卒業研究報告書

題目

機械学習を用いた AI 開発

指導教員

石水 隆 講師

報告者

17–1–037–0120

新堀 穂高

近畿大学理工学部情報学科

令和 2 年 2 月 1 日提出

概要

京都将棋は縦横 5 マスの将棋盤で (香・と),(銀・角),(金・桂),(飛・歩), 王の 5 種類の駒を利用して対戦するミニ将棋である. 通常の将棋のルールと異なり, 京都将棋では王以外の 4 種類の駒は一手ごとに必ず裏返さなければならず, 持ち駒は裏表どちらでも打つ事が出来る. また通常の将棋で禁止されている二歩や, 行く場所のなくなる駒を打つことも出来る. 京都将棋はこのような独特なルールも相まって AI に関する研究が少なく, 学習データも少ない. そこで本研究では機械学習を用いて京都将棋の AI を開発する.AI 開発には Alpha Zero を参考に開発を行う.Alpha Zero とは DeepMind 社によって 2017 年に開発され, プロ棋士の棋譜データを用いることなく, 自己対戦により囲碁・チェス・将棋の学習が出来る AI である.Alpha Zero の強化学習サイクルを繰り返し, モデルの更新を 60 回行った京都将棋 AI を作成した. 作成した京都将棋 AI をランダムに手を選択する AI とウェブアプリケーションに対戦させ, 強さと脆弱性の検証を行う.

目次

1 序論

1.1 京都将棋とは

京都将棋は縦横 5 マスの将棋盤で (香・と),(銀・角),(金・桂),(飛・歩), 王の 5 種類の駒を利用して対戦するミニ将棋である. 通常の将棋のルールと異なり, 京都将棋では王以外の 4 種類の駒は一手ごとに必ず裏返さなければならず, 持ち駒は裏表どちらでも打つ事が出来る. また通常の将棋で禁止されている二歩や, 行く場所のなくなる駒を打つことも出来る. 京都将棋は知名度が低く, 現状ほとんど研究されてない. そのため, 京都将棋の定石等は確立しておらず. 京都将棋特有の手法も未知数である. 既存の京都将棋の AI としては [2] がある.[2] はウェブアプリケーションであり, ネット上でプレイすることができる. また株式会社ねこまどぶろぐ [3] が 2015 年に iOS 版と 2016 年に Android 版の京都将棋アプリを配信していたが,iOS 版は現在はダウンロードが出来なくなっている. また幻冬舎 [4] が商品として販売している.

1.2 将棋 AI の手法

将棋のような可能な局面数の多いゲームでは完全解析を行うことは困難である. かつては強い AI を作成するために評価関数を利用していた. 評価関数は現在の局面の有利不利を評価することにより求められる. 局面の有利不利は駒得, 相手を詰ませられるなど数値化できる基準で評価される.

将棋 AI を作る上で, 最近注目されている手法が機械学習である. 本将棋では過去のプロ棋士の棋譜データが存在するため, 機械学習では Bonanza などの教師あり学習を元に評価関数を決定する事が主流であった. しかし,Alpha Zero ではプロ棋士の棋譜データを利用せずに自己対戦により学習データを作成し, 深層学習によって強い AI を開発する. 深層学習とは大量のデータから規則性を見つけ, 分類や判断などの推論のためのルールを機械に生成させる機械学習の一つである. ルールを決定する際に, ネットワーク構造と調整可能な重みパラメータによって構成されるニューラルネットワークと呼ばれるモデルを利用し, 学習によって重みパラメータを最適化し, ルールを決定する.

1.3 京都将棋 AI

本研究では,Alpha Zero[1] を参考に京都将棋 AI の開発を行う.Alpha Zero とは DeepMind 社によって 2017 年に開発された AI である.Alpha Zero は AlphaGo と AlphaGo Zero を改造したバージョンであり,AlphaGo は DeepMind 社によって開発された囲碁プログラムである.AlphaGo のアルゴリズムはモンテカルロ木探索をベースとしており, 学習にはプロ棋士の棋譜データを利用していた.2017 年 10 月には AlphaGo を改良した AlphaGo Zero が発表された.AlphaGo Zero の特徴は学習にプロ棋士の棋譜データを一切使わず自己対戦のみで学習を行う事である. そして 2017 年の 12 月に AlphaGo Zero の改造バージョン Alpha Zero が開発された.Alpha Zero は囲碁だけでなく, 将棋やチェスの学習が可能で, チェスと将棋の世界チャンピオンの StockFish と Elmo に勝利した.AlphaZero の特徴は AlphaGo Zero と同様にプロ棋士の棋譜データを用いる事なく, 自己対戦により学習が出来る事である.

1.4 本研究の目的

本研究の目的は、機械学習によって京都将棋の AI を開発することである。しかし, 京都将棋はあまり研究されておらず、学習データとして利用できるデータも少ない. そこで Alpha Zero の機械学習サイクルを利用し, 自己対戦における学習データの作成に基づき機械学習を行う。

1.5 本報告書の構成

本報告書の構成は以下の通りである. まず 2 章で本研究の対象である京都将棋について説明する. 3 章では機械学習に利用した Alpha Zero の強化学習サイクルについて述べる.4 章では作成したプログラムについて説明する.5 章ではランダムな AI とウェブアプリケーションの対戦結果及び, 脆弱性を示す. 最後に 6 章にて結論及び今後の課題を述べる.

2 京都将棋

本章では, 本研究の対象である京都将棋について説明する.

2.1 京都将棋の概要

京都将棋は 1976 年に田宮克哉によって発案された. 京都将棋は縦横 5 マスの盤面で (香・と),(銀・角),(金・桂),(飛・歩), 王の 5 種類の駒を利用して対戦するミニ将棋である. 一手ごとに駒を裏返す必要があり, 持ち駒を打つ場合は表裏好きな方で打つことができる.

2.2 京都将棋のルール

本節では京都将棋のルールについて述べる. 図 1 に京都将棋の初期配置を示し, 以下にルールを示す.

5	4	3	2	1	
歩	金	王	銀	と	一
					二
					三
					四
と	銀	玉	金	歩	五

図 1　京都将棋の初期配置

- 盤面は縦横 5 マスで自陣と敵陣は存在しない
- 駒の種類は (香・と),(銀・角),(金・桂),(飛・歩), 王の 5 種類で本将棋と動きは同じである

- 王以外の駒は打った場所にかかわらず必ず裏返す必要がある.
- 本将棋同様, 取った駒は好きな場所に打つことが可能であり裏表どちらで打つことも可能である
- 二歩, 行き所のない駒を打つ, 打ち歩詰めは禁止されていない
- 本将棋同様, 同一譜面 4 回で千日手となり引き分けとなる

3 Alpha Zero の強化学習サイクル

本章では、本研究で Alpha Zero を参考に作成した京都将棋 AI の強化学習サイクルについて述べる. 機械学習をするためには, 学習用のデータを準備する必要がある. 本将棋では, 学習データとしてプロ棋士の対戦から得られた膨大な寄付データを使うことができる. しかし, 京都将棋はマイナーなゲームであるため, そのような棋譜は存在しない. しかし,Alpha Zero は自己対戦による学習データの作成が可能という特徴があるため, プロ棋士の棋譜データを用いることなく学習することが出来る. 強化学習サイクルの流れとして, まず初めにデュアルネットワークの作成を行う. その後, 最新モデルによるモンテカルロ木探索による自己対戦, 作成した学習データを元にデュアルネットワークの学習, 作成したモデルと最新モデルの対戦による更新を繰り返す. 最終的に一番強いモデルが残ることになる.

