Kuhn PokerBot Group 18

Release v1.2.0

Xianbo XU, Yao Lu, Bairui Han, Wen-Hung Huang

Apr 06, 2022

CONTENTS:

Welcome to Kuhn PokerBot Group 18's documentation!

This documentation is for APIs of the PokerBot of Group 18

Kuhn poker is an extremely simplified form of poker developed by Harold W. Kuhn as a simple model zero-sum two-player imperfect-information game, amenable to a complete game-theoretic analysis. In Kuhn poker, the deck includes only three playing cards, for example a King, Queen, and Jack. One card is dealt to each player, which may place bets similarly to a standard poker. If both players bet or both players pass, the player with the higher card wins, otherwise, the betting player wins. [KuhnPoker]

CHAPTER

ONE

AGENT MODULE

For agent class, we treat image classifiers and action generators as two components of the agent. So, two member variables *self.model* and *self.strategy* are introduced to provide services for the agent. By designing like this, our agent class will focus on how to pass information from image classifiers to the strategy generator. And the agent is responsible for passing information between internal components image classifiers, strategy generator and external controller. Also, we can easily use new image classifiers and new strategy generators to replace current components without having to consider the implementation of the agent. We just make sure the two components follow interface specifications and change the loaded *self.model`* and *self.strategy*. Then our agent can run normally without any further changes in the agent class.

class agent.**PokerAgent**

Bases: object

make_action(*state:* client.state.ClientGameState, *round:* client.state.ClientGameRoundState) → str

Next action, used to choose a new action depending on the current state of the game. This method implements your unique PokerBot strategy. Use the state and round arguments to decide your next best move.

Parameters

- **state** (ClientGameState) – State object of the current game (a game has multiple rounds)
- **round** (ClientGameRoundState) – State object of the current round (from deal to showdown)

Returns A string representation of the next action an agent wants to do next, should be from a list of available actions

Return type str in ['BET', 'CALL', 'CHECK', 'FOLD'] (and in round.get_available_actions())

on_error(*error*)

This methods will be called in case of error either from server backend or from client itself. You can optionally use this function for error handling.

Parameters **error** (*str*) – string representation of the error

on_game_end(*state:* client.state.ClientGameState, *result: str*)

This method is called once after the game has ended. A game ends automatically. You can optionally use this method for logging purposes.

Parameters

- **state** (ClientGameState) – State object of the current game
- **result** (*str in ['WIN', 'DEFEAT']*) – End result of the game

on_game_start(*gametype: str*)

This method will be called once at the beginning of the game when server confirms both players have connected.

on_image(*image*)

This method is called every time when card image changes. Use this method for image recongition procedure.

Parameters **image** (*Image*) – Image object

on_new_round_request(*state:* client.state.ClientGameState)

This method is called every time before a new round is started. A new round is started automatically. You can optionally use this method for logging purposes.

Parameters **state** (ClientGameState) – State object of the current game

on_round_end(*state:* client.state.ClientGameState, *round:* client.state.ClientGameRoundState)

This method is called every time a round has ended. A round ends automatically. You can optionally use this method for logging purposes.

Parameters

- **state** (ClientGameState) – State object of the current game
- **round** (ClientGameRoundState) – State object of the current round

CHAPTER

TWO

MODELS PACKAGE

2.1 Model handling:

As for model part, we hoped our model to be robust for different noise images so we generated different noise level images with big rotation and put all these images to training set. Our models would be able to learn many kinds of noisy images and finally the model had great accuracy on even very high noise. The CNN model performed better than FCN and we built both models with 3 and 4 classes outputs.

2.2 Submodules

2.3 models.CNN3 module

class models.CNN3.**model**

Bases: *models.base_model.PokerModelBase*

Model of Image classifier PokerModelBase provide the interfaces definition for image classifiers. model define output with 3 classes using CNN

forward(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **l3** – Output tensor with score for classes

Return type tensor

predict(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **labels** – The predicted labels are returned in shape [n_batches].

Return type torch.tensor

prob(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted probabilities for each class are returned in shape [n_batches,n_classes].

