コンピュータ囲碁講習会（囲碁AI編）

2023/05/20

小林　祐樹

講習会の目的

【講習会の狙い】

- AlphaGo登場以降の要素技術の概要を知ること

- ベースとなる囲碁AIを使って自分独自の囲碁AIを開発できるようになること

講習会の日程

2023/05/20：コンピュータ囲碁講習会（本日）

2023/05/27：9路盤ミニ大会

自己紹介

【名前】：小林　祐樹（こばやし　ゆうき）

【経歴】

- 2016年3月　電気通信大学大学院卒（研究テーマ：コンピュータ囲碁）
- 2016年4月～　（株）日立製作所

- 情報処理学会プログラミングコンテスト委員会（2018-2021）
- コンピュータ囲碁フォーラム理事（2022-）

【趣味】

バドミントン、囲碁AIの開発、コンシューマゲーム

GitHub：https://github.com/kobanium
Twitter：https://twitter.com/goraychan

開発プログラム

- 囲碁AI ”Ray”（使用言語：C++）
  AlphaGo登場以前の技術を使って実装したクラシカルな囲碁AI
    - Webページ：http://computer-go-ray.com/
    - GitHub：https://github.com/kobanium/Ray

- 囲碁AI ”TamaGo” (使用言語：Python）
  最新の囲碁AIの技術を手軽に試せる教育・学習を目的とした囲碁AI
    - GitHub：https://github.com/kobanium/TamaGo

- 将棋“AobaZero” (使用言語：C++, Python)
  将棋版AlphaZeroの追試を目的とした将棋AI
    - Webページ：http://www.yss-aya.com/aobazero/
    - GitHub：https://github.com/kobanium/aobazero

本日の内容

1. コンピュータ囲碁の概要
2. コンピュータ囲碁の技術
3. TamaGo
4. TamaGoを強くするには
5. まとめ
6. 9路盤ミニ大会に向けて

コンピュータ囲碁の概要

コンピュータ囲碁が人間を超えた日

- 2016年3月Google DeepMind社開発のAlphaGoが
  韓国のイ・セドル九段(世界大会優勝18回)に勝利

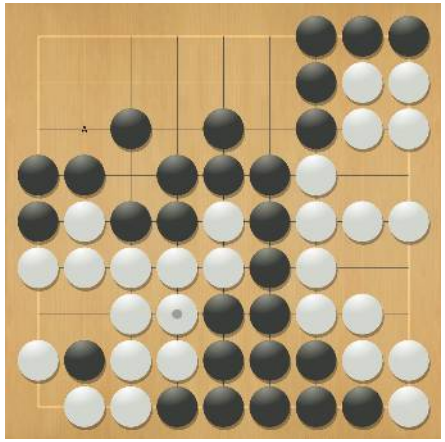
- それまでのコンピュータは人間のプロ相手に3子のハンデで勝てなかった
  ・・・ 東京の各市区町村で一番強い人と同じかちょっと弱いくらい

	先手	後手	結果
第1局	イ・セドル九段	AlphaGo	AlphaGoの勝ち
第2局	AlphaGo	イ・セドル九段	AlphaGoの勝ち
第3局	イ・セドル九段	AlphaGo	AlphaGoの勝ち
第4局	AlphaGo	イ・セドル九段	イ・セドル九段の勝ち
第5局	イ・セドル九段	AlphaGo	AlphaGoの勝ち
AlphaGo 4勝 – イ・セドル九段 1勝			

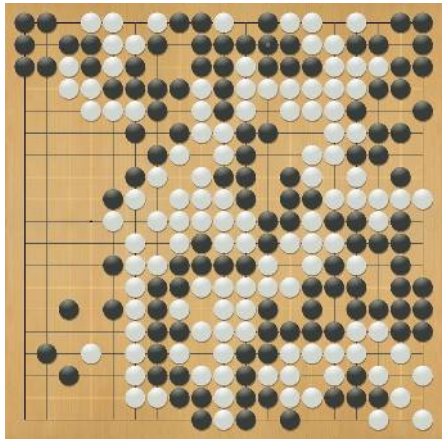
イ・セドル九段
（https://deepmind.com/alphago-korea）

囲碁

- 黒い石と白い石を交互に打つゲーム
- ルールは比較的シンプル（ただし終局の判断・処理が難しい）
- 盤のサイズは9x9、13x13、19x19などがある
- 陣地を多く囲ったほうの勝ち

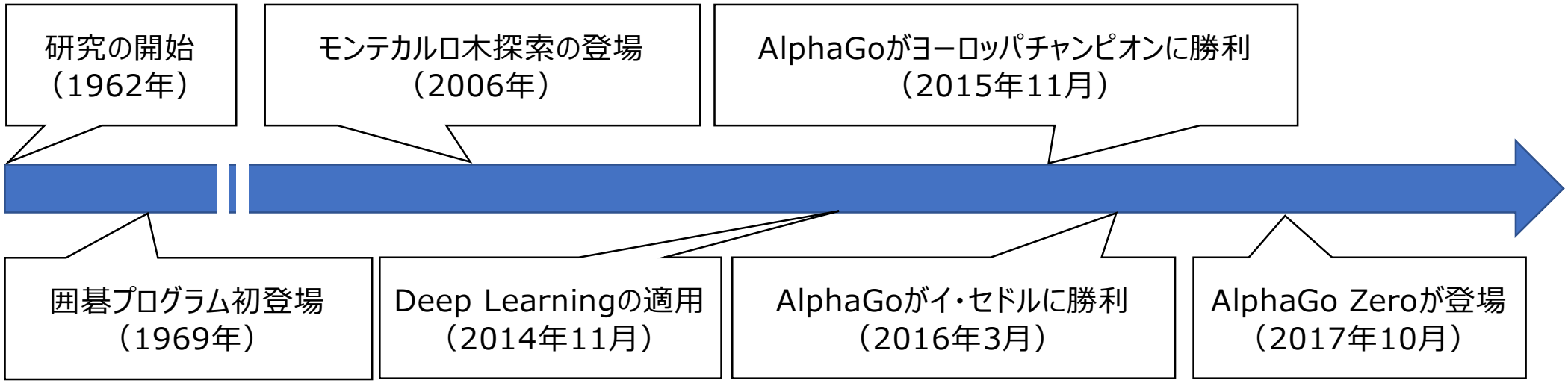


9路盤



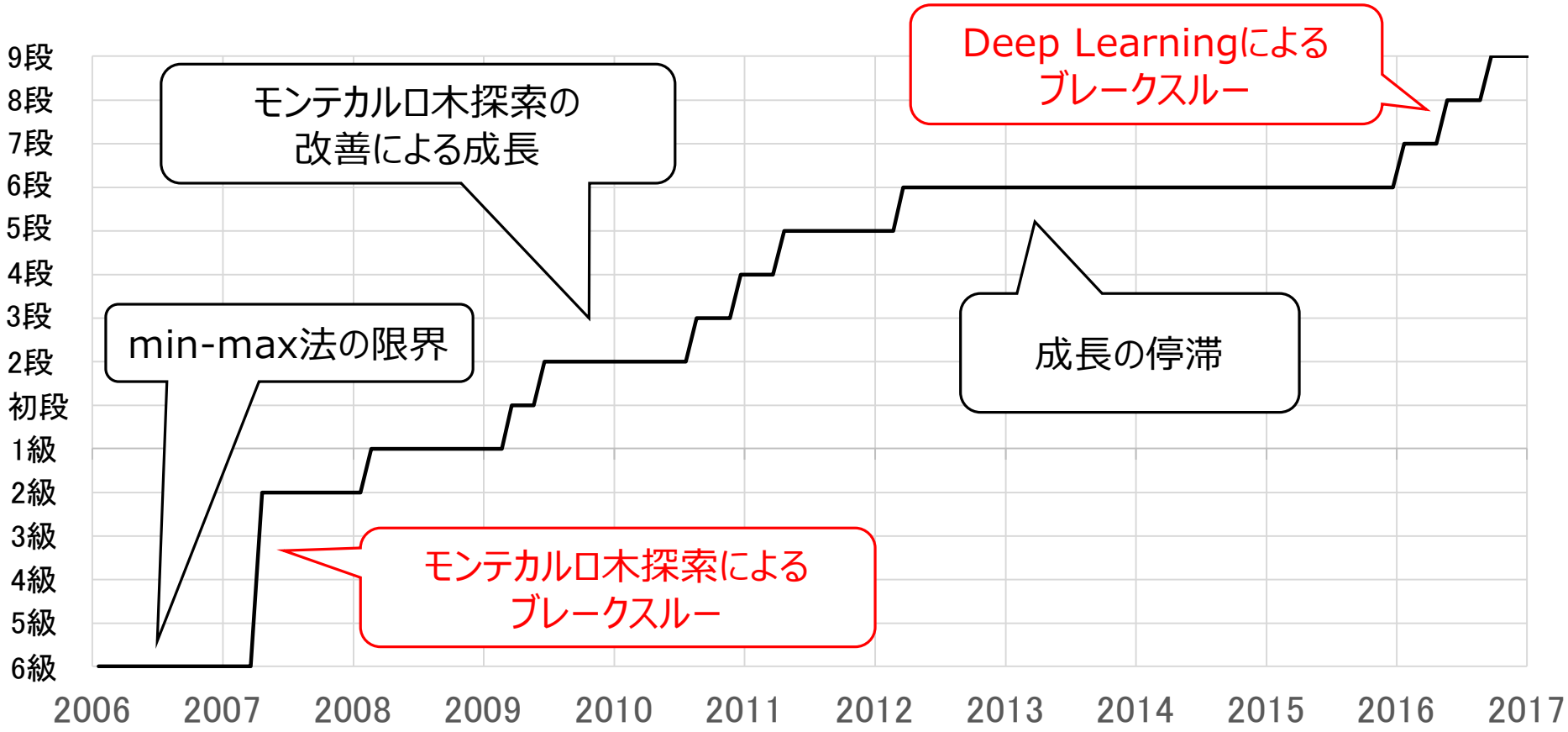
19路盤

コンピュータ囲碁の研究の歴史



コンピュータの強さ

コンピュータ囲碁の強さの変遷



KGS(ネット碁会所)でのコンピュータの強さの変遷

最新の技術動向

- AlphaGo Zeroを起点に適用範囲拡大、学習の効率化の研究が行われている

AlphaGo Zero (2017年)：自己対戦で強くなる強化学習アルゴリズム

→ AlphaZero (2017年)：AlphaGo Zeroの枠組みをチェス、将棋に適用

→ MuZero (2019年)：ゲームのルールも学習

→ Gumbel MuZero/AlphaZero (2021年)：学習の効率化

コンピュータ囲碁の技術

コンピュータ囲碁の技術

1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

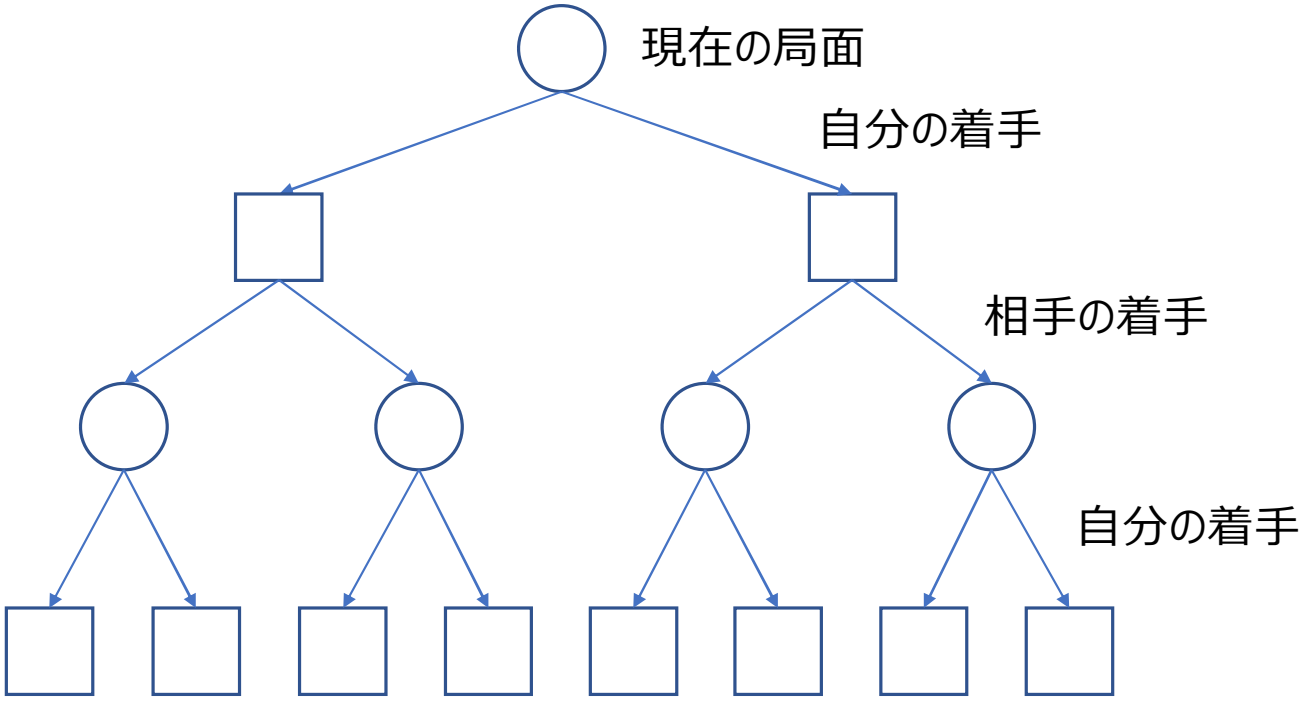
コンピュータ囲碁の技術

1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

min-max法

- 二人零和有限確定完全情報ゲームのプログラムではmin-max法が使われる
- 数手先の局面を調べ、最も自分が得する手を見つける
- 現在のオセロ、チェス、将棋のプログラムはこの手法をベースに実装されている