3.1 デュアルネットワークの作成

本節では, 本研究で作成する京都将棋 AI で用いるニューラルネットワークについて述べる. 本研究で作成するニューラルネットワークは, デュアルネットワークを作成する.[6] デュアルネットワークとは現在の局面に応じて方策と価値を出力する深層学習のモデルである. 方策は次の一手の確率分布、価値は現在の局面の勝敗予測を出力する. デュアルネットワークの入力はゲームの盤面と持ち駒の有無である. 京都将棋は $5*5$ の 盤面であり, 全ての駒の数は駒の裏表を含めると 17 となる. 従って,$5*5$ の二次元配列が自分の駒と相手の駒を合わせた 34 個 となり, 入力シェイプは $(5, 5, 34)$ となる. 入力シェイプとは, ニューラルネットワークに利用する学習データの配列の次元を表している. また, ゲームの盤面は駒が置かれているときは 1, 何も置かれていなければ 0 とする.

次にネットワーク構造について述べる. デュアルネットワークのネットワーク構造は ResNet のモデルにベースに作成する.ResNet は画像認識の分野で高い性能を発揮するニューラルネットワークであり,Alpha Zero ではゲームの盤面を入力として利用するため, 畳み込みニューラルネットワークによる特徴抽出に適している.[7] 畳み込みニューラルネットワークとは畳み込み層を使って抽出するニューラルネットワークであり, 畳み込み層は入力からその特徴を表現した特徴マップに変換する. 仕組みとしては $5*5$ のサイズをカーネルと呼ばれる重みパラメータの配列を利用し $3*3$ の特徴マップに変換する. これは複雑な特徴をもつデータを効率よく抽出できるためである. 次にプーリング層とは, 畳み込み層の特徴マップを縮約してデータの量を削減する層である. 仕組みとしては, データの一部の最大値や平均値を取ることでデータの圧縮を行う. 図 2 および図 3 にネットワーク構造と ResNet の残差ブロックを示す. ここで, 残差ブロックとは, 畳み込みニューラルネットワークのショートカット構造である. 畳み込みニューラルネットワークは層を深くすることでより複雑な特徴を抽出できるようになる. しかし, 単純に層を深くするだけでは性能が悪化する. そのため畳み込み層にショートカットコネクションと呼ばれる迂回ルートを作成している. これにより畳み込み層で学習が不要となった際に迂回することができ, より深い層の学習ができる.

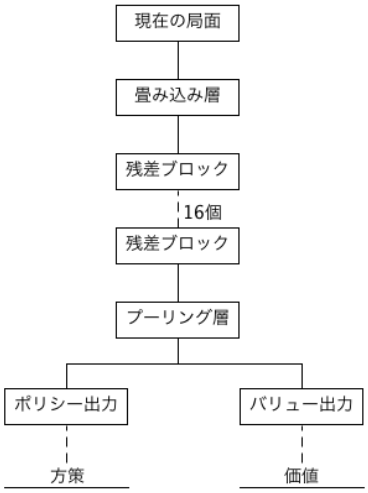

図 2　ネットワーク構造

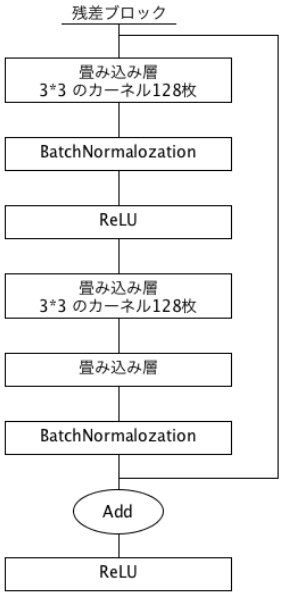

図 3　残差ブロック

3.2　モンテカルロ木探索

本節では, 本研究で作成する京都将棋 AI で用いる探索手法について述べる. 機械学習による探索で用いられる手法としてモンテカルロ木探索がある.[8] 従来のモンテカルロ木探索はランダムにシミュレーションを繰り返し終局まで手を進め勝ち負けを取得し, 最終的に最も選択された手を選択する手法である. しかし何度もシミュレーションを実行する必要があるため時間がかかるという弱点が存在する.Alpha Zero のモンテカルロ木探索ではデュアルネットワークを利用するためシミュレーションを実行する必要がなく, シミュレーション 1 回で新しいノードを作成することができる. モンテカルロ木の初期状態は現在の局面を表すルートノードと, 次の一手である子ノードのみ存在する. また, 各ノードは累計価値と試行回数をそれぞれ保持する. ルートノードから子ノードを選択する際にアーク評価値を利用する. アーク評価値とは局面の評価関数であり, アーク評価値は以下の式で表される.w はノードの累計価値, n はノードの試行回数を示しており $\frac{w}{n}$ は勝率を示している.また t は累計試行回数を示しており, $\frac{\sqrt{t}}{(1+n)}$ はバイアスとなっている.学習データが少ない場合, バイアスは大きな値となり精度に誤差が生じる.

$$\text{アーク評価値} = \frac{w}{n} + p * \frac{\sqrt{t}}{(1+n)}$$

探索の手順は以下の通りであり, 下記の流れを 100 回繰り返した後に最も選択された手を次の一手とする.

1. 初期状態はルートノードから開始する. 現在のノードに子ノードが存在する場合, アーク評価値が最大となる手を選択し移動する. この操作を, 子ノードのないノード, リーフノードに達するまで繰り返す.
2. リーフノードに到達した場合, 子ノードの追加を行い, デュアルネットワークを利用し推論を行い方策と価値を取得する.
3. 価値を取得した後, ノードの累計価値と試行回数を更新しながらルートノードまで戻る.

3.3 最新モデルによる自己対戦

本節では, 最新モデルによる自己対戦について述べる.Alpha Zero ではデュアルネットワークに利用する学習データを自己対戦により作成する. 自己対戦は 500 回行い, 合法手の確率分布を利用しランダムに手を選択する. 学習データの型は自分の駒の配置, 相手の駒の配置, 方策, 価値を保存する. 方策は 1 手ごとにモンテカルロ木探索で取得し, 価値は対戦が終了するたびに取得する. なお 800 手を越えると引き分けとして価値の値は 0 を取得するように作成する.

3.4 デュアルネットワークの学習

本節では, デュアルネットワークの学習について述べる.3.3 節で作成した学習データを利用し学習を行う. まず初めに作成した学習データをデュアルネットワークに利用するため, 学習データの読み込みを行う. 学習データの型は自分の駒の配置, 相手の駒の配置, 方策, 価値を保存したが, これを状態, 方策, 価値のそれぞれで別のリストに変換する. 次に入力データのシェイプ変換を行う. 学習データは入力シェイプを $(5, 5, 34)$ として作成したが, 学習には複数のデータを入力データとすることができるため, 入力シェイプは (データ数, $5, 5, 34$) となる. 今回は 500 個の学習データで推論を行うため $(500, 5, 5, 34)$ に変更する. 次に学習された最新のモデルを読み込みコンパイルを行う. モデルを学習するためにはいくつかの設定をする必要があり, 今回は多クラス交差エントロピー誤差, 平均二乗誤差,Adam を設定している. 最後に学習率の設定を行う. 学習回数は 300 回行い, 学習率は 0.001 から設定し 150 回後に 0.05,240 回後に 0.00025 に設定している. この学習率は [1] をそのまま利用している. 上記の条件で学習を行い学習したモデルを最新モデルとして保存する. 本家の Alpha Zero は 0.2 から始まり 100 回後に 0.02,300 回後に 0.002,500000 回後に 0.0002 となっている.

