラメータ化: $\{\pi_\theta\}_{\theta\in\mathbb{R}}$♣ して、勾配が大きくなる方向へ更新し続ける

> 1. 適当に初期化 θ_0
> 2. 以下繰り返す $(i = 0, 1, ..., T)$
>
> $$\theta_{i+1} = \theta_i + \eta_i \nabla_{\theta_i} J(\pi_{\theta_i})$$



■ **方針**

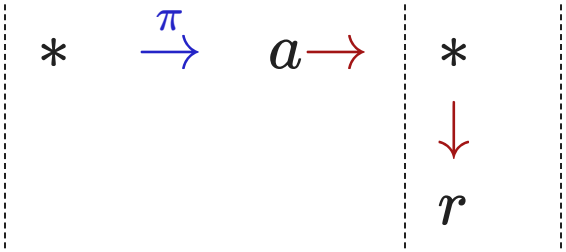
❓ 期待値の勾配：$\nabla_{\theta_i}\langle r_1 + \gamma \cdot r_2 + \gamma^2 \cdot r_3 + \cdots \rangle$ は難しい

↓

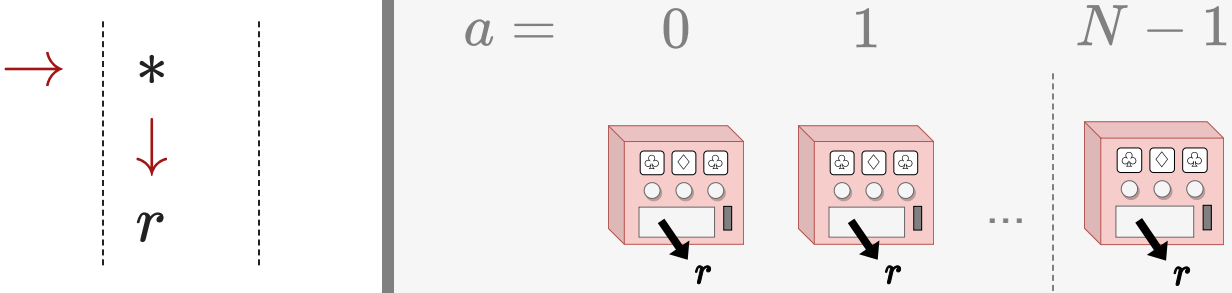
❗ **期待値だけ：$\langle \bullet \rangle$ に書き直せる（方策勾配定理）**