Return type torch.tensor

training: bool

2.4 models.CNN4 module

class models.CNN4.**model**

Bases: *models.base_model.PokerModelBase*

Model of Image classifier PokerModelBase provide the interfaces definition for image classifiers. model define output with 4 classes using CNN

forward(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **l3** – Output tensor with score for classes

Return type tensor

predict(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **labels** – The predicted labels are returned in shape [n_batches].

Return type torch.tensor

prob(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted probabilities for each class are returned in shape [n_batches,n_classes].

Return type torch.tensor

training: bool

2.5 models.FCN3 module

class models.FCN3.**model**

Bases: *models.base_model.PokerModelBase*

Model of Image classifier PokerModelBase provide the interfaces definition for image classifiers. model define output with 3 classes using FCN

forward(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **l3** – Output tensor with score for classes

Return type tensor

predict(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **labels** – The predicted labels are returned in shape [n_batches].

Return type torch.tensor

prob(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted probabilities for each class are returned in shape [n_batches,n_classes].

Return type torch.tensor

training: **bool**

2.6 models.FCN4 module

class models.FCN4.**model**

Bases: *models.base_model.PokerModelBase*

Model of Image classifier PokerModelBase provide the interfaces definition for image classifiers. model define output with 4 classes using FCN

forward(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **l3** – Output tensor with score for classes

Return type tensor

predict(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns **labels** – The predicted labels are returned in shape [n_batches].

Return type torch.tensor

prob(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted probabilities for each class are returned in shape [n_batches,n_classes].

Return type torch.tensor

training: **bool**

2.7 models.base_model module

class models.base_model.**PokerModelBase**

Bases: torch.nn.modules.module.Module

Abstract base class for Image classifier PokerModelBase provide the interfaces definition for image classifiers. By the abstract base class, the implementation details of different models are hidden to the agent.

abstract forward(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Return type Defined by subclasses

abstract predict(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted labels are returned in shape [n_batches].

Return type torch.tensor

abstract prob(*x*)

Parameters **x** (*tensor*) – Input tensor for the neural network

Returns The predicted probabilities for each class are returned in shape [n_batches,n_classes].

Return type torch.tensor

training: bool

2.8 Module contents

CHAPTER

THREE

STRATEGY PACKAGE

3.1 Strategy handling:

We used Counterfactual Regret Minimization(CFR) for deciding which action we would like to use. The CFR has ability to explore every possible result of the action we made and reach Nash equilibrium. Since there does not exist any strategy that can guarantee we could win every single game, using the strategy that can reach Nash equilibrium becomes a good option. The advantage of using the Nash equilibrium strategies is that our exploitability is minimum. Therefore, our agent will not totally malfunction even though the opponent knows our strategy or we gave sufficient exploitative power. Since kuhn poker is a simple poker game, the whole result can be easily calculated by algorithm. We just directly used the result from CFR and used the random seed, which is an uniform distribution, to decide which action we want to take. For the PokerBot, we will engage two types of games, 3 cards and 4 cards, we used CFR sample code from the internet and modified it to generate 3 cards and 4 cards result. The reference website as following url: https://justinsermeno.com/posts/cfr/

3.2 Submodules

3.3 strategy.agent_strategy module

class strategy.agent_strategy.**PokerStrategy**

Bases: *strategy.base_strategy.StrategyBase*

CFR strategy implementation PokerStrategy uses the probabilistic results of actions generated by CFR algorithm to determine what action the poker agent should take each time for a given game type.

card3strategy() → str

Using random seed to decide which action we would do. The probability range is the result from CFR algorithm.

Note: The CFR result3 cards:

Player1 strategy: card history [“CHECK”, “BET”]J [“”] [0.79, 0.21]J [“CB”] [1.00, 0.00]Q [“”] [1.00, 0.00]Q [“CB”] [0.45, 0.55]K [“”] [0.39, 0.61]K [“CB”] [0.00, 1.00]

Player2 strategy: card history [“CHECK”, “BET”]J [“B”] [1.00, 0.00]J [“C”] [0.67, 0.33]Q [“B”] [0.66, 0.34]Q [“C”] [1.00, 0.00]K [“B”] [0.00, 1.00]K [“C”] [0.00, 1.00]

---

card4strategy() → str

Using random seed to decide which action we would do. The probability range is the result from CFR algorithm.