3.5 モデル同士の対戦による更新

本節では, モデル同士の対戦による更新について述べる.3.4 節で学習させたモデルと最新モデルをモンテカルロ木探索を利用し 20 回対戦させ, 学習させたモデルが勝率を超えた場合, 最新モデルに更新を行う. これにより, 学習させたモデルが弱い場合は更新を行わず, 勝ち越した場合に更新を行うため, より強い AI を作成することが出来る.

4 京都将棋のプログラム

本章では, 本研究で作成したプログラムについて述べる.

4.1 京都将棋プログラムの仕様

本研究では [1] のプログラムを参考に python 言語を利用し京都将棋 AI の作成を行った. 表 1 にソースコードの一覧を示す. また,python の開発環境には Google Colab を利用した.Google Colab とは Google が無償で提供しているオンラインサービスである.Google Colab は機械学習のパッケージが揃っているため環境構築の必要がなく便利である. またリアルタイム画像処理の特化した演算装置 GPU を利用することが可能であり, 学習時間を大幅に削減することができる. プログラムの実行の流れとして表 1 のソースコードを Google Colab にアップロードを行い,self_play.py,train_network.py,evalute_network.py を順に実行する.self_play.py では自

己対戦を 500 回行い学習データ (*/history) を作成する. 次に train_network.py では作成された学習データを元に学習を行い, 最新プレイヤーのモデル least.h5 を作成する. 最後に evalute_network.py で best.h5 と least.h5 のモデル同士で対戦を 20 回行い, 勝率が 5 割を超えた場合 least.h5 を best.h5 にコピーし保存する. 保存を行なった best.h5 はローカルの python 実行環境を作成し human_play.py を実行することで人間と best.5 の対戦を行うことが可能である. ローカルの python 実行環境には Anaconda を利用した.Anaconda は python 本体とよく利用されるライブラリをセットにしたパッケージである.

表 1　京都将棋のソースコード一覧

ソースコード	説明
game.py	ゲーム状態
dual_netwrok.py	デュアルネットワーク
pv_mcts.py	モンテカルロ木探索
self_play.py	自己対戦
train_network.py	パラメータ更新
evalute_network.py	新パラメータ評価
human_play.py	ゲーム UI

4.2　game.py

game.py は京都将棋の状態,State クラスを作成する.State のメソッドとその機能を表 2 に示す.

表 2　State クラスのメソッドと機能

メソッド	説明
init(pieces=None,enemy_pieces=None,depth=0)	ゲーム状態の初期化
is_lose()	ゲームの負けを判定
is_draw()	ゲームの引き分けを判定
is_done()	ゲームの終了を判定
pieces_array()	デュアルネットワークの入力の 2 次元配列
position_to_actiton(position,direction)	駒の移動先と移動元を行動に変換
action_to_position(action)	行動を駒の移動先と移動元に変換
legal_actions()	合法手のリストを取得
legal_action_pos(position_src)	駒の移動時の合法手のリストの取得
next(action)	次の状態の取得
is_first_player	先手かどうか

4.3　dual_network.py

dual_network.py はデュアルネットワークを作成する. メソッドとその機能を表 3 に示す.

表 3　dual_network.py のメソッド

conv(filters)	畳み込み層の作成
residual_block()	残差ブロックの作成
dual_network()	デュアルネットワークの作成

4.4　pv_mcts.py

pv_mcts.py はモンテカルロ木探索を作成する. ノードを定義するために Node クラスを作成し, メソッドとその機能を表 4 に示す. また pv_mcts.py のメソッドとその機能を表 5 に示す.

表 4　Node クラスのメソッド

init(state,p)	ノードの初期化
evalute()	局面の価値を計算
next_child_node()	アーク評価値が最大の子のノードを取得

表 5　pv_mcts.py のメソッド

predict(model,state)	推論
nodes_to_scores(nodes)	ノードのリストを試行回数リストに変換
pv_mcts_scores(model, state, temperature)	モンテカルロ木探索のスコアの取得
pv_mcts_action(model, temperature=0)	モンテカルロ木探索で行動選択
boltzman(xs, temperature)	ボルツマン分布

4.5　self_play.py

self_play.py はデュアルネットワークを作成する. メソッドとその機能を表 6 に示す.

表 6　self_play.py のメソッド

first_player_value(ended_state)	最終局面から先手プレイヤーの価値を計算
write_data(history)	学習データの保存
play(model)	1 ゲームの実行
self_play()	自己対戦の実行

4.6　train_network.py

train_network.py は自己対戦によって作成した学習データを利用し, デュアルネットワークの学習を行う. メソッドとその機能を表 7 に示す.

表 7　train_network.py のメソッド

load_data()	学習データの読み込み
train_network()	デュアルネットワークの学習
step_decay(epoch)	学習率の設定

4.7　evalute_network.py

evalute_network.py は train_network.py で学習を行い作成された,least.h5 と best.h5 の対戦を行いモデルの更新を行う. メソッドとその機能を表 8 に示す.

表 8　evalute_network.py のメソッド

first_player_point(ended_state)	最終局面から先手プレイヤーの価値を計算
play(next_actions)	1 ゲームの実行
update_best_player()	モデルの更新
evaluate_network()	モデル同士の対戦による評価

4.8　human_play.py

human_play.py は人間と作成したモデルとの対戦に利用する.GameUI を定義し,GameUI クラスを作成する.GameUI のメソッドとその機能を表 9 に示す.

表 9　GameUI のメソッド

init(self, master=None, model=None)	ゲーム UI の初期化
turn_of_human(self, event)	人間のターン
turn_of_ai(self)	AI のターン
position_to_direction(self, position_src, position_dst)	駒の移動先を駒の移動方向に変換
draw_piece(self, index, first_player, piece_type)	駒の描画
draw_capture(self, first_player, pieces)	持ち駒の描画
draw_cursor(self, x, y, size)	カーソルの描画
on_draw(self)	描画の更新

5　京都将棋 AI の強さと脆弱性

本節では学習させた京都将棋 AI をランダムに手を打つ AI と [2] のウェブアプリケーションの対戦結果を以下に示す. また対戦における京都将棋 AI の脆弱性について述べる.

5.1 京都将棋 AI とランダム AI の対戦結果

3.3 節で述べた Alpha Zero の強化学習サイクルを繰り返し行い, モデルの更新を 60 回行った best モデルとランダムな AI との対戦結果を表 10 に示す. ランダムな手を選択する AI には必ず勝てるという結果が得られた. ランダムな手を選択しているため京都将棋が王手した際の評価値は高くなっており, ランダムな手を選択する AI は回避する事が出来ず負けていると考えられる. 学習によって AI の強さが上がっていることが確認出来る.

表 10　RandomAI との対戦結果 (試行回数 100 回)

	bestAI 先手	bestAI 後手
勝	100	100
負	0	0

5.2 京都将棋 AI とウェブアプリケーションの対戦結果

先ほど述べた best モデルの京都将棋 AI と [2] のウェブアプリケーションを 30 回ずつ先手後手入れ替えて行い,100 手を超えた場合引き分けとした対戦結果を表 11 に示す. ランダムな手を選択する AI 相手には勝てる AI だったが, ウェブアプリケーション相手には best モデルが一度も勝利する事がなく非常に偏った結果となった. さらに [2] は不具合により対局中に持ち駒を一切使わなかったため,100 手を超え引き分けとなった対局が見られた.