方策勾配定理

■ 1ステップだけのケース

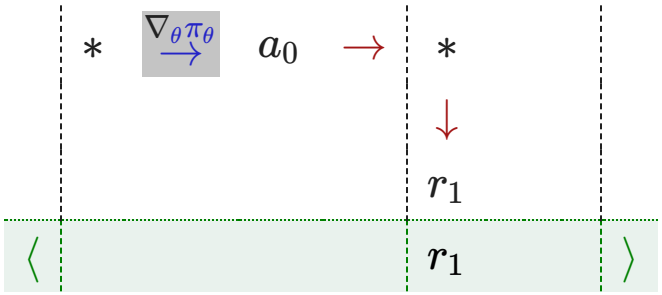


例：当たりやすさの異なるスロットマシン

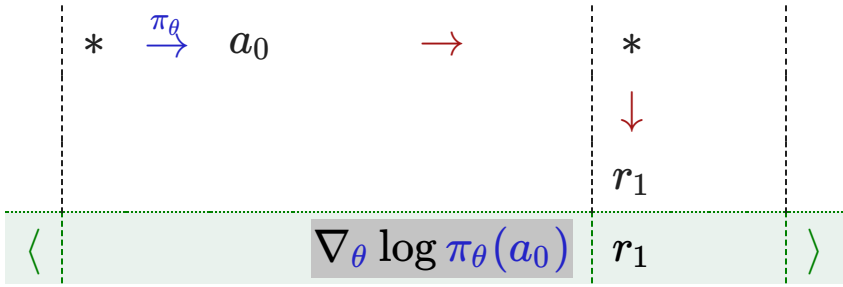


$\nabla\pi = \nabla \log \pi \cdot \pi$ を使うと：

$$\nabla_\theta J(\pi_\theta) =$$



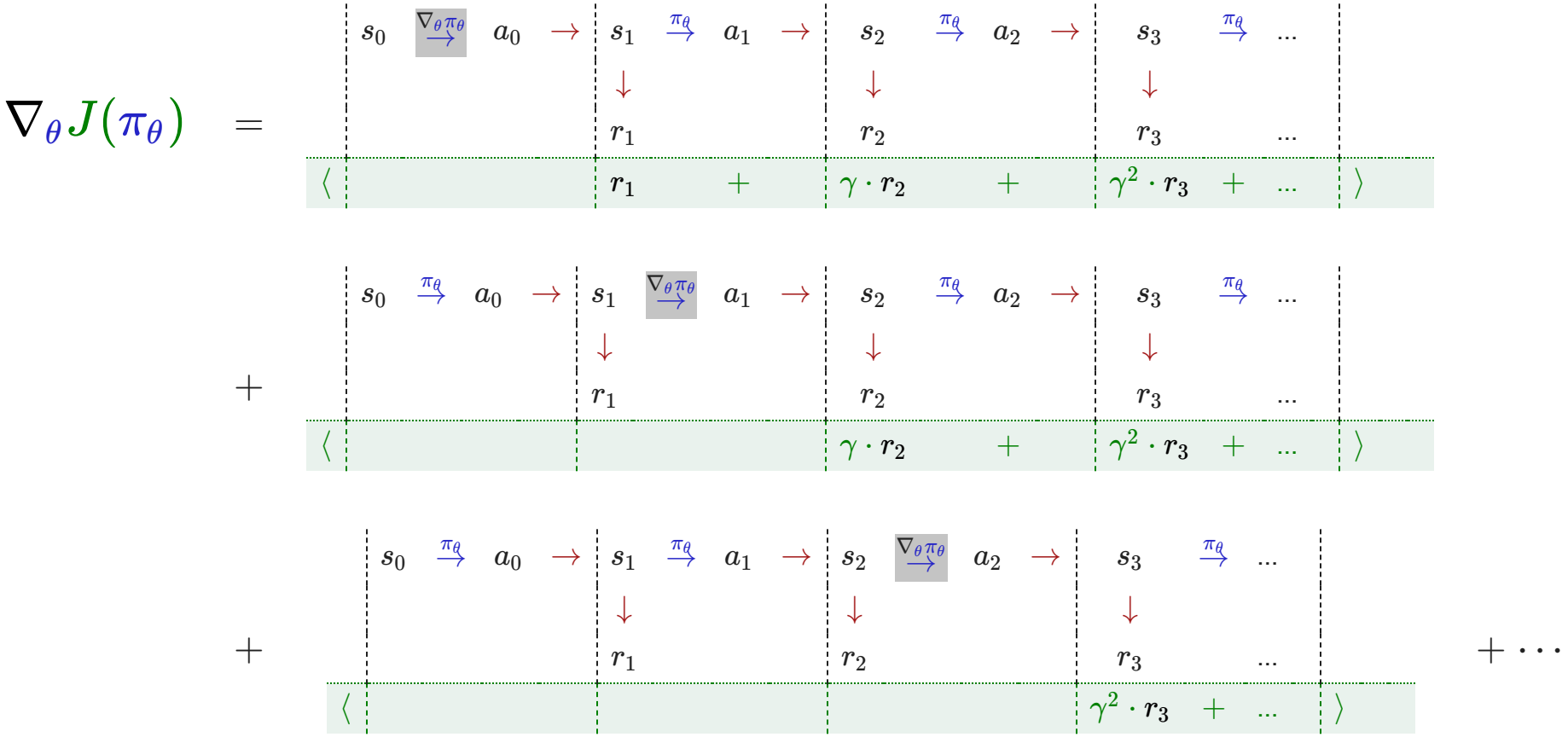
$=$



方策勾配定理

■ ∞ステップのケース

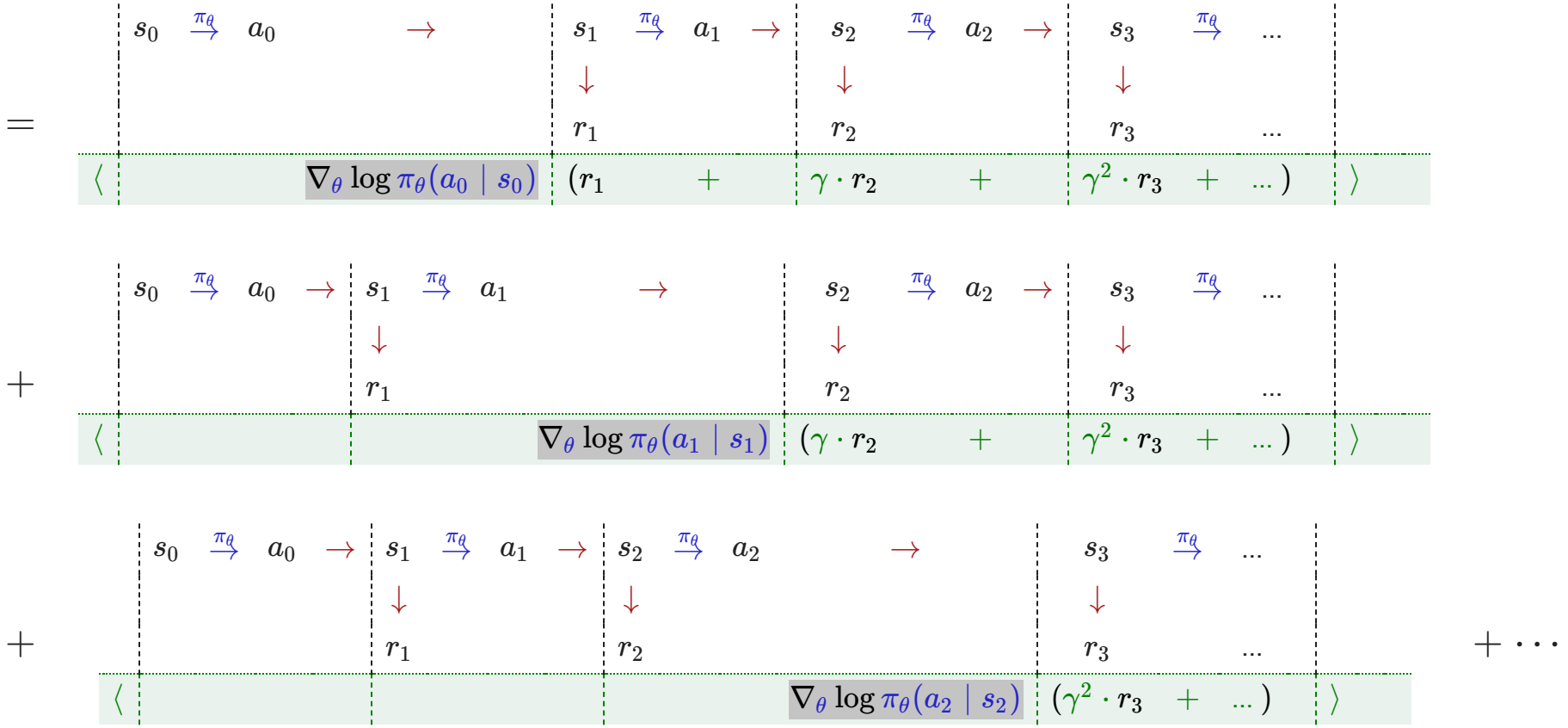
微分の連鎖律と $\nabla\pi = \nabla \log \pi \cdot \pi$ を繰り返し用いているだけ：



方策勾配定理

■ ∞ステップのケース

微分の連鎖律と $\nabla\pi = \nabla \log \pi \cdot \pi$ を繰り返し用いているだけ：



方策勾配定理

■ ∞ステップのケース

$$= \sum_{t=0}^{\infty} \langle \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\gamma^t r_{t+1} + \gamma^{t+1} \cdot r_{t+2} + \gamma^{t+2} \cdot r_{t+3} + \cdots \right) \rangle$$

$$= \sum_{t=0}^{\infty} \langle \nabla_\theta \log \pi_\theta(a_t|s_t) \gamma^t \left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \right) \rangle$$

なのでアルゴリズムは以下のように変形できる

> 1. 適当に初期化 θ_0
> 2. 以下繰り返す $(i = 0, 1, ..., T)$
>
> $$\theta_{i+1} = \theta_i + \eta_i \sum_{t=0}^{\infty} \langle \nabla_{\theta_i} \log \pi_{\theta_i}(a_t|s_t) \gamma^t \left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \right) \rangle$$

