

# Toward Scalable Reinforcement Learning via Massive Batching

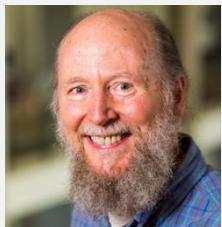
*Sotetsu Koyamada*

# 目次

1. Introduction
2. Massively Vectorized Simulators in Games
3. A Demonstration: Bridge Bidding AI
4. A Potential Disadvantage of Massive Batching
5. Conclusion, Limitations and Future Directions

# **Introduction**

# 強化学習の”父” R. Sutton 曰く



*“The biggest lesson that can be read from 70 years of AI research is that general methods that **leverage computation** are ultimately the most effective, and by a large margin.”*

<https://www.ualberta.ca/en/faculty/2021/05/ai-innovations-made-to-royal-society.html>

— R. Sutton [2019], “The Bitter Lesson.”

“計算機を活用するのが有効”

<http://www.incompleteideas.net/IncIdeas/BitterLesson.html>

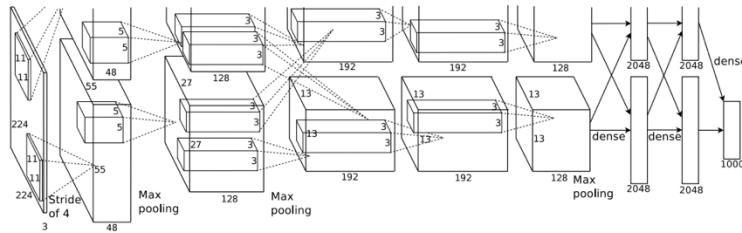
# 計算機活用の具体例

## AlexNet [Krizhevsky et al., 2012]

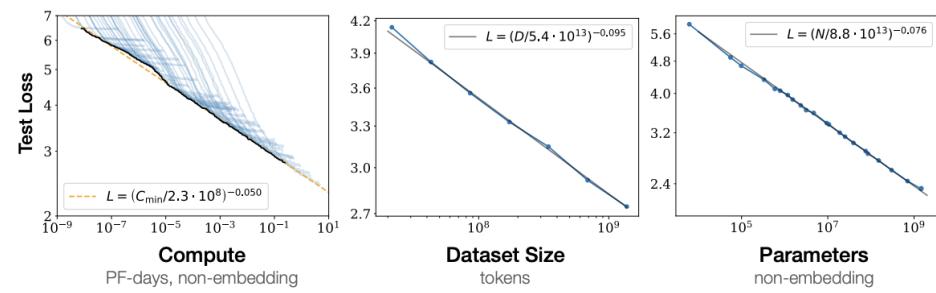
- Krizhevskyらは**GPGPU**を畳み込みニューラルネットワークの学習に活用
- ImageNetコンペティションでSIFT等のヒトが設計した特徴量ベースの手法に大差をつけ優勝

## Transformer [Vaswani et al., 2017]

- Transformerアーキテクチャは系列データの**並列学習**を可能にした
- LLMs (Large Language Models) における *Scaling law* [Kaplan et al., 2020]



From [Krizhevsky et al., 2012]



From [Vaswani et al., 2017]

# Our Focus: 強化学習 (Reinforcement Learning, RL)

## ■ 強化学習 (Reinforcement Learning, RL)

エージェントが環境と相互作用しながら期待累積報酬和を最大化するように方策を学習していく



$$\mathbb{E} \left[ \sum_{t \geq 0} r_t \mid \pi \right]$$

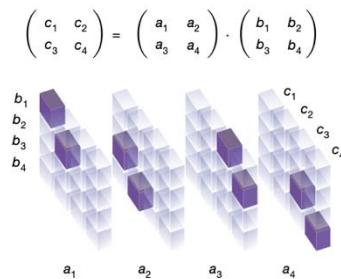
$\pi$

囲碁でトッププロ棋士に勝利



[Silver et al., 2016]

行列分解アルゴリズムの発見



[Mankowitz et al., 2023]

高速なソートアルゴリズムの発見

```
Original
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P // P = A
mov Memory[1] Q // Q = B
mov Memory[2] R // R = C

mov R S
cap P R
cmov P R // R = max(A, C)
cmov P S // S = min(A, C)
mov S P // P = min(A, C)
cmov Q P // P = min(A, B, C)
cmov S Q // Q = max(min(A, C), B)

mov R Memory[0] // R = min(A, B, C)
mov Q Memory[1] // = max(min(A, C), B)
mov R Memory[2] // = max(A, C)
```

```
AlphaDev
Memory[0] = A
Memory[1] = B
Memory[2] = C

mov Memory[0] P // P = A
mov Memory[1] Q // Q = B
mov Memory[2] R // R = C

mov R S
cap P R
cmov P R // R = max(A, C)
cmov P S // S = min(A, C)

cmp S Q
cmov Q P // P = min(A, B)
cmov S Q // Q = max(min(A, C), B)

mov R Memory[0] // R = min(A, B)
mov Q Memory[1] // = max(min(A, C), B)
mov R Memory[2] // = max(A, C)
```

ロボット制御



From <https://research.google/blog/multi-task-robotic-reinforcement-learning-at-scale/>

単なる模倣ではなく、ヒトを超える性能を發揮できる可能性のある学習フレームワーク

# 強化学習の問題点: 必要な学習データ量が膨大

## ■ AlphaZeroの例

[Silver et al., 2018]

20M対局（試合）



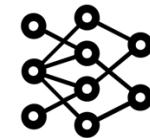
人間換算なら30分/対局として千年間休まず対局する必要がある

**セルフプレイ: TPU 7000 days >> 学習: TPU 90 days**

AlphaZeroは極端な例だが、他の環境・アルゴリズムでもデータをサンプリングするのがボトルネックになるのが一般的（後述）

# Overview: 深層強化学習 (Deep RL)

Deep RL: 方策・価値関数をニューラルネットワークで近似



\* 価値関数: ある状態からの期待累積報酬和を推定

---

## Algorithm 1 High-level overview of PPO [Schulman et al., 2017]

---

- 1: initialize  $N$  vectorized environments, initial policy  $\theta$ , and batch size  $B$ .
  - 2: for iteration = 1, ... do
  - 3:     Generate  $N$  rollouts for  $T$  timesteps from the  $N$  environments using policy  $\theta$ .
  - 4:     Update policy  $\theta$  for  $K$  epochs and for  $\lfloor \frac{NT}{B} \rfloor$  minibatches in each epoch.
- 

For details, see “The 37 Implementation Details of PPO.” [Huang et al., 2022]

(誤解を恐れずに言えば) 以下の繰り返し

- (1) 学習データのサンプリング (by policy and env interactions)
- (2) データを使って方策の改善

**(通常) ボトルネックは (1) データのサンプリング**

≥ 60% in Atari Breakout [Weng et al., 2021], 約75% in Ant [Freeman et al., 2021]

# 強化学習における並列化: MIMD と SIMD による分類

(注) フリンの分類（並列処理におけるコンピューターアーキテクチャの分類）から概念を借用

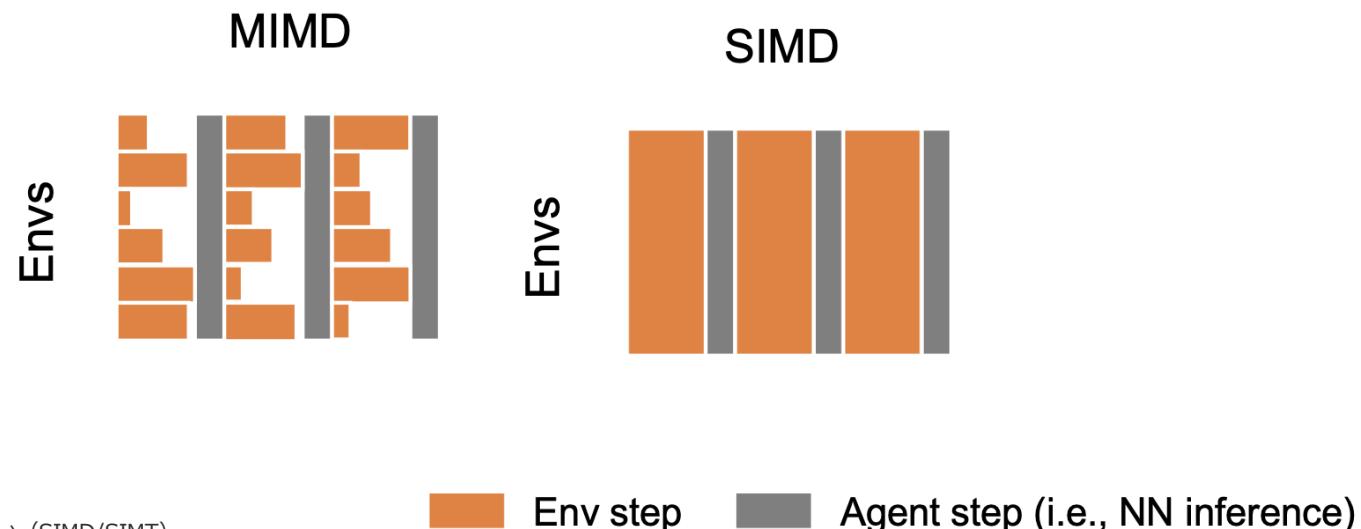
環境遷移 (env step) の並列方法に応じて分類

## MIMD (Multiple Instruction, Multiple Data stream)

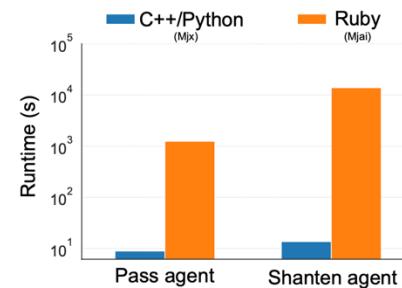
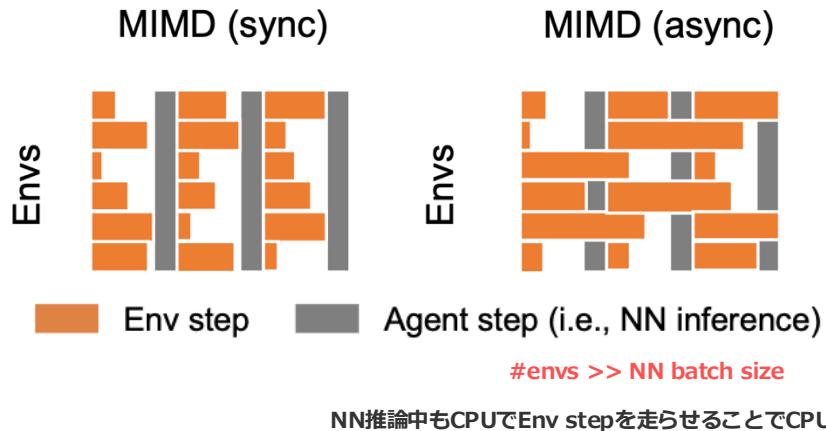
マルチプロセス・マルチスレッドで環境遷移を並列実行