表 11　[2] との対戦結果 (試行回数 30 回)

	bestAI 先手	bestAI 後手
勝	0	0
負	29	28
引分	1	2

5.3 京都将棋 AI の脆弱性

ランダムな AI と [2] との対戦結果から得られたデータは偏ったものとなった. ウェブアプリケーションは持ち駒を使わなかったにもかかわらず, 京都将棋 AI は一度もかつ事が出来なかった. 京都将棋 AI が負けた原因として, 王手回避と逆王手になる手を認識しない事が見受けられた. モンテカルロ木探索で行動を選択する際に王手を回避するなどのプログラムは作成していないため, アーク評価値の評価が正しく出来ず, そのような手を選択する事が発生したと考えられる. また玉への詰め方も単調なものであり, 図 4 の局面で京都将棋 AI は桂・金を持ち駒として持っていたが, △ 2 四金と打ち,5 のように王に駒を取られ, 明らかに駒損と考えられる手を選択していた. 上記のことから作成した京都将棋 AI はある程度の強さを持ち合わせていないと考えられる.

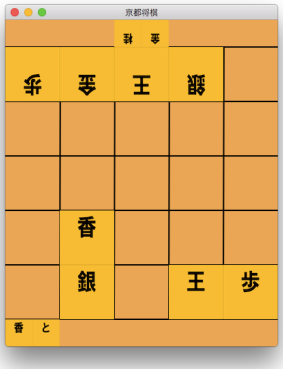

図 4　ウェブアプリケーションと京都将棋 AI の盤面 1

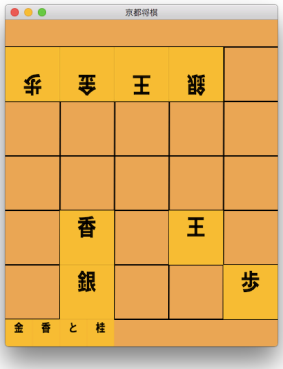

図 5　ウェブアプリケーションと京都将棋 AI の盤面 2

6　結論・今後の課題

本研究で作成した AI は, ランダムな手を選択する AI 相手には非常に高い勝率が得られたが, ウェブアプリケーションや人間との対戦において, 王手を回避しない, 逆王手になる手を選択するなどの脆弱な部分が確認された. このような脆弱な部分が確認された原因として 2 つあげられる.1 つ目に学習回数が不十分であることが考えられる. 本研究で作成した強化学習サイクルは 1 台のマシンで動作するように,Alpha Zero のスケールダウンしたものを実装しているため学習に時間がかかり,60 回の学習では十分な学習データが得られなかったことが考えられる.2 つ目にモンテカルロ木探索を行う際のアーク評価値の精度が低いことが考えられる. アーク評価値の精度が低いと, 王手をかけられているにもかかわらず, 王手を回避せず, 他の手を選択することが考えられる.

今後の課題として, 原因を解決する方法はモンテカルロ木探索を行う際の 1 推論のシミュレーション回数を増やすことが考えられる. シミュレーション回数を増やすことで出来るだけ多くのノードから手を選択する事が出来るため, 王手を回避する手などの勝率の高い手を選びやすくなり, アーク評価値の精度上げる事が出来ると考えられる.

謝辞

本研究を行うにあたり, ご指導いただきました石水隆講師には, コロナの影響もありオンラインでのミーティングが多かったにもかかわらず, 研究に対する丁寧かつ熱心なご指導をいただきました. この場を借りて感謝を申し上げます.

参考文献

[1] 布留川 英一：Alpha Zero 深層学習・強化学習・探索 人工知能プログラミング実践入門, 株式会社ボーンデジタル (2019)

[2] 将棋ゲームの時間, (2012) https://syouginojikan.web.fc2.com/kyouto.html

[3] 京都将棋, Nekomado Co.Ltd, (2015), https://itunes.apple.com/jp/app/京都将棋/id1037596970?mt=8

[4] 京都将棋, 株式会社幻冬舎エデュケーション, (2014)

[5] https://deepmind.com/research/case-studies/alphago-the-story-so-far

[6] Mastering the game of Go with deep neural networks and tree search（深層ニューラルネットワークと木探索により囲碁を究める）』https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf

[7] Deep Residual Learning for Image Recognition https://arxiv.org/abs/1512.03385

[8] Multiple Policy Value Monte Carlo Tree Search https://arxiv.org/abs/1905.13521

付録 A　ソースファイル

本研究で作成したプログラムのソースファイルを以下に示す.