Note: The CFR result4 cards:

Player1 strategy: card history [“CHECK”, “BET”]J [“”] [0.75, 0.25]J [“CB”] [1.00, 0.00]Q [“”] [1.00, 0.00]Q [“CB”] [0.75, 0.25]K [“”] [1.00, 0.00]K [“CB”] [0.00, 1.00]A [“”] [0.25, 0.75]A [“CB”] [0.00, 1.00]

Player2 strategy: card history [“CHECK”, “BET”]J [“B”] [1.00, 0.00]J [“C”] [0.51, 0.49]Q [“B”] [0.79, 0.21]Q [“C”] [1.00, 0.00]K [“B”] [0.20, 0.80]K [“C”] [0.51, 0.49]A [“B”] [0.00, 1.00]A [“C”] [0.00, 1.00]

get_strategy() → str

3.4 strategy.base_strategy module

class strategy.base_strategy.**StrategyBase**

Bases: object

Base class for game strategy StrategyBase provides the interface definitions for game strategy, and defines the set value functions for inheritance.

get_strategy()

set_avaliable_actions(*avaliable_actions: list*)

set_bank(*bank: int*)

set_current_card(*current_card: str*)

set_gametype(*gametype: int*)

set_move_history(*move_history: list*)

set_order(*order: int*)

set_roll_the_dice(*RollTheDice: bool*)

3.5 Module contents

CHAPTER

FOUR

DATA_SETS MODULE

4.1 Data handling

For data handling, we generated noisy images with different rotate angle and noise level for later training. After generated data we extracted features from the images which aimed to remove the noise as much as possible. Because our noise followed uniform distribution so most of the noise intensity was bigger than 0(the pure black) so we regarded those pixels as background.

data_sets.**extract_features**(*img: <module 'PIL.Image' from '/home/xianbo/miniconda3/lib/python3.9/site-packages/PIL/Image.py'>*)

Convert an image to features that serve as input to the image classifier.

Parameters **img** (*Image*) – Image to convert to features.

Returns **features** – Extracted features in a format that can be used in the image classifier.

Return type list/matrix/structure of int, int between zero and one

data_sets.**generate_data_set**(*n_samples: int*, *data_dir: str*, *num_label: int = 4*, *noise_level: float = 0.2*) → None

Generate n_samples noisy images by using generate_noisy_image(), and store them in data_dir.

Parameters

- **n_samples** (*int*) – Number of train/test examples to generate
- **data_dir** (*str*) – Directory for storing images. *TRAINING_IMAGE_DIR*, *TEST_IMAGE_DIR* are predefined path for training and testing.
- **num_label** (*int in [3,4], default: 4*) – Number of unique labels to generate. First 'num_label' lables in predefined LABELS will be used.
- **noise_level** (*flat in range [0,1], default: 0.2*) – Probability with which a given pixel is randomized.

Examples

```
>>> generate_data_set(30,TRAINING_IMAGE_DIR,4,0.2)
30 pictures will be saved to TRAINING_IMAGE_DIR
```

data_sets.**generate_noisy_image**(*rank*, *noise_level*)

Generate a noisy image with a given noise corruption. This implementation mirrors how the server generates the images. However the exact server settings for noise_level and ROTATE_MAX_ANGLE are unknown. For the PokerBot assignment you won't need to update this function, but remember to test it.

Parameters

- **rank** (*str in ['J', 'Q', 'K','A']*) – Original card rank.
- **noise_level** (*int between zero and one*) – Probability with which a given pixel is randomized.

Returns **noisy_img** – A noisy image representation of the card rank.