## SIMD (Single Instruction, Multiple Data stream),

環境遷移もGPU/TPU上で実行 (2021-) ここではベクトル化と同義



# MIMD並列での強化学習



麻雀環境の例

コア数による制限から单ースレッド・プロセスでの実行速度が重要 (e.g., C++実装が必要)

**X シミュレータ実装コストが高い・スケールしない**

スケールする高性能な MIMD (async) はしばしば分散学習が必要 (e.g., IMPALA, SEED RL)

**X 分散処理によるアルゴリズム実装コストも高い**

: Env step @ CPU machines (e.g., 数千コア)  
NN推論 @ GPU machine (e.g., A100 x 8)

## 余談

### Microsoft製麻雀AI・Suphx開発体験記

- 実タスク（難易度の高いタスク）でのMIMD並列による強化学習は**非常につらい**
- 小手先の手法改善よりサンプル数を稼げるアーキテクチャで学習するのが効率的と強く実感

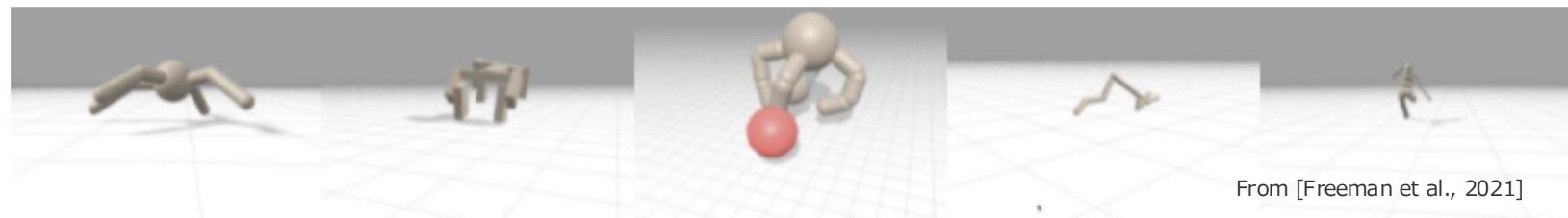
<https://www.microsoft.com/en-us/research/project/suphx-mastering-mahjong-with-deep-reinforcement-learning/>

[J. Li, S. Koyamada, et al., 2020]

The screenshot shows the Microsoft Research website for the Suphx Mahjong AI project. The main content area features a green banner with the text "Suphx: The World Best Mahjong AI". To the right is a logo for "SUPHX" featuring a stylized flame or hand icon. Below the logo is a map of Mahjong tiles with the text "世界最強 麻雀AI Suphx の衝撃" and "お知らせ". At the bottom, there is a note: "麻雀(=不完全情報ゲーム)の真理に最強AI「Suphx」(スーパーフラッシュ)が迫る" followed by a small logo.

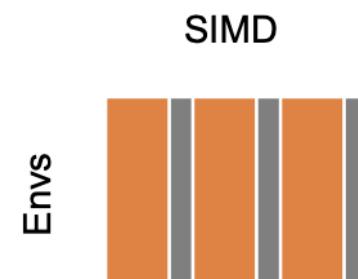
# SIMD並列での強化学習

2021～ ベクトル化が比較的容易な連続状態行動空間・古典制御で成功  
(IsaacGym/Brax/gymnax)



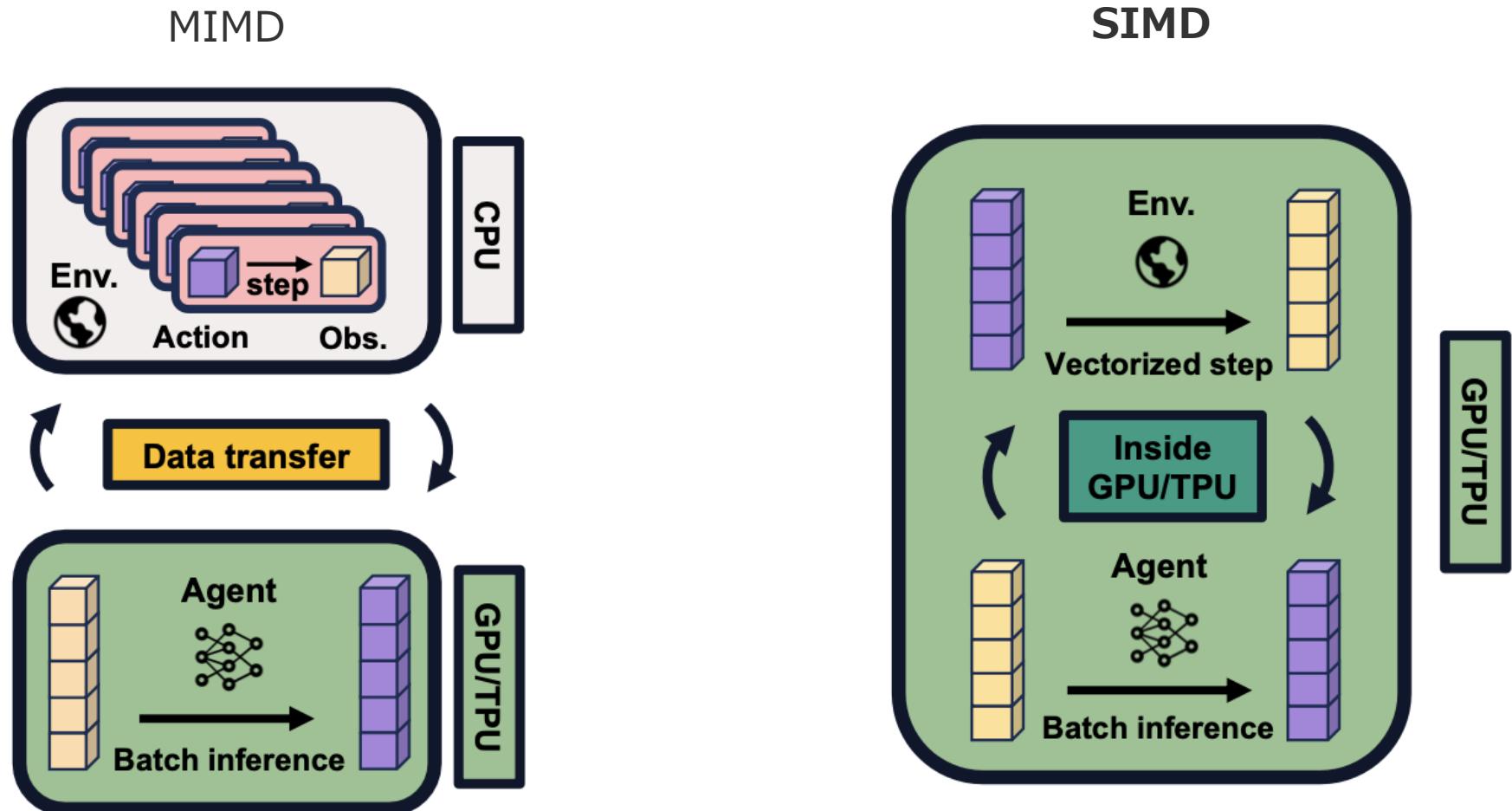
物理モデルを数式で書き下すことによって並列計算を行列演算として実行

- ✓ GPU上でスケール（数千バッチサイズ）
- ✓ デバイス間のデータ転送コストがない
- ✓ アルゴリズム実装も容易（分散処理不要）



後述する自動ベクトル化によって、バッチ次元を意識せずに簡潔にプログラムをかける [Hessel et al., 21]

# 深層強化学習アーキテクチャ



[Makoviychuk et al., 2021, Freeman et al., 2021, Hessel et al., 2021]

- ✓ デバイス間 (CPU  $\leftrightarrow$  GPU) でのデータ転送コストがない
- ✓ 環境遷移 (env step) がベクトル化され、GPU上でスケールする

# **Massively Vectorized Simulators in Games**

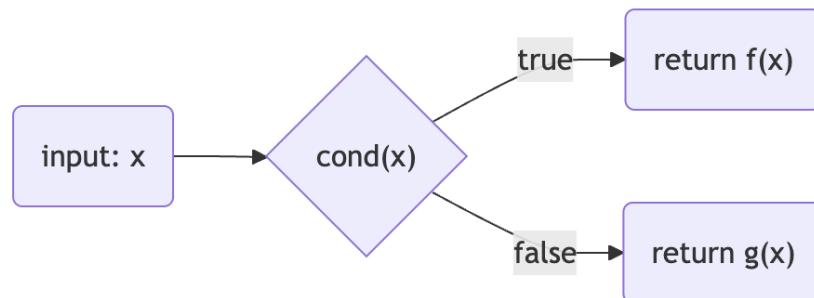
# SIMD並列における制約

SIMD並列をするための制約: 動的に計算グラフが変わってはいけない

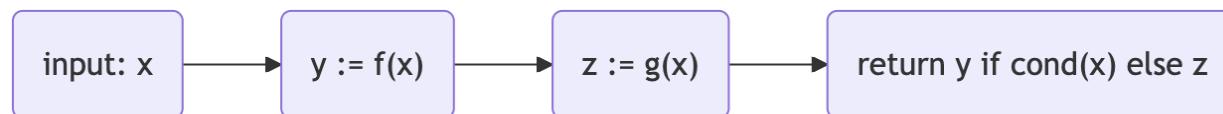
(動的だとsingle instructionで操作できない)