モンテカルロ法

期待値 をモンテカルロ法で近似する（**REINFORCEアルゴリズム**[3]）

> 実際に得られた (s_t, a_t, r_{t+1}) を使って
>
> $$\theta_{i+1} = \theta_i + \eta_i \sum_{t=0}^{\infty} \nabla_{\theta_i} \log \pi_{\theta_i}(a_t|s_t) \gamma^t \left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \right)$$

■ 疑似コード

```
class REINFORCE():
  ...
  def reset(self):
    self.history = []

  def step(self, s, a, rn):
    self.history.append((s, a, rn))

  def flush(self):
    for (s, a, rn) in self.history:
      self.update(s, a, self.history)
```

```
opt = REINFORCE(agt)

for episode in range(N_EPISODE):
    env.reset()                          # 環境をリセット
    opt.reset()                          # 履歴をリセット
    while not env.is_terminated():       # 実際に動かす
        s = env.get_state()
        a = agt.play(s)
        rn = env.step(a)
        opt.step(s, a, rn)               # 履歴に記録
    opt.flush()                          # モンテカルロ更新
```

ε貪欲法

基本的なアイデア

$J(\pi_{\text{new}}) \geq J(\pi_{\text{old}})$となる（**方策改善**する）ように新しい方策を決めれば良い

❓ 勾配更新以外で、どのような決め方が方策改善するか？

■ ε貪欲法

確率 $(1-\epsilon)$ で π_{old} での最適行動、確率 ϵ で一様ランダム

$$
s \xrightarrow{\pi_{\text{new}}} a
$$

$$
= \begin{cases} s \to a \in \underset{\tilde{a} \in A}{\arg\max} \Big(\; s,\ \tilde{a} \to s_1 \xrightarrow{\pi_{\text{old}}} a_1 \to s_2 \xrightarrow{\pi_{\text{old}}} a_2 \to \cdots \\ \qquad\qquad\qquad\qquad \downarrow \qquad\qquad\qquad\qquad \downarrow \\ \qquad\qquad\qquad\qquad r_1 \qquad\qquad\qquad\qquad r_2 \qquad\qquad \cdots \\ \qquad\quad \langle \qquad r_1 \qquad + \qquad \gamma \cdot r_2 \qquad + \qquad \cdots \qquad \rangle \; \Big) & \text{確率 } 1-\epsilon \\ s \xrightarrow{\frac{1}{|A|}} a \in A & \text{確率 } \epsilon \end{cases}
$$

方策改善定理

■ ポイントとなる不等式

$$\forall s, a, \quad 0 <^{\exists} \epsilon < 1, \quad s.t. \sum_{a \in A} \left(\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|} \right) f(a) \leq (1-\epsilon) \max_{\tilde{a} \in A} f(\tilde{a})$$

証明：

$$\begin{aligned}
(\text{左辺}) - (\text{右辺}) &= (1-\epsilon) \left\{ \sum_{a \in A} \frac{\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|}}{1-\epsilon} f(a) - \max_{\tilde{a} \in A} f(\tilde{a}) \right\} \\
&= \underbrace{(1-\epsilon)}_{\geq 0} \sum_{a \in A} \underbrace{\frac{\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|}}{1-\epsilon}}_{(\clubsuit)} \underbrace{\left\{ f(a) - \max_{\tilde{a} \in A} f(\tilde{a}) \right\}}_{\leq 0} \\
&\leq 0
\end{aligned}$$

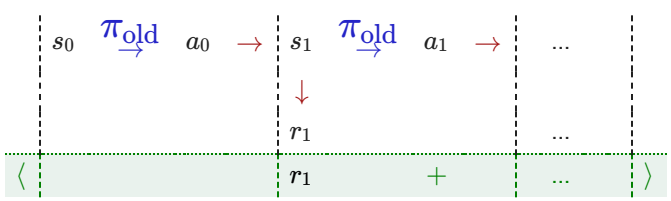
※ ϵ を十分小さくとれば $(\clubsuit) \geq 0$ にできる。