game.py ファイル

```
import random
import math

class State:
    def __init__(self,pieces=None, enemy_pieces=None, depth = 0):
        #方向定数
        self.dxy = ((0,-1),(1,-1),(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),
                    (1,-2),(-1,-2),
                    (0,-2),(0,-3),(0,-4),
                    (2,0),(3,0),(4,0),
                    (0,2),(0,3),(0,4),
                    (-2,0),(-3,0),(-4,0),
                    (2,-2),(3,-3),(4,-4),
                    (2,2),(3,3),(4,4),
                    (-2,2),(-3,3),(-4,4),
                    (-2,-2),(-3,-3),(-4,-4))



        #駒の配置

        self.pieces = pieces if pieces != None else [0] * (25+8) #盤面 25+ 持ち駒 +8

        self.enemy_pieces = enemy_pieces if enemy_pieces != None else [0]*(25+8) #敵 も
同様（盤面は反対から見る）
        self.depth = depth

        #駒の初期配置
        if pieces == None or enemy_pieces == None: #
            self.pieces =       [0,0,0,0,0, 0,0,0,0,0,  0,0,0,0,0, 0,0,0,0,0, 5,2,9,3,1, 0,0,0,0,0,
            self.enemy_pieces = [0,0,0,0,0, 0,0,0,0,0,  0,0,0,0,0, 0,0,0,0,0, 5,2,9,3,1, 0,0,0,0,0,



    def is_lose(self):
```

```
        for i in range(25):
            if self.pieces[i] == 9: #9 は王様を指す
                return False
        return True

    def is_draw(self):
        return self.depth >= 800 #300 手

    def is_done(self):
        return self.is_lose() or self.is_draw()
     # デュアルネットワークの入力の 2 次元配列の取得
    def pieces_array(self):
        # プレイヤー毎のデュアルネットワークの入力の 2 次元配列の取得
        def pieces_array_of(pieces):
            table_list = []
            # 0:歩，1:銀，2:金，3:王，4:ト，5:飛 6:角 7:桂 8:香
            #駒が存在している箇所は 1 そうでないなら 0
            for j in range(1, 10):
                table = [0] * 25
                table_list.append(table)
                for i in range(25):
                    if pieces[i] == j:
                        table[i] = 1

            # 6:歩の持ち駒，7:銀の持ち駒，8:金の持ち駒，9：トの持ち駒 10:飛車の持ち駒 11:角の
持ち駒 12:桂馬の持ち駒 13:香車の持ち駒
            #持ち駒ありなら 1 なしなら 0
            for j in range(1, 9):
                flag = 1 if pieces[24+j] > 0 else 0
                table = [flag] * 25
                table_list.append(table)
            return table_list

        # デュアルネットワークの入力の 2 次元配列の取得
        return [pieces_array_of(self.pieces), pieces_array_of(self.enemy_pieces)]

    def position_to_action(self,position,direction): #駒の移動先を行動に（駒の移動先*駒の移
動元数（方向＋持ち駒種類） + 移動元）
        return position * (34+8) + direction
```

```
def action_to_position(self,action):#行動を駒の移動先と移動元に変換
    return (int(action/42),action%42)

def legal_actions(self): #合法手の取得
    actions = []
    for p in range(25):
        #駒の移動時
        if self.pieces[p] != 0:
            actions.extend(self.legal_actions_pos(p))
        #持ち駒を置く時
        if self.pieces[p] == 0 and self.enemy_pieces[24-p] == 0:
            for capture in range(1,9):
                if self.pieces[24+capture] != 0:
                    actions.append(self.position_to_action(p,33+capture)) ###

    return actions

def legal_actions_pos(self, position_src): #駒の移動時の合法手
    actions = []
    piece_type = self.pieces[position_src]
    if piece_type > 9: piece_type-9
    directions = []
    if piece_type == 1: #歩
        directions = [0]
    elif piece_type == 2: #銀
        directions = [0,1,3,5,7]
    elif piece_type == 3: #金
        directions = [0,1,2,4,6,7]
    elif piece_type == 4:# 香車
        directions = [0,10,11,12]
    elif piece_type == 5:#ト
        directions = [0,1,2,4,6,7]
    elif piece_type == 6:#飛
        directions = [0,2,4,6,10,11,12,13,14,15,16,17,18,19,20,21]
    elif piece_type == 7:#角
        directions = [1,3,5,7,22,23,24,25,26,27,28,29,30,31,32,33]
    elif piece_type == 8:#桂馬
        directions = [8,9]
    elif piece_type == 9:#王
        directions = [0,1,2,3,4,5,6,7]
```

```
#合法手取得
if piece_type == 1 or piece_type == 2 or piece_type == 3  or piece_type == 5  or piece_type
    for direction in directions:
        #駒の移動元を計算
        x = position_src%5 + self.dxy[direction][0]
        y = int(position_src/5) + self.dxy[direction][1]
        #(x = 0, y = 4)
        p = x + y * 5
        if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
            actions.append(self.position_to_action(p,direction))

elif piece_type == 4 or piece_type == 6: #香車と飛車
    for direction in directions:
        check = True
        if self.dxy[direction][0] > 0:
            for i in range(1,self.dxy[direction][0]):
                x = position_src%5 + i
                y = int(position_src/5)+0
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                    check = True
                else:
                    check = False
                    break
            if check == True:
                x = position_src%5+self.dxy[direction][0]
                y = int(position_src/5)
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                    actions.append(self.position_to_action(p,direction))

        elif self.dxy[direction][0] < 0:
            for i in range(-1,self.dxy[direction][0],-1):
                x = position_src%5 + i
                y = int(position_src/5)+0
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
```

```
                check = True
            else:
                check = False
                break
        if check == True:
            x = position_src%5+self.dxy[direction][0]
            y = int(position_src/5)
            p = x + y * 5

            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                actions.append(self.position_to_action(p,direction))

    elif self.dxy[direction][1] > 0:
        for i in range(1,self.dxy[direction][1]):
            x = position_src%5+0
            y = int(position_src/5)+i
            p = x + y * 5

            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                check = True
            else:
                check = False
                break
        if check == True:
            x = position_src%5+0
            y = int(position_src/5)+self.dxy[direction][1]
            p = x + y * 5

            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                actions.append(self.position_to_action(p,direction))

    elif self.dxy[direction][1] < 0:
        for i in range(-1,self.dxy[direction][1],-1):
            x = position_src%5+0
            y = int(position_src/5)+i
            p = x + y * 5

            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                check = True
            else:
```

```
                    check = False
                    break
            if check == True:
                x = position_src%5+0
                y = int(position_src/5)+self.dxy[direction][1]
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                    actions.append(self.position_to_action(p,direction))
elif piece_type == 7: #角
    for direction in directions:
        check = True
        if self.dxy[direction][0] > 0 and self.dxy[direction][1] < 0: #右下
            for i in range(1,self.dxy[direction][0]):
                x = position_src%5 + i
                y = int(position_src/5)+(-1*i)
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                    check = True
                else:
                    check = False
                    break
            if check == True:
                x = position_src%5+self.dxy[direction][0]
                y = int(position_src/5)+(self.dxy[direction][1])
                p = x + y * 5

                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                    actions.append(self.position_to_action(p,direction))

        elif self.dxy[direction][0] < 0 and self.dxy[direction][1] < 0:#左下
            for i in range(-1,self.dxy[direction][0],-1):
                x = position_src%5 + i
                y = int(position_src/5)+i
                p = x + y * 5
                if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                    check = True
                else:
                    check = False
```

```
                break
        if check == True:
            x = position_src%5+self.dxy[direction][0]
            y = int(position_src/5)+self.dxy[direction][1]
            p = x + y * 5
            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                actions.append(self.position_to_action(p,direction))

    elif self.dxy[direction][1] > 0 and self.dxy[direction][0] > 0: #右上
        for i in range(1,self.dxy[direction][1]):
            x = position_src%5+i
            y = int(position_src/5)+i
            p = x + y * 5
            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                check = True
            else:
                check = False
                break
        if check == True:
            x = position_src%5+self.dxy[direction][0]
            y = int(position_src/5)+self.dxy[direction][1]
            p = x + y * 5
            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                actions.append(self.position_to_action(p,direction))

    elif self.dxy[direction][1] > 0 and self.dxy[direction][0] < 0:#左上
        for i in range(-1,self.dxy[direction][1],-1):
            x = position_src%5+i
            y = int(position_src/5)+(-1*i)
            p = x + y * 5

            if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0 and self.e
                check = True
            else:
                check = False
                break
        if check == True:
            x = position_src%5+self.dxy[direction][0]
            y = int(position_src/5)+self.dxy[direction][1]
            p = x + y * 5
```

```
                    if 0 <= x and x <= 4 and 0 <=y and y <= 4 and self.pieces[p] == 0:
                        actions.append(self.position_to_action(p,direction))

    return actions



#行動に応じた状態を取得
def next(self,action):
    #状態作成
    state = State(self.pieces.copy(), self.enemy_pieces.copy(), self.depth+1)

    #移動先と移動元に変換
    position_dst , position_src = self.action_to_position(action)

    #駒の移動
    if position_src < 34:
        #駒の移動元
        x = position_dst%5 - self.dxy[position_src][0]
        y = int(position_dst/5) - self.dxy[position_src][1]
        position_src = x + y * 5

    #駒の移動
        state.pieces[position_dst] = state.pieces[position_src] #移動後
        state.pieces[position_src] = 0 #移動前の場所は 0 に

        #歩と飛車
        if state.pieces[position_dst] == 1:
            state.pieces[position_dst] = 6
        #銀と角
        elif state.pieces[position_dst] == 2:
            state.pieces[position_dst] = 7
        #金と桂
        elif state.pieces[position_dst] == 3:
            state.pieces[position_dst] = 8
        #香車とト
        elif state.pieces[position_dst] == 4:
```

```
        state.pieces[position_dst] = 5

    elif state.pieces[position_dst] == 6:
        state.pieces[position_dst] = 1
    elif state.pieces[position_dst] == 7:
        state.pieces[position_dst] = 2
    elif state.pieces[position_dst] == 8:
        state.pieces[position_dst] = 3
    elif state.pieces[position_dst] == 5:
        state.pieces[position_dst] = 4
```