If の例

## MIMD (Multiple Instruction, Multiple Data stream)



## SIMD (Single Instruction, Multiple Data stream)

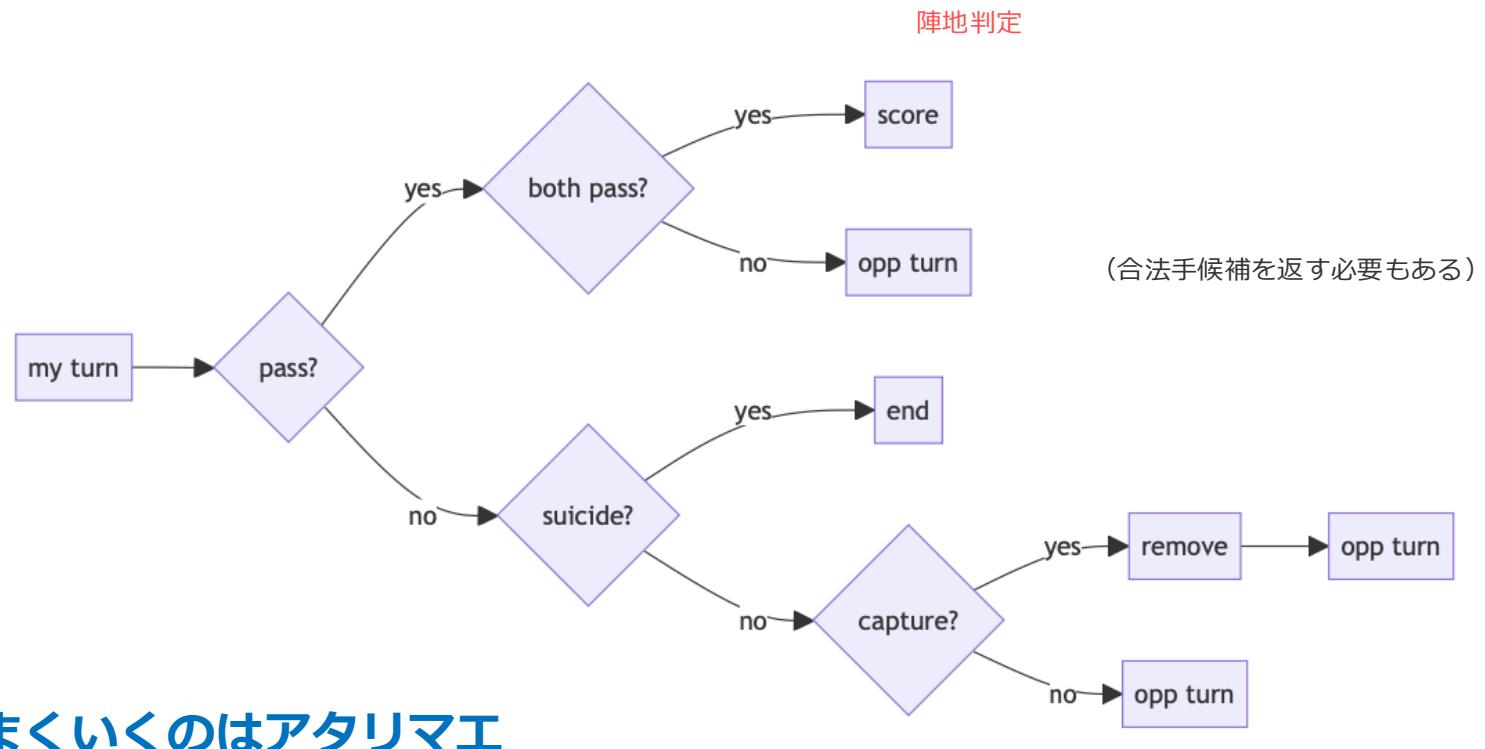
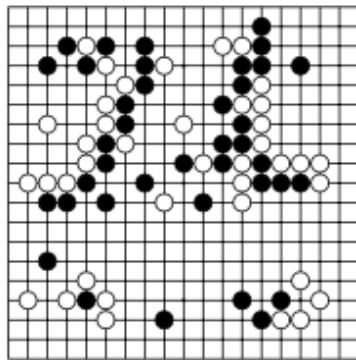


# SIMD並列の問題点

連続状態行動空間・古典制御などでうまくいっているが…

条件分岐・動的な制御が多いヘテロな環境に向き

囲碁の例

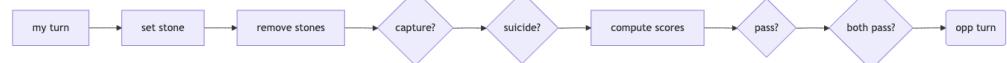


向いている環境でうまくいくのはアタリマエ  
不向きな環境でうまくいくか？構成的に検証

陣地判定

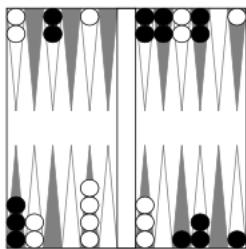
(合法手候補を返す必要もある)

囲まれているか判定

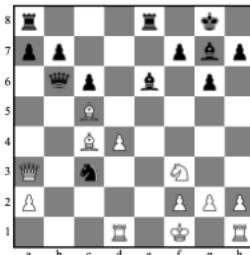


# ドメイン：（古典的な）ゲーム環境

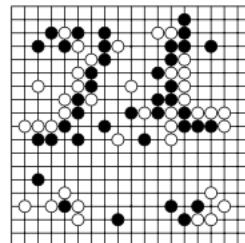
Backgammon



Chess



Go



Shogi



## AI研究の重要なマイルストーン

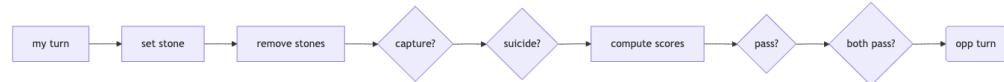
- ゲームは知能 (intelligence) を測るベンチマークの一つとして捉えられている
- 例えば、「藤井聰太は“頭が良い”」という主張は多くの人に受け入れられている

Backgammon (TD-Gammon), Chess (Deep Blue), Atari (DQN), Go (AlphaGo), Go, Shogi (AlphaZero), Poker (Libratus, Pluribus), Mahjong (Suphx), StarCraft 2 (AlphaStar) …

多くの条件分岐・動的な制御

一見するとベクトル化に不向き

# どうSIMD並列によって実装するか？



1. SIMD並列でプログラムが正しく動くか？
2. SIMD並列で効率的に実行が可能か？

容易にベクトル化できる制約の範囲で実装する

任意のプログラム

ベクトル化が容易なプログラム

ベクトル化したときに効率的なプログラム

# ベクトル化が容易なプログラム

## すべて純粋関数 (pure function) として実装をする

任意のプログラム

ベクトル化が容易なプログラム

純粋関数

ベクトル化したときに効率的なプログラム

### 純粋関数 (pure function)

純粋関数入力に対して常に同じ出力を返し、外部の状態に依存せず、副作用（例えば、グローバル変数の変更やI/O操作）がない関数。同じ入力に対して異なる結果が生じることがない。

効率的に並列処理を行うためには、個々の計算が相互に干渉しないことが重要。純粋関数は、同じ入力に対して常に同じ結果を返し、外部状態に依存しないため、並列化による競合や同期の問題を避けることができる。

### NG例

- グローバル変数の更新
- elseのないif
- 早期リターン
- ブロック外を変更するループ
- ...

### 純粋関数はJAXで容易にベクトル化可能



```
import jax
import pgx

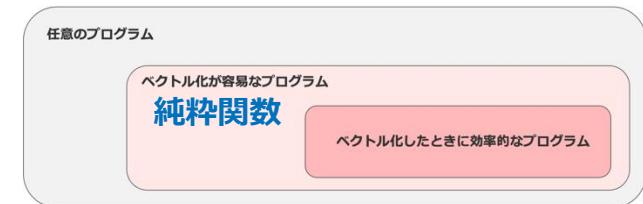
env = pgx.make("go_19x19")
...
# state : vectorized (e.g., 1024 different Go boards)
# action: vectorized (e.g., 1024d vector)
state = jax.vmap(env.step)(state, action)
```

自動ベクトル化 純粋関数

# ベクトル化したときに効率的なプログラム

純粋関数の形で書けばどんな処理でも高速に動くというわけではない。以下は検証において有用だった原則（優先順位の高い順）

（余談） チェスは3回スクラッチで作り直した



## 1. 数学的なトリックの利用

[囲碁] 呼吸点判定にコーラー・シュワルツの不等式を用いて探索を避けて盤面全体の演算を行なった

## 2. 多少冗長にしてでも行列演算に変換する

[囲碁] ある地点の周辺だけで終わる可能性が高い探索よりも、盤面全体で計算

[チェス・将棋] 各位置から駒の動きを行列で表現

## 3. 静的な情報は行列か Lookup table (LUT) へ事前計算しておいて利用する

[チェス・将棋] 駒の動きは各64 (81) マスと駒種類について事前に計算可能

(行列で持ちにくくても静的なLUTが利用できる)

## 4. 逐次的なloop処理が独立ならばベクトル化して実行する

[チェス] 例えは合法手生成は、各行動について、駒を実際に動かしてみてチェックされないか確認する必要がある。これらは独立なのでベクトル化して実行する

# 【例】逐次的なloop処理が独立ならばベクトル化して実行する

チェスにおける実際のボトルネック箇所（ゲームAIでは合法手を学習時に使うのが一般的・なおチェスではスタイルメイトのため合法手生成は必須）

loop

```
def generate_legal_actions(state):

    def not_in_check(action): 実際に一手動いてみてチェックされてないか確認（自殺手・王手放置）
        state_copy = apply_move(state, action)
        king_pos = find_king(state_copy)
        return not under_attack(state_copy, king_pos)

    legal_action_candidates = ... # clearly illegal actions have been removed
    is_legal_move = []
    for action in legal_action_candidates:
        is_legal_move.append(not_in_check(action))
    ...


```

w/ vmap

```
def generate_legal_actions(state):

    def not_in_check(action):
        state_copy = apply_move(state, action)
        king_pos = find_king(state_copy)
        return not under_attack(state_copy, king_pos)

    legal_action_candidates = ... # clearly illegal actions have been removed
    is_legal_move = jax.vmap(not_in_check)(legal_action_candidates)
    ...

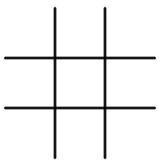
```

自動ベクトル化

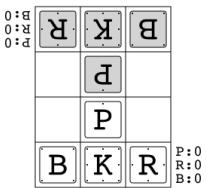
独立なのでベクトル化して計算

Env Name	# Players	Obs. shape	# Actions	Tag
2048	1	$4 \times 4 \times 31$	4	perfect info. (w/ chance)
Animal shogi	2	$4 \times 3 \times 194$	132	perfect info.
Backgammon	2	34	156	perfect info. (w/ chance)
Bridge bidding	4	480	38	imperfect info.
Chess	2	$8 \times 8 \times 119$	4672	perfect info.
Connect Four	2	$6 \times 7 \times 2$	7	perfect info.
Gardner chess	2	$5 \times 5 \times 115$	1225	perfect info.
Go 9x9	2	$9 \times 9 \times 17$	82	perfect info.
Go 19x19	2	$19 \times 19 \times 17$	362	perfect info.
Hex	2	$11 \times 11 \times 4$	122	perfect info.
Kuhn poker	2	7	4	imperfect info.
Leduc hold'em	2	34	3	imperfect info.
MinAtar Asterix	1	$10 \times 10 \times 4$	5	Atari-like
MinAtar Breakout	1	$10 \times 10 \times 4$	3	Atari-like
MinAtar Freeway	1	$10 \times 10 \times 7$	3	Atari-like
MinAtar Seaquest	1	$10 \times 10 \times 10$	6	Atari-like
MinAtar Space Invaders	1	$10 \times 10 \times 6$	4	Atari-like
Othello	2	$8 \times 8 \times 2$	65	perfect info.
Shogi	2	$9 \times 9 \times 119$	2187	perfect info.
Sparrow mahjong	3	11 × 15	11	imperfect info.
Tic-tac-toe	2	$3 \times 3 \times 2$	9	perfect info.