方策改善定理

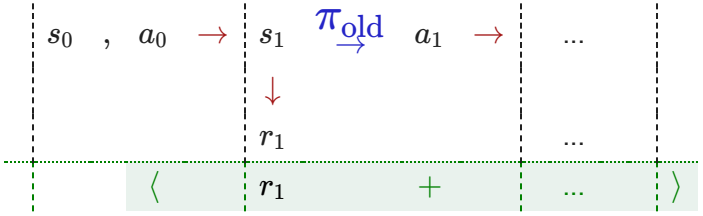
■ ポイントとなる不等式

$$\forall s, a, \quad 0 <^{\exists} \epsilon < 1, \quad s.t. \sum_{a \in A} \left(\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|} \right) f(a) \leq (1-\epsilon) \max_{\tilde{a} \in A} f(\tilde{a})$$

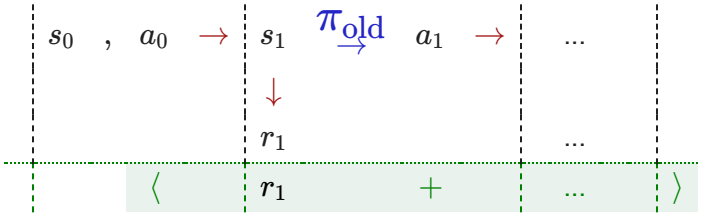
∞ステップの最初に適用すると



$$= \sum_{a_0 \in A} \left(\pi_{\text{old}}(a_0|s_0) - \frac{\epsilon}{|A|} \right)$$



$$+ \quad \frac{\epsilon}{|A|} \sum_{a_0 \in A}$$

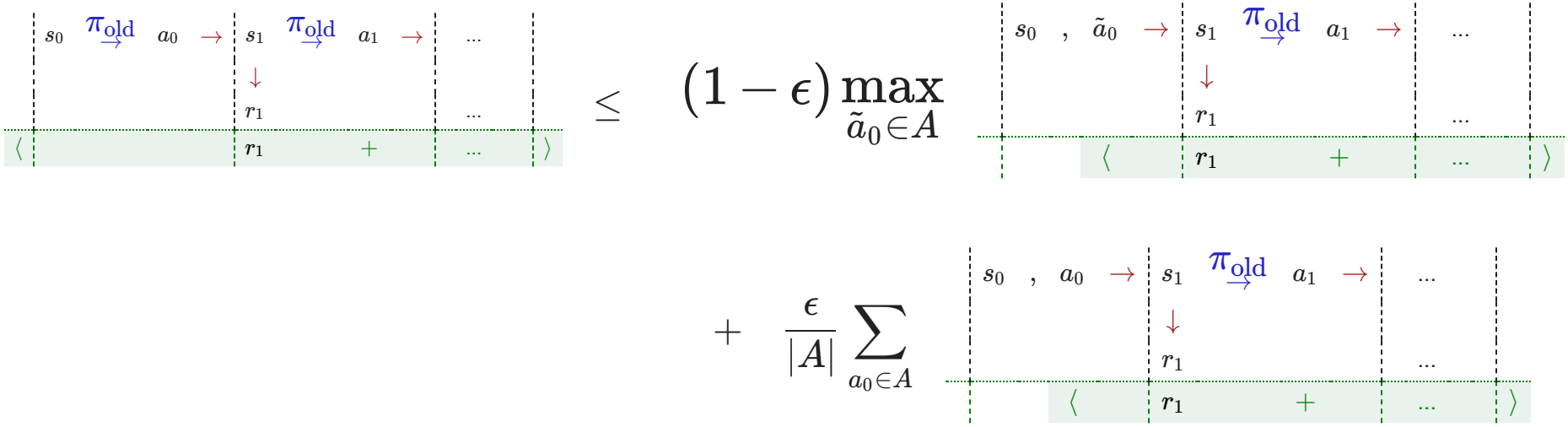


方策改善定理

■ ポイントとなる不等式

$$\forall s, a, \quad 0 <^{\exists} \epsilon < 1, \quad s.t. \sum_{a \in A} \left(\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|} \right) f(a) \leq (1-\epsilon) \max_{\tilde{a} \in A} f(\tilde{a})$$

