Fatjack

An introduction to reinforcement learning with Blackjack game

Javier Mañá Solsona

July 2021

Preliminary observations:

Objectives: this document aims to discuss the implementation of reinforcement learning algorithms to solve the casino game blackjack.

To the target audience: I provide some basic information about both the core concepts of reinforcement learning and the rules of blackjack with the hope of not making the document abstruse for those who do not have previous knowledge of those topics.

Document structure: I first introduce the personal motivations behind this project, what to expect from it, why I have chosen reinforcement learning and blackjack, and how I have approached the task in comparison with previous works; then, I undertake the task of outlining the main ideas behind reinforcement learning and which version of Blackjack I have attempted to solve. Subsequently, I depict the architecture of the project, which I have designed to obtain some operational leverage; next, I broach the methodology, constraints, and the results thereof; finally, I draw the conclusions and discuss some future extensions of this work; additionally, I have added some guidelines to interact with my final data product, a web interface.

Caveat: although this memorandum seeks to be succinct but complete, the Jupyter Notebooks attached in the repository therewith also contain relevant information; for the sake of clarity, I have chosen to set out some concepts or decisions before or after their associated pieces of code. Thus, having a look at those notebooks is advisable to gain full insight into this project and explicit references to them are used to shorten the present document (avoiding prolixity). Moreover, a great deal of effort has been made to present Notebooks in a clean and didactic manner. Some knowledge of Python programming language is recommended to understand them.

Table of contents:

Introduction

Why reinforcement learning?

As Henderson et al. (2017) point out, the number of whitepapers dedicated to reinforcement learning (also abbreviated as RL hereunder) has dramatically grown over the last decade, albeit far from the current prominence of supervised and unsupervised learning. Therefore, it could be easily stated that RL is on the up, especially after some notorious achievements such as AlphaGo (AlphaGo - The movie, 2017) – the software that beat the go world champion.

Nonetheless, since most of the efforts in the field have been devoted to both solving games and expand the theoretical body of knowledge, some authors are sceptical about the advent of monetising opportunities for RL and consider that this branch of machine learning will be tried and found wanting when applied to real business problems (Lorica, 2021). Those concerns stem from the fact that, although one of the advantages of RL over other machine learning techniques is that it hardly relies on large datasets, learning is achieved by building simulated environments. Consequently, how well an environment mimics reality becomes the overarching aspect and the number of simulations needed to achieve acceptable results drastically increases for complex tasks.

Anyways, such a lively debate portends that there is lot to come with regards to this area of machine learning in the forthcoming future and that it is worth paying close attention to how developments unfold from its very beginning.

As a learning experience for a data-scientist-in-the-making, this project has offered me a string of opportunities:

First, I have honed my programming skills, ability that I consider of paramount importance. Unlike supervised learning and unsupervised learning, there are not so many vastly used out-of-the-box resources for RL such as Scikit-learn or Keras. Their counterparts in RL are basically focus on developing benchmark environments to try algorithms out, not algorithms themself. The most well-known library may be Gym. Therefore, I have put a great deal of effort to rise to the challenge of coding the agents myself.

Second, it is undeniable that making use of out-of-the-box libraries is very handy and alluring but also somewhat deceiving: their adoption is so simple that glossing over details of the underlying theory becomes tempting. I have not let myself be beset by those temptations and followed a study-first-then-work approach. I think that practising that ability is also important: one cannot expect that there is always a quick and already implemented solution for a given problem.

Third, one ends up gaining resilience when a task is started at its very bottom: with no prior knowledge of it at all.

Finally, the more challenging it becomes, the funnier it is. And the funnier, the better.

Why blackjack?

The main reason why I have chosen blackjack is that, given that I did not have any previous knowledge of both RL and data science in general, it did not seem sound to undertake a complex task as a first foray into such a complex topic. Moreover, my focus has been on understanding the basic theory and developing methodology.

Having said so, there are other important reasons:

First, it is a well-known game so that it might be easier to reach a wider audience and generate interest in the project and the skills of the author; heaviness on details and complexity in rules are usually deterrent.