#相手の駒が存在する時

```
    piece_type = state.enemy_pieces[24-position_dst]
    if piece_type != 0:
        if piece_type == 1 or piece_type == 6:
            state.pieces[24+1] += 1
            state.pieces[24+6] += 1
        elif piece_type == 2 or piece_type == 7:
            state.pieces[24+2] += 1
            state.pieces[24+7] += 1
        elif piece_type == 3 or piece_type == 8:
            state.pieces[24+3] += 1
            state.pieces[24+8] += 1
        elif piece_type == 4 or piece_type == 5:
            state.pieces[24+4] += 1
            state.pieces[24+5] += 1
        state.enemy_pieces[24-position_dst] = 0
```

#持ち駒の配置

```
else:
    capture = position_src - 33
    state.pieces[position_dst] = capture
    if capture == 1 or capture == 6:
        state.pieces[24+1] -= 1
        state.pieces[24+6] -= 1
    elif capture== 2 or capture == 7:
        state.pieces[24+2] -= 1
        state.pieces[24+7] -= 1
    elif capture == 3 or capture == 8:
```

```
            state.pieces[24+3] -= 1
            state.pieces[24+8] -= 1
        elif capture == 4 or capture == 5:
            state.pieces[24+4] -= 1
            state.pieces[24+5] -= 1

    #駒の交代
    w = state.pieces
    state.pieces = state.enemy_pieces
    state.enemy_pieces = w
    return state


def is_first_player(self):
    return self.depth%2 != 0


def __str__(self):
    pieces0 = self.pieces if self.is_first_player() else self.enemy_pieces
    pieces1 = self.enemy_pieces if self.is_first_player() else self.pieces
    koma0 = ('','歩','銀','金','香','ト','飛','角','桂','王')
    koma1 = ('','ふ','ぎ','き','や','と','ひ','か','け','お')

    str = '['
    for i in range(25,33):
        if pieces1[i] >= 2: str += koma1[i-24]
        if pieces1[i] >= 1: str += koma1[i-24]
    str += ']\n'

    for i in range(25):
        if pieces0[i] != 0:
            str += koma0[pieces0[i]]
        elif pieces1[24-i] != 0:
            str += koma1[pieces1[24-i]]
        else:
            str += '一'
        if i % 5 == 4:
            str += '\n'

    str += '['
```

```
        for i in range(25,33):
            if pieces0[i] >= 2: str += koma0[i-24]
            if pieces0[i] >= 1: str += koma0[i-24]
        str += ']\n'
        return str

def random_action(state):
    legal_actions = state.legal_actions()
    return legal_actions[random.randint(0, len(legal_actions)-1)]

def human_action(state):
    legal_actions = state.legal_actions()
    for i in legal_actions:
        print(int(i/42),i%42)
    return legal_actions[int(input())]
```

dual_network.py ファイル

```
# パッケージのインポート
from tensorflow.keras.layers import Activation, Add, BatchNormalization, Conv2D, Dense, GlobalAverag
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
from tensorflow.keras import backend as K
import os

# パラメータの準備
DN_FILTERS  = 128 # 畳み込み層のカーネル数（本家は 256）
DN_RESIDUAL_NUM =  16 # 残差ブロックの数（本家は 19）
DN_INPUT_SHAPE = (5, 5, 34) # 入力シェイプ（配置が 5*5 の）
DN_OUTPUT_SIZE = 1050 # 行動数（駒の移動先 (25)*駒の移動元 (42)）

# 畳み込み層の作成
def conv(filters):
    return Conv2D(filters, 3, padding='same', use_bias=False,
        kernel_initializer='he_normal', kernel_regularizer=l2(0.0005))

# 残差ブロックの作成
def residual_block():
    def f(x):
        sc = x
```

```
        x = conv(DN_FILTERS)(x)
        x = BatchNormalization()(x)
        x = Activation('relu')(x)
        x = conv(DN_FILTERS)(x)
        x = BatchNormalization()(x)
        x = Add()([x, sc])
        x = Activation('relu')(x)
        return x
    return f

# デュアルネットワークの作成
def dual_network():
    # モデル作成済みの場合は無処理
    if os.path.exists('./model/best.h5'):
        return

    # 入力層
    input = Input(shape=DN_INPUT_SHAPE)

    # 畳み込み層
    x = conv(DN_FILTERS)(input)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # 残差ブロック x 16
    for i in range(DN_RESIDUAL_NUM):
        x = residual_block()(x)

    # プーリング層
    x = GlobalAveragePooling2D()(x)

    # ポリシー出力
    p = Dense(DN_OUTPUT_SIZE, kernel_regularizer=l2(0.0005),
              activation='softmax', name='pi')(x)

    # バリュー出力
    v = Dense(1, kernel_regularizer=l2(0.0005))(x)
    v = Activation('tanh', name='v')(v)

    # モデルの作成
```

```
    model = Model(inputs=input, outputs=[p,v])

    # モデルの保存
    os.makedirs('./model/', exist_ok=True) # フォルダがない時は生成
    model.save('./model/best.h5') # ベストプレイヤーのモデル

    # モデルの破棄
    K.clear_session()
    del model
```

pv_mcts.py ファイル

```
from game import State
from dual_network import DN_INPUT_SHAPE
from math import sqrt
from tensorflow.keras.models import load_model
from pathlib import Path
import numpy as np

# パラメータの準備
PV_EVALUATE_COUNT = 100 # 1 推論あたりのシミュレーション回数（本家は 1600）

# 推論
def predict(model, state):
    # 推論のための入力データのシェイプの変換
    a, b, c = DN_INPUT_SHAPE
    x = np.array(state.pieces_array())
    x = x.reshape(c, a, b).transpose(1, 2, 0).reshape(1, a, b, c)

    # 推論
    y = model.predict(x, batch_size=1)

    # 方策の取得
    policies = y[0][0][list(state.legal_actions())] # 合法手のみ
    policies /= sum(policies) if sum(policies) else 1 # 合計 1 の確率分布に変換

    # 価値の取得
    value = y[1][0][0]
    return policies, value

# ノードのリストを試行回数のリストに変換
```

```
def nodes_to_scores(nodes):
    scores = []
    for c in nodes:
        scores.append(c.n)
    return scores

# モンテカルロ木探索のスコアの取得
def pv_mcts_scores(model, state, temperature):
    # モンテカルロ木探索のノードの定義
    class Node:
        # ノードの初期化
        def __init__(self, state, p):
            self.state = state # 状態
            self.p = p # 方策
            self.w = 0 # 累計価値
            self.n = 0 # 試行回数
            self.child_nodes = None  # 子ノード群

        # 局面の価値の計算
        def evaluate(self):
            # ゲーム終了時
            if self.state.is_done():
                # 勝敗結果で価値を取得
                value = -1 if self.state.is_lose() else 0

                # 累計価値と試行回数の更新
                self.w += value
                self.n += 1
                return value

            # 子ノードが存在しない時
            if not self.child_nodes:
                # ニューラルネットワークの推論で方策と価値を取得
                policies, value = predict(model, self.state)

                # 累計価値と試行回数の更新
                self.w += value
                self.n += 1

                # 子ノードの展開
```

```
            self.child_nodes = []
            for action, policy in zip(self.state.legal_actions(), policies):
                self.child_nodes.append(Node(self.state.next(action), policy))
            return value

        # 子ノードが存在する時
        else:
            # アーク評価値が最大の子ノードの評価で価値を取得
            value = -self.next_child_node().evaluate()

            # 累計価値と試行回数の更新
            self.w += value
            self.n += 1
            return value

    # アーク評価値が最大の子ノードを取得
    def next_child_node(self):
        # アーク評価値の計算
        C_PUCT = 1.0
        t = sum(nodes_to_scores(self.child_nodes))
        pucb_values = []
        for child_node in self.child_nodes:
            pucb_values.append((-child_node.w / child_node.n if child_node.n else 0.0) +
                C_PUCT * child_node.p * sqrt(t) / (1 + child_node.n))

        # アーク評価値が最大の子ノードを返す
        return self.child_nodes[np.argmax(pucb_values)]

# 現在の局面のノードの作成
root_node = Node(state, 0)

# 複数回の評価の実行
for _ in range(PV_EVALUATE_COUNT):
    root_node.evaluate()

# 合法手の確率分布
scores = nodes_to_scores(root_node.child_nodes)
if temperature == 0: # 最大値のみ 1
    action = np.argmax(scores)
    scores = np.zeros(len(scores))
```

```
        scores[action] = 1
    else: # ボルツマン分布でバラつき付加
        scores = boltzman(scores, temperature)
    return scores

# モンテカルロ木探索で行動選択
def pv_mcts_action(model, temperature=0):
    def pv_mcts_action(state):
        scores = pv_mcts_scores(model, state, temperature)
        return np.random.choice(state.legal_actions(), p=scores)
    return pv_mcts_action

# ボルツマン分布
def boltzman(xs, temperature):
    xs = [x ** (1 / temperature) for x in xs]
    return [x / sum(xs) for x in xs]
```