Return type Image

Examples

```
>>> generate_noisy_image("J",0.2)
```

data_sets.**load_data_set**(*data_dir*, *n_validation*)

Prepare features for the images in data_dir and divide in a training and validation set.

Parameters

- **data_dir** (*str*) – Directory of images to load
- **n_validation** (*int*) – Number of images that are assigned to the validation set

Returns

- **training_features** (*list*) – Containing the arrays of int between 0 and 1 which are the training images features.
- **training_labels** (*list*) – Containing the training labels of str which contain alphabet among' J,Q,K,A'.
- **validation_features** (*list*) – Containing the arrays of int between 0 and 1 which are the validation images features.
- **validation_labels** (*list*) – Containing the validation labels of str which contain alphabet among' J,Q,K,A'.

CHAPTER

FIVE

CLIENT PACKAGE

This part is provided from the teacher. So, we don't add much information about this part.

5.1 Submodules

5.2 client.controller module

5.3 client.events module

class client.events.**ClientRequestEventsIterator**

Bases: object

close()

is_closed()

make_request(*request*)

next()

set_initial_request(*request*)

5.4 client.state module

class client.state.**ClientGameRoundState**(*coordinator_id*, *round_id*)

Bases: object

ClientGameRoundState tracks the state of the current round, from deal to showdown. Attributes should be accessed through their corresponding getter and setter methods. For the PokerBot assignment you should not modify the setter methods yourself (only test them).

_coordinator_id

Unique game coordinator identifier (token), duplicate from ClientGameState._coordinator_id

Type str

_round_id

Round counter, starts from 1

Type int

_card

Current card in hand; '?' means the exact card rank is unknown and has to be recognized from _card_image

Type str, in ['J', 'Q', 'K', '?']

_card_image

Current card image in hand

Type Image

_turn_order

Player turn position for the current round, player '1' acts first

Type int, in [1, 2]

_moves_history

Previously made actions of both players. Actions in the list alternate between players, i.e., the first element is the first action of player '1', and the second element is the first action of player '2', etc. The last element of _moves_history is the last action made by your opponent. If you're the first to move, _moves_history will be empty.

Type list of str

_available_actions

Available actions this turn, e.g., on the first move, _available_actions = ['BET', 'CHECK', 'FOLD'].

Type list of str, where str in subset of ['BET', 'CHECK', 'FOLD', 'CALL']

_outcome

Amount of chips won this round. Negative values indicate a loss.

Type str

_cards

Cards at showdown for both players, concatenated in player order. I.e., 'KJ' indicates player '1' holds a 'K', and player '2' holds a 'J'. If the opposing player folds, a question-mark is returned for that player's card; i.e. 'K?' indicates the card for player '2' was not revealed at showdown.

Type str

add_move_history(*move*)

get_available_actions()

get_card()

get_card_image()

get_cards()

get_coordinator_id()

get_moves_history()

get_outcome()

get_round_id()

get_turn_order()

is_ended()

set_available_actions(*available_actions*)

set_card(*card*)

set_card_image(*card_image*)

set_cards(*cards*)

set_moves_history(*moves_history*)

set_outcome(*outcome*)

set_turn_order(*order*)

class client.state.**ClientGameState**(*coordinator_id*, *player_token*, *player_bank*)

Bases: object

A ClientGameState object tracks a specific game between two players. A game consists of multiple rounds from deal to showdown. Attributes should be accessed through their corresponding getter and setter methods. For the PokerBot assignment you should not modify the setter methods yourself (only test them).

_coordinator_id

Game coordinator identifier token

Type str

_player_token

Unique player identifier token

Type str

_player_bank

Amount of player credit chips

Type int

_rounds

Tracks the individual rounds played in this game

Type list of ClientGameRoundState

get_coordinator_id()

get_last_round_state() → *client.state.ClientGameRoundState*

get_player_bank()

get_player_token()

get_rounds()

start_new_round()

update_bank(*outcome*)

5.5 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[KuhnPoker] Kuhn poker - Wikipedia https://en.wikipedia.org/wiki/Kuhn_poker

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N

O

P

S

T

U