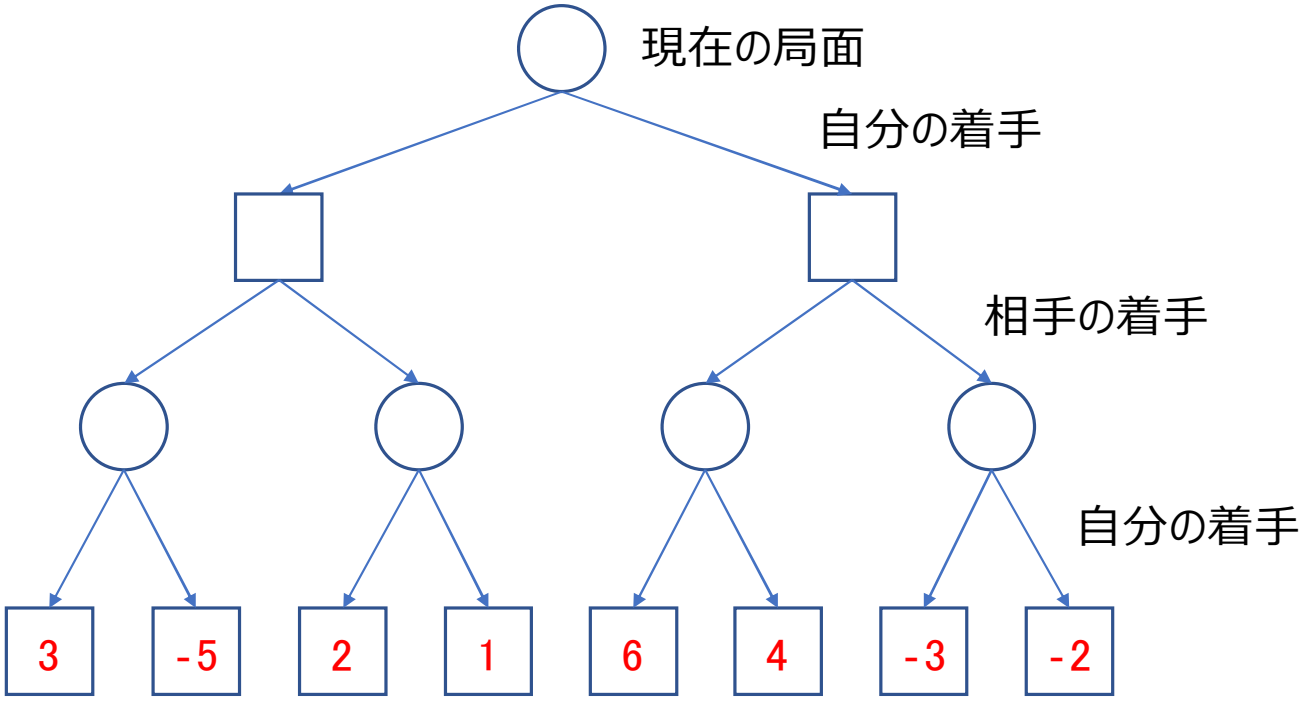
min-max法



：自分の手番

：相手の手番

min-max法

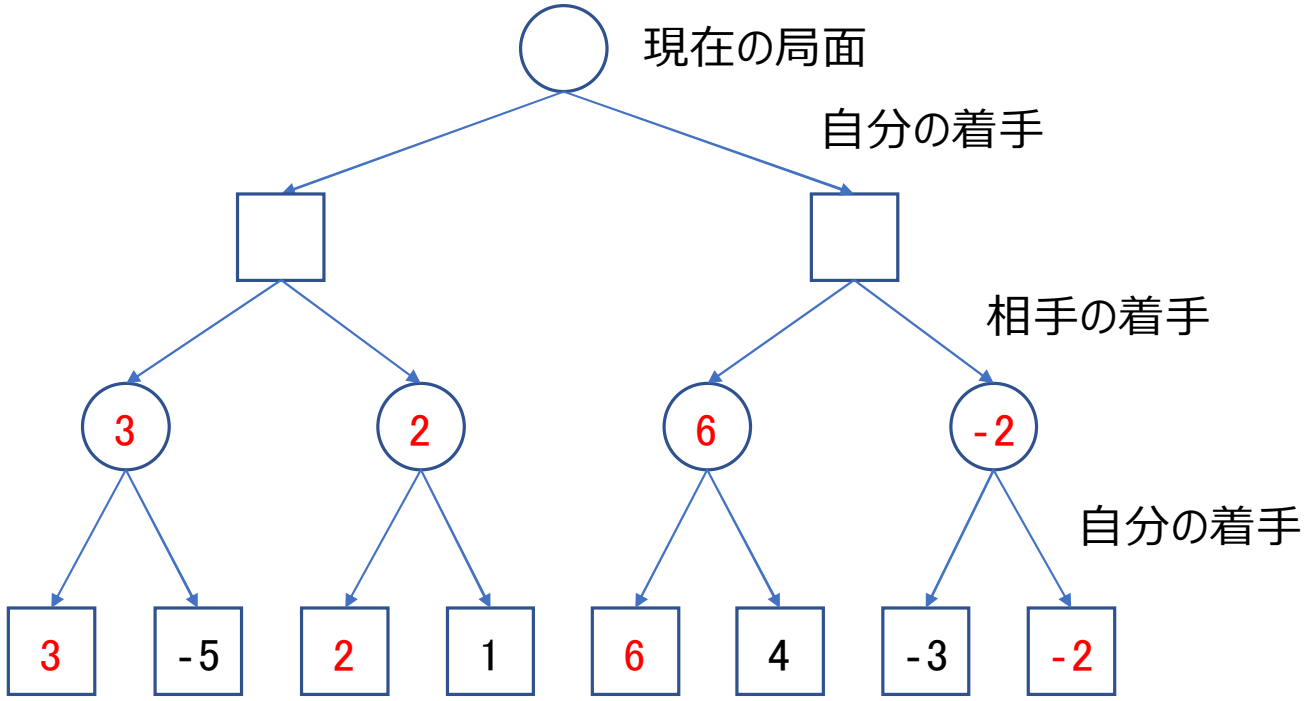


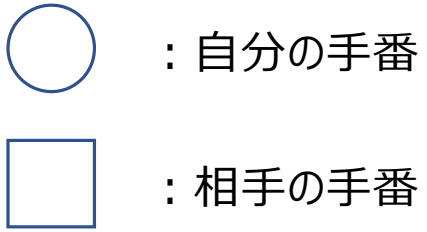
：自分の手番

：相手の手番

局面評価関数を使って、末端にスコアをつける

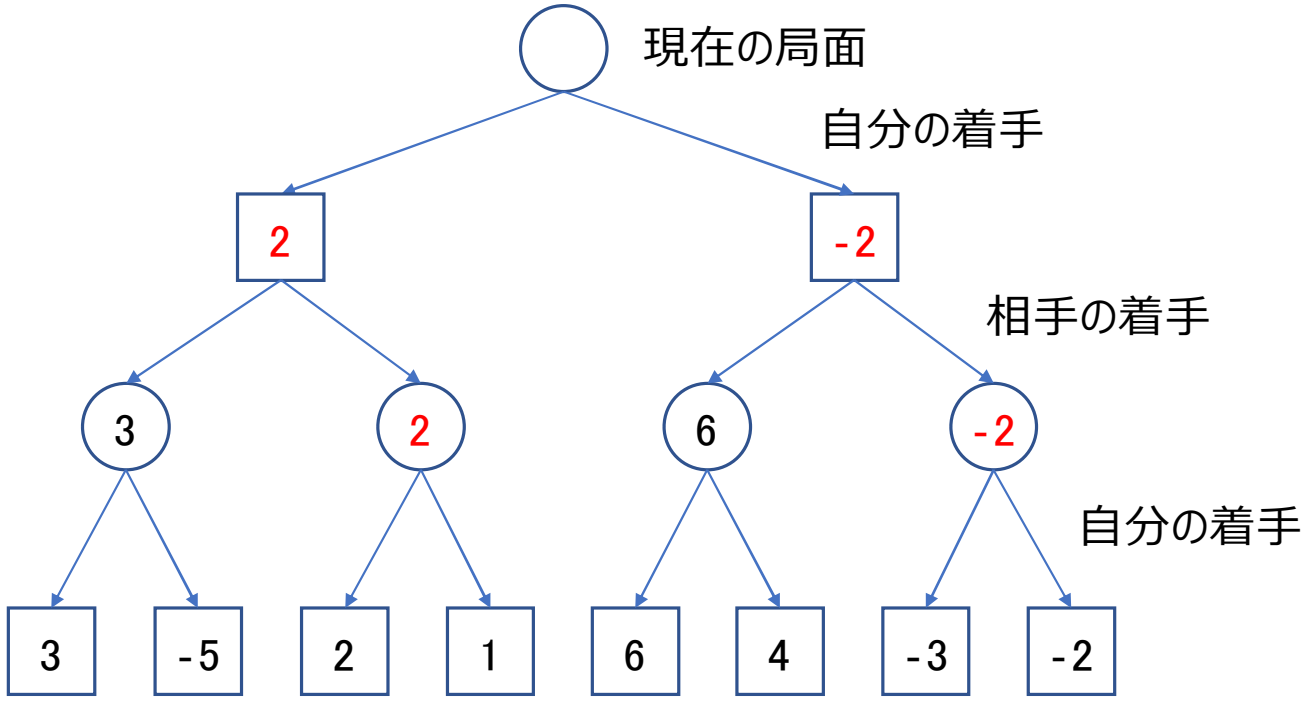
min-max法

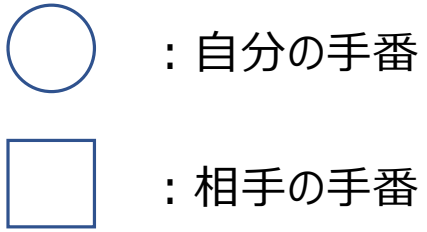




自分の手番なので、各々最も良いスコアを反映する

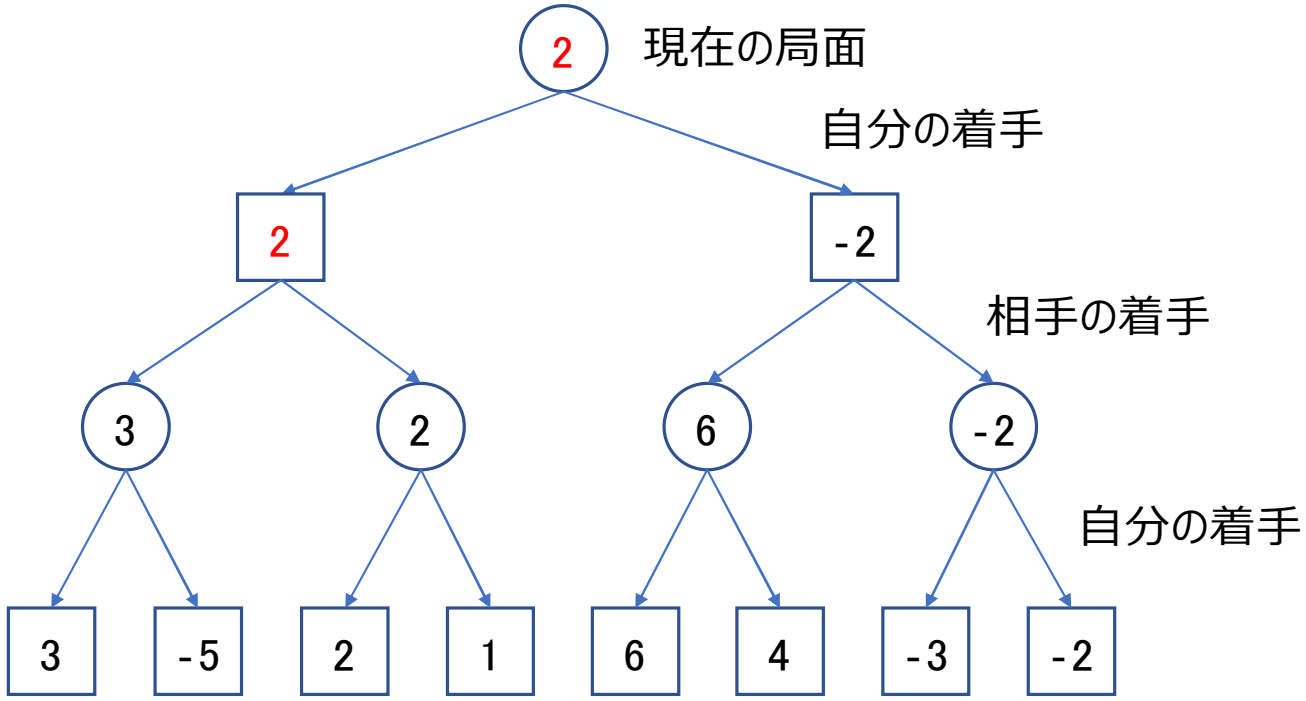
min-max法

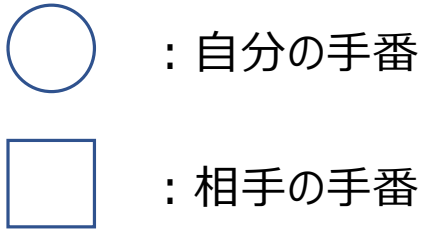




相手の手番なので、各々最も悪いスコアを反映する

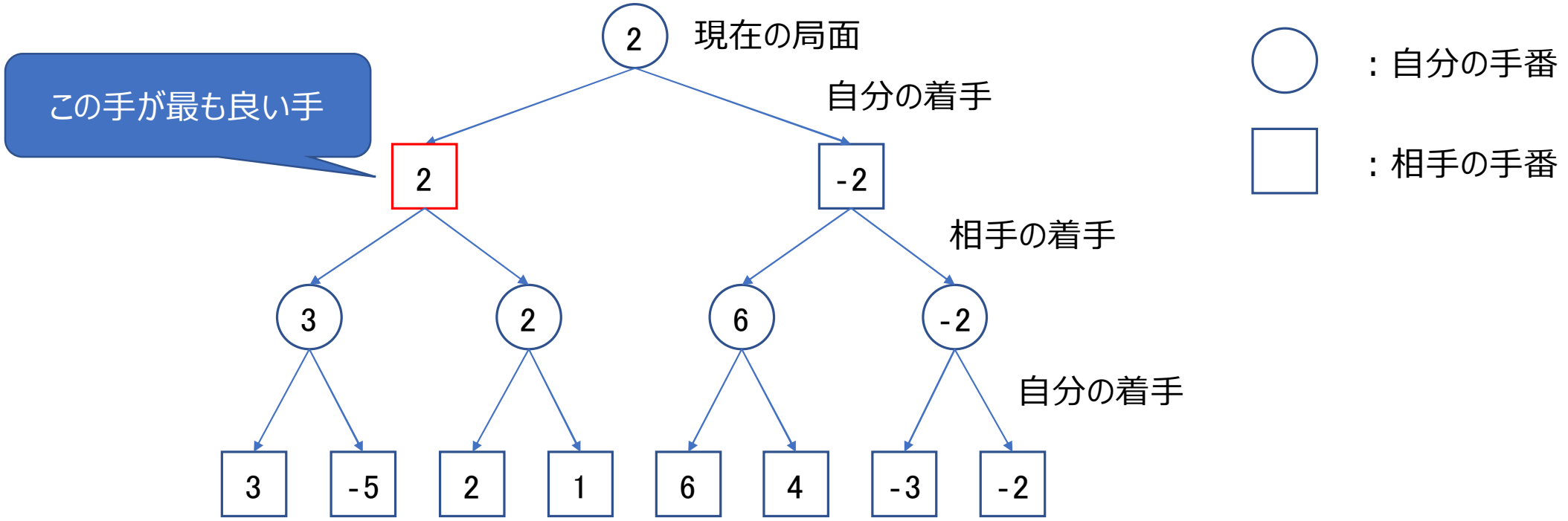
min-max法





自分の手番なので、最も良いスコアを反映する

min-max法


2
現在の局面
自分の着手
この手が最も良い手
2
-2
相手の着手
3
2
6
-2
自分の着手
3
-5
2
1
6
4
-3
-2
：自分の手番
：相手の手番

min-max法の特徴

- 計算にかかる時間はb^d
  b：1局面当たりの読む手の数 (branching factor)
  d：読みの深さ (depth)

- 末端の局面を評価する局面評価関数が必須

- 強くするためには正確な評価関数を用意し、深く読む必要あり
  着手評価関数：読む手を絞り込むために
  局面評価関数：優劣の正確な評価のために

囲碁におけるmin-max法

- 2006年までは囲碁でもmin-max法が主流（アマチュア初段未満の強さ）
  厳密にはより効率的なalpha-beta探索

- 強くならなかった要因は
  1. 正確な局面評価関数の作成が困難（人手での設計はほぼ不可能）
  2. 1局面あたりの読む手の数が膨大で深く読めない
     （19路盤で平均180手程度）

→ 2006年、モンテカルロ木探索（モンテカルロ法 + 木探索）の登場

コンピュータ囲碁の技術

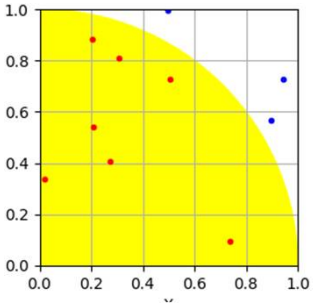
1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

モンテカルロ木探索とは

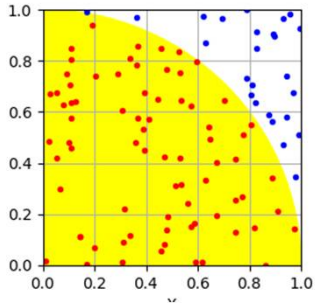
- 英語名：Monte Carlo Tree Search (MCTS)
- モンテカルロ法 + 木探索
- 乱数による対局シミュレーションの勝率を局面評価関数として使用

モンテカルロ法

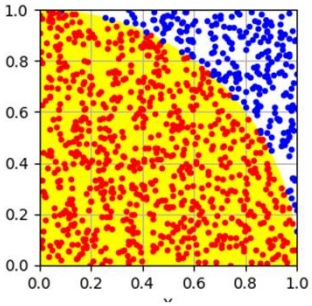
- 乱数を用いてシミュレーションや数値計算を行う手法
- 複数回試行することで近似解を求められる
- 試行回数を増やすと精度が上がる



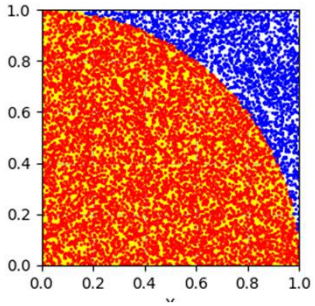
pi=2.8
(N=10)



pi=3.00
(N=100)



pi=3.036
(N=1000)



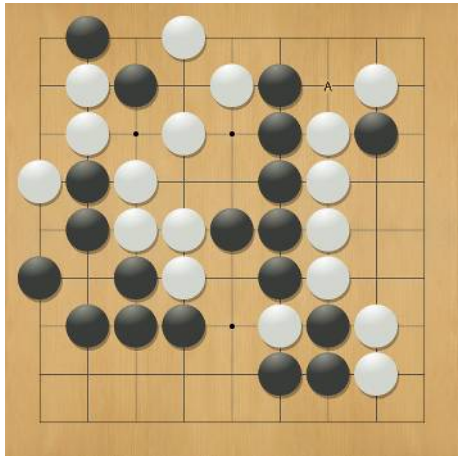
pi=3.1588
(N=10000)

```python
import random
import matplotlib.pyplot as plt
import matplotlib.patches as pat

def montecarlo_method(N=1000):
    """
    モンテカルロ法で円周率を求める
    """
    point = 0

    for i in range(N):
        x = random.random()
        y = random.random()

        if x * x + y * y <= 1.0:
            point += 1
            plt.plot(x, y, "ro")
        else:
            plt.plot(x, y, "bo")

    return 4.0 * point / N

# 黄色の四分円を描画
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111)
w = pat.Wedge(center=(0,0), r=1.0, theta1=0, theta2=90, color="yellow")
ax.add_patch(w)

N = 1000

pi = montecarlo_method(N)

# 試行回数と求めた円周率を出力
print("All Points : {}".format(N))
print("Pi : {}".format(pi))

plt.grid(True)
plt.xlim([0,1])
plt.ylim([0,1])
plt.xlabel("X")
plt.ylabel("Y")

# 各点の描画
plt.show()
```

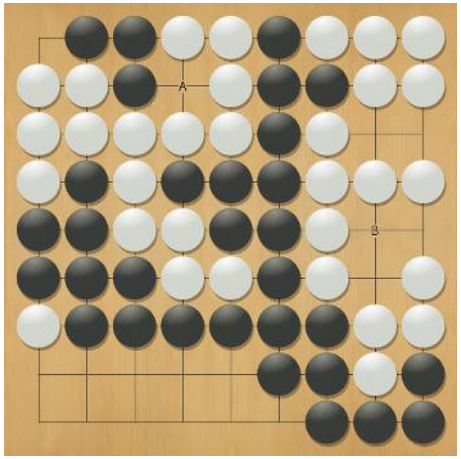
モンテカルロ法で円周率を求めるスクリプト

3.3 囲碁とモンテカルロ法

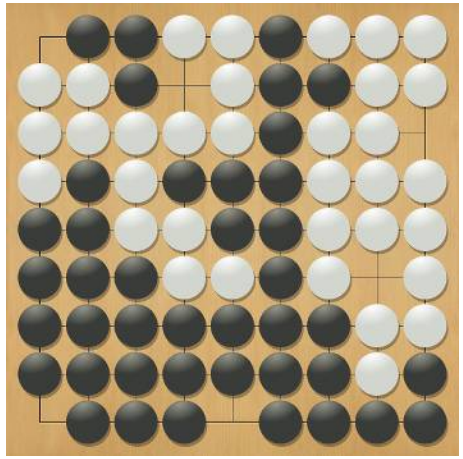
- 評価したい局面から終局までランダムに着手して、勝敗の平均（勝率）を算出
  評価する局面は現在の局面から1手（評価したい手）を打った後の局面
- 何回か対局シミュレーションして得られた勝率を局面評価関数として使用
  → 勝率が高い手＝良い手



この状態で勝負の判断は困難
（実際は白番勝ちの局面）

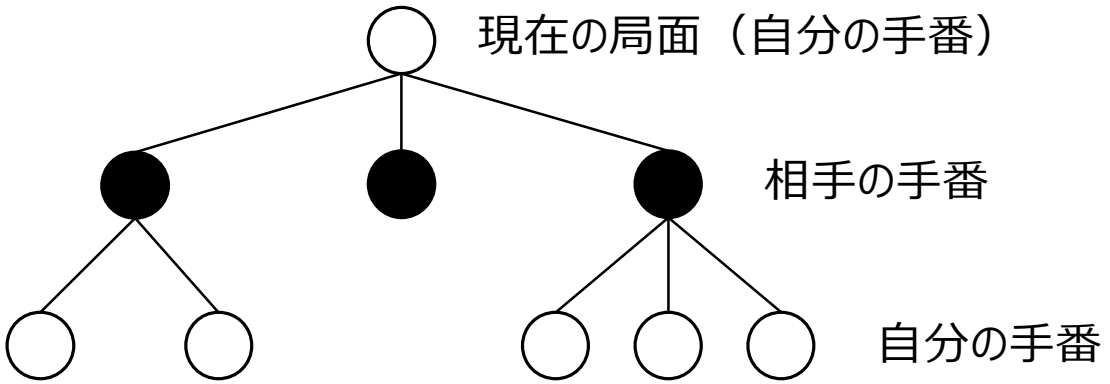


人間目線からの終局
（なんとなく白勝ちがわかる）

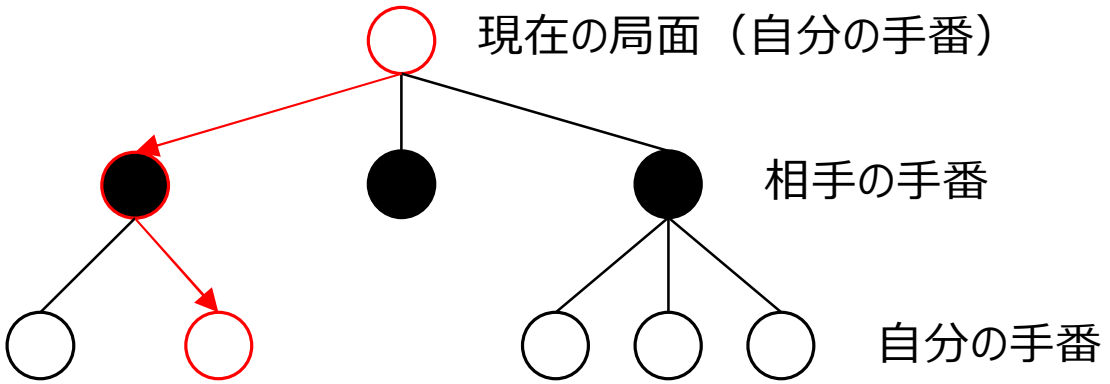


コンピュータ目線での終局
（ここまで行くと機械的に勝敗判定可能）

モンテカルロ木探索の処理

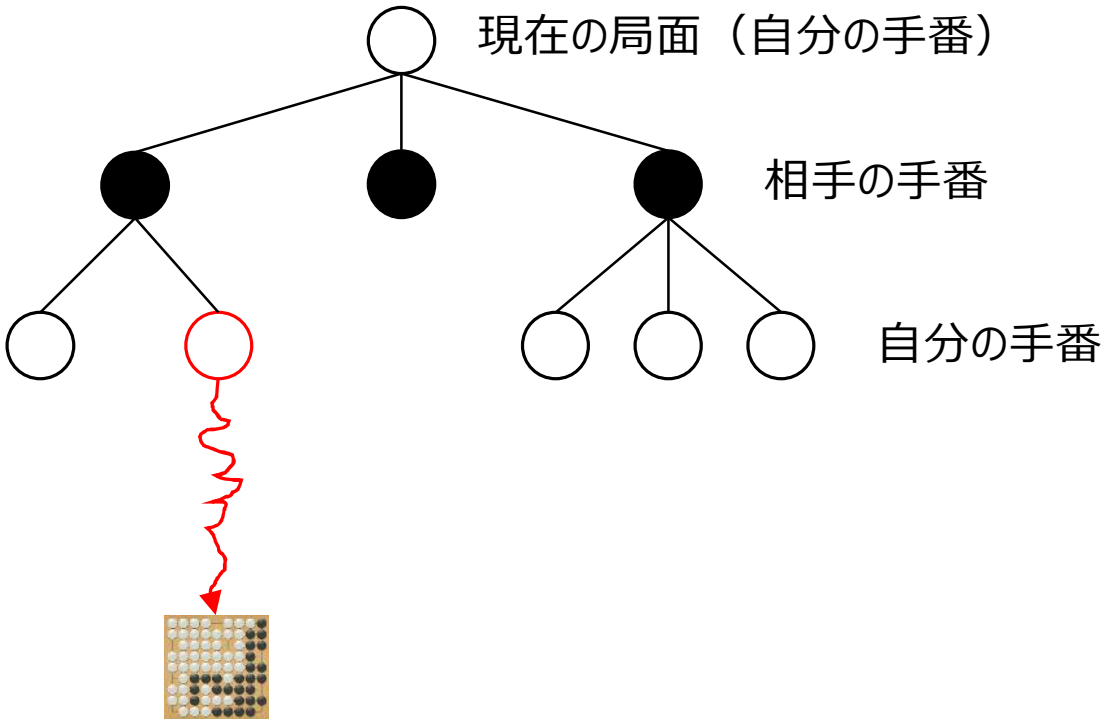


モンテカルロ木探索の処理



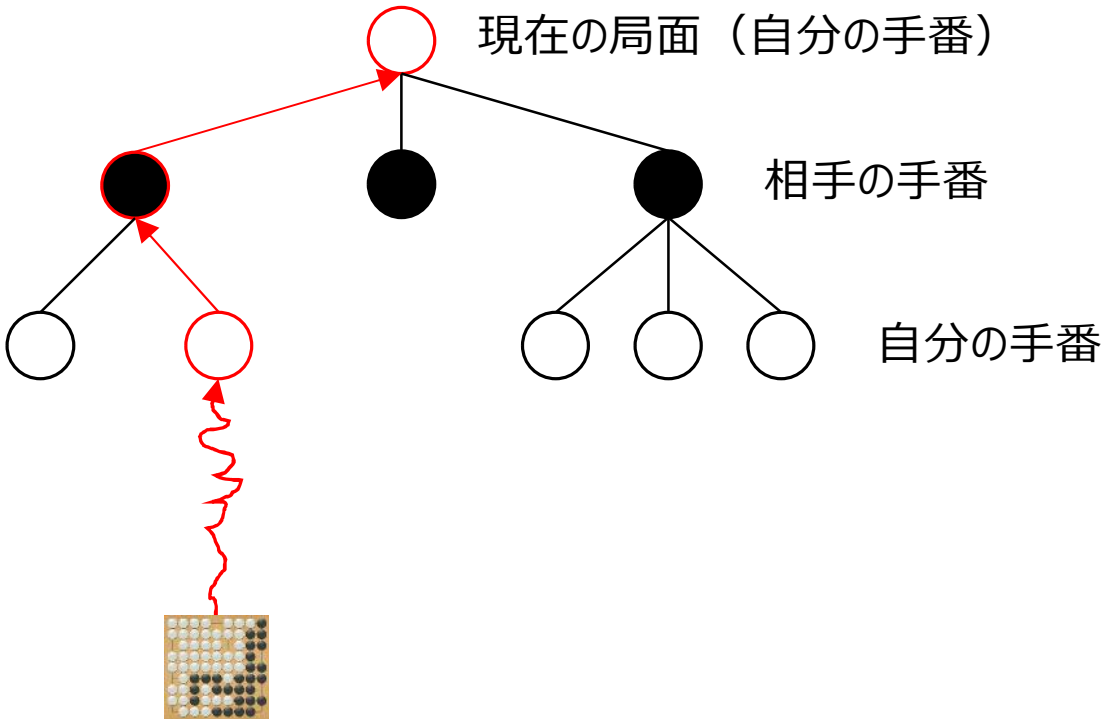
① 末端に到達するまで評価したい手を選ぶ

モンテカルロ木探索の処理



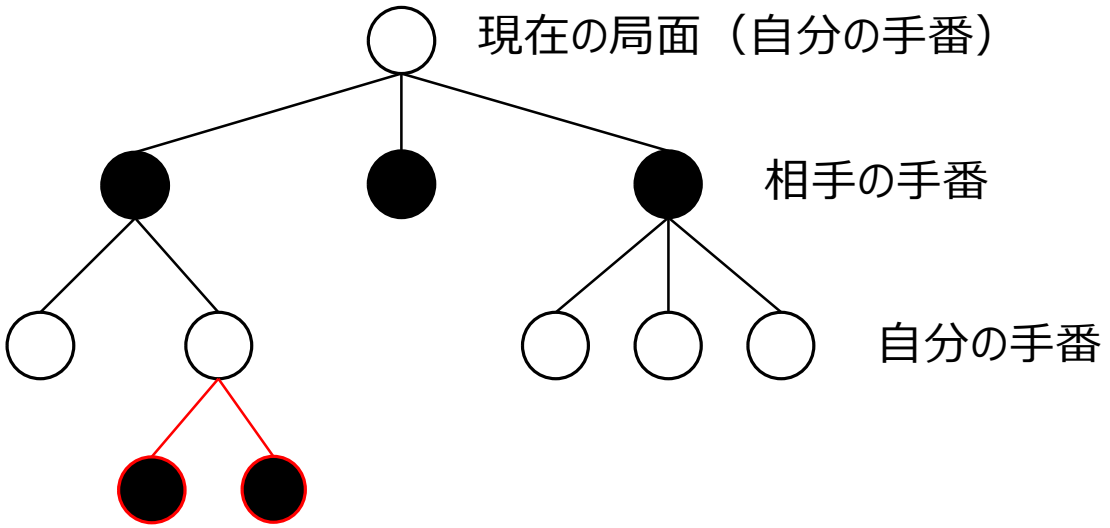
② 終局まで乱数に従って着手する
（対局のシミュレーション）

モンテカルロ木探索の処理



③ 結果を反映する
(以下①～③繰り返し)

モンテカルロ木探索の処理



④ 有望そうな手を1手深く読む

評価する手を選ぶ基準は・・・

- MCTSで評価したい手を選ぶときにジレンマが発生する…
  より勝率の高い有望な手 vs まだ十分に探索していない手

- このジレンマをうまいこと扱うためにUCB1値を使う

$$\mathrm{UCB1}(j) = \bar{x_j} + \sqrt{\frac{2\ln n}{n_j}}$$

x_j：着手jの勝率
n_j：着手jの探索回数
n：現局面の探索回数

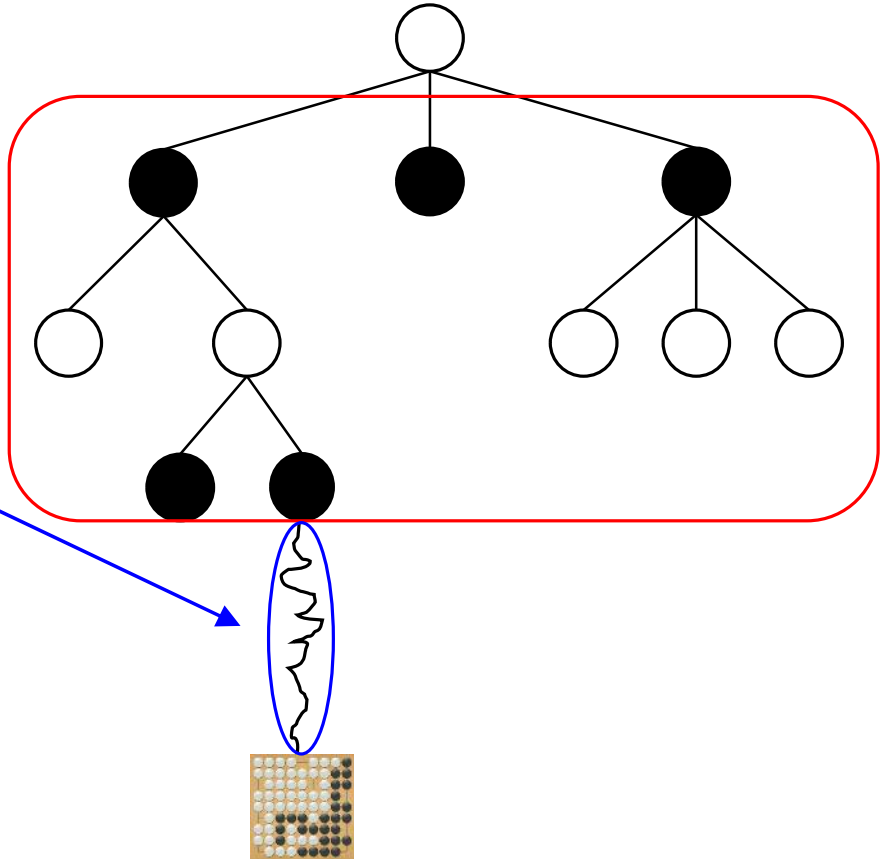
- かなり研究が進んでいるため、他にもいろいろ指標がある

強くするためには

- 評価する手を絞り込む
  →無駄な手を評価しない（深く読むため）

- 囲碁らしい手を打ってシミュレーションする
  →得られる勝敗を確からしいものにする

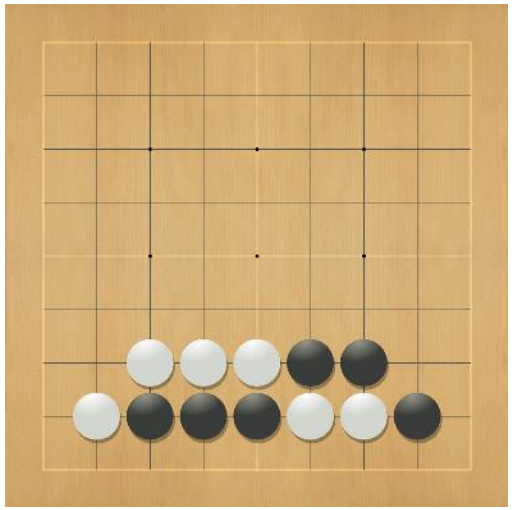
- たくさんシミュレーションする
  → 1．勝率が正確な値に近づく
  　 2．読みが深くなる