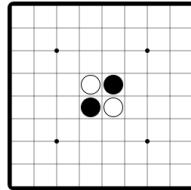
Tic-tac-toe



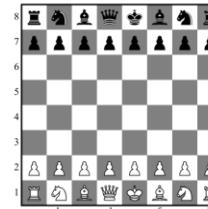
Animal shogi



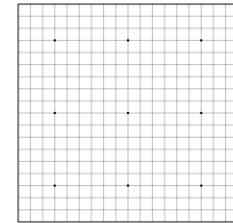
Othello



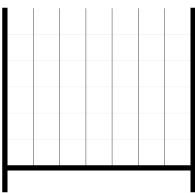
Chess



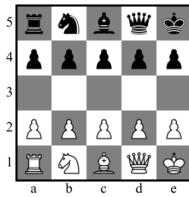
Go (9x9, 19x19)



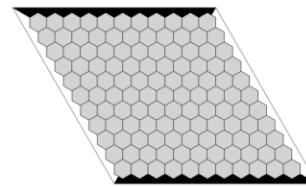
Connect four



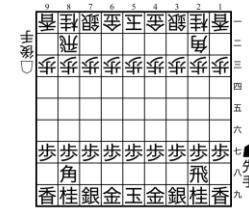
Gardner chess



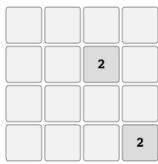
Hex



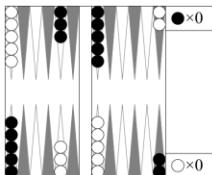
Shogi



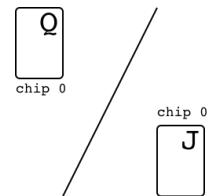
2048



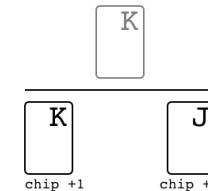
Backgammon



Kuhn poker



Leduc Hold'em



Bridge bidding

North	Vul.
♦ K J 2	
♦ A-J 10 8 6	
♦ 2	
♦ A J 8 2	

West	Vul.
♦ 10 9	
♦ 5 4 2	
♦ A 3 8 6 4	
♦ 10 7 4	

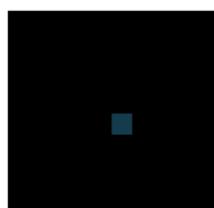
  

East	Vul.
♦ Q 8 7	
♦ K 9 7	
♦ X 10 9 5	
♦ K 6 5	

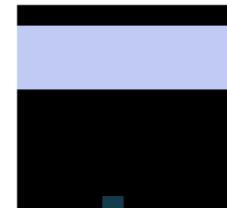
  

South(Dealer)	Vul.
♦ A 6 5 4 3	
♦ Q 3	
♦ O 7 3	
♦ Q 9 3	

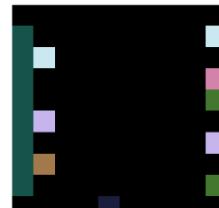
Asterix



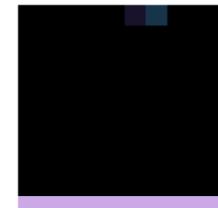
Breakout



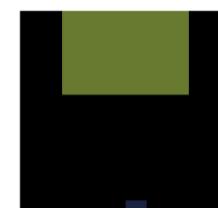
Freeway



Seaquest



Space Invaders



# スループットの比較

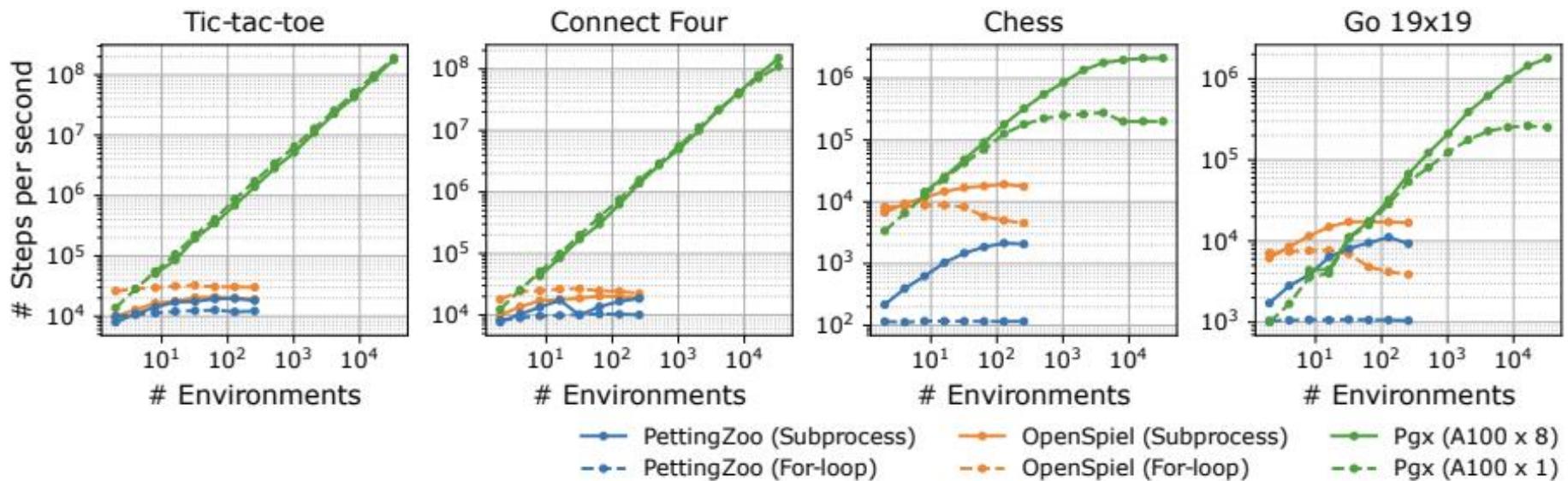
競合ライブラリ (PettingZoo/OpenSpiel) の並列化方法

(ライブラリから公式の並列化方法は提供されてない)

- For-loop: (dummy) 実際にはシーケンシャルに実行
- Subprocess: Python の multi-processing モジュールを使って実装

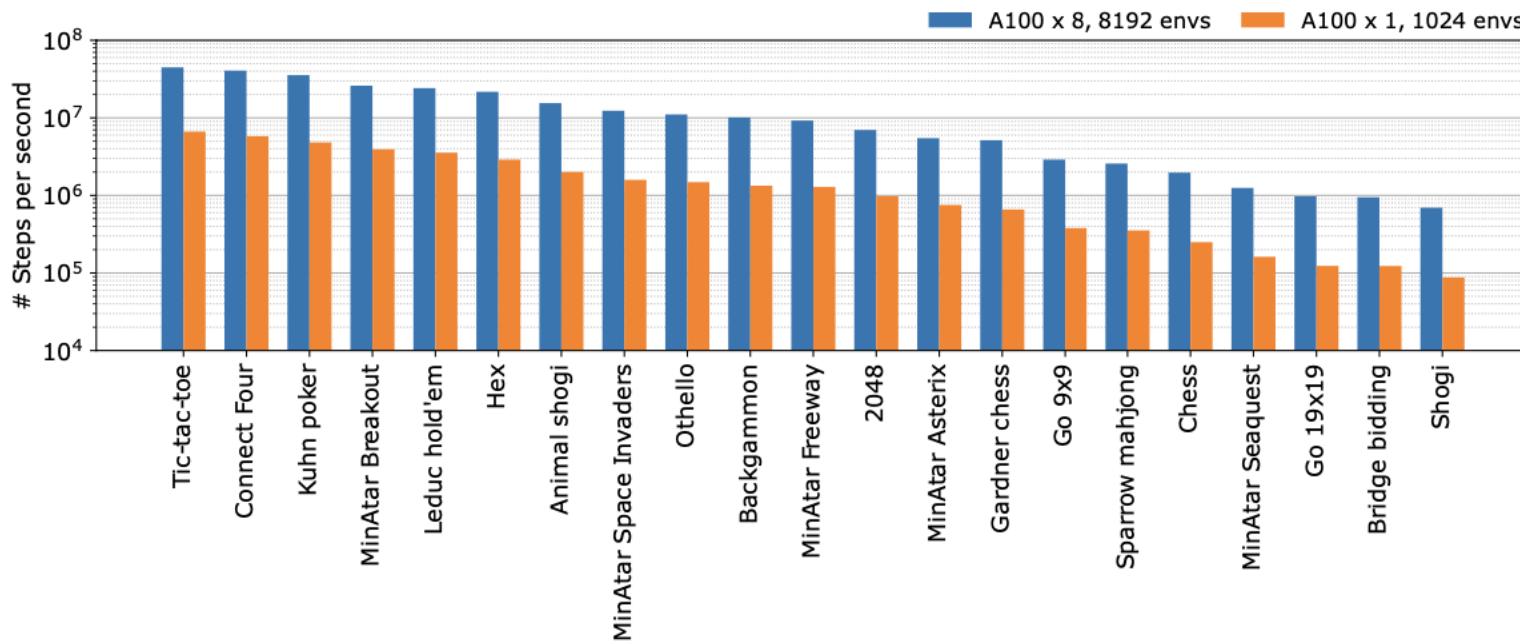
(定評のある実装TianShouからのフォーク)