∞ステップの最初に適用すると

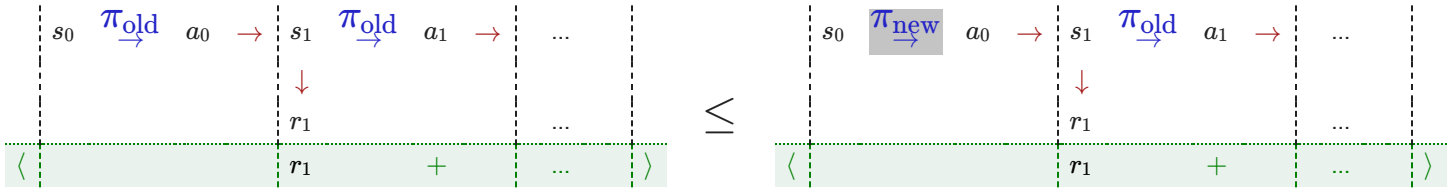


方策改善定理

■ ポイントとなる不等式

$$\forall s, a, \quad 0 <^{\exists} \epsilon < 1, \quad s.t. \sum_{a \in A} \left(\pi_{\text{old}}(a|s) - \frac{\epsilon}{|A|} \right) f(a) \leq (1-\epsilon) \max_{\tilde{a} \in A} f(\tilde{a})$$