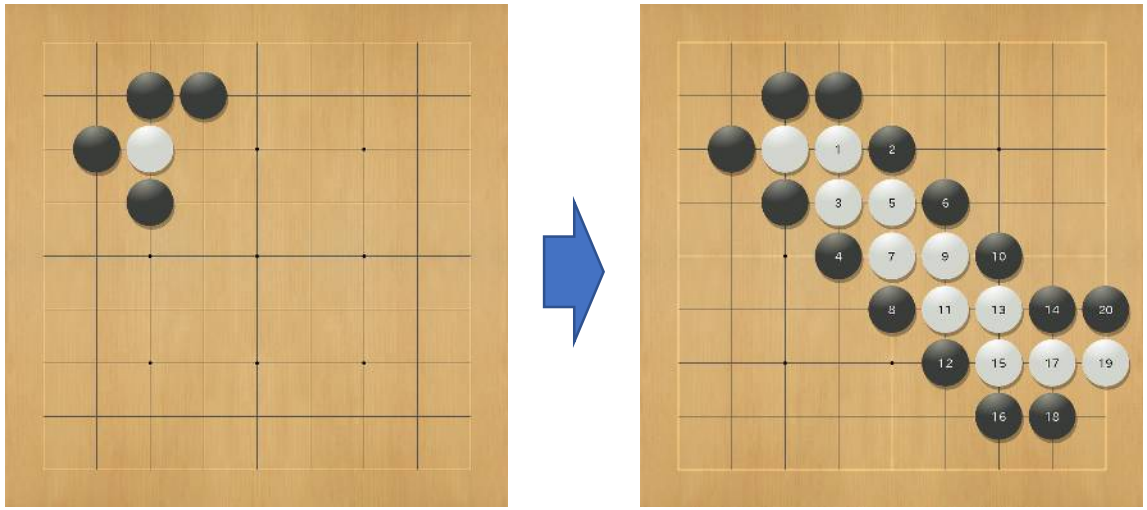
モンテカルロ木探索の弱点

- 圧倒的に優勢（または劣勢）の時の着手がおかしくなりがち

- 攻め合いやシチョウなど正確な応手を打ち続ける必要がある局面では
  シミュレーションの勝敗の信頼性が低下



攻め合い



シチョウ

コンピュータ囲碁の技術

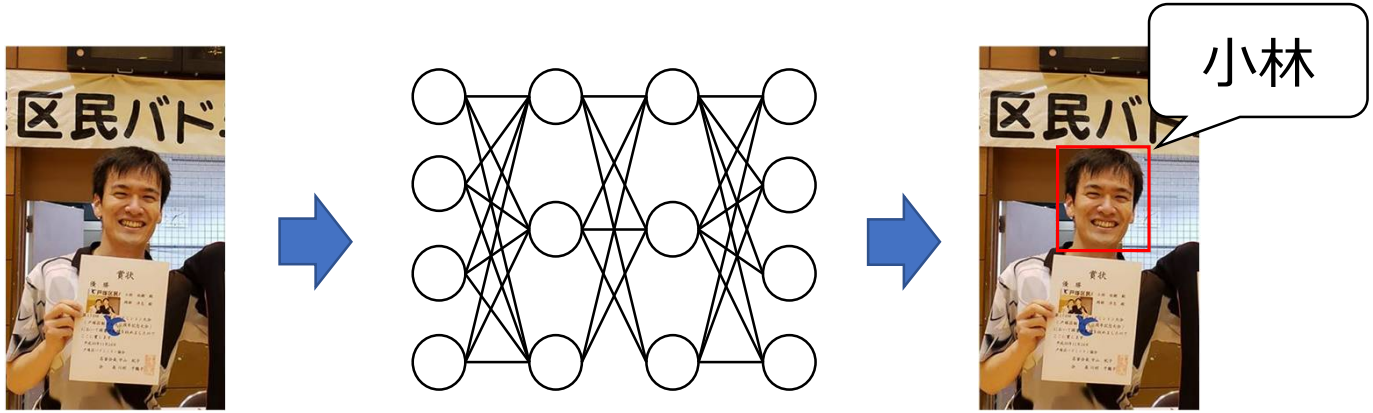
1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

Deep Learning

- 4層以上の多層Neural Networkによる機械学習手法
  Neural Network：神経細胞（ニューロン）とそのつながりを
  数式的なモデルで表現したもの

- 近年の音声・画像認識、自然言語処理の性能向上を実現させた技術

- 2012年、画像認識コンテストで圧倒的な性能で優勝
  （現在の人工知能ブームのきっかけ）

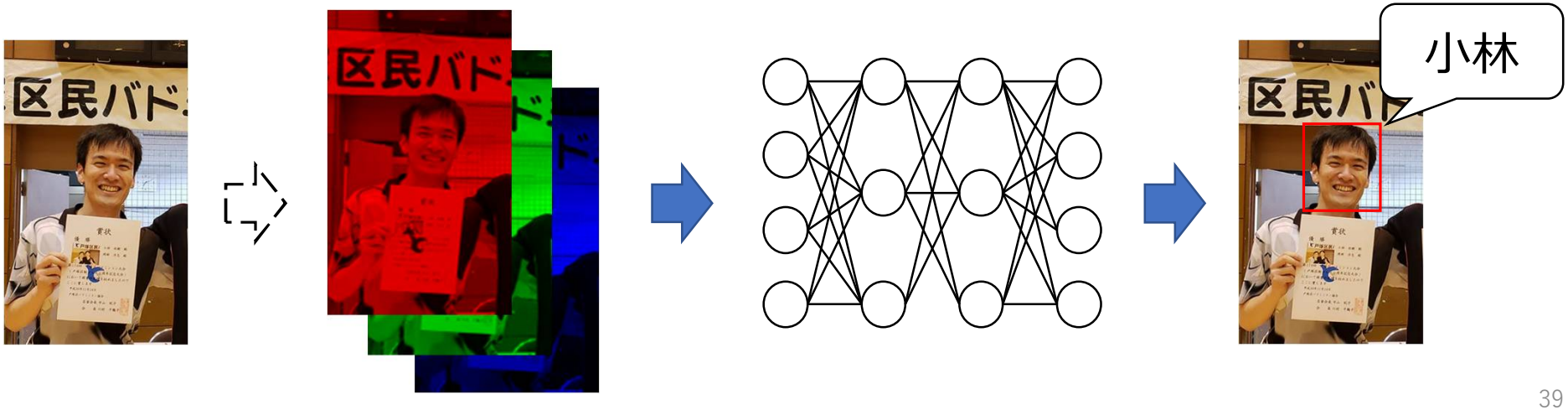


画像認識におけるDeep Learning

- 畳み込みニューラルネットワークを用いるのが一般的
  （最近はVision Transformerなど違うアーキテクチャを用いるケースもある）

- 人間がやるべき処理は画像のRGB成分を分けて入力するだけ

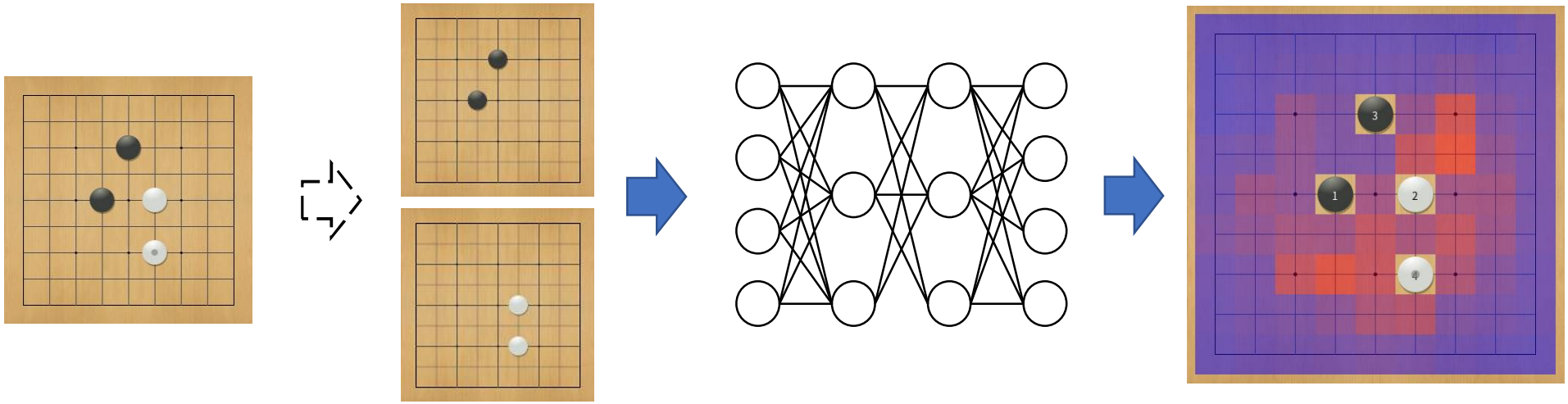
- 線や濃淡の特徴などはニューラルネットワークが自動的に抽出する



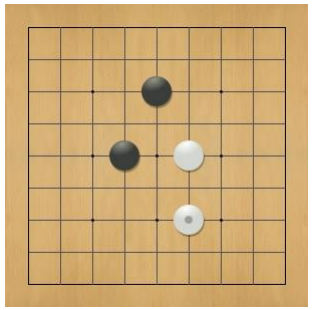
囲碁は画像に近い

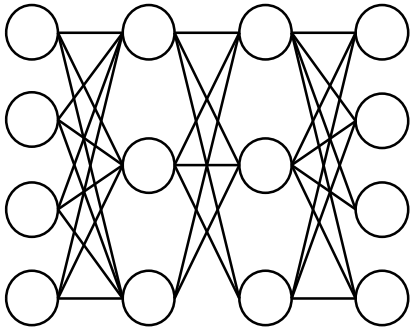
- 囲碁は9x9, 13x13, 19x19の画像として捉えられる

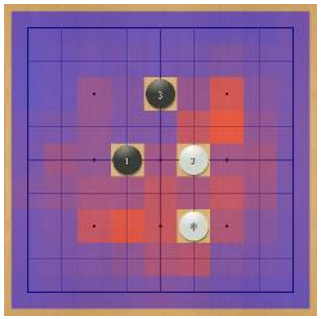
- 各交点は黒石、白石、空点の3つのどれかの状態を取る



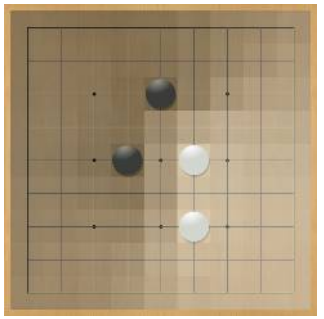
囲碁で推測すること







黒の勝率 48.2%



白が0.6目優勢

Policy

Value

Ownership

Score

予測は

Score, Value, Ownership, Policy

の順で難しいと思われる

コンピュータ囲碁の技術

1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

AlphaGo

- Google DeepMind社が開発した囲碁AI

- モンテカルロ木探索 + Deep Learning

- モンテカルロ木探索実行時に2つのネットワークを利用
  Policy Network：現局面での着手の優劣の数値化（着手評価関数）
  Value Network：現局面でのある手番の勝率（局面評価関数）

- Policy Networkは人間の棋譜から教師あり学習

- Value Networkは学習するために自己対戦と強化学習を利用

教師あり学習と強化学習

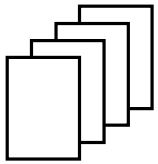
教師あり学習

お手本（正解）を真似する
（お手本が必須）

強化学習

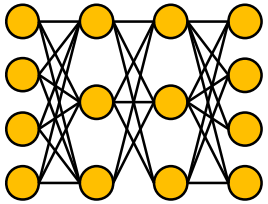
行動した結果をフィードバックする

AlphaGoの作り方



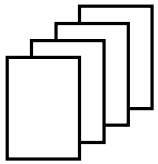
人間の対局データ
（約10万局）

①人間の手を
教師として学習



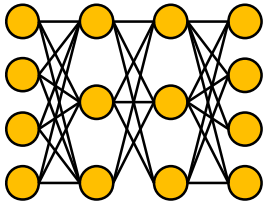
教師あり学習で作った
Policy Network

AlphaGoの作り方



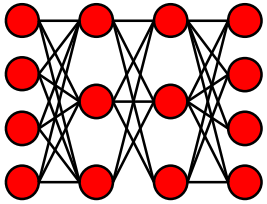
人間の対局データ
（約10万局）

①人間の手を
教師として学習



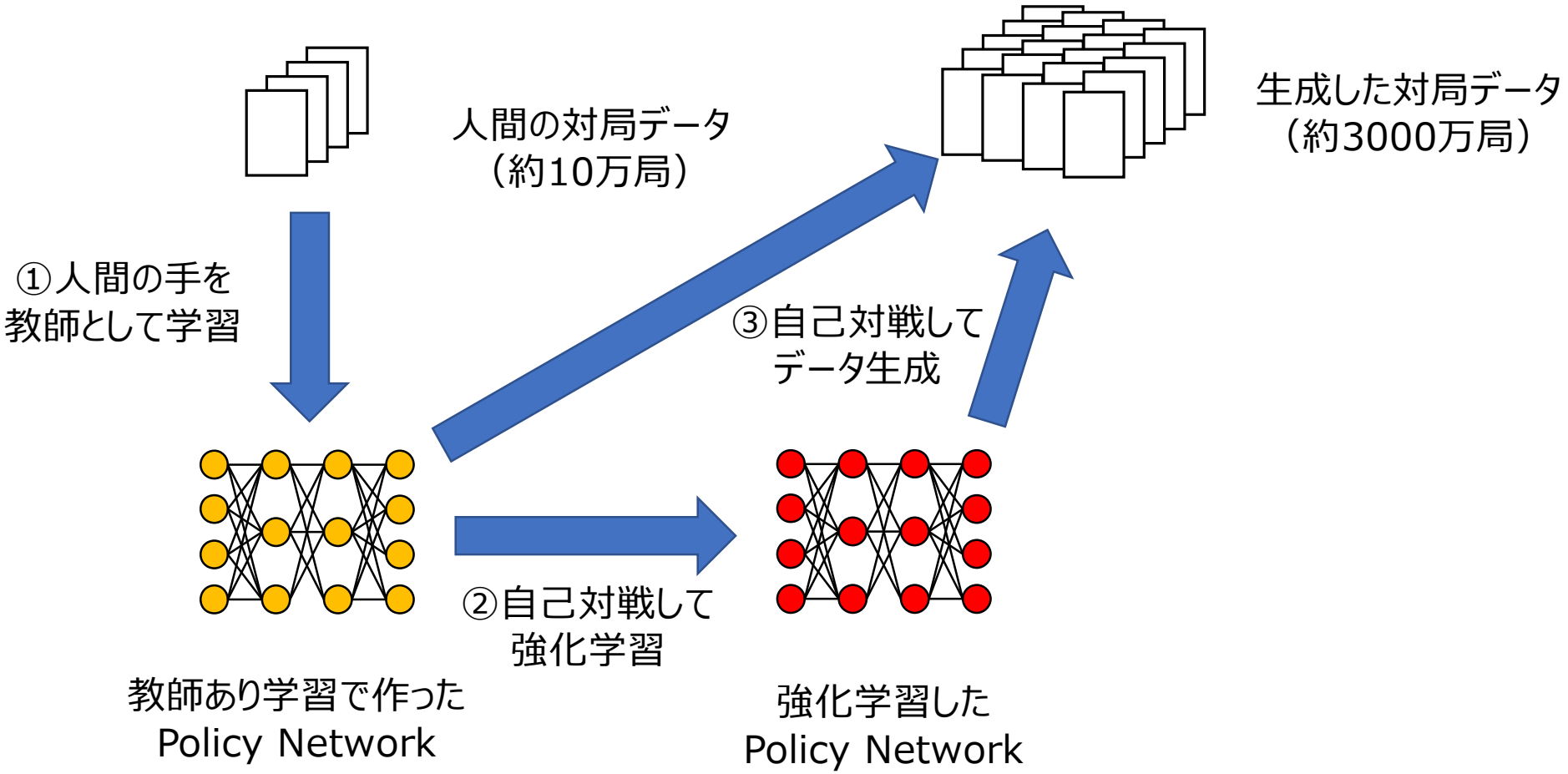
教師あり学習で作った
Policy Network

②自己対戦して
強化学習

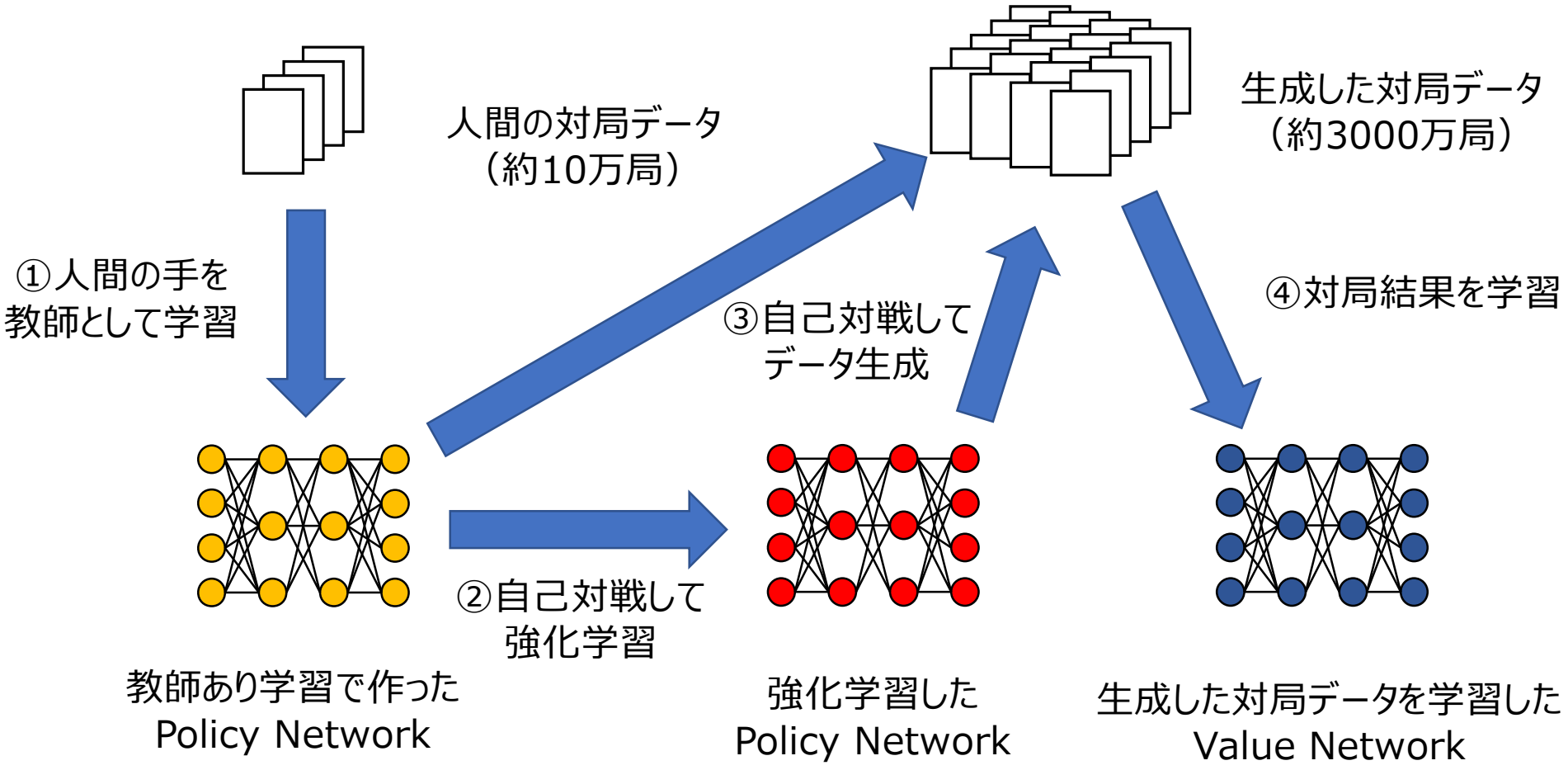


強化学習した
Policy Network

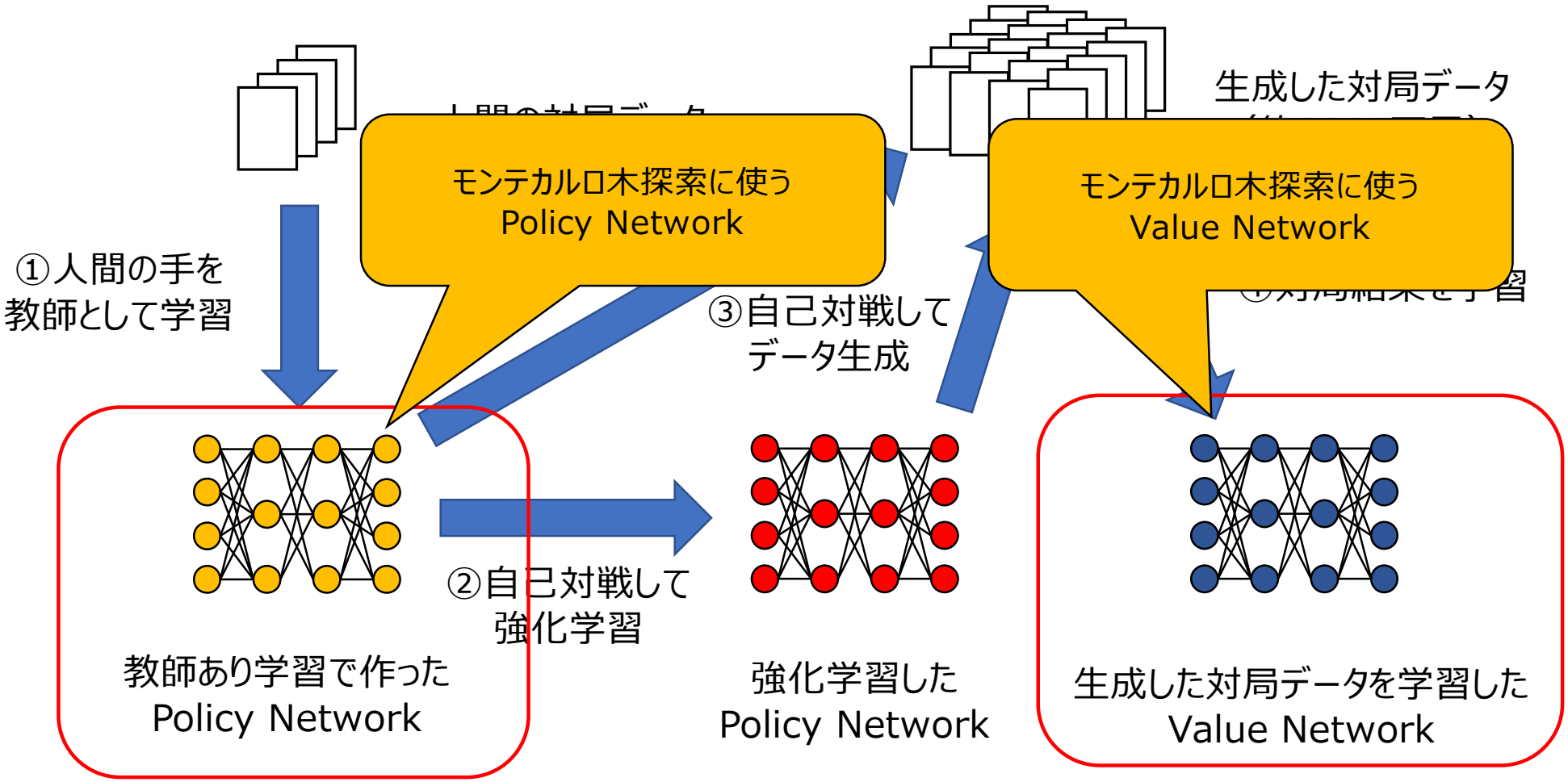
AlphaGoの作り方



AlphaGoの作り方

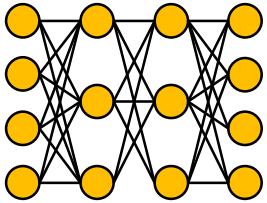


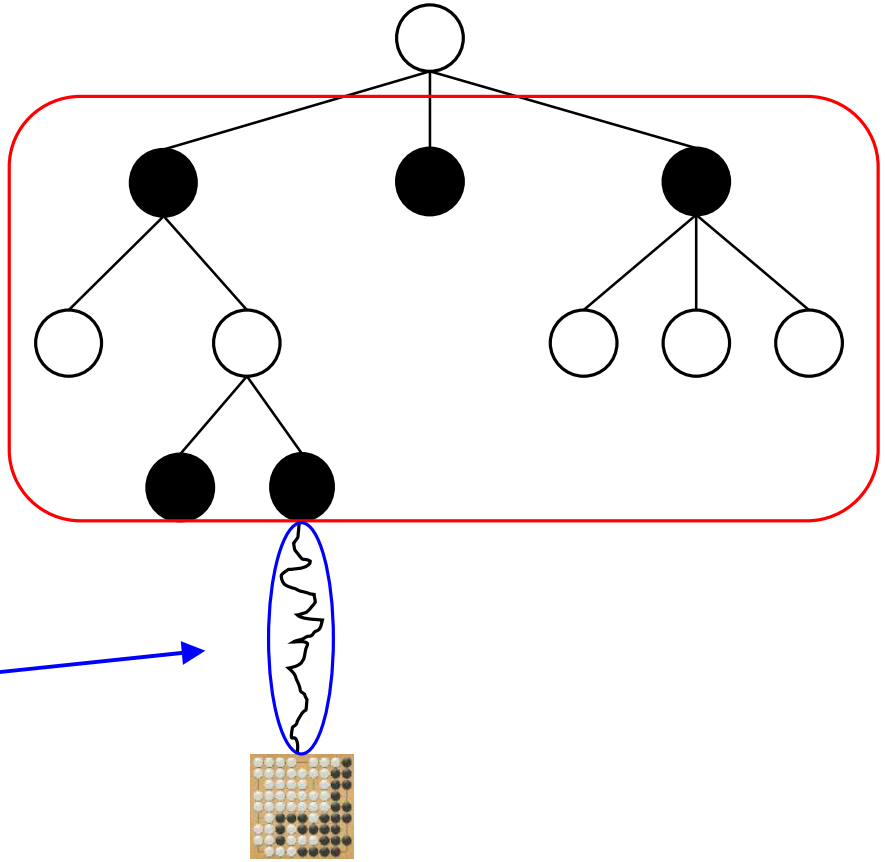
AlphaGoの作り方



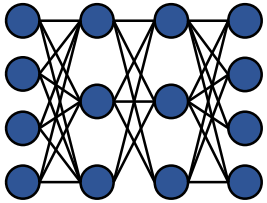
モンテカルロ木探索へのNetworkの組み込み

- 読む手を絞り込むためにPolicy Networkを利用





- 対局のシミュレーション結果を補正するために
  Value Networkを利用



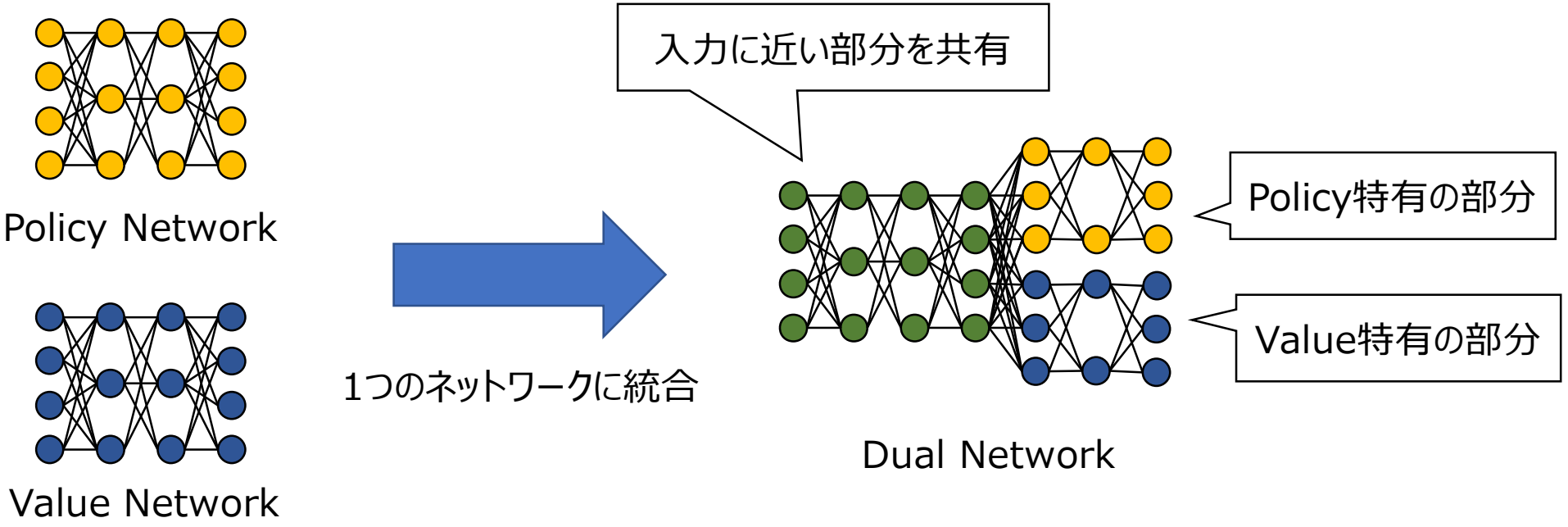
コンピュータ囲碁の技術

1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

AlphaGo Zero

- Google DeepMind社が開発した囲碁AI
- 強化学習のみで人間が生み出したデータは一切使わない
- Policy NetworkとValue Networkを統合したDual Networkを利用
- 膨大な計算機資源を利用