@ DGX A100 workstation



十分大きいバッチサイズで 10x ~ 100x のスループット改善

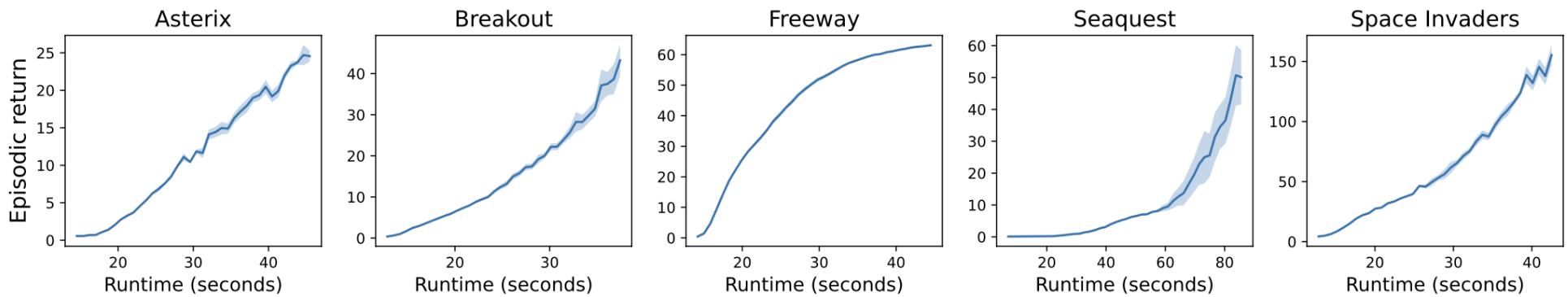
# すべての環境が妥当に高速か？（スループットは実装依存）



すべての環境で妥当な速度・スケーラビティ

# デモ: PPO @MinAtar [Young&Tian, 2019]

Batch size = 4096, A100 x 1



50-90秒以内で学習完了（性能は先行研究と比べて遜色ないレベル）

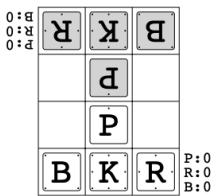
数年前まで12hかかっている研究も [Obando-Ceron et al., 2021]

注: [Lange et al., 2021] もGPU上で動作する MinAtar 実装しているが、すべての環境を実装できていない

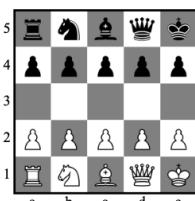
# デモ: Gumbel AlphaZero [Danihelka et al., 2022]

Batch size = 1024, A100 x 1

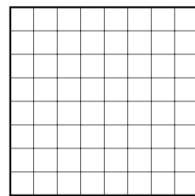
Animal shogi



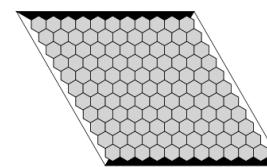
Gardner chess



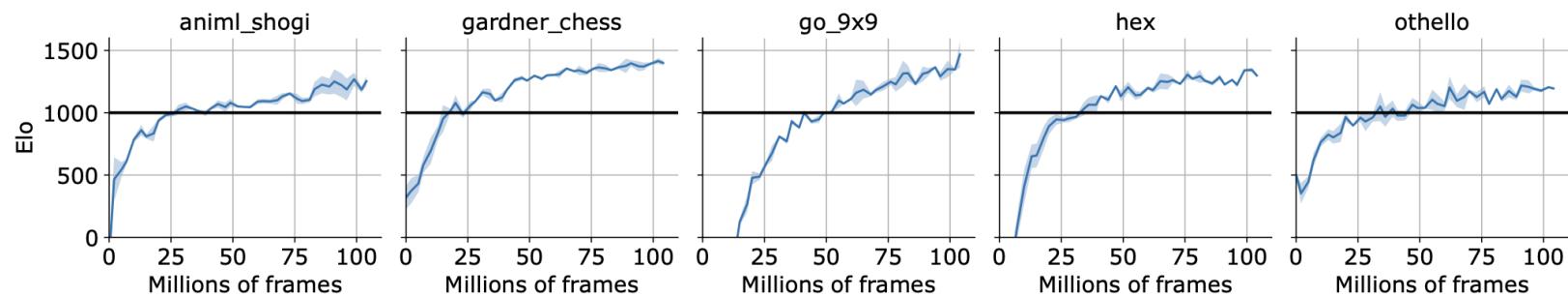
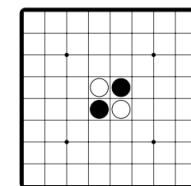
Go 9x9



Hex



Othello

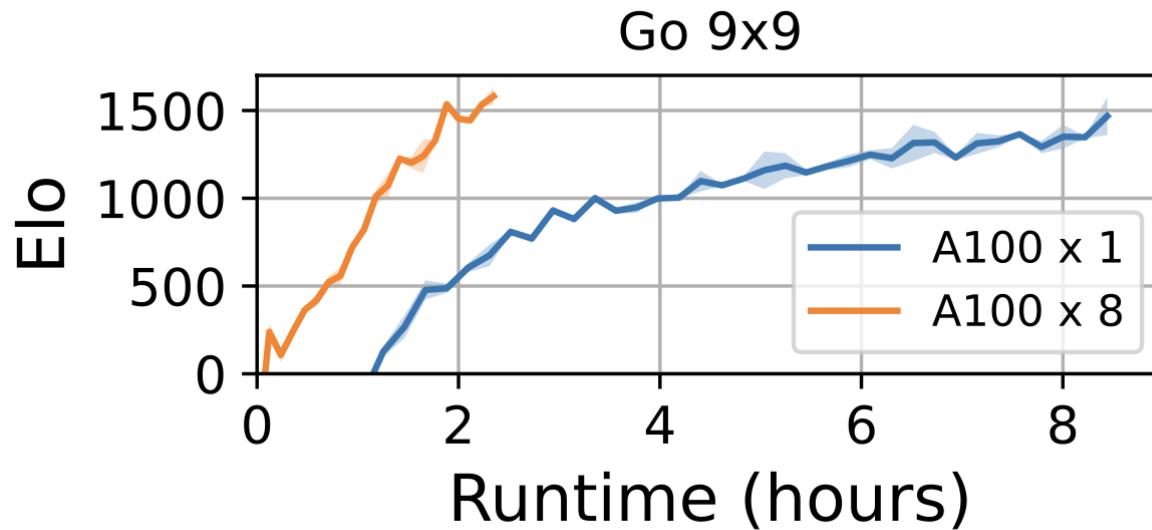


囲碁 (19x19) や チェスの実験はそれでもコスト大

より小さいが十分な難易度のタスク・ベースラインモデルを提供  
Google DeepMindのようなビッグテック以外でも研究を可能に

Go 9x9 で Elo 1000 のモデルはPachi (w/ 10K sim) [Baudiš and Gailly, 2012] より強い (Baseline in [Danihelka et al., 2022])

# 複数GPUでの学習も容易



GPUを増やし、バッチサイズを大きくするだけで4x高速化

注: TPUは4096TPUコアまで利用可能

# 関連研究: GPU/TPU上で動作する強化学習環境

Library	Environments	arXiv sub.	
Brax [Freeman et al., 2021]	Continuous control	2021/6/24	NeurIPS 21
Gymnax [Lange, 2022]	Classic control, bsuite, MinAtar		
Pgx (ours)	Board Games	2023/05/29	NeurIPS 23
Jumanji [Bonnet et al., 2024]	Combinatorial optimization	2023/06/16	ICLR 24
Waymax [Gulino et al., 2023]	Autonomous driving	2023/10/12	NeurIPS 23
JaxMARL [Rutherford et al., 2024]	Multi-agent RL	2023/11/16	NeurIPS 24 (to appear)
XLand-MiniGrid [Nikulin et al., 2023]	Meta RL, Mini-grid	2023/12/19	NeurIPS 24 (to appear)
Craftax [Matthews et al., 2024]	Open-ended RL	2024/02/26	ICML 24
TORAX [Citrin et al., 2024]	Tokamak Transport	2024/06/10	
NAVIX [Pignatelli et al., 2024]	Mini-grid	2024/07/28	

2024年における強化学習のベストプラクティス

あなたの解きたい問題がシミュレータを作れるドメインであれば、  
GPU上で動作するシミュレータを実装して強化学習する（数倍以上の高速化）

(既にあればそれを使う)

# A Demonstration: Bridge Bidding AI

# 先行研究の再現だけ？

## Bridge Bidding AI

- マルチエージェント不完全情報ゲーム
- 欧米で人気のあるゲーム（“グランドスラム”の由来）
- Microsoft, Google DeepMind, Metaから先行研究有

### コントラクトブリッジ

ブリッジは4人で行う不完全情報カードゲームで、2人ずつのチームに分かれる。各プレイヤーには、52枚のカードデッキから13枚の手札が配られる。

**入札フェーズ:** このオークション形式の段階では、プレイヤーは自分のチームが何回の“トリック”（各プレイヤーから1枚ずつカードが出されるセット）を取れるかを予測し、手札の強さや可能性をパートナーに伝えるために入札を行なう。また、“トランプ”となる絵札（他の絵札よりも強いカードとして使われる）を選択します。入札を行って“コントラクト”を設定し、チームが取ることを目指すトリックの数や最終的な契約を決めたプレイヤーである“ディクレアラー”を決定する。

**プレイフェーズ:** プレイヤーは順番に1枚ずつカードを出し、最初に出された絵札とトランプの中で最も“強い”カードがそのトリックを取る。このプロセスを13回繰り返す。チームの得点は、コントラクトを満たすか、それを上回るトリックを取ることで決まるが、達成できなかった場合にはペナルティが課される。

効果的なコミュニケーションと戦略が重要であり、プレイヤーは入札を通じてパートナーに手札の可能性を伝え、勝利するためのコントラクトを形成する必要がある。

North		Vul.
♠	K J 2	
♥	A J 10 8 6	
♦	2	
♣	A J 8 2	

West		Vul.
♠	10 9	
♥	5 4 2	
♦	A J 8 6 4	
♣	10 7 4	

S (D)	W	N	E	Vul.