Additionally, I deem blackjack ideal to make RL a tangible data product for those who might just want to scan the project or its very outcome (recruiters, for example); episodes are short, and it takes little time to evaluate the overall course of the game.

Finally, as claimed above, RL implementations tend to make use of video games and grown-ups playing computer games may still carry stigma. Avoiding such possibility seems appropriate.

Considerations on previous works and founding ground for this project:

By surfing the net, it swiftly becomes undeniable that RL has been already applied to blackjack and some of those past attempts seem an outright success. Even more so, the blackjack problem has been employed for didactical purposes.

Nevertheless, I have noticed some aspects that might qualify the assertions above and be worth addressing in this project.

1. **Lack of scalability**: if showcased; the agent's computations, the game logic, and the environment design are highly entangled in most solutions (Mahajan, 2021 as an example). Consequently, those approaches are task-specific and cannot scale up to more complex problems. In other words, code is not easily reusable. I thought that it would be appropriate to build three independent and interactive layers, so that new future tasks can be undertaken by simply extending the current project (adding new environments).
2. **Lack of replicability:** as Henderson et al. (2019) claim, how RL dissertations and projects are commonly approach poses some questions in terms of replicability, especially when it comes to comparing multiple algorithms. Whether or not the code is disclosed, whether the hyperarameters' values are explicitly stated, the intrinsic non-deterministic nature of the environments, the lack of confidence bounds when presenting results, and the code itself are factors that hinder performance assessment of others' work. Heeding Henderson's advice, I discuss how to generate confidence bounds to present consistent results and account for the stochasticity of the task.

Framing the problem in dialectic terms:

This project addresses the following questions:

1. Is it possible to develop a playing-to-win strategy for blackjack by using RL techniques?
2. If so, how good is that strategy?
3. If not, how close to not losing money can RL get?
4. Is it possible to induce a blackjack strategy without drawing on the vast literature on the topic and combinatorial probability?

From an operational point of view:

1. Can this project be a spawning ground for more complex tasks in the future by embracing scalability as a design principle?

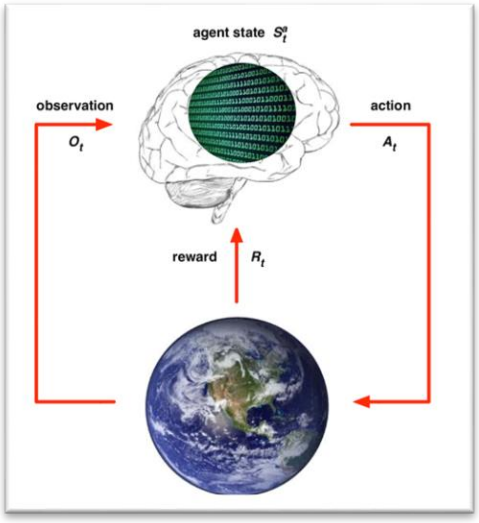
Theoretical background

In this section, I would like to outline the basic concepts for those who might find themselves in my starting point (no prior knowledge of the topic).

First, I have chiefly used the **Reinforcement Learning – An Introduction book (Sutton et al., 2018)** as an ushering guidance into the minimally necessary theory to handle this project.

Before briefly summarising the main concepts, it is worth having a quick word on the most basic RL terminology, which is repeatedly used over the whole project:

- Agent: it is a software program that can receive input signals, make decisions with respect to them, and improve its decision-making skills over time. No prior knowledge of the task at hand is necessary.
- Environment: it is the software program a reinforcement learning agent interacts with to learn a particular task.
- Time-step: every interaction between the agent and the environment.
- Episode: a set of time-steps that represents the execution of the whole task. An episode ends when the agent hits a final state, also known as terminal state.
- Observation: it is the signal sent to the agent by the environment at each time-step.
- Reward: it is a scalar value that indicates how well an agent does at a given time-step.
- Action: it is the decision an agent makes at a given time-step.
- State: it is the agent's internal representation of the environment and comprises both observation and reward, at least.
- Policy: the behaviour of the agent. The goal is to find the optimal policy; that is, the sequence of actions that maximise reward. It is commonly denoted by π symbol.