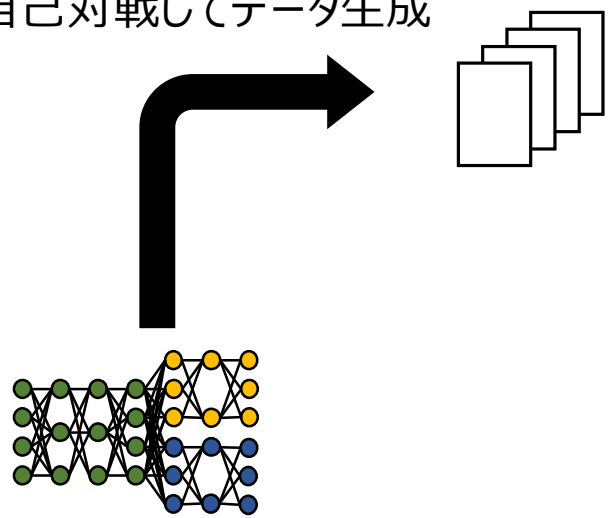
Dual Network



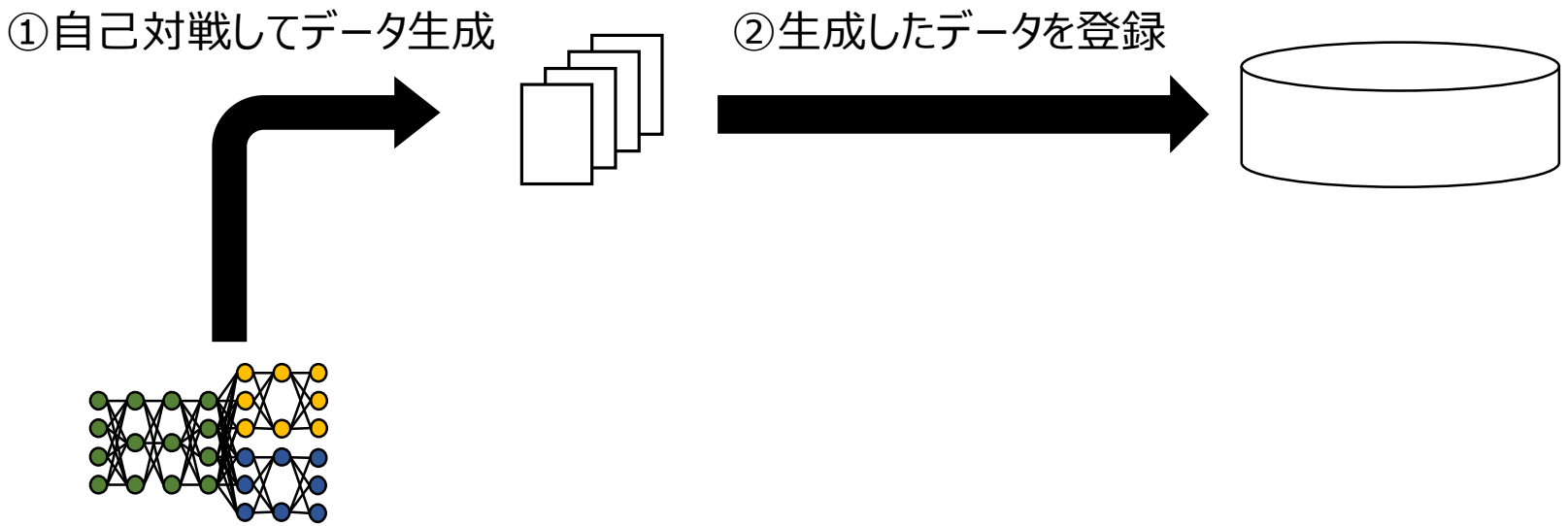
1つに統合した方が相互に作用して効率的に学習できる

強化学習のサイクル

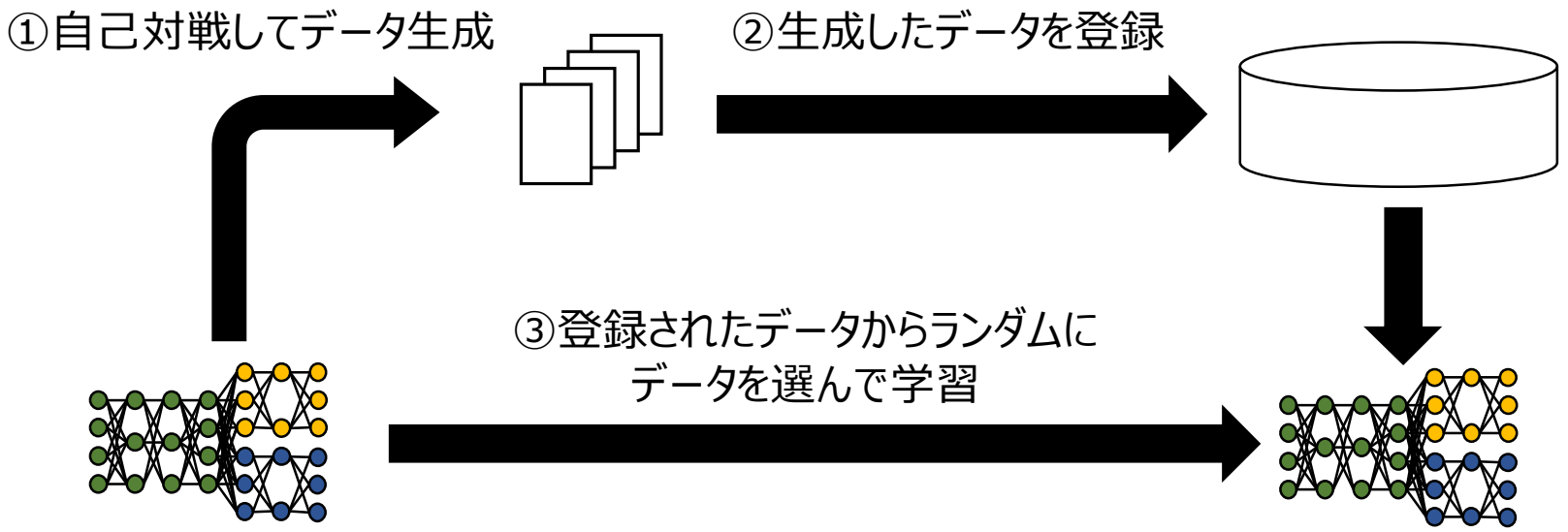
①自己対戦してデータ生成



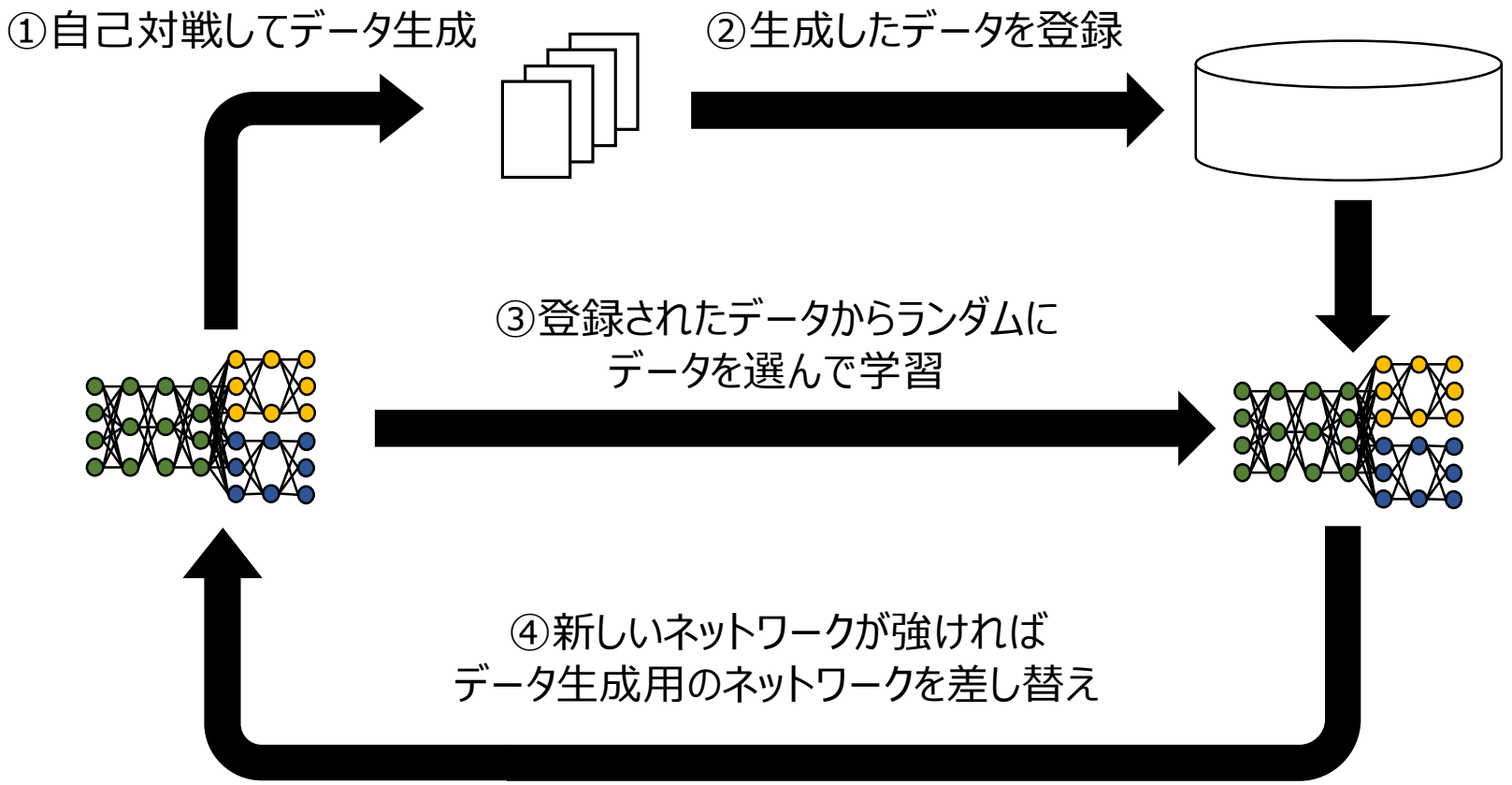
強化学習のサイクル



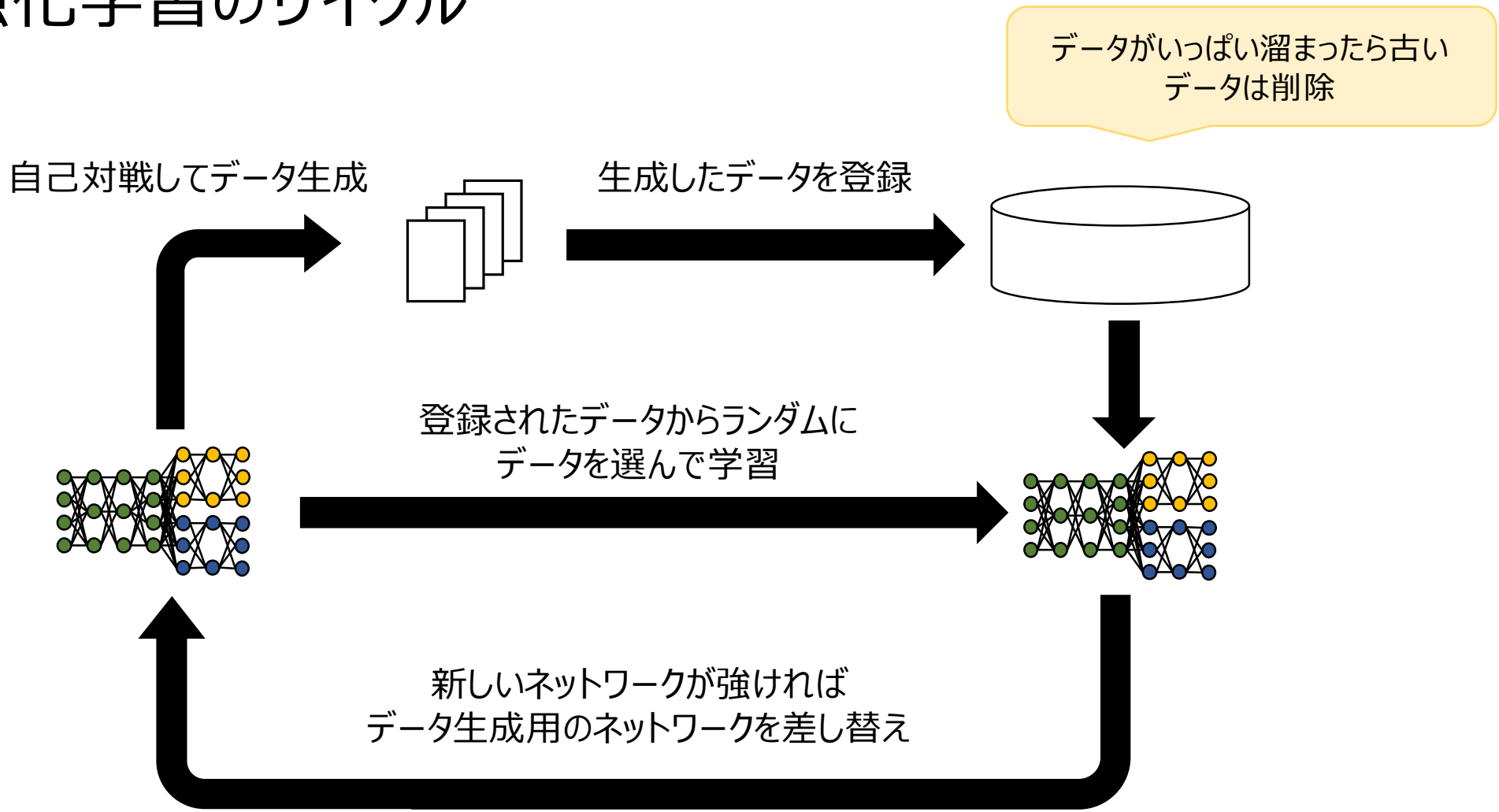
強化学習のサイクル



強化学習のサイクル


①自己対戦してデータ生成
②生成したデータを登録
③登録されたデータからランダムに
データを選んで学習
④新しいネットワークが強ければ
データ生成用のネットワークを差し替え

強化学習のサイクル


データがいっぱい溜まったら古い
データは削除
自己対戦してデータ生成
生成したデータを登録
登録されたデータからランダムに
データを選んで学習
新しいネットワークが強ければ
データ生成用のネットワークを差し替え

自己対戦時の設定

- 自己対戦ではPUCTアルゴリズム（MCTSの一種）を使用する

$$\mathrm{PUCB}(j) = v_j + c_{\mathrm{pucb}} p_j \frac{\sqrt{n}}{1 + n_j}$$

v_j：着手jのValueの平均値
p_j：着手jのPolicyの値
n_j：現局面における着手jの探索回数
n：現局面の探索回数
c_{pucb}：定数

- 局面にばらつきが出るように下記ランダム性を入れ込む
  - PolicyとDirichlet分布に従ったノイズの加重平均をp_jに使用
  - 30手目まで探索回数の分布に従って着手

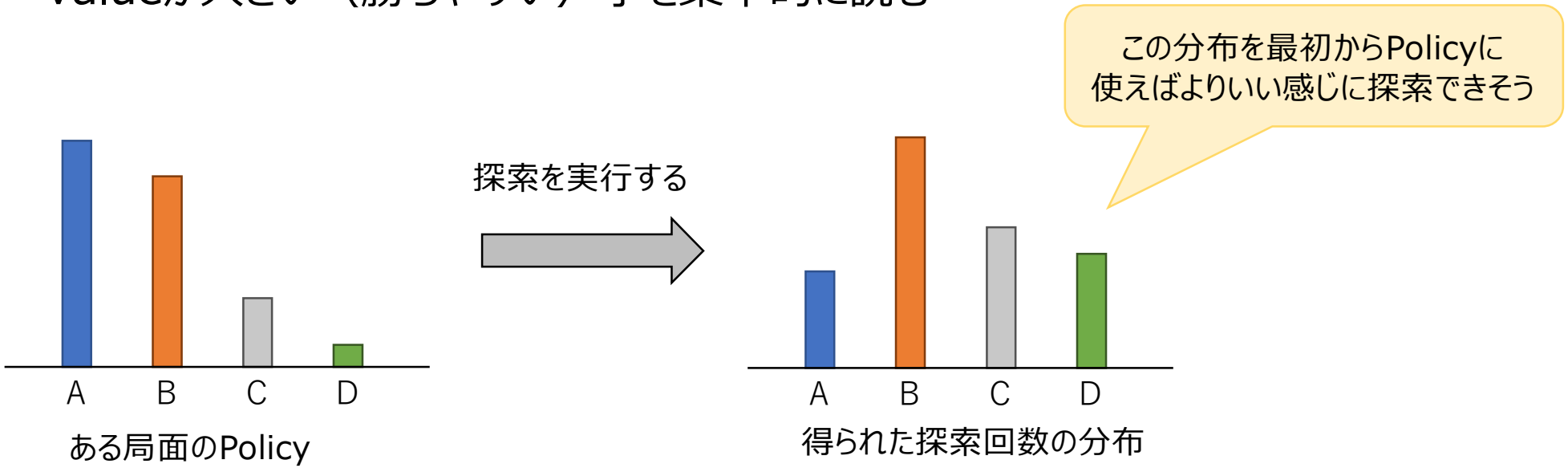
- 1手あたりの探索回数は1600回

Policyを改善するには

- PUCB値はPolicyとValueをバランスよく評価

$$PUCB(j) = v_j + p_j \frac{\sqrt{n}}{1 + n_j}$$

- たくさん探索していくと第2項の値は小さくなっていく（分子の方が分母より小さくなる）
  → Valueが大きい（勝ちやすい）手を集中的に読む



PolicyとValueの学習

- Policyは局面に対する探索回数の分布を学習
  - Policy：その局面だけを見た手の良し悪しの評価
  - 探索回数の分布：その局面をある程度読んで得られた手の良し悪しの評価

→ 感覚的に前者より後者の方が強そう
（“Valueが正確であれば”という前提がある）

- Valueは自己対戦の勝敗をそのまま学習

→ 自己対戦の勝敗は強いプレイヤであるほど妥当と思われる
（PolicyとValueの正確さが棋力に影響する）

PolicyとValueが相互に作用しながら協調して強くなる

AlphaZero

- AlphaGo Zeroの強化学習の枠組みを汎用的にまとめたもの
  （囲碁に加えて、チェス、将棋で実験・検証）

- 常に最新のネットワークを使用して自己対戦
  AlphaGo Zeroは一番強いネットワークを使用

- それぞれのゲームで当時最強のプログラムに勝ち越し
  チェス：Stockfish（※1）に28勝72分
  将棋：elmo（※2）に90勝8敗2分

(※1) 世界最強のオープンソースチェスプログラム
(※2) 第27回(2017年)世界コンピュータ将棋選手権優勝プログラム

コンピュータ囲碁の技術

1. Min-Max法
2. モンテカルロ木探索
3. Deep Learning
4. AlphaGo
5. AlphaGo Zero
6. Gumbel AlphaZero

AlphaZeroの課題

- AlphaZeroの要求計算資源量は非常に膨大

- 小林1人、PC1台で囲碁AIを作ろうとするとざっくりこんな感じ…

	マシン費用	電気代	開発期間	得られる強さ
AlphaGo以前	¥100,000(※1)	¥45,000	3ヶ月	ネット碁2段
AlphaGoの手法	¥600,000(※2)	¥360,000	1年	プロ級以上
AlphaZeroの手法	¥600,000(※2)	¥36,000,000	100年	超人級

マシンのスペックは以下を想定
(※1) 8コアCPU、メモリ16GB、SSD500GB
(※2) 16コアCPU、メモリ64GB、SSD2TB+HDD6TB
ハイエンドゲーミングGPU

実際にAlphaZeroと同じ程度の強さにするだけであれば、様々な工夫で1～2年くらいでできそう

なぜこんなに時間がかかる？

- 自己対戦でのデータ生成がリソース使用量の大部分を占める
  - 探索1回あたりニューラルネットワークの計算を1回する必要がある
  - 生成すべきデータ数が数百万局オーダー