East		Vul.
♠	Q 8 7	
♥	K 9 7	
♦	K 10 9 5	
♣	K 6 5	

South (Dealer)		Vul.
♠	A 6 5 4 3	
♥	Q 3	
♦	Q 7 3	
♣	Q 9 3	



## ■ Supervised Learning (SL)

- WBridge5 (SAYC System) からのデータで学習  
(後に評価に使う WBridge5 エージェントとは別のビデオシステム)

## ■ Reinforcement Learning (RL) with self-play

- PPO [Schulman et al., 2017]
- with Fictitious Self-Play (**FSP**; [Heinrich et al., 2015])

\* Fictitious Self-Play: 過去のモデルチェックポイントから uniform に対戦相手をサンプリング

# 結果

- $10^4$  PPO update steps
- 対戦相手: **WBridge5** (世界コンピュータブリッジ選手権優勝)  
[2005, 2007, 2008, and 2016-2018]

Paper	IMPs/board ( $\pm$ SE)	# games
Rong et al. [2019]	+0.25 ( $\pm$ N/A)	64
Gong et al. [2019]	+0.41 ( $\pm$ 0.27)	64
Tian et al. [2020]	+0.63 ( $\pm$ 0.22)	1K
Lockhart et al. [2020]	+0.85 ( $\pm$ 0.05)	10K
Qiu et al. [2024]	+0.98 ( $\pm$ 0.05)	10K (concurrent work)
Ours	<b>+1.24</b> ( $\pm$ 0.19)	1K

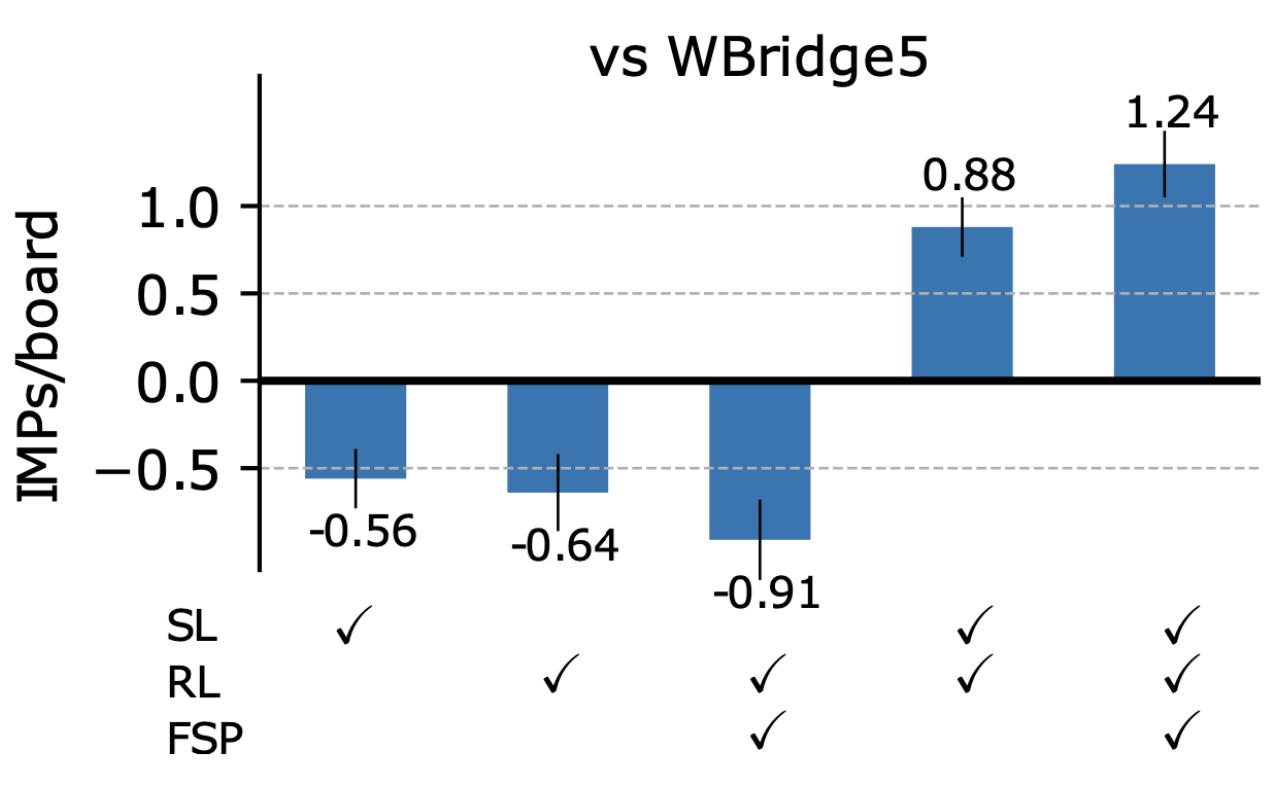
**Pgxを使って Batch size = 8192, A100 × 8 で1時間以内で学習可能**

注: 図の性能はGPU1台で学習したモデルのもの

余談

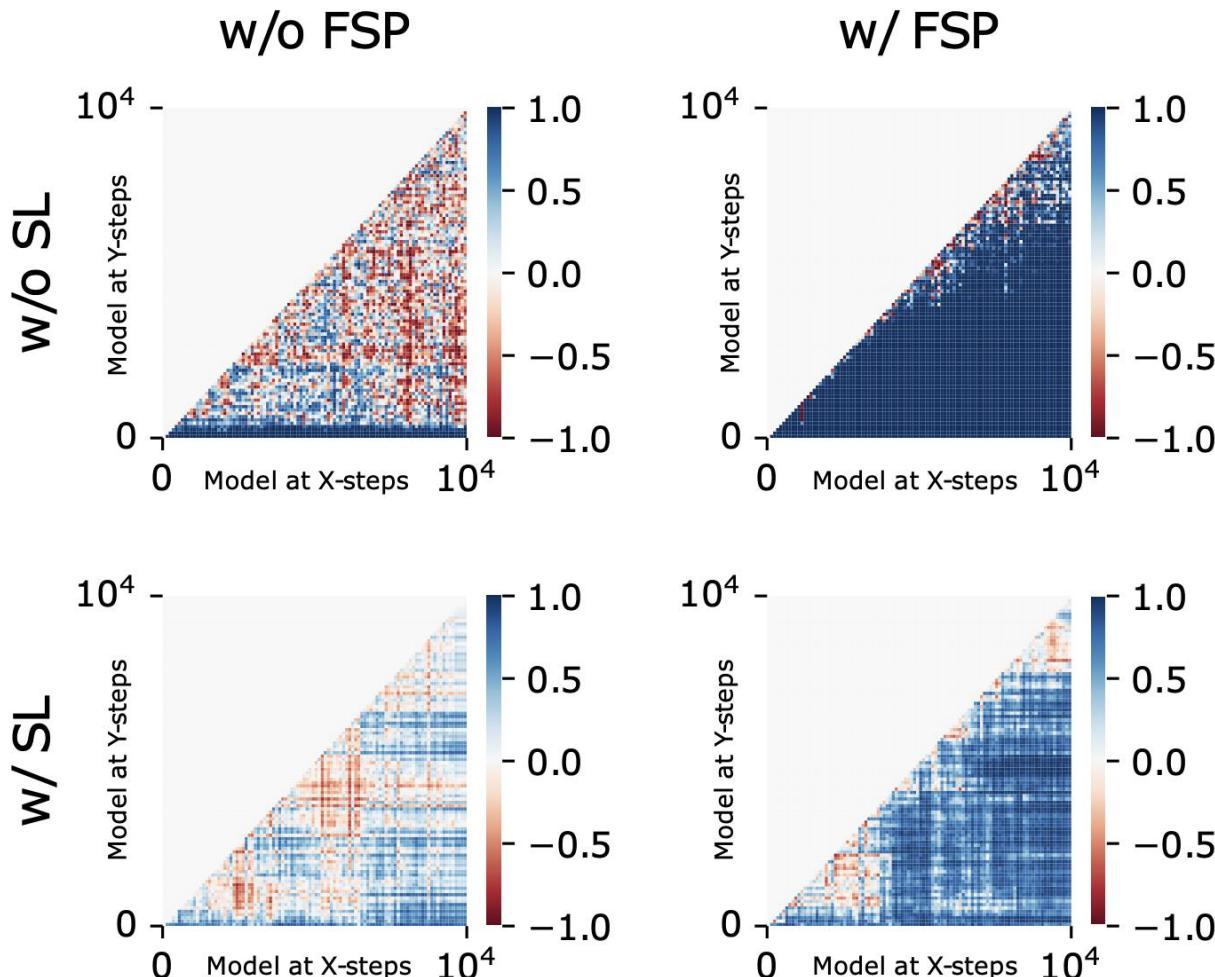
京大石井研では5年以上前からブリッジAIの研究に着手していたが、WBridge5に勝つことがなかなかできていなかった。Pgxを使うことでこれまで一週間単位でかかっていた実験が数時間で終わるようになり、大幅に性能が上がった。

# Ablation study



SL, RL, FSPすべての要素が性能改善に寄与

# FSPはどのように影響を与えるか？



各要素はモデルXのモデルYに対する性能指標 (IMPs/board) をtanhでスケールしたもの

**FSPがないと特定のモデルにだけ勝ち越すことを学習する傾向**

# **A Potential Disadvantage of Massive Batching**

# 大規模なバッチ（ベクトル化された環境）での学習は良いことだけ？

## ■ メリット: 高い計算効率

GPU/TPUアクセラレータの活用で高速に計算可能

## ■ デメリット: Delayed feedback & Reduced adaptability

逐次的な場合と比べて、報酬の観測・方策の更新の間隔が空いてしまう

(極端な) 例

注: 合計ステップ数は共に100K

- (A) **100K** 回逐次: 方策を細やかに100K回更新
- (B) バッチサイズ **5K** で **20** 回バッチ: 20回しか方策を更新できない

**(A) の方が高い性能を示す可能性が高い**

# 確率的バンディットにおける純探索問題 (pure exploration)

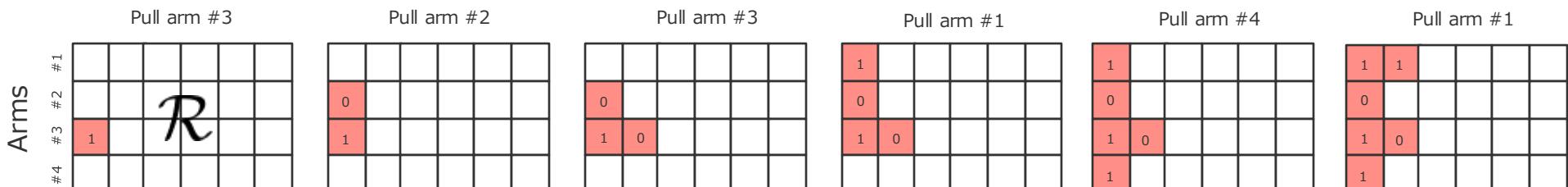
(最後にモンテカルロ木探索への応用を考える)

- この問題について、最も単純化された問題設定であるバンディット問題で考える
- 純探索問題 (pure exploration) にフォーカス (RLではプランニングに応用例有)