∞ステップの最初に適用すると

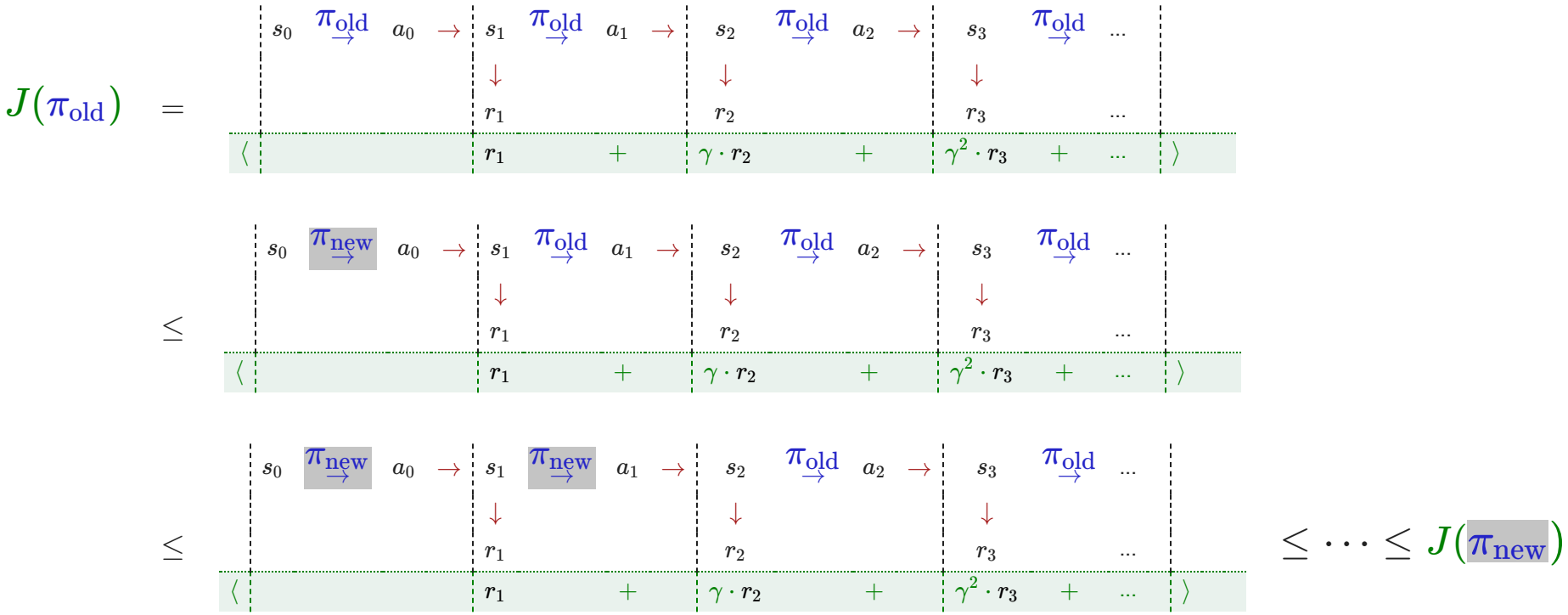


後ろがずっと π_{old} なら、先頭を π_{new} に置き換えて、不等式 $\leq$ で繋げる

方策改善定理

■ 方策改善

$J(\pi_{\text{old}})$ の $t = 0$ から順に不等式を適用していく（$\gamma \geq 0$ に注意）



方策改善定理

■ 方策改善

なので、以下のような更新ルールが考えられる

1. 適当に初期化 π_0
2. 以下繰り返す $(i = 0, 1, ..., T)$

$$s \overset{\pi_{i+1}}{\to} a$$

$$= \begin{cases} s \to a \in \underset{\tilde{a} \in A}{\arg\max} \left(\begin{array}{|cccccccc|} \hline s, \ \tilde{a} & \to & s_1 & \overset{\pi_i}{\to} & a_1 & \to & s_2 & \overset{\pi_i}{\to} \ a_2 \ \to \ ... \\ & & \downarrow & & & & \downarrow & & & \\ & & r_1 & & & & r_2 & & ... & \\ \hline \langle & & r_1 & + & & & \gamma \cdot r_2 & + & ... & \rangle \\ \hline \end{array} \right) & \text{確率 } 1 - \epsilon_i \\ s \overset{\frac{1}{|A|}}{\to} a \in A & \text{確率 } \epsilon_i \end{cases}$$

モンテカルロ法

期待値 をモンテカルロ法で近似する

> 実際に得られた (s_t, a_t, r_{t+1}) を使って
>
> $$s_t \overset{\pi_{i+1}}{\to} a_t = \begin{cases} \arg\max_{\tilde{a}_t \in A} \left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots\right) & \text{確率 } 1-\epsilon_i \\ s_t \overset{\frac{1}{|A|}}{\to} a_t & \text{確率 } \epsilon_i \end{cases}$$

■ 疑似コード

```
class MonteCarlo():
  ...
  def reset(self):
    self.history = []

  def step(self, s, a, rn):
    self.history.append((s, a, rn))

  def flush(self):
    for (s, a, rn) in self.history:
      self.update(s, a, self.history)
```

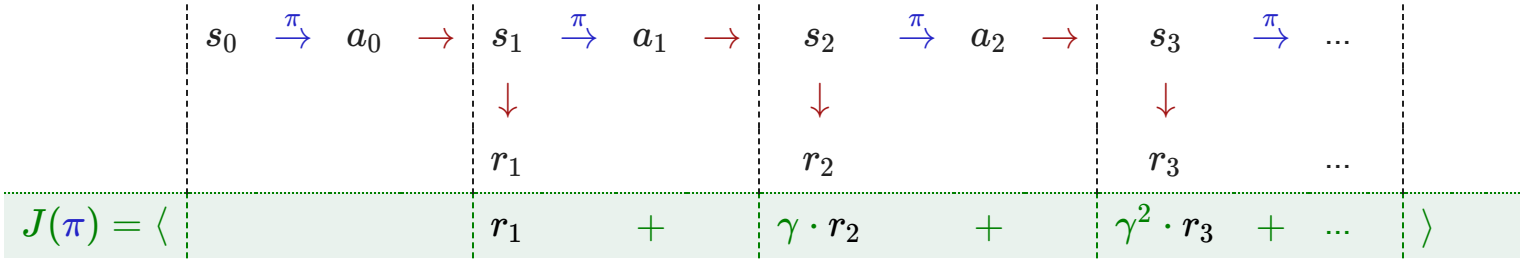
```
opt = MonteCarlo(agt)

for episode in range(N_EPISODE):
    env.reset()                          # 環境をリセット
    opt.reset()                          # 履歴をリセット
    while not env.is_terminated():       # 実際に動かす
        s = env.get_state()
        a = agt.play(s)
        rn = env.step(a)
        opt.step(s, a, rn)               # 履歴に記録
    opt.flush()                          # モンテカルロ更新
```

まとめ

本日のまとめ

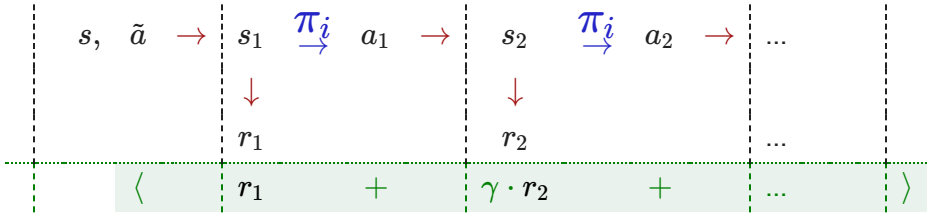
- 強化学習の基礎について
  - 「目的関数」= MDPについての期待値



- 二つのモンテカルロ的アプローチ
  - 方策勾配法（REINFORCE）

$\langle \nabla_{\theta_i} \log \pi_{\theta_i}(a_t|s_t)\gamma^t \left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots \right) \rangle$ をモンテカルロ近似