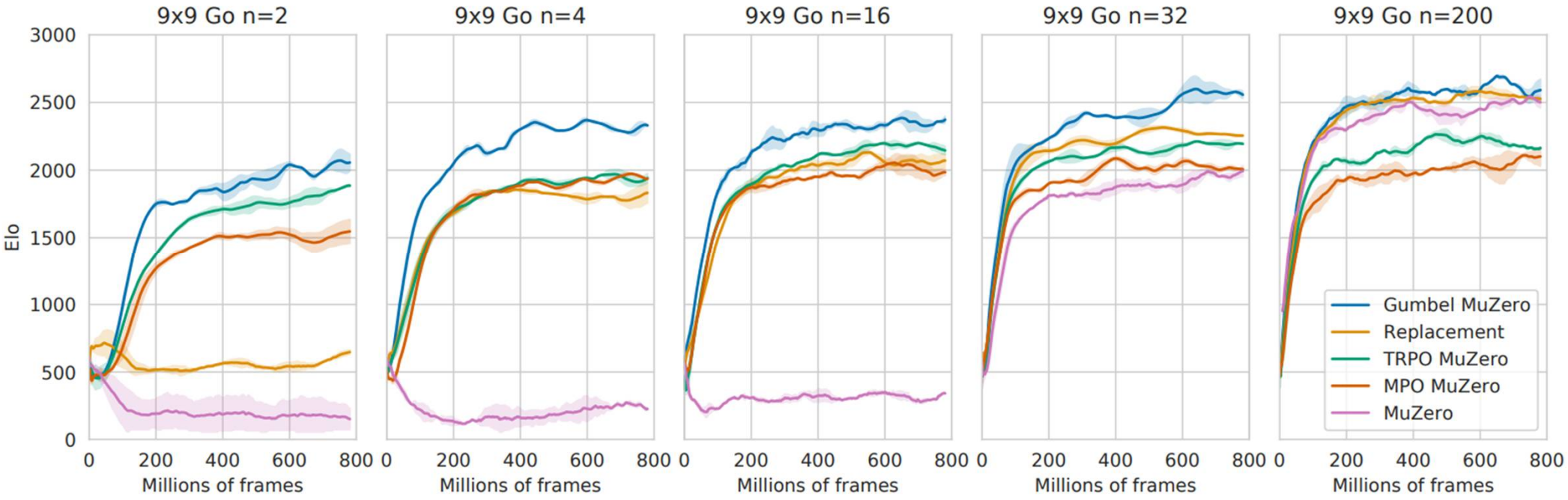
  → ニューラルネットワークの計算は重い上に実行回数も多い

- ニューラルネットワークの計算回数を減らせば、学習時間を短縮できるか？
  - 1手あたりの探索回数を減らす

    → 探索回数が少ないと探索回数の分布に対するノイズの影響が大きくなる
  - 1局あたり学習データとして使用する局面数を増やす

    → Valueが過学習を起こして学習に失敗する

自己対戦の探索回数の重要性



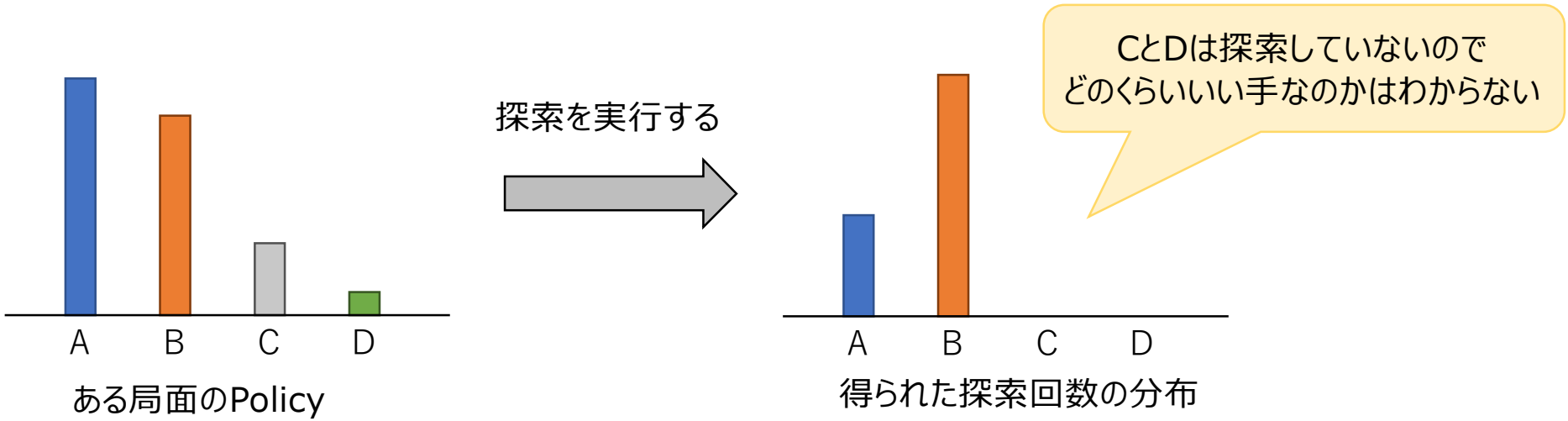
"Policy Improvement by Planning with Gumbel", Figure 2

探索回数が少ないと強くなるどころか弱くなる場合がある

探索回数が少ないとなぜ弱くなる？

- Policyに付与されたノイズの影響が大きい
  → Valueを踏まえたいい感じの探索回数の分布が得られない

- 探索されない手が増える
  → 探索していない手は全て等しく0としてPolicyを学習



Gumbel AlphaZero

- 自己対戦の探索回数が少なくても学習が進むように変形したAlphaZeroの改良版

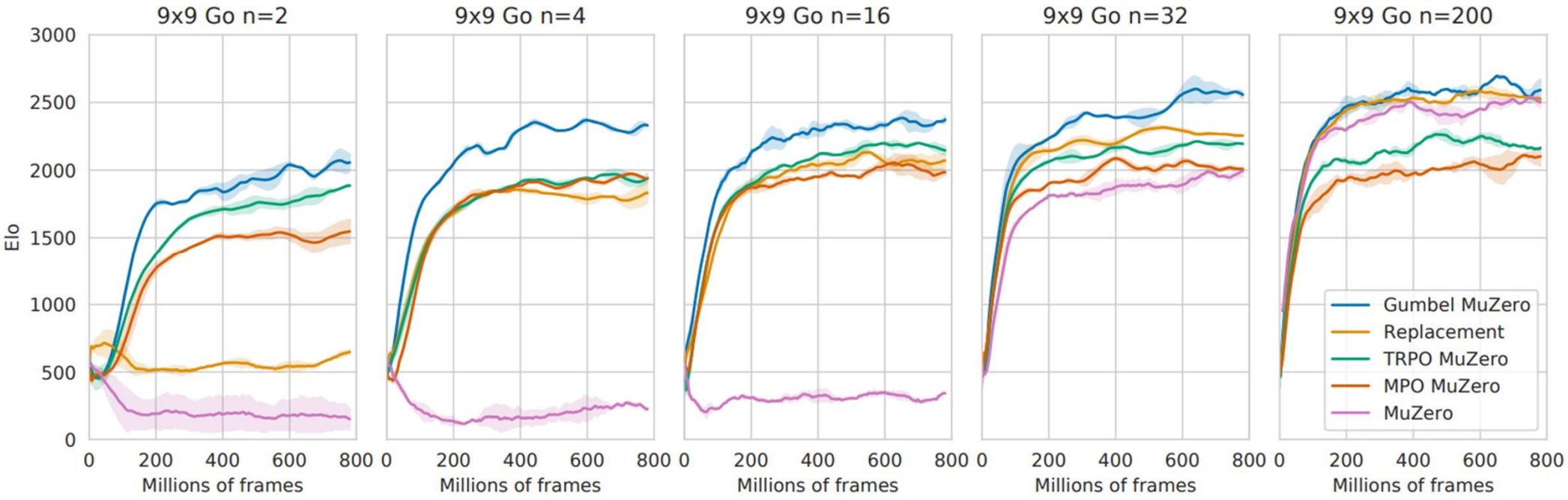
	AlphaZero	Gumbel AlphaZero
探索アルゴリズム	PUCT	SHOT
Policyに付与するノイズ	Dirichlet分布	Gumbel分布
Policyのターゲット	探索回数の分布	Improved Policy

- 少ない探索回数でも強くなる
  （ただしプレイヤとしては弱くなるので最終的な強さは弱め）

（詳細を解説すると非常に時間がかかるのでここでは説明を割愛）

詳細はPolicy improvement by planning with Gumbelを参照

探索回数と棋力向上の関係



“Policy Improvement by Planning with Gumbel”, Figure 2

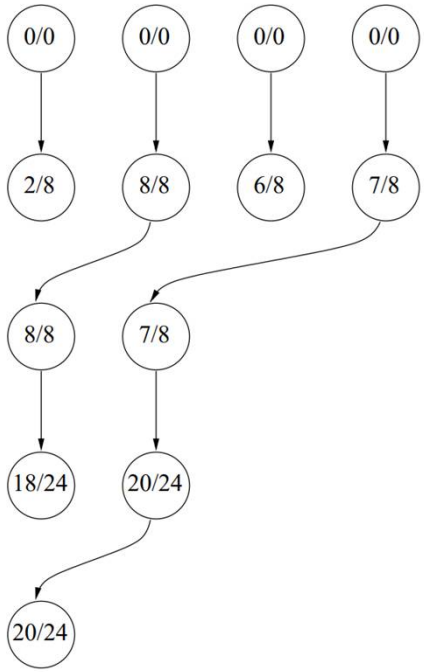
Gumbel AlphaZeroは探索回数が2回でもちゃんと強くなる

SHOT (Sequential Halving applied to trees)

- PUCTアルゴリズムはPUCB値に従って着手を選ぶ
  → 探索する手の幅が局面に依存して変わる
    - 必然の1手があったときに探索が集中（読みの幅が狭い）
    - いい手が見つからないときにたくさん試す（読みの深さが浅い）

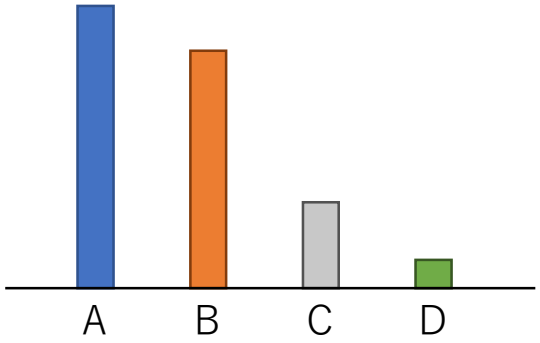
- SHOTは読む幅を決めて探索する
  → 読みの幅と深さが安定する

- 探索回数が少ないときはPUCTよりSHOTの方が強い



Sequential Halving Applied to Trees, Figure 1

Improved Policy

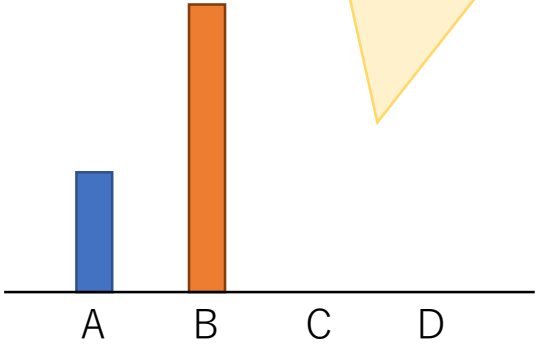


ある局面のPolicy

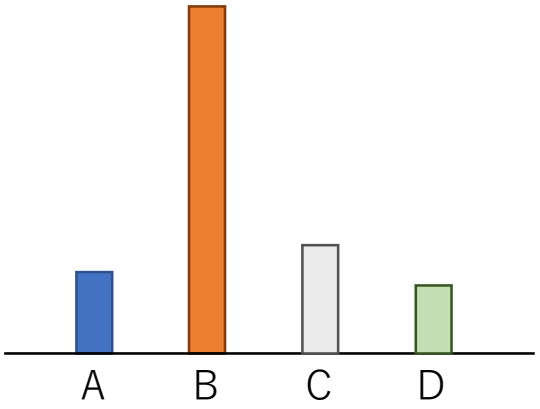
探索を実行する

探索を実行してImproved Policyを求める

このままだとC, Dの良し悪しがわからない



得られた探索回数の分布



Improved Policy

Improved Policyの求め方

探索した手 ： ValueでPolicyの値を補正

未探索の手 ： 探索した手のValueとPolicyからその手のPolicyを推定

※ 最低2つの手を探索していないと未探索の手の推定ができない

TamaGo

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

TamaGoとは

- 最新の囲碁AIの技術を取り入れた教育・学習用の囲碁AI

- 開発言語はPython

- 機能的な特徴は以下の通り
  - ニューラルネットワークを用いた教師あり学習・強化学習をサポート
  - 大会参加に必要な一通りの機能を実装済み
    * GTP (Go Text Protocol) 対応
    * 持ち時間を考慮した思考時間の配分
    * ニューラルネットワークを使用したモンテカルロ木探索

- ソースコード：https://github.com/kobanium/TamaGo

TamaGo開発のモチベーション

- 最新の囲碁AIを改造するにはハードルが高い
  1. C++とPythonのスキルが必要

  2. 開発コミュニティのコミュニケーションは英語

- 個人環境で囲碁AIの強化学習を試せるようにしたい
  （使用する定石や手筋が変化するのを見て楽しめるように）

- 最新のRayの開発ノウハウの公開

TamaGo開発のモチベーション

- 最新の囲碁AIを改造するにはハードルが高い
  1. C++とPythonのスキルが必要
     → 速度の低下を許容してPythonのみで実装
  2. 開発コミュニティのコミュニケーションは英語
     → 小林が開発することによって日本語でのアクセスを容易化

- 個人環境で囲碁AIの強化学習を試せるようにしたい
  （使用する定石や手筋が変化するのを見て楽しめるように）
  → Gumbel AlphaZeroの手法を使うことで、要求計算資源量を低減

- 最新のRayの開発ノウハウの公開
  → 学習時の設定や各種ハイパーパラメータをTamaGoに流用

TamaGoのプロジェクト構成

#	フォルダ名	概要
1	archive	自己対戦生成棋譜格納場所
2	board	碁盤の実装スクリプト
3	common	共通利用するスクリプト
4	data	中間データ格納場所
5	doc	ドキュメント
6	gtp	GTP実装スクリプト
7	img	ドキュメントで使用する画像
8	mcts	MCTS実装スクリプト
9	model	ニューラルネットワークモデルファイル格納場所
10	nn	ニューラルネットワーク実装スクリプト
11	selfplay	自己対戦実装スクリプト
12	sgf	SGFファイルの入出力実装スクリプト

#	スクリプト名	概要
1	get_final_status.py	自己対戦の結果を補正するためのスクリプト
2	learning_param.py	学習で使用するハイパーパラメータ定義
3	main.py	GTPクライアントのエントリーポイント
4	pipeline.sh	強化学習パイプラインの定義
5	program.py	プログラム情報の定義
6	selfplay_main.py	自己対戦のエントリーポイント
7	train.py	学習処理のエントリーポイント

赤字は開発で主に編集するファイル

ちゃんと動く囲碁AIを作るためには…

- 大きく分けて3つの要素が必要
    1. GTP (Go Text Protocol) 処理部
       → gtpフォルダ内のスクリプト
    2. 碁盤を表すデータとその処理部
       → boardフォルダ内のスクリプト
    3. 思考処理部
       → mctsフォルダ内のスクリプト（モンテカルロ木探索の実装）
          nnフォルダ内のスクリプト（ニューラルネットワークの実装）

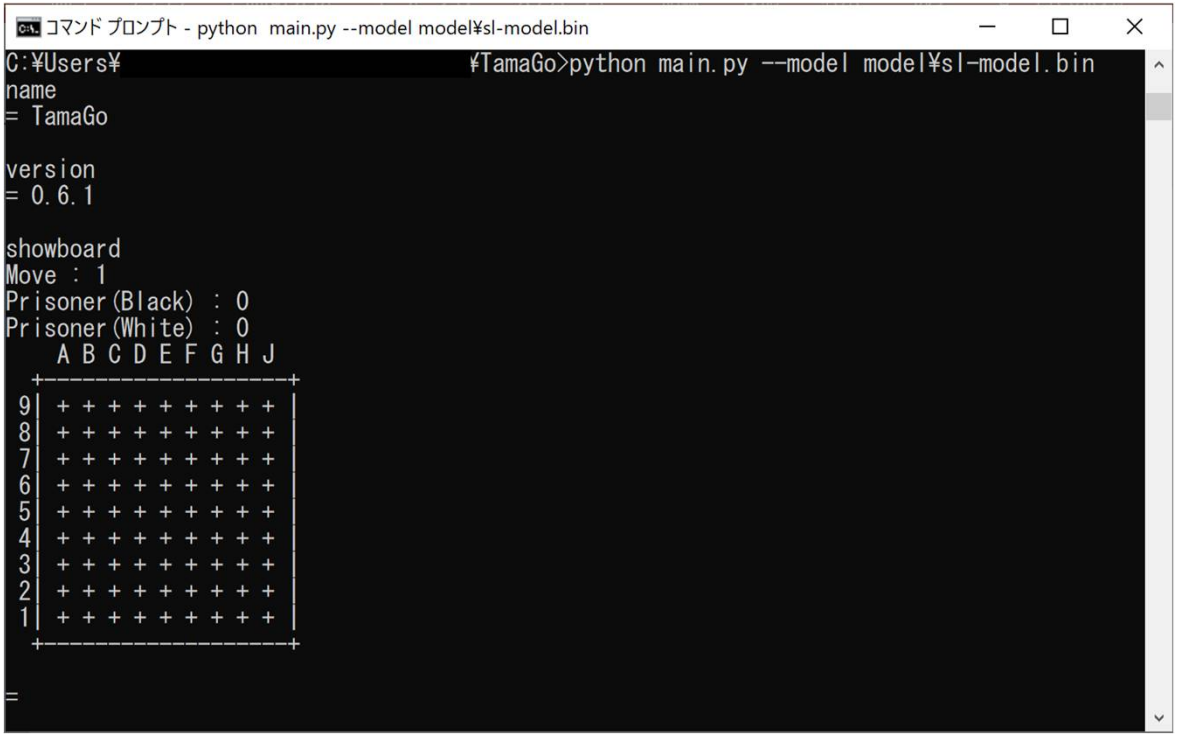
時間が足りないので3のモンテカルロ木探索の実装については割愛

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

Go Text Protocol

- 囲碁AIで使用される通信規約



TamaGoの実行例

- GTPに対応することでGoGUI, Lizzie, Sabakiなどで思考エンジンとして使用可能

TamaGoがサポートするGTPコマンド一覧（一部抜粋）

#	コマンド	実行される処理
1	version	プログラムのバージョンの表示
2	protocol_version	プログラムがサポートするGTPバージョンの表示
3	name	プログラムの名前の表示
4	quit	プログラムの終了
5	list_commands	プログラムがサポートするGTPコマンド一覧の表示
6	play	指定した色の石を指定した座標に着手
7	genmove	指定した手番で思考して着手生成
8	clear_board	碁盤の初期化
9	boardsize	碁盤の大きさの変更
10	time_left	指定した手番の残り時間の設定
11	komi	コミの設定
12	showboard	現在の局面の表示

→ 対局に必要な各種GTPコマンドは実装済み

実際に動かしてみる

name, version, protocol_versionの応答

name, version, protocol_versionの応答はprogram.pyの内容を参照
【注意点】PROTOCOL_VERSIONのみ変更不可

```
PROGRAM_NAME="TamaGo"
PROTOCOL_VERSION="2"

# Version 0.0.0 : ランダムプレイヤの実装。
# Version 0.1.0 : SGFファイルの読み込み処理の実装。load_sgfコマンドの対応。
#                 配石パターンのデータ構造の追加。眼の判定、上下左右の空点判定等改善。
#                 着手履歴、Zobrist Hash、超劫の判定の実装。
# Version 0.2.0 : ニューラルネットワークの教師あり学習の実装。
#                 Policy Networkを使用した着手生成ロジックの実装。
# Version 0.2.1 : Residual Blockの構造を修正。学習の再実行。
# Version 0.3.0 : モンテカルロ木探索の実装。
# Version 0.3.1 : モンテカルロ木探索のValue更新処理のバグ修正。komi, get_komiコマンドのサポート。
# Version 0.4.0 : Sequential Halving Applied to Trees (SHOT) の実装。
# Version 0.5.0 : 探索時間の制御、time_left、time_settingsコマンドのサポート。
# Version 0.6.0 : Gumbel AlphaZero方式の強化学習の実装。ネットワークの構造改善。
# Version 0.6.1 : --batch-sizeオプションの追加。
VERSION="0.6.1"
```

GTPクライアント起動時のオプション

#	オプション	概要
1	--size	碁盤のサイズの指定
2	--superko	Positional Super Koルールの有効化
3	--model	ニューラルネットワークモデルファイルの指定
4	--use-gpu	ニューラルネットワーク推論のGPU実行有効化
5	--policy-move	Policyの分布に従って着手生成（デバッグ用）
6	--sequential-halving	SHOTで探索を実行（デバッグ用）
7	--komi	コミの指定
8	--visits	1手あたりの探索回数の指定
9	--const-time	1手あたりの思考時間の指定（秒）
10	--time	持ち時間の指定（秒）
11	--batch-size	ニューラルネットワーク推論時のミニバッチサイズの指定

GTPクライアントとしての使い方

main.pyがGTPクライアントのエントリーポイント

[1手あたり400回探索して着手生成]

python main.py --model model¥sl-model.bin --visits 400

[コンピュータ囲碁大会で使うときの一例]

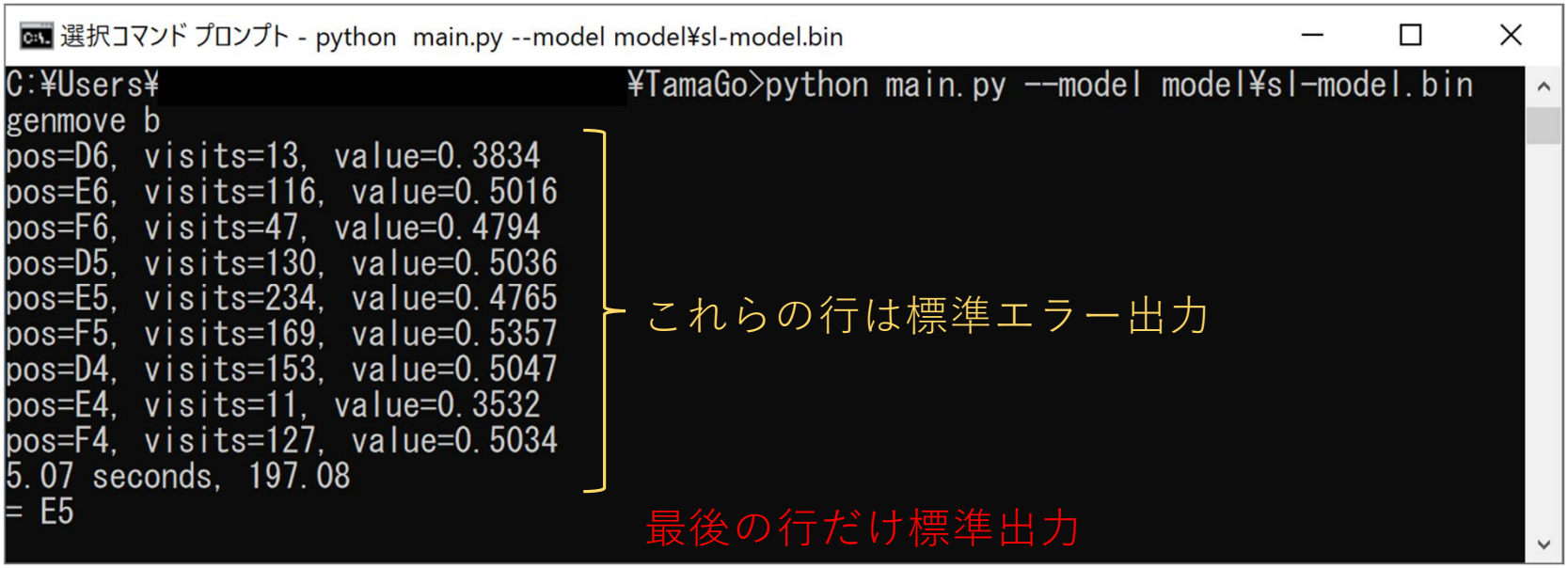
python main.py --model model¥sl-model.bin --time 600 --use-gpu true –-batch-size 13 –superko true

【注意点】

--time, --const-time, --visitsオプションを同時指定した場合、1つだけ有効になる
  --timeオプション指定時は--const-time, --visitsは無視される
  --const-timeオプション指定時は--visitsは無視される

開発時の注意点

- コンソール出力の際にはGTPの応答以外は標準エラー出力にする
  → 開発時に追加するコンソール出力は原則標準エラー出力
  common/print_console.pyにあるprint_err関数を使用すると良い

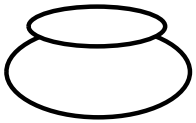


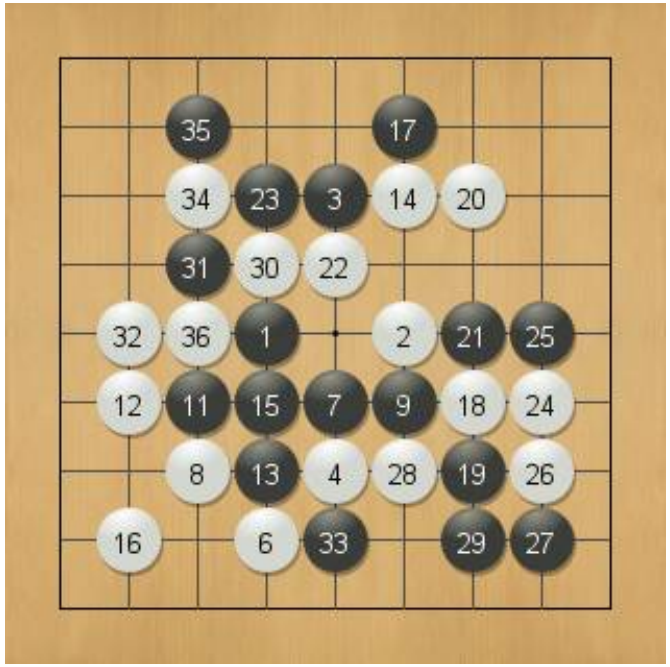
TamaGo

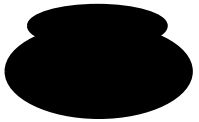
1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

局面情報で表現したいこと

- 盤上の石の並び ＋ 打ち上げた石 ＋ …







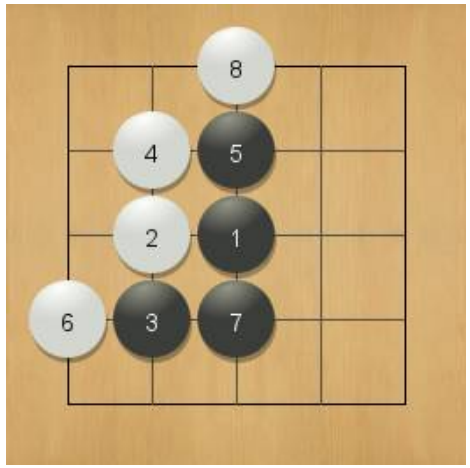
- 他にも様々な処理の補助になる情報はいろいろある
  （着手の記録、コウの発生個所など）

碁盤の表現

- 碁盤のデータ構造はboard/go_board.pyのGoBoardクラスで定義

- 各交点の状態はboard/stone.pyのStoneクラスで表現
  EMPTY：空点
  BLACK：黒石
  WHITE：白石
  OUT_OF_BOARD：盤外