## 純探索問題

- アーム本数:  $n$ . 各アームの報酬の期待値:  $1 \geq \mu_1 \geq \mu_2 \geq \dots \geq \mu_n \geq 0$
- 予算:  $T$ .  $T$  回アームを引いた後で最適だと思うアーム  $a_T$  を1本選ぶ
- 報酬行列  $\mathcal{R} \in \{0, 1\}^{n \times T}$  各要素  $\mathcal{R}_{i,j} \in \{0, 1\}$  は  $i$  番目のアームを  $j$  回目に引いたときの報酬  
(アーム毎に独立にカウント)
- アルゴリズム:  $\pi : \{0, 1\}^{n \times T} \rightarrow [n]$   
( $[n] := \{1, \dots, n\}$ )
- 評価指標: シンプルリグレット

$$\mathbb{E}_{\mathcal{R}}[\mu_1 - \mu_{a_T}]$$



$$n = 4, T = 6$$

# SH: Sequential Halving [Karnin et al., 2013]

---

## Algorithm 1 SH: Sequential Halving (Karnin et al., 2013)

---

```
1: input number of arms:  $n$ , budget:  $T$ 
2: initialize best arm candidates  $\mathcal{S}_0 := [n]$  n本すべてのアームで候補集合を初期化
3: for round  $r = 0, \dots, \lceil \log_2 n \rceil - 1$  do
4:   pull each arm  $a \in \mathcal{S}_r$  for  $J_r = \left\lfloor \frac{T}{|\mathcal{S}_r| \lceil \log_2 n \rceil} \right\rfloor$  times 残っているアームを同じ回数引く
5:    $\mathcal{S}_{r+1} \leftarrow$  top- $\lceil |\mathcal{S}_r|/2 \rceil$  arms in  $\mathcal{S}_r$  w.r.t. the empirical rewards 下位半分のアーム取り除く
6: return the only arm in  $\mathcal{S}_{\lceil \log_2 n \rceil}$  約  $\log_2 n$  ラウンド繰り返すことで最後に1本残る
```

---

- ✓ シンプル
- ✓ タスク依存のハイパーパラメータがない
- ✓ 効率的 (シンプルリグレットが対数因子を除いて最適 [Zhao et al., 2023])  $\tilde{O}(\sqrt{n/T})$

### 応用例

- ハイパーパラメータの探索 [Jamieson&Talwalkar, 2016]
- State-of-the-art AlphaZero/MuZero family [Danihelka et al., 2022]

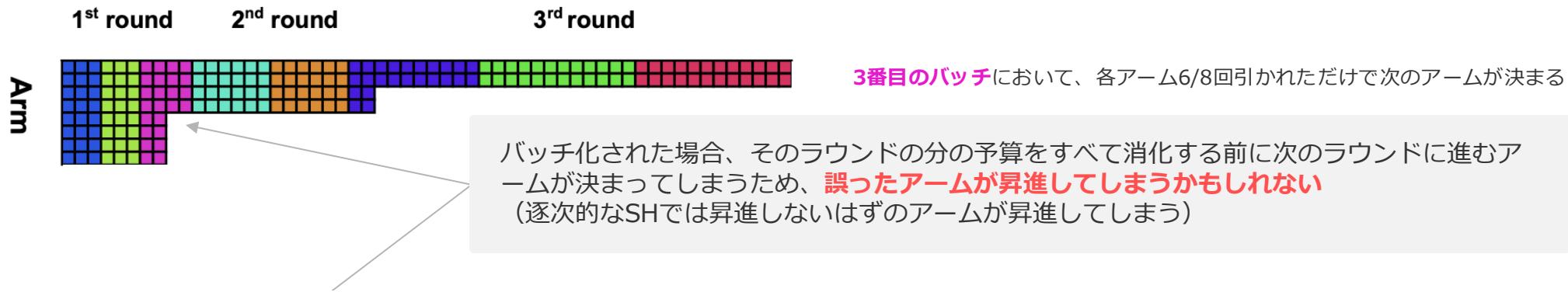
# SHをバッチで実行するとどうなる？

2種類のバッチ化方法を考える

## BSH (Breadth-first SH)

引かれる回数がなるべく均一になるように選択

注: BSH は [Jun et al., 2016] で提案されたバッチ版SHに類似



## ASH (Advance-first SH)

次のラウンドに進むアームの決定 (昇進) をなるべく遅らせる



図において、同じ色は一つのバッチ (batch size = 24) に対応します  
例えば、最初の 青いバッチ において、BSHは8本を3回ずつ、ASHは3本を8回ずつ引きます

# SHとASHの等価性: 誤ったアームの昇進は起こらない (条件付)

**Theorem 1** Given a stochastic bandit problem with  $n \geq 2$  arms, let  $b \geq 2$  be the batch size and  $B$  be the batch budget satisfying  $B \geq \max\{4, n\} \lceil \log_2 n \rceil$ . Then, the ASH algorithm is algorithmically equivalent to the SH algorithm with the same total budget  $T = b \times B$  — the mapping  $\pi_{\text{ASH}}$  is identical to  $\pi_{\text{SH}}$ .

同じ予算において ( $T = b \times B$ ) バッチ予算  $B$  が条件

$$B \geq \max\{4, \frac{n}{b}\} \lceil \log_2 n \rceil$$

バッチ予算  $B$  が極端に小さくない限り

を満たす限り SHとASHで選ばれるアームは同じ

例

$n = 32$  のとき

- (A) **100K** 回逐次 ( $T=100K$ )
- (B) バッチサイズ **5K** で **20** 回バッチ ( $B=20, b=5K$ )

注: 合計ステップ数は共に100K

条件を満たすので完全に同じアームを選べる

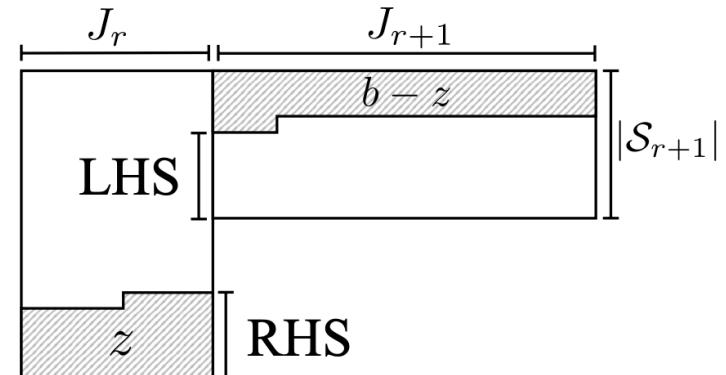
# 証明の概略

バッチがラウンドをまたぐ場合に誤ったアームの昇進が起こらないことを示す

サイズ  $b$  のバッチを  $z$  と  $b - z$  に分割する 任意の  $z$  について次を示す

$$|S_{r+1}| - \left\lceil \frac{b-z}{J_{r+1}} \right\rceil \geq \left\lceil \frac{z}{J_r} \right\rceil$$

$\underbrace{\hspace{10em}}$  バッチを引いたあとに残る次のラウンド  
に進めるアームの余剰本数       $\underbrace{\hspace{10em}}$  バッチを引く時点で引き終  
わっていないアームの本数

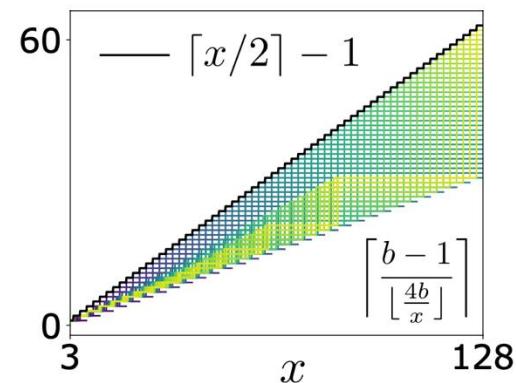


**仮に現時点で引き終わってないアームが  
全て昇進する必要があつても問題ない**

$J_r$  ラウンド  $r$  で各アームが引かれる回数  
 $S_r$  ラウンド  $r$  で生き残っているアーム集合

式を変形していくと下記を示せばよくなる（右図のように成立）

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{[4b/x]} \right\rceil$$



# 条件 $B \geq \max\{4, \frac{n}{b}\} \lceil \log_2 n \rceil$ についての議論

$$(C1) \quad B \geq \frac{n}{b} \lceil \log_2 n \rceil$$

SH でもアルゴリズム実行に必要

(各アーム一度は引かれる必要がある)

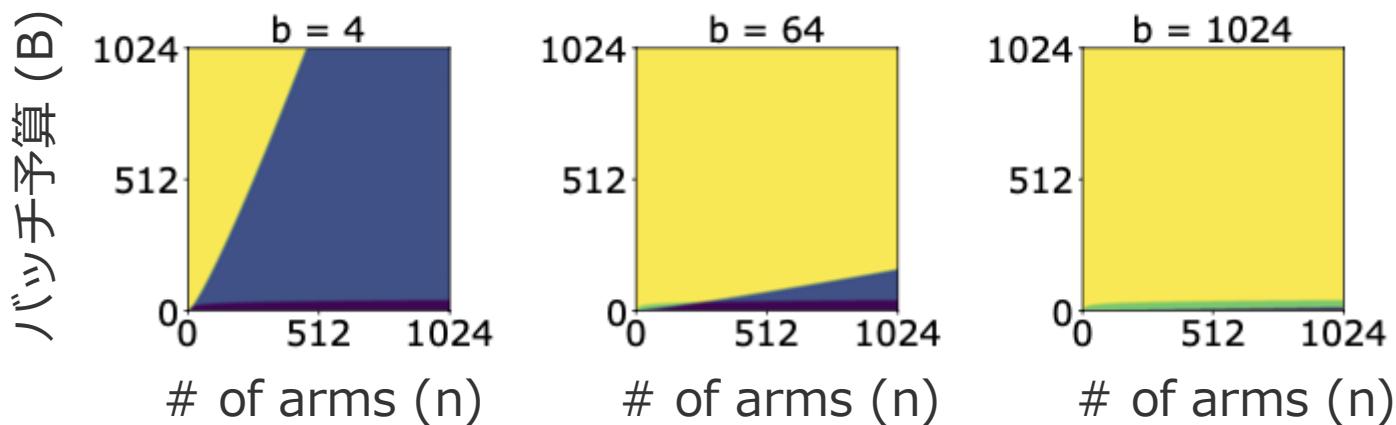
かつ

$$(C2) \quad B \geq 4 \lceil \log_2 n \rceil$$

SH と ASH の一致性のために必要

注: 条件C2はタイト

条件C1が支配的なので、追加の条件C2は問題ではない



■ Both (C1) and (C2) hold (i.e., ASH is equivalent to SH).

■ Only (C1) holds (i.e., SH is executable but ASH may not be equivalent to SH).

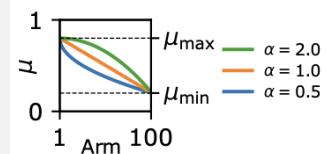
■ Only (C2) holds (i.e., SH is not executable).

■ Neither (C1) nor (C2) holds.