  - ε貪欲法



をモンテカルロ近似

まとめ

本日触れられなかったこと

■ **on policy / off policy**

- 今回：サンプリングの方策 $=$ 本番で使う方策、**on-policy** という
- サンプリングの方策 $\neq$ 本番で使う方策 でも良い。**off-policy** という

大雑把には、重み再定義と同じ

$$
\begin{aligned}
\langle r(a) \rangle_{\pi_{\text{本番}}(a)} &= \sum_{a \in A} \pi_{\text{本番}}(a) r(a) \\
&= \sum_{a \in A} \pi_{\text{サンプリング}}(a) \frac{\pi_{\text{本番}}(a)}{\pi_{\text{サンプリング}}(a)} r(a) \\
&= \left\langle \frac{\pi_{\text{本番}}(a)}{\pi_{\text{サンプリング}}(a)} r(a) \right\rangle_{\pi_{\text{サンプリング}}(a)}
\end{aligned}
$$

有名なoff-policy：Q学習

まとめ

本日触れられなかったこと

■ Temporal Difference (TD)更新

TD更新：サンプリングしながら同時に π の更新

```
class ActorCritic():
  ...
  def reset(self):
    self.t = 0

  def step(self, s, a, rn, sn):
    self.update(s, a, rn, sn)
    self.t += 1

  def flush(self):
    pass
```

```
opt = ActorCritic(agt)

for episode in range(N_EPISODE):
    env.reset()                          # 環境をリセット
    opt.reset()                          # 内部情報をリセット
    while not env.is_terminated():       # 実際に動かす
        s = env.get_state()
        a = agt.play(s)
        rn = env.step(a)
        opt.step(s, a, rn)               # TD更新
    opt.flush()                          # ダミー
```

有名なTD更新法

名前		説明
Actor-Critic	方策勾配法	方策＝actor、価値＝critic を学習
SARSA	ε貪欲法	行動価値を学習
Q学習	[2:1]	最良の行動価値を学習

まとめ

本日触れられなかったこと

■ モデルフリー / モデルベース

環境 / Environment

- **状態**s で **行動**a をとった時の状態変化の確率

$$P_s(s_{\text{next}}|s,a)$$

- その際の**報酬**の確率

$$P_r(r_{\text{next}}|s,a,s_{\text{next}})$$

エージェント / Agent

- **状態**s で **行動**a を取る確率

$$\pi(a|s) \quad \cdots \textbf{方策}\text{と呼ばれる}$$

- その他の構造
  - Environment (P) のシミュレータ　⬆このモデル
  - 状態評価関数, ...etc

これまで：モデル（環境のシミュレータ）なし：**モデルフリー**

モデル（環境のシミュレータ）あり：**モデルベース**

実践

2次元迷路の環境での実装 https://github.com/AkinoriTanaka-phys/gridworld_rl を作ってみたので、これを用います。

Google colabノートブックはこちら：https://colab.research.google.com/github/AkinoriTanaka-phys/deeplearning_notes/blob/master/appendix/rl_tutorial.ipynb

1. ここに更に **観測可能状態** の集合を加えて、エージェントからは真の状態 s が直接見えないようにする定式化 https://ja.wikipedia.org/wiki/部分観測マルコフ決定過程 もあり、この場合は隠れマルコフ過程に対応します。応用上重要ですが今回は割愛します。 ↩

2. Q学習と呼ばれる手法では、今回説明するような「段々と方策改善する」というよりは「最適な方策が満たすべき条件」に向かって学習が進みます。 ↩ ↩

3. 実際には誤差（分散）を抑えるために$\left(r_{t+1} + \dots\right)$を状態に依存する適当な定数でシフトした$\left(r_{t+1} + \dots\right) - v\left(s_t\right)$に置き換えることが多いです。この置き換えもサンプル数が十分多ければ厳密に元の勾配と一致します。このシフト分を**ベースライン**と呼びます。 ↩