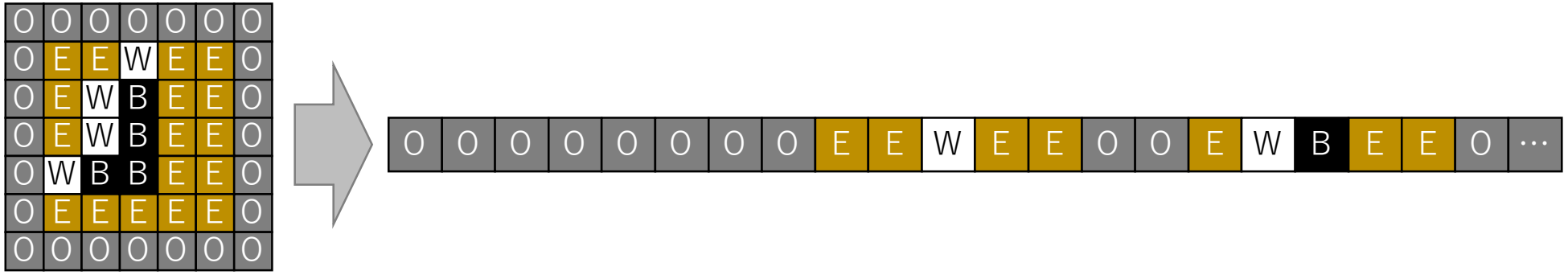
処理を楽にするために盤外で囲う



O	O	O	O	O	O	O
O	E	E	W	E	E	O
O	E	W	B	E	E	O
O	E	W	B	E	E	O
O	W	B	B	E	E	O
O	E	E	E	E	E	O
O	O	O	O	O	O	O

碁盤の表現

- boardは1次元配列で実装（碁盤の座標の始点は左上）



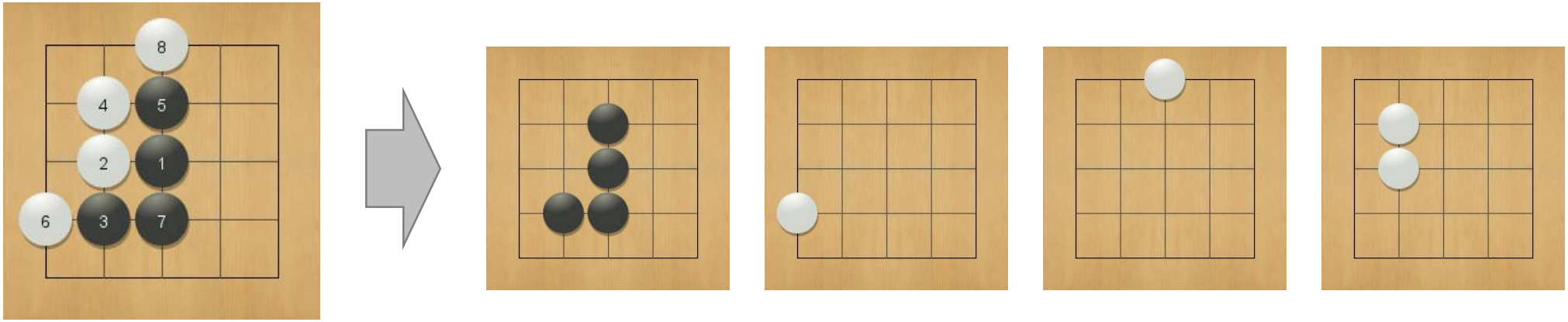
- 2次元座標で表現したものを1次元配列のインデックスに変換するのがPOS
  ex. POS(2,5) = 2 + (1 + 5 + 1) x 5 = 37

- 盤外を除いたインデックスをまとめたものがonboard_pos

石の塊をまとめて扱う

- 上下左右に隣接した同じ色の石は運命共同体
  → 打ち上げられるときは全て取り上げられる（一部だけ打ち上げられることはない）

- この石の連なりを”連（れん）”と呼称

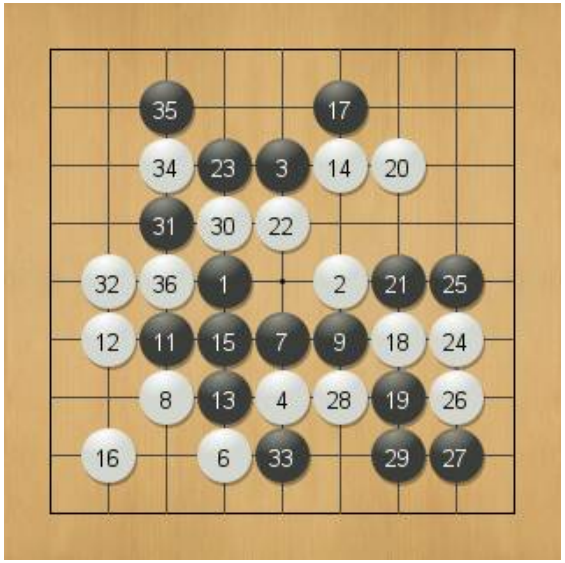


左の局面は4つ連がある

連のデータ構造

- 盤上にあるの連を統合的に管理するのがboard/string.pyのStringDataクラス

- 1つの連に関する情報を保持するのがboard/string.pyのStringクラス



Stringクラスのメンバ

#	メンバ変数	保持する情報	値を取得するメソッド
1	color	連を構成する石の色	get_color
2	libs	連が持つ呼吸点の数	get_num_liberties
3	lib	連が持つ呼吸点の座標	get_liberties
4	neighbors	隣接する敵の連の数	（なし）
5	neighbor	隣接する敵の連のID	get_neighbors
6	origin	連を構成する石の始点	get_origin
7	size	連を構成する石の個数	get_size
8	flag	連の存在フラグ	exist

StringDataクラスのメンバ（意味のあるもののみ抜粋）

#	メンバ変数	保持する情報	値を取得するメソッド
1	string	各連の情報（Stringクラス）	（なし）
2	string_id	各交点の連ID	get_id
3	string_next	各交点の連の繋がり	get_stone_coordinates

- get_id
  引数：座標（1次元表現）
  戻り値：指定した座標の連ID（連がないときは0）
- get_stone_coordinates
  引数：連ID
  戻り値：指定した連IDを構成するすべての石の座標
  【注意点】指定した連IDの連が存在することを必ず確認すること

GoBoardクラスのメンバ変数

#	メンバ	保持する情報・役割
1	board_size	碁盤のサイズ
2	board_size_with_ob	盤外を含む碁盤のサイズ
3	komi	コミの値
4	board	各交点の状態
5	pattern	8近傍の配石パターン
6	strings	連の情報
7	record	着手の履歴
8	onboard_pos	盤上のインデックス列
9	moves	現在の手数

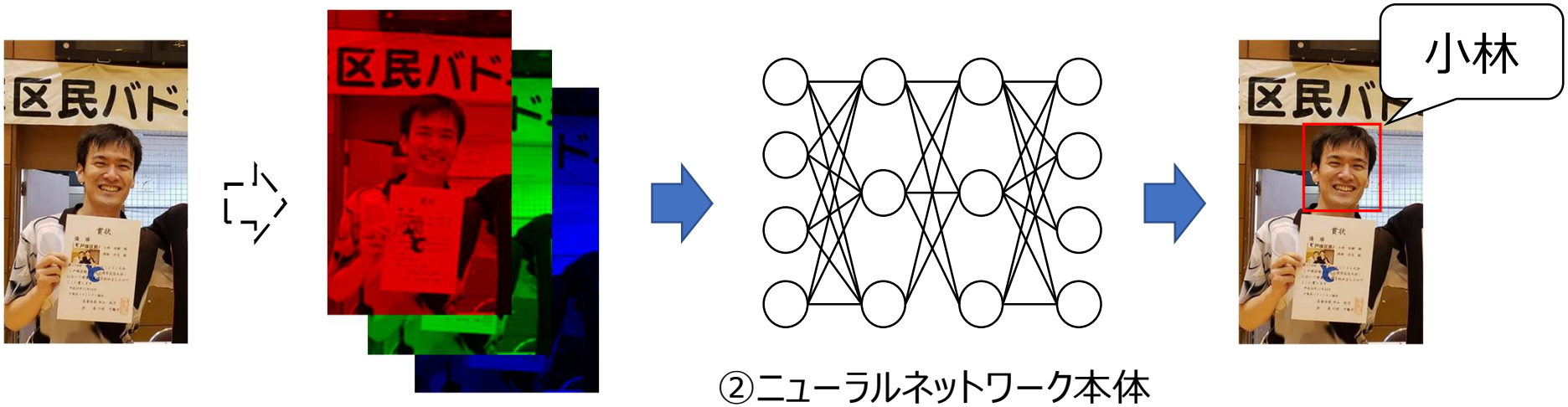
#	メンバ	保持する情報・役割
10	coordinate	座標フォーマット変換処理
11	ko_moves	コウが発生した手数
12	ko_pos	コウが発生した座標
13	prisoner	各手番の打ち上げた石の数
14	positional_hash	現在の局面のハッシュ値
15	check_superko	超コウ確認フラグ
16	board_start	2次元表現での盤の左端（上端）のインデックス
17	board_end	2次元表現での盤の右端（下端）のインデックス
18	sym_map	座標の8対称変換マップ

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

ニューラルネットワークの構成

- ニューラルネットワークを実装するのためには下記2つの要素の実装が必要
  ①. 入力特徴生成処理
  ②. ニューラルネットワーク本体



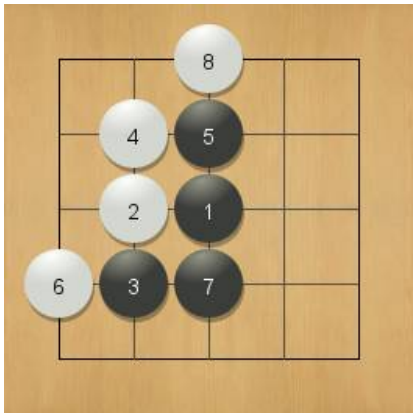
TamaGoの入力特徴

- 入力特徴生成処理はnn/feature.pyのgenerate_input_planes関数で実装

- 生成される入力特徴は下記6種類

#	特徴	概要
1	空点	石がない交点は1、そうでなければ0
2	自分の石	自分の石がある交点は1、そうでなければ0
3	相手の石	相手の石がある交点は1、そうでなければ0
4	直前の着手箇所	直前の相手の着手箇所は1、そうでなければ0
5	直前の手がパスか否か	直前の相手の着手がパスならすべて1、そうでなければすべて0
6	手番の色	黒番ならすべて1、白番ならすべて-1

入力特徴の一例



①空点

1	1	0	1	1
1	0	0	1	1
1	0	0	1	1
0	0	0	1	1
1	1	1	1	1

②自分の石

0	0	0	0	0
0	0	1	0	0
0	0	1	0	0
0	1	1	0	0
0	0	0	0	0

③相手の石

0	0	1	0	0
0	1	0	0	0
0	1	0	0	0
1	0	0	0	0
0	0	0	0	0

④直前の手

0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

⑤パスか否か

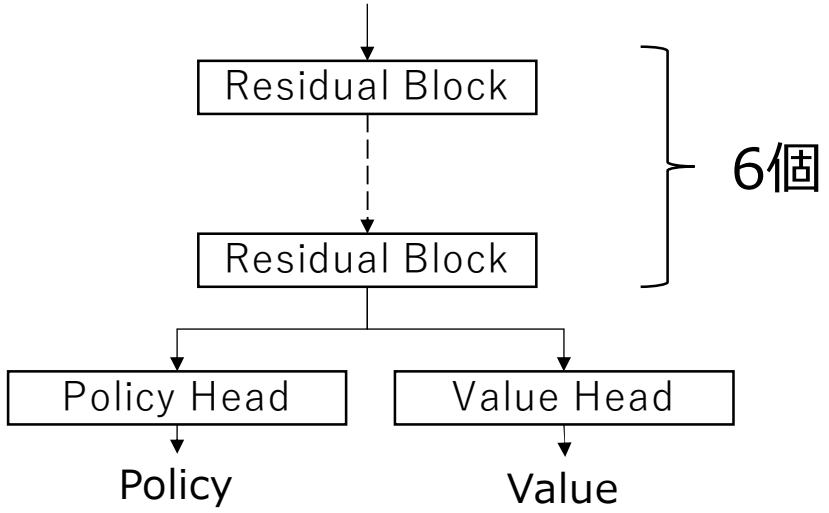
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

⑥手番の色

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

ニューラルネットワークの実装

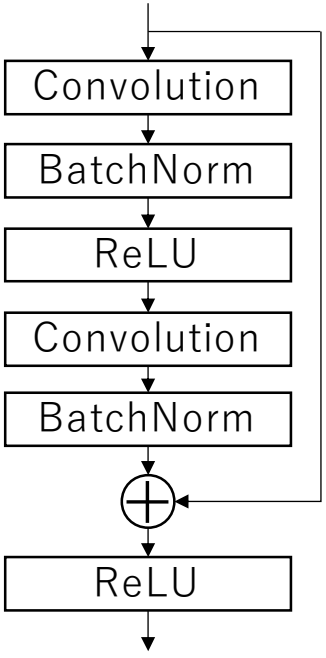
- ニューラルネットワーク本体はnn/network/dual_net.pyのDualNetクラスで実装
- デフォルトの実装はResidual Blockを6段重ねたResidual Network



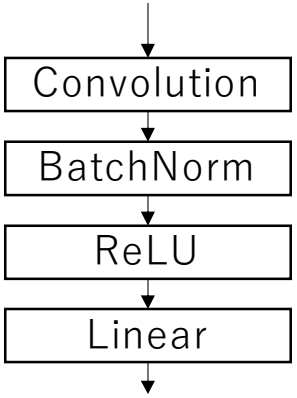
Residual Networkの構造

各構成要素の定義

- DualNetクラスの各構成要素はそれぞれ下記ファイルのクラスで定義
  Residual Block：nn/network/dual_net.pyのResidualBlockクラス
  Policy Head：nn/network/head/policy_head.pyのPolicyHeadクラス
  Value Head：nn/network/head/value_head.pyのValueHeadクラス

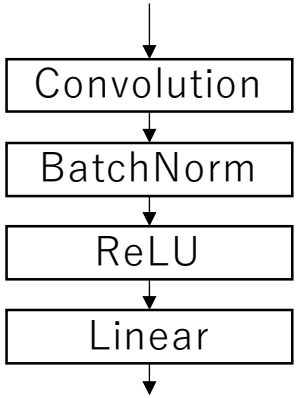


Residual Blockの構造



82出力

Policy Headの構造

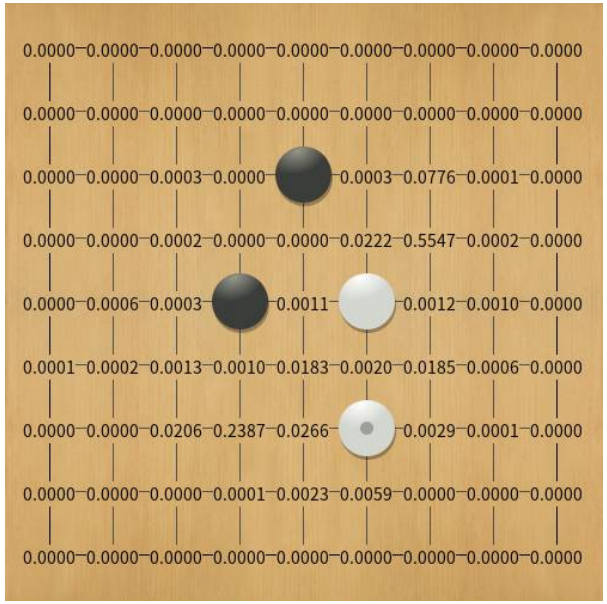


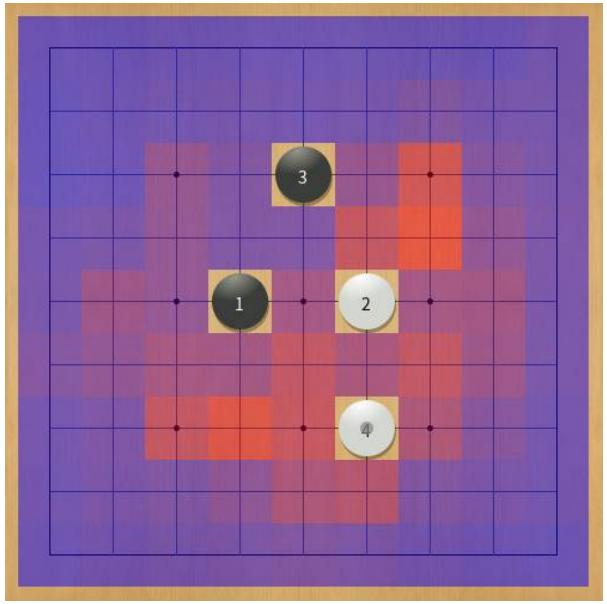
3出力

Value Headの構造

Policy Headの出力

- Policy Headの出力は各交点の着手の良し悪しを数値化したもの
  （推論時はSoftmax関数を適用するので出力値の合計は1になる）





- 出力の個数は交点の個数 + パス（9路盤なら81 + 1 = 82個）

Value Headの出力

- Value Headの出力は現局面の勝ち、引き分け、敗けになる確率
  ex.
  勝ち：0.43
  引き分け：0.04
  敗け：0.53
  → 引き分けを0.5勝とみなすと勝率44.0%

DualNetクラスのメソッド

#	メソッド	Softmax関数の適用（Policy）	Softmax関数の適用（Value）	使用するタイミング
1	forward	無	無	強化学習実行時
2	forward_for_sl	有	無	教師あり学習実行時
3	forward_with_softmax	有	有	（特になし）
4	Inference	有	有	MCTSの探索時
5	Inference_with_policy_logits	無	有	Gumbel AlphaZeroの自己対戦時の探索時

※ inferenceとinference_with_policy_logitsはデバイス間のデータ転送も含む

Policy HeadやValue Headの出力にはSoftmaxをかけないようにしておくこと

DualNetクラスで定義するパラメータ

- DualNetクラスの構造を決めるパラメータとしてfilters, blocksの2つがある

#	パラメータ	パラメータの意味	デフォルト値
1	filters	Convolution Layerのフィルタ数（≒パラメータ数）	64
2	blocks	Residual Blockのブロック数	6

- filtersもblocksも値を大きくするとPolicyやValueの予測が正確になる
  → 計算量が増えるので、計算速度は遅くなる

- おおよそ各パラメータに対する計算時間の増加は下記の通り
  filters：2倍にすると計算時間4倍（いろいろあって実際は4倍よりは小さい）
  blocks：2倍にすると計算時間は2倍

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

TamaGoの教師あり学習

- TamaGoはニューラルネットワークを使用した教師あり学習をサポート

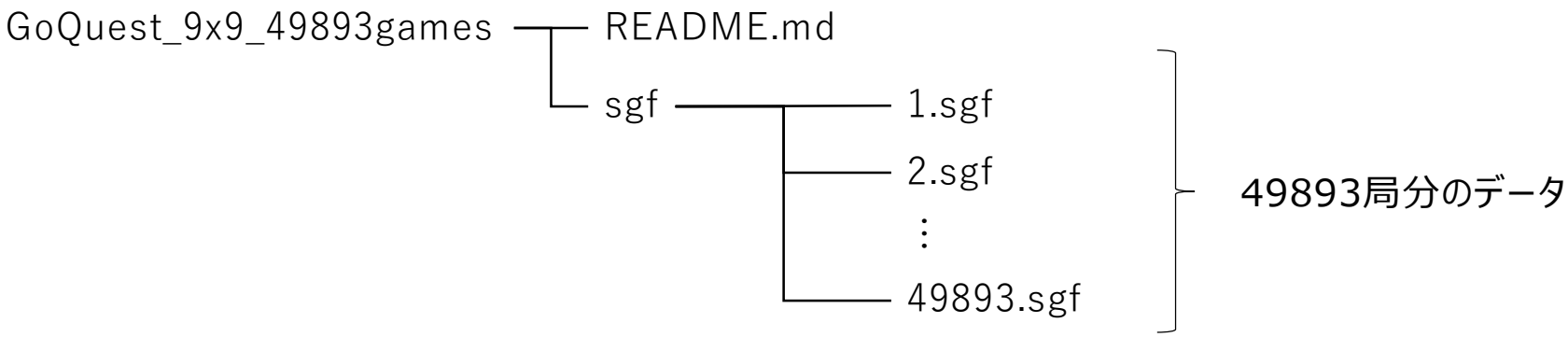
- 必要なのは教師データとなるSGFファイル
  SGF：囲碁の棋譜を記録するのにつかわれるファイル形式（.sgf）

- CPUを使用した実行、GPUを使用した実行のどちらもサポート
  → CPUだけでもニューラルネットワークの規模が小さければ実行可能
  試行錯誤の反復を早めるためにGPUの使用を強く推奨

教師あり学習で使用するデータ

- 下記URLから9路盤の棋譜49,893局分のSGFファイルを配布
  http://computer-go-ray.com/file/GoQuest_9x9_49893games.zip
  （データを用意するにあたり囲碁クエスト開発者の棚瀬さんにご協力いただきました）

- GoQuest_9x9_49893games.zipを展開すると以下のような構成のフォルダが生成される



教師あり学習の実行方法

- 教師あり学習を実行する場合はtrain.pyを実行する
  （train.pyは教師あり学習、強化学習共通のエントリーポイント）

#	コマンドライン オプション	概要	デフォルト値	備考
1	--kifu-dir	教師あり学習に使用するSGFファイルを格納したディレクトリのパスの指定	（なし）	
2	--size	碁盤のサイズの指定	BOARD_SIZE (=9)	基本的には指定不要
3	--use-gpu	GPU使用フラグの指定	True	
4	--rl	強化学習実行フラグの指定	False	教師あり学習の際は必ずFalseを指定
5	--window-size	（強化学習用のオプションのため後回し）	300000	教師あり学習では無視される

【実行の一例】

```
# python train.py --kifu-dir GoQuest_9x9_49893games¥sgf --use-gpu True --rl False
```

教師あり学習の設定項目

- 教師あり学習の設定項目（ハイパーパラメータ）はlearning_param.pyに定義

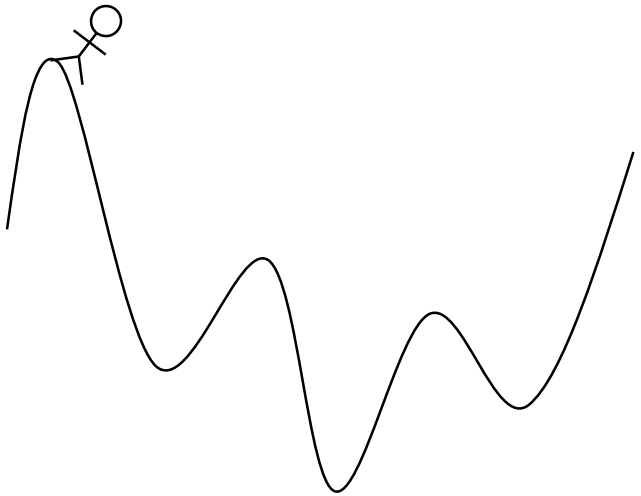
	ハイパーパラメータ	概要
1	SL_LEARNING_RATE	学習率の初期値
2	BATCH_SIZE	1ステップ学習するデータの個数
3	MOMENTUM	オプティマイザのモーメンタムパラメータ
4	WEIGHT_DECAY	L2正則化の重み
5	EPOCHS	データセットを学習する回数
6	LEARNING_SCHEDULE	学習率変更スケジュール
7	DATA_SET_SIZE	npzファイル1つ当たりのデータ数
8	SL_VALUE_WEIGHT	Policyに対するValueの損失関数の重み

- ある程度うまく動くことを確認している値を設定しているため、最初はデフォルトの値をそのまま使うことを推奨

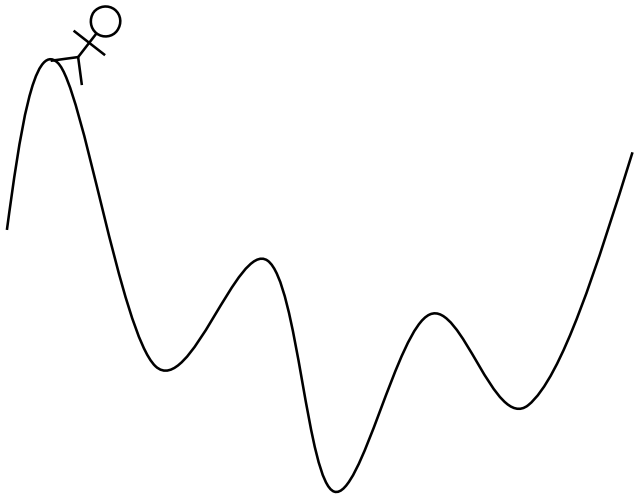
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