# 実験による検証（人工データ）

## 実験設定

- 10K個の人工バンディット問題
- 各バンディット問題は次の4つ組で特徴付けられる:  $(n, \alpha, \mu_{\min}, \mu_{\max})$
- 報酬の分布は  $\Delta_a := \mu_1 - \mu_a$  が  $\Delta_a \propto (n/a)^\alpha$  に従う [Zhao et al., 2023]
- 各インスタンスに対して、100個の疑似乱数を生成し、SHとASHをそれぞれ適用した

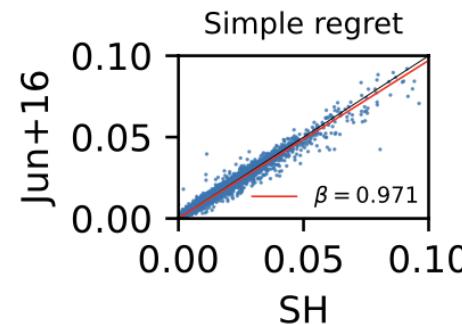
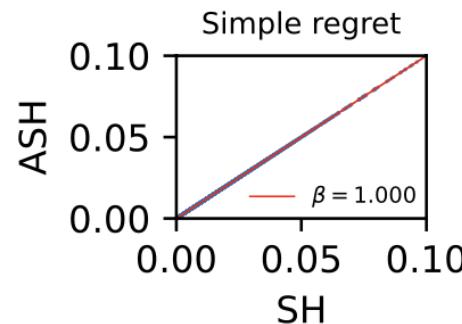
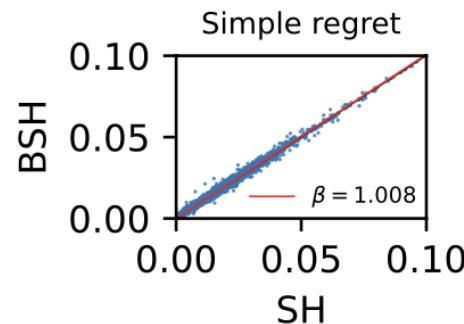


条件が成立しているとき 10K個のバンディット問題と100個の疑似乱数すべてで SHとASHで選ばれたアームは一致

# 実験による検証（人工データ）

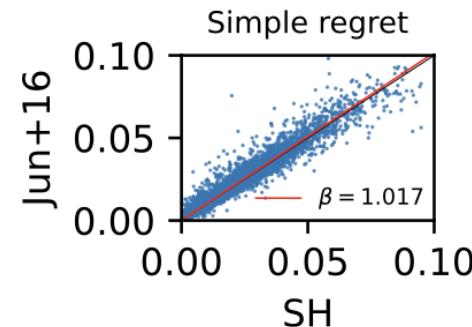
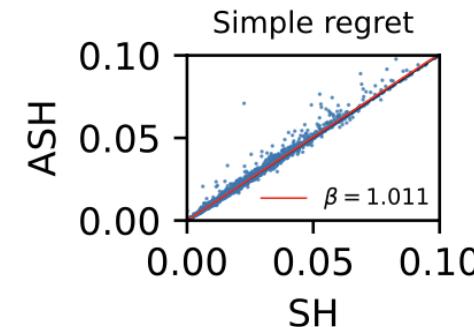
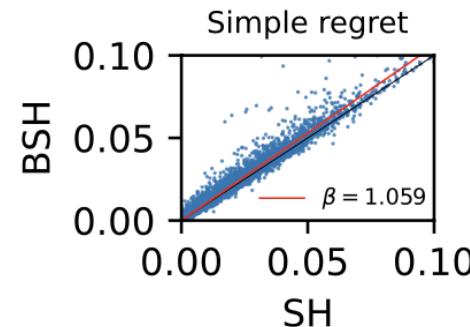
## ■ バッチ予算が大きいとき: $B \geq 4[\log_2 n]$

$\beta$ : 最小二乗法で推定された傾き



\* Jun+16: BSHに似たバッチ化の仕方

## ■ バッチ予算が小さいとき: $B < 4[\log_2 n]$

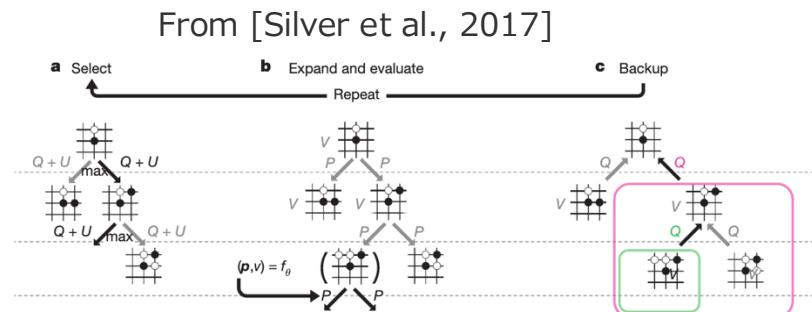
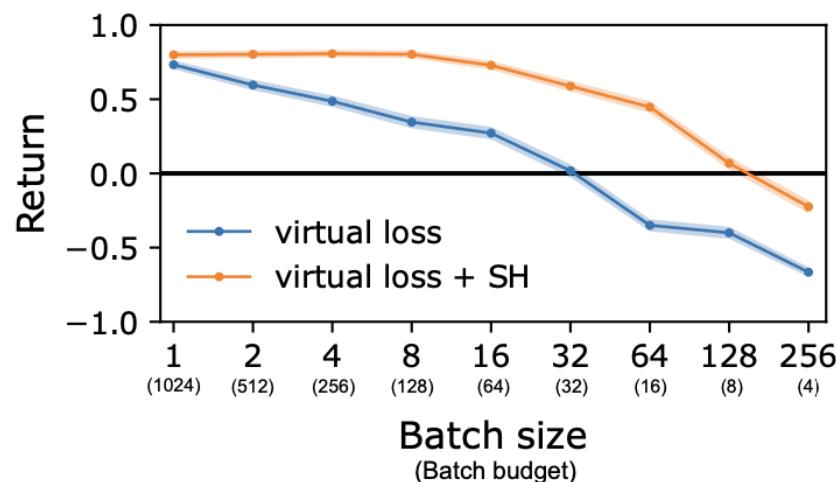


実験的にはバッチ化の仕方によらず  $B = \log_2 n$  程度まではほぼ変わらない

# モンテカルロ木探索 (MCTS) のバッチ化への応用

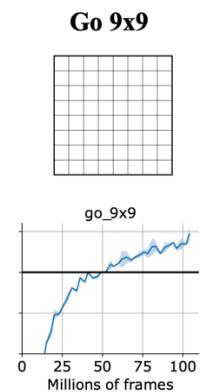
【想定】プロ棋士と対戦する際には時間制限がある。  
NN推論スループットから何回NN推論できるか決まる。  
限られたNN推論回数の中でできるだけ良い行動選択をしたい

#sims (予算) = 1024 で固定



MCTSはノードが一つづつ増えていく  
バッチでまとめて増やしたい

9x9 Go  
vs 100 sim



100回逐次と同等の性能を  
8回バッチで達成可能

Virtual loss [Chaslot et al., 2008]

既存の最もポピュラーな並列化（バッチ化）手法

Virtual loss + SH

ルートノードは代わりにバッチ化したSH

(注) MCTSでは報酬の分布がi.i.d.ではないので理論的な保証はない

# **Conclusion, Limitations, and Future Directions**

## Conclusion

GPU上で動くベクトル化されたRL環境の有効性を（古典的）ゲームで示した

- スループットの大幅な改善 (**10x-100x**)
- PPO / AlphaZero での学習事例

ゲームは分岐や動的処理があるため**一見不向き** → ドメインの拡張に貢献

## Limitations

- 原理的に完全な再現が不可能な対象もある (e.g., 別のバイナリを呼び出す・ロボット実機)
- 極端に条件分岐が多い場合には不向き

## Conclusion

- GPU上で動作するベクトル化されたシミュレータの有効性を実証的に示した
  - 単なる追試やベンチマークではなく、先行研究より高い性能
  - シミュレータを除けば構成要素は非常にシンプルな既存手法の組み合わせ

*"general methods that **leverage computation** are ultimately the most effective"*

# A Potential Disadvantage of Massive Batching

## Conclusion

100K回逐次 vs 20回バッチ ( batch size = 5K )

- 大規模なバッチ化ではDelayed Feedback & Reduced Adaptabilityが問題
- 一方で SH は大規模なバッチ化に対してロバストなことを理論的・実験的に立証
- MCTSにも応用可であることも実証した

Sequential Halving (SH; Karnin et al., 2013) はシンプルかつ高性能で最先端のRLアルゴリズムにも組み込まれている (e.g., Gumbel MuZero)

## Limitations

- 報酬分布がi.i.d.でない場合に適用が困難

例

- \* ハイパーパラメータサーチの文脈では報酬が学習中のロス (train loss)
- \* バッチで将来のロスは観測できない

# Conclusion and Future work

## Conclusion

*"general methods that leverage computation are ultimately the most effective" — Sutton [2019]*

1. GPU上で動くベクトル化されたRL環境の適応可能範囲を広げた（ゲームは一見不向き）

2. 有効性を実際に Bridge Bidding AI で示した

シンプルでスケーラブル

GPU/TPUマシンが進化すれば性能向上

3. 大規模バッチ化に適したアルゴリズムの存在を示した

Library	Environments	arXiv sub.
Brax [Freeman et al., 2021]	Continuous control	2021/6/24
Gymnas [Lange, 2022]	Classic control, bsuite, MinAtar	
Fgx (ours)	Board Games	2023/05/29
Jumanji [Bonnet et al., 2024]	Combinatorial optimization	2023/06/16
Waymax [Gulino et al., 2023]	Autonomous driving	2023/10/12
JaxMARL [Rutherford et al., 2024]	Multi-agent RL	2023/11/16
XLand-MiniGrid [Nikulin et al., 2023]	Meta RL, Mini-grid	2023/12/19
Craftax [Matthews et al., 2024]	Open-ended RL	2024/02/26
TORAX [Citrin et al., 2024]	Tokamak Transport	2024/06/10
NAVIX [Pignatelli et al., 2024]	Mini-grid	2024/07/28

→ “これからの強化学習は大規模なバッチ化を前提としてアルゴリズムを考える必要性がある”

(Deep learningとの組み合わせを踏まえても)

## Future work

- 自動での環境シミュレータのGPU対応 (e.g., LLMによる自動実装)
- モデルベース強化学習との組み合わせ (e.g., 低次元環境モデルは既知・高次元画像観測は学習)
- 大規模なバッチ化をexploitできるアルゴリズムの開発

\* 現在の強化学習のいくつかの構成要素は歴史的経緯からバッチ化に不向き (e.g., モンテカルロ木探索)

\* バイアスがある低分散な手法より、バイアスのない高分散な手法を大規模なバッチ化で実行した方が効率的かもしれない