self_play.py ファイル

```
from game import State
from pv_mcts import pv_mcts_scores
from dual_network import DN_OUTPUT_SIZE
from datetime import datetime
from tensorflow.keras.models import load_model
from tensorflow.keras import backend as K
from pathlib import Path
import numpy as np
import pickle
import os

# パラメータの準備
SP_GAME_COUNT = 250 # セルフプレイを行うゲーム数（本家は25000）
SP_TEMPERATURE = 1.0 # ボルツマン分布の温度パラメータ

# 先手プレイヤーの価値
def first_player_value(ended_state):
    # 1:先手勝利，-1:先手敗北，0:引き分け
    if ended_state.is_lose():
        return -1 if ended_state.is_first_player() else 1
    return 0
```

```
# 学習データの保存
def write_data(history):
    now = datetime.now()
    os.makedirs('./data/', exist_ok=True) # フォルダがない時は生成
    path = './data/{:04}{:02}{:02}{:02}{:02}{:02}.history'.format(
        now.year, now.month, now.day, now.hour, now.minute, now.second)
    with open(path, mode='wb') as f:
        pickle.dump(history, f)

# 1 ゲームの実行
def play(model):
    # 学習データ
    history = []

    # 状態の生成
    state = State()

    while True:
        # ゲーム終了時
        if state.is_done():
            break

        # 合法手の確率分布の取得
        scores = pv_mcts_scores(model, state, SP_TEMPERATURE)

        # 学習データに状態と方策を追加
        policies = [0] * DN_OUTPUT_SIZE
        for action, policy in zip(state.legal_actions(), scores):
            policies[action] = policy
        history.append([state.pieces_array(), policies, None])

        # 行動の取得
        action = np.random.choice(state.legal_actions(), p=scores)

        # 次の状態の取得
        state = state.next(action)

    # 学習データに価値を追加
    value = first_player_value(state)
    for i in range(len(history)):
```

```
        history[i][2] = value
        value = -value
    return history

# セルフプレイ
def self_play():
    # 学習データ
    history = []

    # ベストプレイヤーのモデルの読み込み
    model = load_model('./model/best.h5')

    # 複数回のゲームの実行
    for i in range(SP_GAME_COUNT):
        # 1 ゲームの実行
        h = play(model)
        history.extend(h)

        # 出力
        print('\rSelfPlay {}/{}'.format(i+1, SP_GAME_COUNT), end='')
    print('')

    # 学習データの保存
    write_data(history)

    # モデルの破棄
    K.clear_session()
    del model
```

train_network.py ファイル

```
from dual_network import DN_INPUT_SHAPE
from tensorflow.keras.callbacks import LearningRateScheduler, LambdaCallback
from tensorflow.keras.models import load_model
from tensorflow.keras import backend as K
from pathlib import Path
import numpy as np
import pickle

# パラメータの準備
RN_EPOCHS = 900 # 学習回数
```

```
# 学習データの読み込み
def load_data():
    history_path = sorted(Path('./data').glob('*.history'))[-1]
    with history_path.open(mode='rb') as f:
        return pickle.load(f)

# デュアルネットワークの学習
def train_network():
    # 学習データの読み込み
    history = load_data()
    xs, y_policies, y_values = zip(*history)

    # 学習のための入力データのシェイプの変換
    a, b, c = DN_INPUT_SHAPE
    xs = np.array(xs)
    xs = xs.reshape(len(xs), c, a, b).transpose(0, 2, 3, 1)
    y_policies = np.array(y_policies)
    y_values = np.array(y_values)

    # ベストプレイヤーのモデルの読み込み
    model = load_model('./model/best.h5')

    # モデルのコンパイル
    model.compile(loss=['categorical_crossentropy', 'mse'], optimizer='adam')

    # 学習率
    def step_decay(epoch):
        x = 0.001
        if epoch >= 450: x = 0.0005
        if epoch >= 720: x = 0.00025
        return x
    lr_decay = LearningRateScheduler(step_decay)

    # 出力
    print_callback = LambdaCallback(
        on_epoch_begin=lambda epoch,logs:
                print('\rTrain {}/{}'.format(epoch + 1,RN_EPOCHS), end=''))

    # 学習の実行
```

```
    model.fit(xs, [y_policies, y_values], batch_size=128, epochs=RN_EPOCHS,
            verbose=0, callbacks=[lr_decay, print_callback])
    print('')

    # 最新プレイヤーのモデルの保存
    model.save('./model/latest.h5')

    # モデルの破棄
    K.clear_session()
    del model
```

evalute_network.py ファイル

```
from game import State
from pv_mcts import pv_mcts_action
from tensorflow.keras.models import load_model
from tensorflow.keras import backend as K
from pathlib import Path
from shutil import copy
import numpy as np

# パラメータの準備
EN_GAME_COUNT = 20 # 1評価あたりのゲーム数（本家は400）
EN_TEMPERATURE = 1.0 # ボルツマン分布の温度

# 先手プレイヤーのポイント
def first_player_point(ended_state):
    # 1:先手勝利，0:先手敗北，0.5:引き分け
    if ended_state.is_lose():
        return 0 if ended_state.is_first_player() else 1
    return 0.5

# 1ゲームの実行
def play(next_actions):
    # 状態の生成
    state = State()

    # ゲーム終了までループ
    while True:
        # ゲーム終了時
        if state.is_done():
```

```
            break;

        # 行動の取得
        next_action = next_actions[0] if state.is_first_player() else next_actions[1]
        action = next_action(state)

        # 次の状態の取得
        state = state.next(action)

    # 先手プレイヤーのポイントを返す
    return first_player_point(state)

# ベストプレイヤーの交代
def update_best_player():
    copy('./model/latest.h5', './model/best.h5')
    print('Change BestPlayer')

# ネットワークの評価
def evaluate_network():
    # 最新プレイヤーのモデルの読み込み
    model0 = load_model('./model/latest.h5')

    # ベストプレイヤーのモデルの読み込み
    model1 = load_model('./model/best.h5')

    # PV MCTS で行動選択を行う関数の生成
    next_action0 = pv_mcts_action(model0, EN_TEMPERATURE)
    next_action1 = pv_mcts_action(model1, EN_TEMPERATURE)
    next_actions = (next_action0, next_action1)

    # 複数回の対戦を繰り返す
    total_point = 0
    for i in range(EN_GAME_COUNT):
        # 1 ゲームの実行
        if i % 2 == 0:
            total_point += play(next_actions)
        else:
            total_point += 1 - play(list(reversed(next_actions)))

        # 出力
```

```
        print('\rEvaluate {}/{}'.format(i + 1, EN_GAME_COUNT), end='')
    print('')

    # 平均ポイントの計算
    average_point = total_point / EN_GAME_COUNT
    print('AveragePoint', average_point)

    # モデルの破棄
    K.clear_session()
    del model0
    del model1

    # ベストプレイヤーの交代
    if average_point > 0.5:
        update_best_player()
        return True
    else:
        return False
```