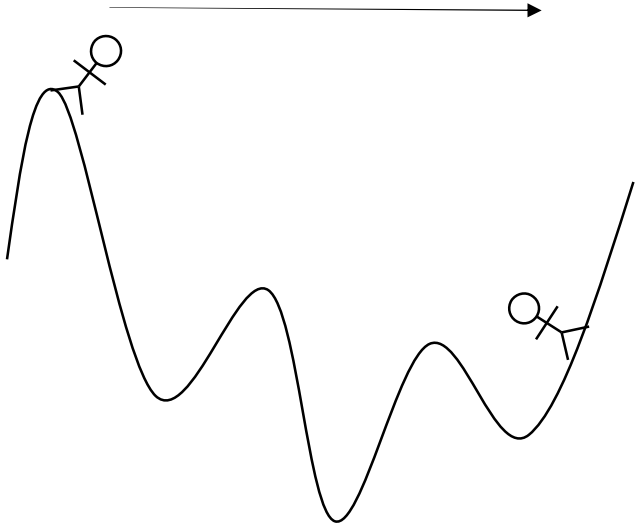
【学習率小】



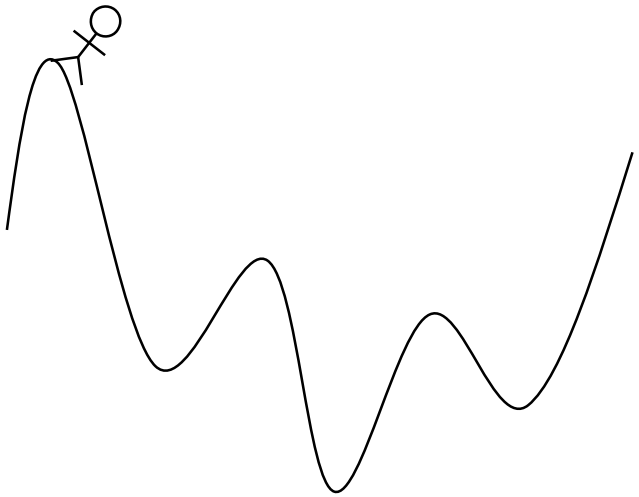
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



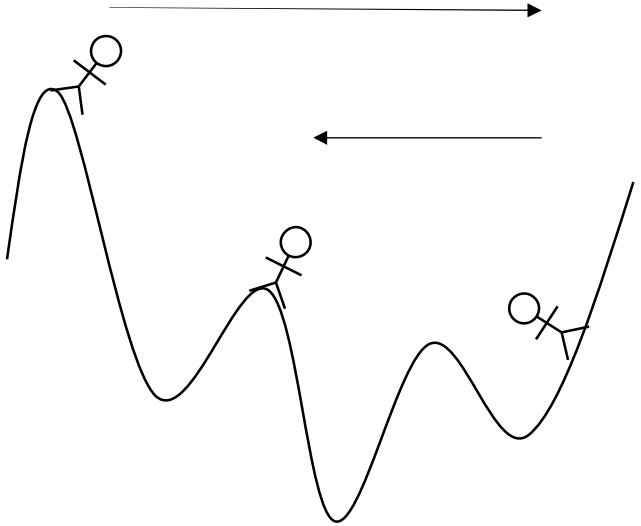
【学習率小】



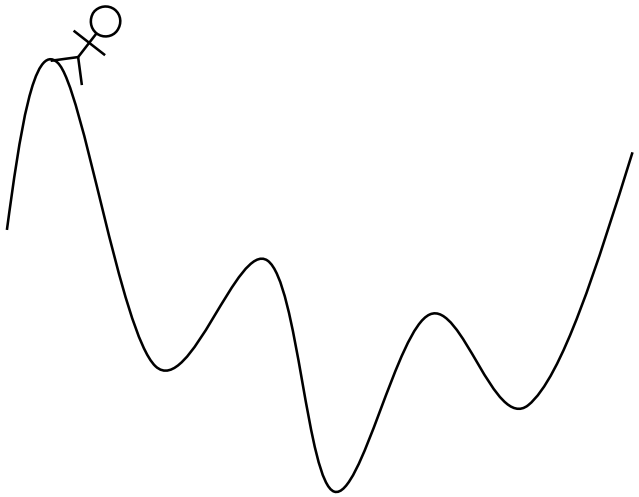
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



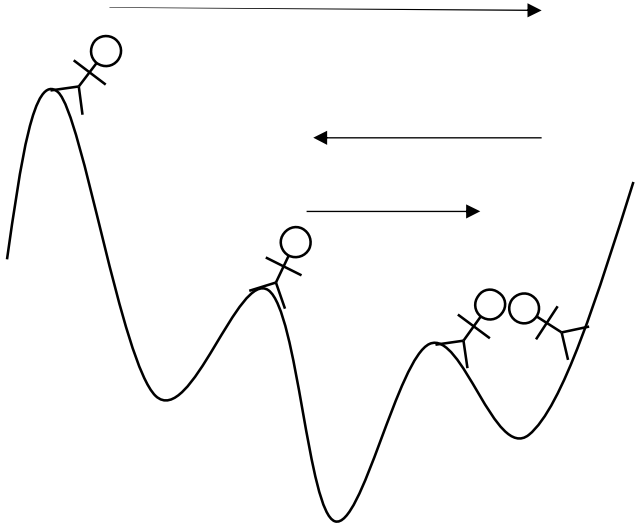
【学習率小】



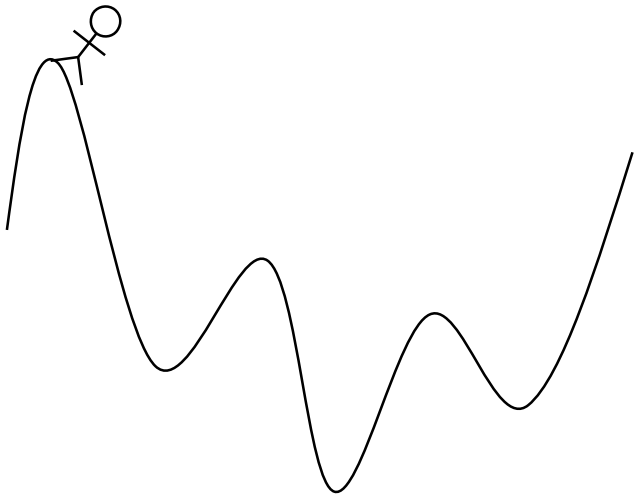
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



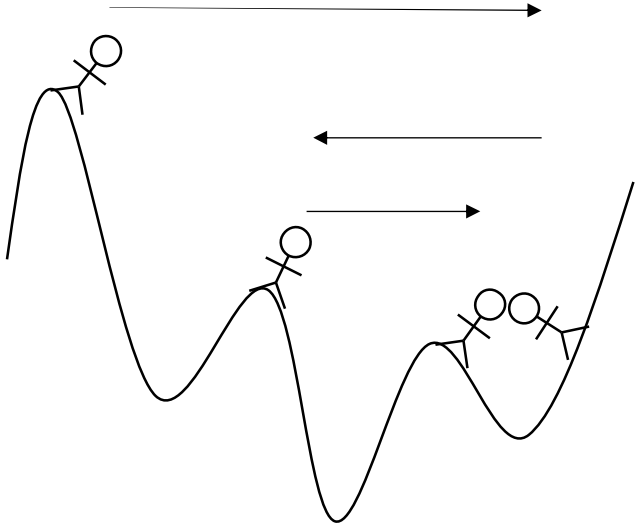
【学習率小】



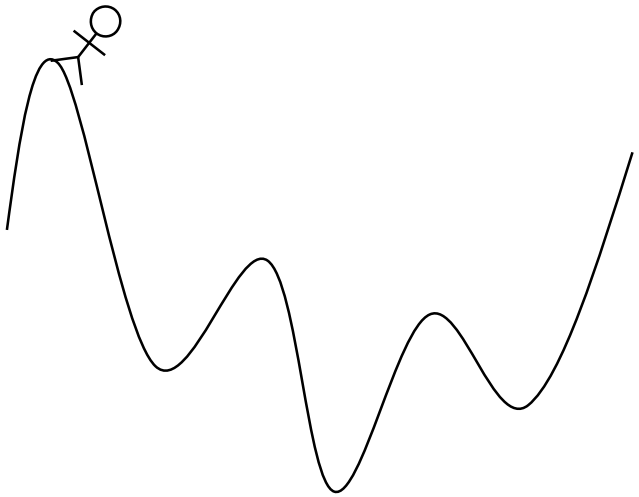
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



【学習率小】

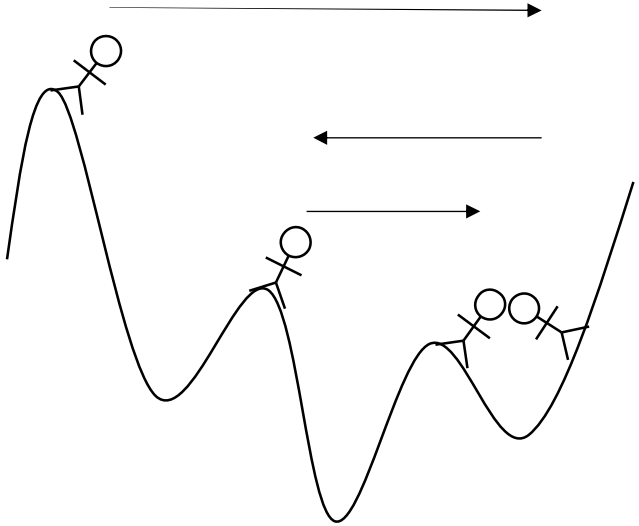


長所：目標まで速く近づける
短所：本当の目標までは近づけずに振動する

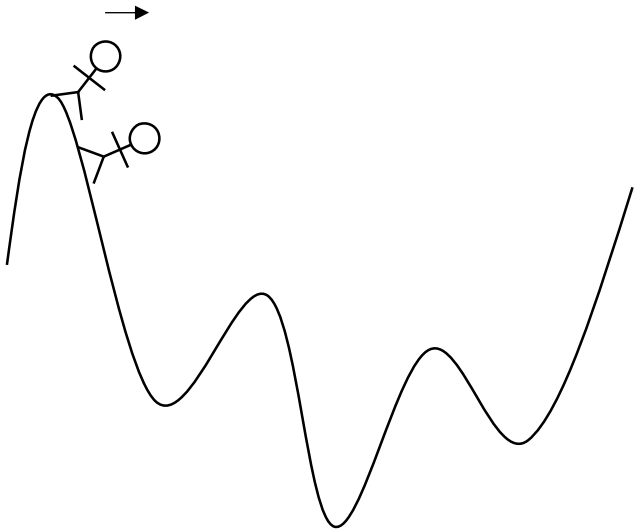
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



【学習率小】

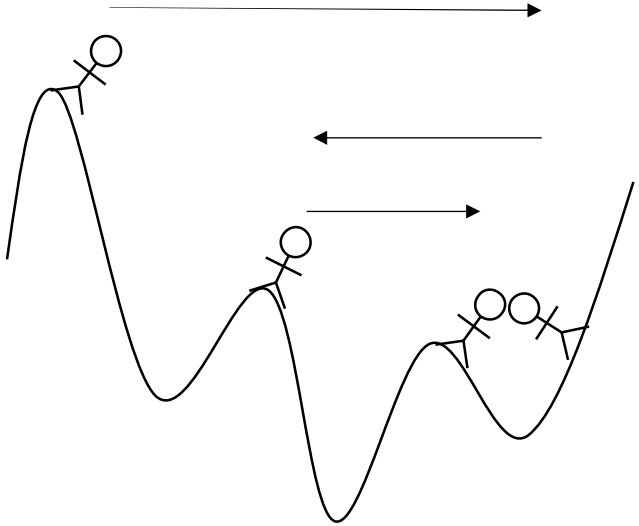


長所：目標まで速く近づける
短所：本当の目標までは近づけずに振動する

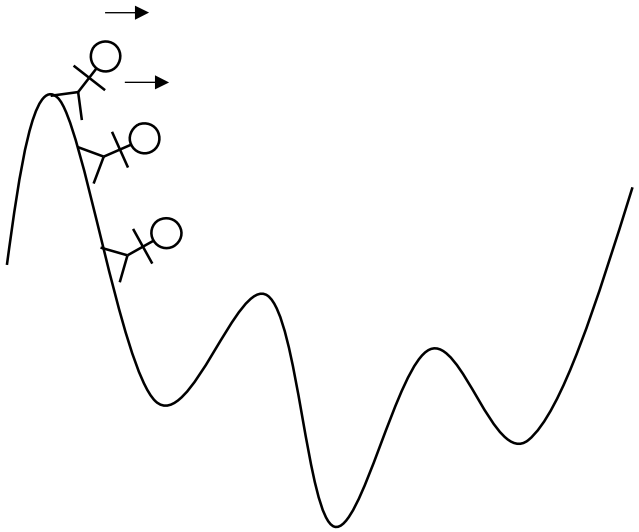
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



【学習率小】

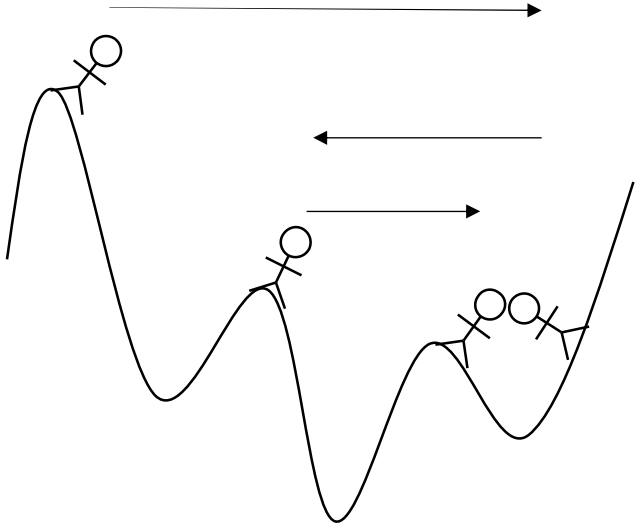


長所：目標まで速く近づける
短所：本当の目標までは近づけずに振動する

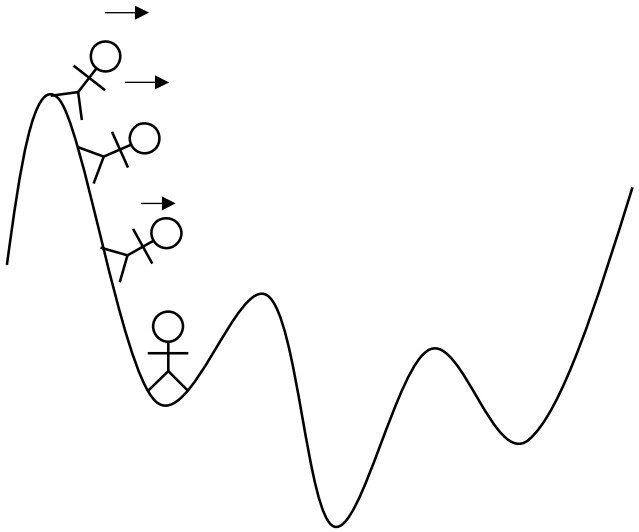
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



【学習率小】

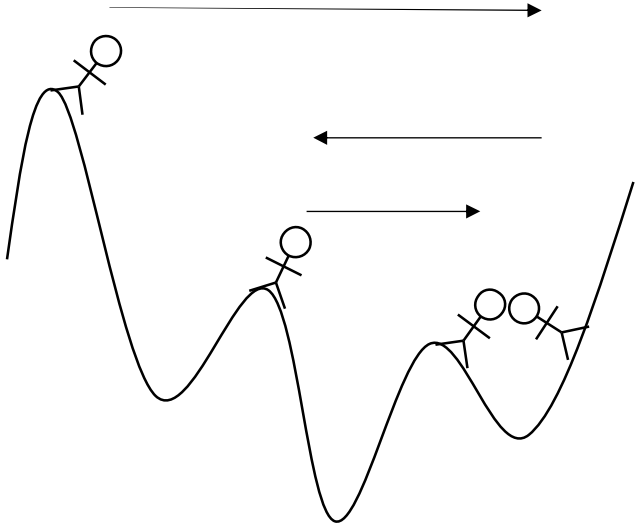


長所：目標まで速く近づける
短所：本当の目標までは近づけずに振動する

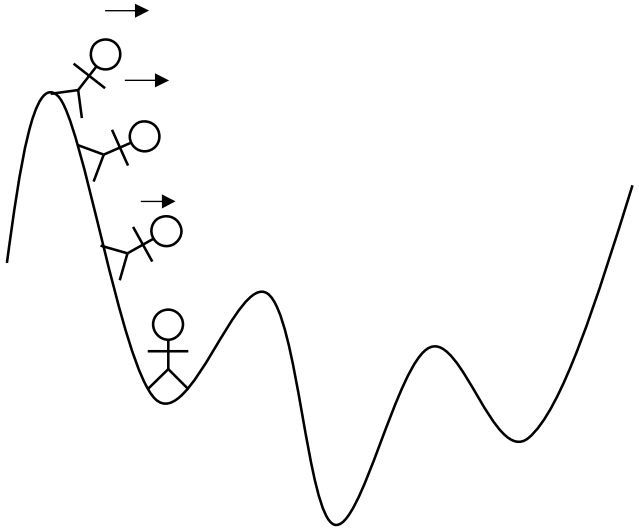
学習率とは？

- 学習率とは1つのミニバッチに対してどの程度学習するかを表した割合

【学習率大】



【学習率小】



長所：目標まで速く近づける
短所：本当の目標までは近づけずに振動する

長所：極小値に落ち着きやすい
短所：遅い、はまったときに抜け出せない

学習率を設定するコツ

- 最初は学習率を大きめに、そこから徐々に小さくしていくことがポイント

- デフォルト値だと学習率の調整スケジュールは
  1～ 5周目：0.01
  6～ 8周目：0.001
  9～10週目：0.0001
  11～15週目：0.00001
  となっている

- EPOCHSと合わせて調整すると良い
  （EPOCHSに合わせて学習時間が伸びるので注意）

```
# 教師あり学習実行時の学習率
SL_LEARNING_RATE = 0.01

# 学習の反復回数
EPOCHS = 15

# 学習率を変更するエポック数と変更後の学習率
LEARNING_SCHEDULE = {
    "learning_rate": {
        5: 0.001,
        8: 0.0001,
        10: 0.00001,
    }
}
```

その他のハイパーパラメータについて

- SL_VALUE_WEIGHT, MOMENTUM, WEIGHT_DECAYは変更不要
  （特にSL_VALUE_WEIGHTは大きくするとValueの学習がうまくいかないので注意）

- ニューラルネットワークのブロック数やフィルタ数を増やしたら、
  GPUメモリが溢れないようにBATCH_SIZEを調整
  （BATCH_SIZEはなるべく2のべき乗の大きい値にしておく方がよい）

- npzファイル生成でプログラムが異常終了する場合はDATA_SET_SIZEを小さくする

教師あり学習の処理内容

- 教師あり学習の処理は大きく2ステップからなる
    1. SGFファイルから学習用データをまとめた.npzファイルをdataフォルダに出力
       （--kifu-dirオプションが指定されたときのみ実行）
       → generate_supervised_learning_data関数
    2. dataフォルダ内の.npzファイルを使用して教師あり学習を実行
       → train_on_gpu関数、またはtrain_on_cpu関数
- 実行終了後、modelフォルダ以下にsl-model.binが生成される

- .npzファイルは再利用可能（ただし再利用の可否については下記に従う）

#	開発者が行う変更	.npzファイル再利用
1	使用するSGFファイルの変更	不可
2	ニューラルネットワークのブロック数・フィルタ数の変更	可能
3	ニューラルネットワークの入力特徴の変更	不可
4	各種学習に使うハイパーパラメータの変更	可能

教師あり学習の実行例

```
# python train.py --rl false --use-gpu true
Training data set : ['data/sl_data_0.npz', 'data/sl_data_1.npz', 'data/sl_data_10.npz', 'data/sl_data_11.npz',
'data/sl_data_12.npz', 'data/sl_data_13.npz', 'data/sl_data_14.npz', 'data/sl_data_15.npz', 'data/sl_data_16.npz',
'data/sl_data_17.npz', 'data/sl_data_18.npz', 'data/sl_data_2.npz', 'data/sl_data_3.npz', 'data/sl_data_4.npz',
'data/sl_data_5.npz']
Testing data set  : ['data/sl_data_6.npz', 'data/sl_data_7.npz', 'data/sl_data_8.npz', 'data/sl_data_9.npz']
epoch 0, data-0 : loss = 2.238818852812052, time = 78.59641933441162 sec.
            policy loss : 2.221566588103771
            value loss  : 0.8626131917238236
epoch 0, data-1 : loss = 1.792825085401535, time = 78.3811445236206 sec.
            policy loss : 1.7757179414629936
            value loss  : 0.855357280254364
...
epoch 14, data-14 : loss = 1.248982057750225, time = 79.46439051628113 sec.
            policy loss : 1.2350193989276885
            value loss  : 0.698132973343134
Test 14 : loss = 1.3059230829179287, time = 95.98968648910522 sec.
            policy loss : 1.2919065376631915
            value loss  : 0.7008273023441434
```

PolicyとValueの学習目標

- PolicyとValueはともにSGFファイルを目標に学習している
  Policy：SGFファイルに記録されている各局面の着手
  Value：SGFファイルに記録されている対局結果

- Valueは過剰適合しないようにPolicyに対して重み0.02で学習
  （0.02はSL_VALUE_WEIGHTに由来）

学習データとテストデータの分割

- デフォルトの設定では学習用データと検証用データを8:2で分割
  学習用データ：学習に使用するデータ
  検証用データ：学習に使用せず、学習の進行の評価に使うデータ

- データの分割はnn/learn.pyの下記行で処理

```
train_data_set, test_data_set = split_train_test_set(data_set, 0.8)
```

- 全部のデータを学習に使用したい場合は0.8を1.0にすればよい
  → 変更箇所がtrain_on_gpuとtrain_on_cpuの2か所あるため注意

TamaGo

1. TamaGoの概要
2. GTPクライアントとしてのTamaGo
3. TamaGoのデータ構造
4. ニューラルネットワークの実装
5. 教師あり学習
6. 強化学習

TamaGoの強化学習

- TamaGoはGumbel AlphaZero方式の強化学習をサポート
- 自己対戦でデータを生成するため、SGFファイルの準備は不要
- ニューラルネットワークを使用した探索で自己対戦をするためGPU使用はほぼ必須
- TamaGoの終局時の判定がお粗末なのでGNUGoによる判定結果を使うことを推奨
  （なくても学習が進むが、あった方が安定して速く学習が進む）

強化学習のパイプライン

- 強化学習のパイプラインはpipeline.shに実装（Linux環境向け）

```
for i in `seq 1 100` ; do
    python3.6 selfplay_main.py --save-dir archive --model model/rl-model.bin --use-gpu true
    python3.6 get_final_status.py
    python3.6 train.py --rl true --kifu-dir archive
done
```

- 3つの処理を繰り返している
    1. 自己対戦（selfplay_main.py）
    2. GNUGoによる対局結果の補正（get_final_status.py）
    3. 自己対戦で生成したデータを利用した強化学習（train.py）

（2のget_final_status.pyの実行はオプション）

自己対戦の実行方法

- 自己対戦を実行する際はselfplay_main.pyを実行する

#	コマンドライン オプション	概要	デフォルト値
1	--save-dir	生成した棋譜を格納するディレクトリの指定	archive
2	--process	自己対戦実行ワーカ数の指定	NUM_SELF_PLAY_WORKERS
3	--num-data	生成する棋譜の数の指定	NUM_SELF_PLAY_GAMES
4	--size	碁盤のサイズの指定	BOARD_SIZE
5	--use-gpu	GPU使用フラグの指定	True
6	--visits	自己対戦時の1手あたりの探索回数	SELF_PLAY_VISITS
7	--model	使用ニューラルネットワークモデルファイルのパスの指定	model/rl-model.bin

- 赤字のパラメータはlearning_param.pyに定義
- コマンドラインオプションによる指定より、learning_param.pyの値を編集することを推奨（ほかの項目はデフォルト値を使うことを強く推奨）

自己対戦の設定項目

- 自己対戦の設定項目はlearning_param.pyに定義

	ハイパーパラメータ	概要
1	SELF_PLAY_VISITS	自己対戦時の1手あたりの探索回数
2	NUM_SELF_PLAY_WORKERS	自己対戦実行ワーカ数
3	NUM_SELF_PLAY_GAMES	生成する棋譜の数

- SELF_PLAY_VISITSについては以下トレードオフがある
  値を大きくする：棋譜の質は良い、生成速度は遅い
  値を小さくする：生成速度は速い、棋譜の質は悪い
  → 強化学習がある程度進んだら大きくすることを考える
- NUM_SELF_PLAY_WORKERSは大きいほど生成速度が上がる
  → ただしCPUかGPUのいずれかがボトルネックになって頭打ちになる
- NUM_SELF_PLAY_GAMESは5000か10000が妥当
  → 値が小さすぎると学習に失敗しやすく、値が大きいと学習の進行が遅くなる

強化学習で生成される棋譜の内容

Valueが学習する勝敗

(;FF[4]GM[1]SZ[9]
AP[TamaGo]PB[TamaGo-Black]PW[TamaGo-White]RE[W+R]KM[7.0];B[ef]C[82 A9:4.635e-12 B9:1.070e-05
C9:1.427e-05 D9:6.679e-10 E9:1.207e-05 F9:1.264e-05 G9:1.027e-05 H9:8.614e-06 J9:7.135e-06
A8:1.297e-05 B8:1.253e-09 C8:5.696e-09 D8:1.986e-05 E8:2.646e-05 F8:4.150e-08 G8:2.150e-05
H8:1.373e-05 J8:9.391e-06 A7:6.919e-11 B7:2.534e-05 C7:2.204e-07 D7:6.949e-05 E7:7.535e-04
F7:7.094e-05 G7:1.892e-07 H7:1.949e-05 J7:1.279e-05 A6:1.140e-05 B6:2.677e-05 C6:3.587e-06
D6:1.830e-01 E6:2.328e-03 F6:3.212e-01 G6:3.205e-06 H6:8.839e-09 J6:1.330e-05 A5:1.156e-05
B5:2.802e-05 C5:7.050e-05 D5:1.139e-01 E5:1.488e-04 F5:4.396e-03 G5:1.384e-04 H5:2.476e-05
J5:1.438e-05 A4:1.029e-05 B4:1.856e-05 C4:8.017e-05 D4:7.015e-02 E4:2.751e-01 F4:4.241e-03
G4:2.272e-02 H4:2.339e-05 J4:1.183e-05 A3:2.121e-10 B3:1.875e-05 C3:1.754e-06 D3:9.246e-04
E3:2.580e-06 F3:5.766e-05 G3:3.297e-05 H3:1.774e-05 J3:1.041e-05 A2:9.105e-06 B2:1.400e-05
C2:2.019e-05 D2:3.016e-08 E2:3.030e-05 F2:2.588e-08 G2:1.722e-05 H2:1.263e-05 J2:9.752e-06
A1:5.753e-06 B1:1.106e-05 C1:1.054e-05 D1:1.431e-05 E1:1.364e-05 F1:1.210e-05 G1:1.048e-05
H1:8.875e-06 J1:2.627e-12 PASS:2.455e-09];W[ce]C[81 A9:8.225e-08 B9:1.539e-07 C9:1.654e-07
D9:1.949e-07 E9:1.959e-07 …

Policyが学習する
Improved Policy

上記内容のSGFファイルが--save-dirで指定したフォルダ内に出力される
ファイルの内容は変更NG（閲覧はOK）

対局結果の補正

- get_final_status.pyは対局結果のGNUGoによる補正を実行する
  （TamaGoの終局判定が非常にお粗末なため）

- 各環境で下記のようにGNUGoのインストールが必要
  【Ubuntu】
  # apt install gnugo
  【Mac】
  # brew install gnu-go
  【Windows】
  http://gnugo.baduk.org/からzipファイルをダウンロード
  解凍後、下記どちらかの対応をする
    1. gnugo.exeにパスを通す
    2. 右記の”gnugo”をgnugo.exeの絶対パスに変更

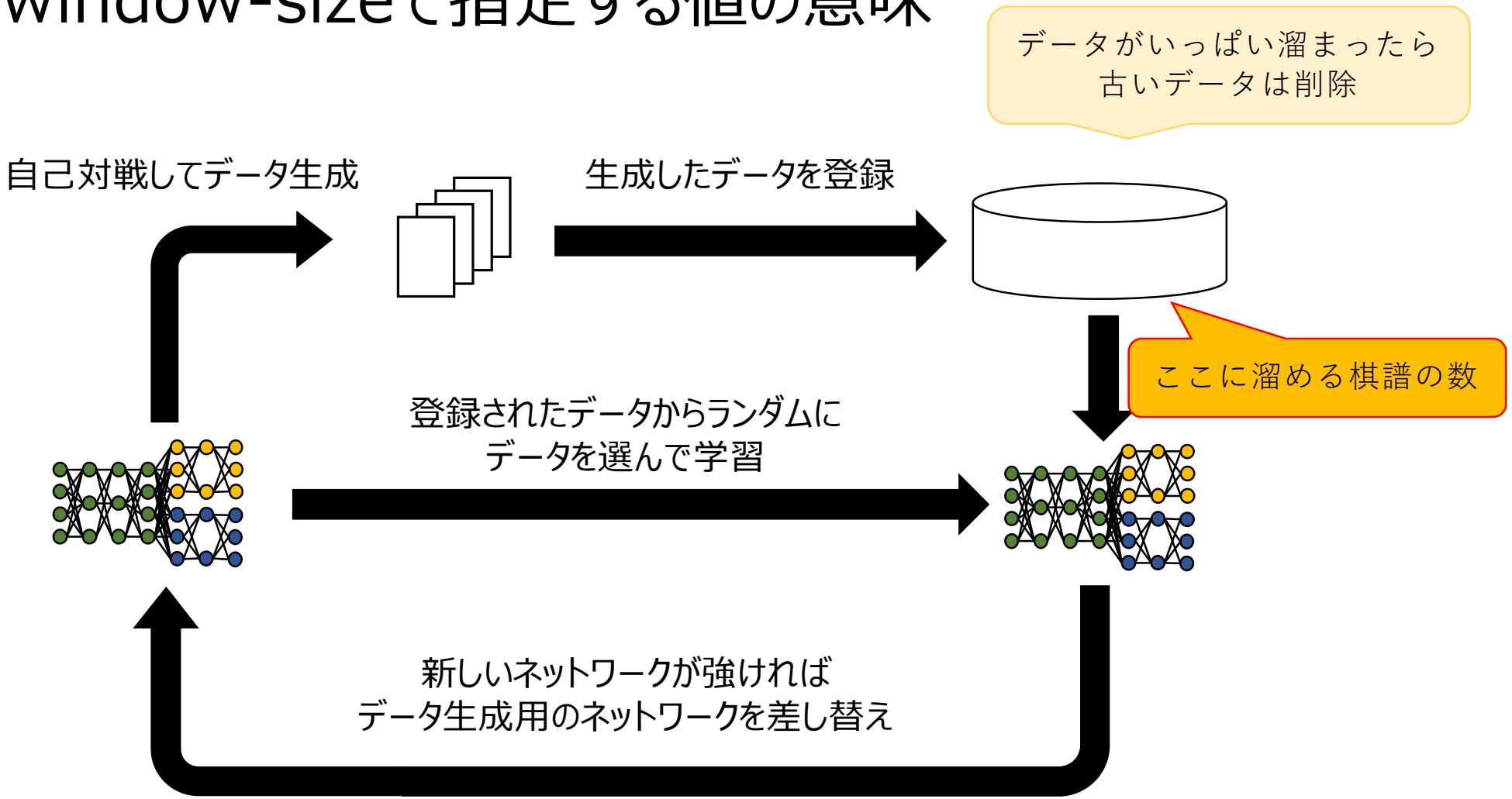
```
gnugo_command = [
    "gnugo",
    "--mode",
    "gtp",
    "--level",
    "10"
]
```

強化学習の実行方法

- 強化学習を実行する場合はtrain.pyを実行する
  【再掲】（train.pyは教師あり学習、強化学習共通のエントリーポイント）

#	オプション	概要	デフォルト値	備考
1	--kifu-dir	強化学習に使用するSGFファイルを格納したディレクトリのパスの指定	（なし）	selfplay_main.pyの--save-dirオプションと同じ値を指定する
2	--size	碁盤のサイズの指定	BOARD_SIZE (=9)	基本的には指定不要
3	--use-gpu	GPU使用フラグの指定	True	
4	--rl	強化学習実行フラグの指定	False	強化学習の際は必ずTrueを指定
5	--window-size	学習の使用する棋譜の最大数	300000	

--window-sizeで指定する値の意味



強化学習の設定項目

- 強化学習の設定項目（ハイパーパラメータ）はlearning_param.pyに定義

	ハイパーパラメータ	概要
1	RL_LEARNING_RATE	学習率
2	BATCH_SIZE	1ステップ学習するデータの個数
3	MOMENTUM	オプティマイザのモーメンタムパラメータ
4	WEIGHT_DECAY	L2正則化の重み
5	DATA_SET_SIZE	npzファイル1つ当たりのデータ数
6	RL_VALUE_WEIGHT	Policyに対するValueの損失関数の重み

- ある程度うまく動くことを確認している値を設定しているため、最初はデフォルトの値をそのまま使うことを推奨

- 強化学習がある程度進んだら、手でRL_LEARNING_RATEを小さくする

強化学習の処理内容

- 強化学習の処理は大きく2ステップからなる
    1. SGFファイルから学習用データをまとめた.npzファイルをdataフォルダに出力
       （--kifu-dirオプションが指定されたときのみ実行）
       → generate_reinforcement_learning_data関数
    2. dataフォルダ内の.npzファイルを使用して強化学習を実行
       → train_with_gumbel_alphazero_on_gpu関数
          またはtrain_with_gumbel_alphazero_on_cpu関数

- 実行終了後、modelフォルダ以下にrl-model.binとrl-state.ckptが生成される
  （これらのファイルはtrain.py実行のたびに読み込まれる）

- 強化学習パイプライン実行時には--kifu-dirの指定は必須
  → 指定なしだと既存の.npzを繰り返し学習するため
  （強化学習サイクルが正しく動かない）

TamaGoを強くするには

強化する方法

- 強くする方法は大まかに以下3つに分けられる
    1. 教師あり学習、強化学習共通の方法
    2. 教師あり学習のみに適用できる方法
    3. 強化学習のみに適用できる方法

【注意】

下記で入手できるsl-model.bin, rl-model.binを基準に強くできるという意味

https://github.com/kobanium/TamaGo/releases/tag/v0.6.0

KataGoとかLeelaZeroより必ず強くなるという意味ではない

強くする方法一覧

#		改良方針	要求プログラミングスキル
1	共通	ニューラルネットワークの フィルタ数、ブロック数を増やす	無
2		ニューラルネットワークの入力特徴を増やす	中
3		ResBlockの代わりにSEBlockを使う	中
4		ReLUの代わりにSwishを使う	低
5		PythonからC++に移植する	高
6	教師あり学習	学習データを増やす	無
7		Optimizerを変える	中
8	強化学習	自己対戦時の探索回数を増やす	無
9		自己対戦数を増やす	無

① ニューラルネットワークのフィルタ数、ブロック数を増やす

- nn/network/dual_net.pyの右の部分の数値を増やす

```
filters = 64
blocks = 6
```

- 値を増やすと学習や探索に時間がかかるようになるので要調整
  フィルタ数 ： 2倍にすると計算時間が約4倍
  ブロック数 ： 2倍にすると計算時間が2倍

【参考】KataGoのフィルタ数とブロック数のスケーリング

#	フィルタ数	ブロック数
1	96	6
2	128	10
3	192	15
4	256	20

② ニューラルネットワークの入力特徴を増やす

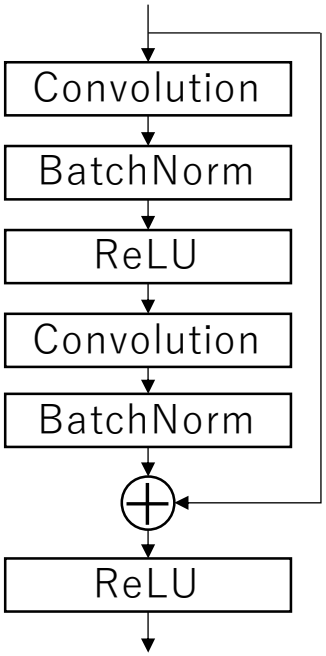
- 囲碁のドメイン知識を入力特徴として採用する
  - 相手の石をアタリにする手
  - アタリにされている石をノビる手
  - 連が持つダメの個数
  - …

#	特徴	概要
1	空点	石がない交点は1、そうでなければ0
2	自分の石	自分の石がある交点は1、そうでなければ0
3	相手の石	相手の石がある交点は1、そうでなければ0
4	直前の着手箇所	直前の相手の着手箇所は1、そうでなければ0
5	直前の手がパスか否か	直前の相手の着手がパスならすべて1、そうでなければすべて0
6	手番の色	黒番ならすべて1、白番ならすべて-1

→ TamaGoの入力特徴は非常にナイーブ（ドメイン知識は不採用）

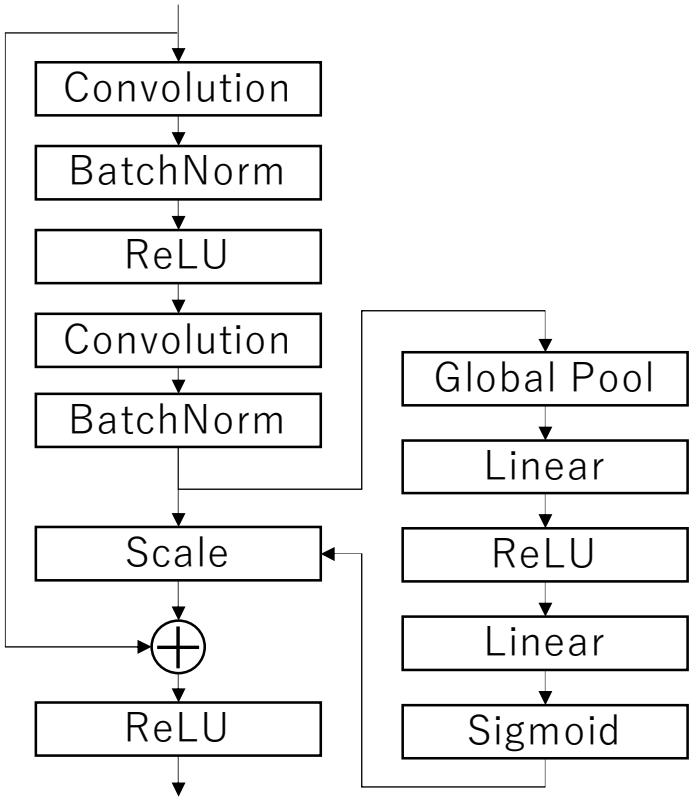
③ ResBlockの代わりにSEBlockを使う

SEはSqueeze and Excitationの略



Residual Block

遅くなる代わりに精度が良くなる



SE Block

④ ReLUの代わりにSwishを使う

- SwishはReLUよりも良い性能を示すことが多い
  （計算が重くなるのでわずかに遅くなる）

$$\mathrm{ReLU}(x) = \max(x, 0)$$

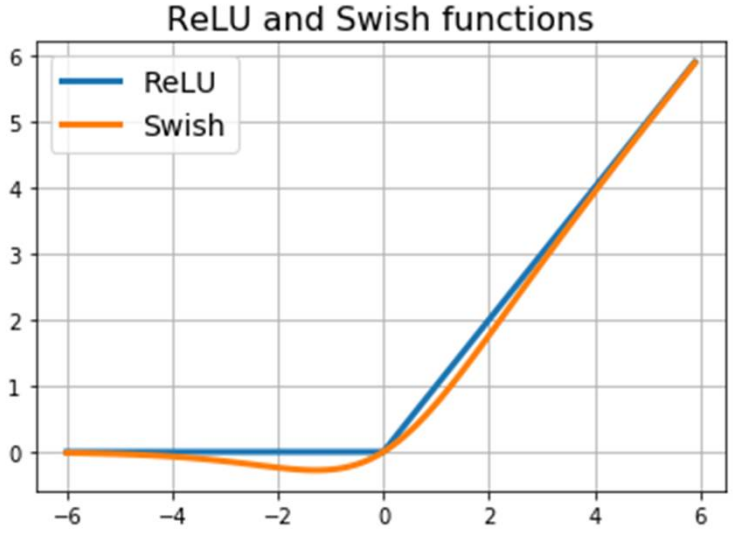
$$\mathrm{Swish}(x) = \frac{x}{1 + e^{-x}}$$

- nn/network/dual_net.pyを
  以下のように書き換えればいいはず

```
self.relu = nn.ReLU()
```

```
self.relu = nn.SiLU()
```

（ちょっと自信ない…）



https://www.renom.jp/ja/notebooks/tutorial/basic_algorithm/activation_swish/notebook.html

⑤ PythonからC++に移植する

- Pythonは実行速度が遅いのでC++に移植すると探索が速くなる
  【効果1】. 探索が速くなると読みが深くなって強くなる
  【効果2】. 強化学習の棋譜生成速度が向上して強化学習サイクルを速くできる

- TamaGoの碁盤のデータ構造や探索の実装はRayを参考にしているので、
  C++移植の際の参考になると思われる
  https://github.com/kobanium/Ray

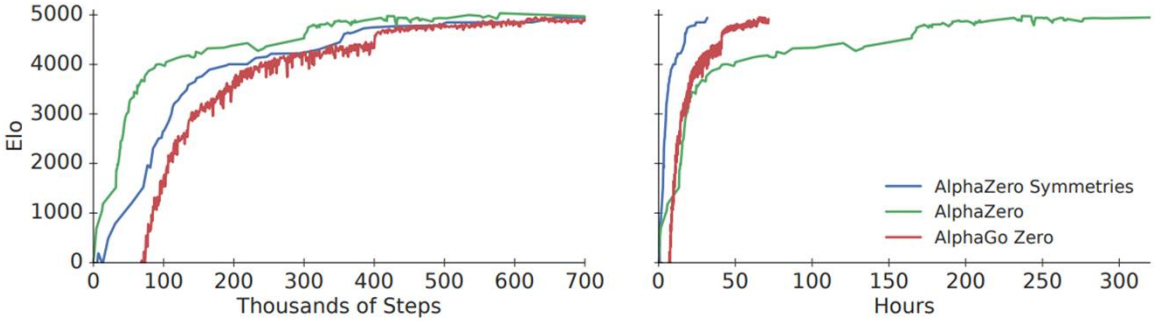
⑥ データを増やす

- データを増やす方法は大きく2つある
  - 棋譜ファイルを増やす
  - 局面を回転・反転させて水増しする

- 後者の手法は既に取り込み済み
  → データを約8倍に水増し
  強化学習にも効果あり(右図)

- 19路盤はKGSなど公開データあり

- 9路盤の追加データ入手は困難



"A general reinforcement learning algorithm that masters chess, shogi and Go through self-play", Figure S1

⑦ Optimizerを変える

- train.pyのOptimizerは
  確率的勾配法
  モーメンタム法
  Nesterovの加速勾配法
  を組み合わせたOptimizerを使用

```
optimizer = torch.optim.SGD(dual_net.parameters(),
                            lr=RL_LEARNING_RATE,
                            momentum=MOMENTUM,
                            weight_decay=WEIGHT_DECAY,
                            nesterov=True)
```

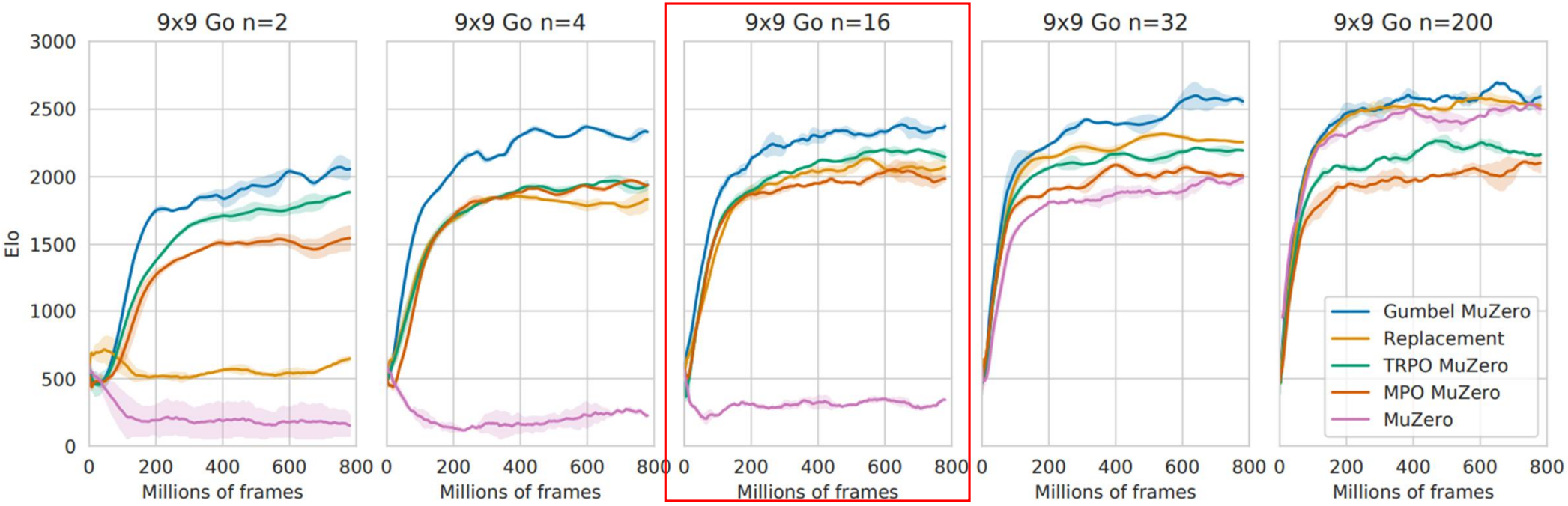
強化学習で使用するOptimizer

（教師あり学習も同じtorch.optim.SGDを使用）

- より優れている手法としては
  AdamW（torch.optimに実装あり）
  NAdam（torch.optimに実装あり）
  SAM（https://github.com/davda54/samのスクリプト導入要）
  がある

⑧ 自己対戦時の探索回数を増やす

- TamaGoの強化学習モデルはvisits ＝ 16で棋譜生成しているので、visitsを16以上にすると強くできる（はず）



“Policy Improvement by Planning with Gumbel”, Figure 2

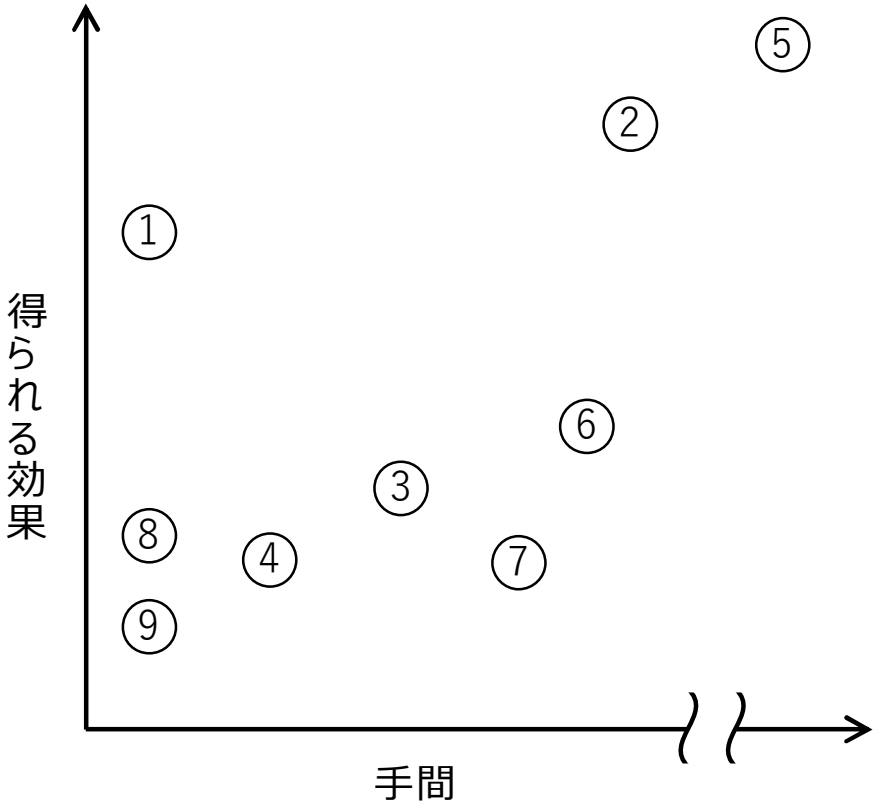
⑨ 自己対戦数を増やす

- TamaGoの強化学習モデル(rl-model.bin)は80万局生成、学習したモデル

- ただ既存のモデルをそのまま流用して強化学習を進めても強くならなそう…
  70万局時点のモデルと公開モデルの強さがほぼ一緒であることは確認済み

- ニューラルネットワークの規模が大きいほど、強さが収束するまでの必要自己対戦数は大きくなる傾向がある

それぞれの手法の効果は…



#		改良方針
1	共通	ニューラルネットワークの フィルタ数、ブロック数を増やす
2		ニューラルネットワークの入力特徴を増やす
3		ResBlockの代わりにSEBlockを使う
4		ReLUの代わりにSwishを使う
5		PythonからC++に移植する
6	教師あり学習	学習データを増やす
7		Optimizerを変える
8	強化学習	自己対戦時の探索回数を増やす
9		自己対戦数を増やす

何はともあれ①をトライするのがおすすめ

強くする方法一覧

#		改良方針	要求プログラミングスキル
1	共通	ニューラルネットワークの フィルタ数、ブロック数を増やす	無
2		ニューラルネットワークの入力特徴を増やす	中
3		ResBlockの代わりにSEBlockを使う	中
4		ReLUの代わりにSwishを使う	低
5		PythonからC++に移植する	高
6	教師あり学習	学習データを増やす	無
7		Optimizerを変える	中
8	強化学習	自己対戦時の探索回数を増やす	無
9		自己対戦数を増やす	無

まとめ

まとめ

【講習会の狙い】

- AlphaGo登場以降の要素技術の概要を知ること
  → MCTS, AlphaGo, AlphaGo Zeroの仕組みを説明

- ベースとなる囲碁AIを使って自分独自の囲碁AIを開発できるようになること
  → TamaGoの実装のポイントと学習の実行方法、強くする方針を説明

9路盤ミニ大会に向けて

宿題事項

- program.pyのPROGRAM_NAMEを自分のプログラム名に
- なにがしかの工夫を入れてみる
  （必ずしも強くなる必要はなし）
- 対局に向けてGUIを準備する（おすすめはGoGUI）

不明点や疑問点があればSlackにて遠慮なくご質問ください

ミニ大会のルール

碁盤のサイズ	9路盤
コミ	7.0目
持ち時間	10分
進行方式	7～8人のグループの総当たり

※ 進行方式と持ち時間については参加人数によって変更の可能性あり

おわりに

おわりに

- 囲碁AIの開発に必要なリソースは非常に膨大です。
  ただ9路盤は個人所有できるレベルのPCで超人レベルになれます。

- 莫大なリソース要求に対応できるのは囲碁の知識と種々の工夫です。

- 強くする方法ばかりお話ししましたが、強さに関係しない工夫をしてみたり、
  自分の棋譜だけ学習したクローンAIを作ったり、開発の楽しみ方はいろいろあります。

- 本講習会をきっかけに囲碁AIの大会でまた皆様にお会いできれば幸いです。
  CGFオープン：2023/7/15 (9路盤)、2023/7/16 (19路盤)
  UEC杯：2023/11/4 – 5 (19路盤)