human_play.py ファイル

```
from game import State
from pv_mcts import pv_mcts_action
from tensorflow.keras.models import load_model
from pathlib import Path
from threading import Thread
import tkinter as tk
from PIL import Image, ImageTk

# ベストプレイヤーのモデルの読み込み
model = load_model('./model/best.h5')

# ゲーム UI の定義
class GameUI(tk.Frame):
    # 初期化
    def __init__(self, master=None, model=None):
        tk.Frame.__init__(self, master)
        self.master.title('京都将棋')

        # ゲーム状態の生成
        self.state = State()
```

```
        self.select = -1 # 選択(-1:なし, 0〜24:マス, 25〜32:持ち駒)

        # 方向定数
        self.dxy = ((0,-1),(1,-1),(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),
                    (1,-2),(-1,-2),
                    (0,-2),(0,-3),(0,-4),
                    (2,0),(3,0),(4,0),
                    (0,2),(0,3),(0,4),
                    (-2,0),(-3,0),(-4,0),
                    (2,-2),(3,-3),(4,-4),
                    (2,2),(3,3),(4,4),
                    (-2,2),(-3,3),(-4,4),
                    (-2,-2),(-3,-3),(-4,-4))

        # PV MCTSで行動選択を行う関数の生成
        self.next_action = pv_mcts_action(model, 0.0)

        # イメージの準備
        self.images = [(None, None, None, None, None, None, None, None, None)]
        for i in range(1, 10):
            image = Image.open('piece{}.png'.format(i))
            self.images.append((
                ImageTk.PhotoImage(image),
                ImageTk.PhotoImage(image.rotate(180)),
                ImageTk.PhotoImage(image.resize((40, 40))),
                ImageTk.PhotoImage(image.resize((40, 40)).rotate(180))))

        # キャンバスの生成
        self.c = tk.Canvas(self, width = 400, height = 480, highlightthickness = 0)
        self.c.bind('<Button-1>', self.turn_of_human)
        self.c.pack()

        # 描画の更新
        self.on_draw()

    # 人間のターン
    def turn_of_human(self, event):
        # ゲーム終了時
        if self.state.is_done():
            self.state = State()
```

```
        self.on_draw()
        return

    # 先手でない時
    if not self.state.is_first_player():
        return self.master.after(1, self.turn_of_ai)

    # 持ち駒の種類の取得
    captures = []
    for i in range(8):
        if self.state.pieces[25+i] >= 2: captures.append(1+i)
        if self.state.pieces[25+i] >= 1: captures.append(1+i)

    # 駒の選択と移動の位置の計算（0〜24:マス，25〜32:持ち駒）
    p = int(event.x/80) + int((event.y-40)/80) * 5 #5 は縦横の盤面
    if 40 <= event.y and event.y <= 440:
        select = p
    elif event.x < len(captures) * 40 and event.y > 440:
        select = 25 + int(event.x/40)
    else:
        return

    # 駒の選択
    if self.select < 0:
        self.select = select
        self.on_draw()
        return

    # 駒の選択と移動を行動に変換
    action = -1
    if select < 25:
        # 駒の移動時
        if self.select < 25:
            action = self.state.position_to_action(p, self.position_to_direction(self.select, p))
        # 持ち駒の配置時
        else:
            action = self.state.position_to_action(p, 33+captures[self.select-25])

    # 合法手でない時
    if not (action in self.state.legal_actions()):
```

```
            self.select = -1
            self.on_draw()
            return

        # 次の状態の取得
        self.state = self.state.next(action)
        self.select = -1
        self.on_draw()

        # AI のターン
        self.master.after(1, self.turn_of_ai)

    # AI のターン
    def turn_of_ai(self):
        # ゲーム終了時
        if self.state.is_done():
            return

        # 行動の取得
        action = self.next_action(self.state)

        # 次の状態の取得
        self.state = self.state.next(action)
        self.on_draw()

    # 駒の移動先を駒の移動方向に変換
    def position_to_direction(self, position_src, position_dst):
        dx = position_dst%5-position_src%5
        dy = int(position_dst/5)-int(position_src/5)
        for i in range(34):
            if self.dxy[i][0] == dx and self.dxy[i][1] == dy: return i
        return 0

    # 駒の描画
    def draw_piece(self, index, first_player, piece_type):
        x = (index%5)*80
        y = int(index/5)*80+40
        index = 0 if first_player else 1
        self.c.create_image(x, y, image=self.images[piece_type][index],  anchor=tk.NW)
```

```
# 持ち駒の描画
def draw_capture(self, first_player, pieces):
    index, x, dx, y = (2, 0, 40, 440) if first_player else (3, 200, -40, 0)
    captures = []
    for i in range(8):
        if pieces[25+i] >= 2: captures.append(1+i)
        if pieces[25+i] >= 1: captures.append(1+i)
    for i in range(len(captures)):
        self.c.create_image(x+dx*i, y, image=self.images[captures[i]][index],  anchor=tk.NW)

# カーソルの描画
def draw_cursor(self, x, y, size):
    self.c.create_line(x+1, y+1, x+size-1, y+1, width = 4.0, fill = '#FF0000')
    self.c.create_line(x+1, y+size-1, x+size-1, y+size-1, width = 4.0, fill = '#FF0000')
    self.c.create_line(x+1, y+1, x+1, y+size-1, width = 4.0, fill = '#FF0000')
    self.c.create_line(x+size-1, y+1, x+size-1, y+size-1, width = 4.0, fill = '#FF0000')

# 描画の更新
def on_draw(self):
    # マス目
    self.c.delete('all')
    self.c.create_rectangle(0, 0, 400, 480, width = 0.0, fill = '#EDAA56')
    for i in range(1,5):
        self.c.create_line(i*80+1, 40, i*80, 440, width = 2.0, fill = '#000000')
    for i in range(6):
        self.c.create_line(0, 40+i*80, 400, 40+i*80,  width = 2.0, fill = '#000000')

    # 駒
    for p in range(25):
        p0, p1 = (p, 24-p) if self.state.is_first_player() else (24-p, p)
        if self.state.pieces[p0] != 0:
            self.draw_piece(p, self.state.is_first_player(), self.state.pieces[p0])
        if self.state.enemy_pieces[p1] != 0:
            self.draw_piece(p, not self.state.is_first_player(), self.state.enemy_pieces[p1])

    # 持ち駒
    self.draw_capture(self.state.is_first_player(), self.state.pieces)
    self.draw_capture(not self.state.is_first_player(), self.state.enemy_pieces)

    # 選択カーソル
```

```
        if 0 <= self.select and self.select < 25:
            self.draw_cursor(int(self.select%5)*80, int(self.select/5)*80+40, 80)
        elif 25 <= self.select:
            self.draw_cursor((self.select-25)*40, 360, 40)
```

#ゲーム UI の実行

```
f = GameUI(model = model)
f.pack()
f.mainloop()
```