A basic scheme on how the concepts relate to each other (source: David Silver Course)

One of the first computations an agent should consider is that of accumulated reward at a given time-step t, which can be recursively expressed as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + + \gamma^{T-1} r_T$$
$$= r_{t+1} + \gamma(r_{t+2} + \gamma\, r_{t+3} + \gamma^2 r_{t+4} + \cdots + + \gamma^{T-2} r_T)$$
$$= r_{t+1} + \gamma(G_{t+1})$$

The accumulated reward is discounted at a gamma rate [0,1]. It helps both adjust how far-sighted/short-sighted the agent is and account for infinity when the task is continuous (when there is no terminal state at all).

Any RL environment is formally described as a Markov decision process in which all states are Markov (the states capture all the relevant information from the past). It consists of:

1. A finite set of states (S).
2. A finite set of actions (A).
3. A state transition probability matrix (P), where $P^a_{ss'} = P\,[S_{t+1} = s' \mid S_t = s, A_t = a]$.
4. A reward function, where $R^a{}_s = E\,[R_{t+1} \mid S_t = s, A_t = a]$.
5. A discount factor.

As the ulterior goal is to maximise reward, reinforcement learning algorithms entail estimating how well an agent fares in terms of expected reward from a random state *s* onwards. This idea can be mathematically shaped in two ways:

- The state-value function: it computes the expected return when starting in *s* and following a policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

- The action-value function: it computes the expected return starting from *s*, taking the action *a*, and following π from then onwards.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Following the foundations of dynamic programming, the legendary mathematician Richard Bellman broke down the value function into two components: the immediate reward and the future discounted values. In such fashion, the problem becomes recursive. The so-called Bellman equation is:

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S}
\end{aligned}
$$

A RL task is completed/solved when the optimal state-value and action-value functions are found. Such situation is mathematically defined by:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a)$$

Bellman also demonstrated that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s', r} p(s', r \,|\, s, a) \big[r + \gamma v_*(s') \big].
\end{aligned}
$$

The last formula is known as the Bellman optimality equation and its q version is as follows:

$$
\begin{aligned}
q_*(s, a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a \Big] \\
&= \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma \max_{a'} q_*(s', a') \Big].
\end{aligned}
$$

The Bellman optimality equation is non-linear, so iterative methods must be used for such tasks. Furthermore, one of the features that makes reinforcement learning attractive is that those functions are learned from experience, and it is not necessary to know the transition probability matrix (p) beforehand.

At this point, I would like to table some standard classifications of the RL area that help narrow down the scope of this project.

As far as the environment representation and computational design are concerned, RL algorithms are mainly divided into two categories:

1. Tabular methods: the state universe and the associated value function are encapsulated in a table and the agent keeps track of rewards and states to update either the action-value or state-value function for a policy π. Tables are generally called V-table or Q-table, depending on the goal of agents.
2. Function approximation methods: instead of using a table, neural networks are trained to map states and actions.

This project only deals with algorithms of the first category.

RL agent can pursue two types of goal:

1. Prediction: agents estimate the state-value function for a given policy. There is no policy improvement. I have used two algorithms to carry out prediction:
    a. Montecarlo
    b. Temporal Difference (TD) Lambda
2. Control: agents attempt to improve policy. I have implemented the following algorithms for such purpose:
    a. Sarsa
    b. Q-Learning
    c. Every visit Montecarlo
    d. Off-Policy Montecarlo
    e. Sarsa Lambda
    f. Watkins Lambda

Additionally, RL algorithms are classified according to when they internally update their estimations:

1. Montecarlo (MC): agents run entire episodes and update the value function backwards (also known as off-line update). Montecarlo methods have high variance but zero bias. My chosen MC algorithms are:
    a. Every visit Montecarlo
    b. Off-Policy Montecarlo

2. Temporal Difference (TD): agents update the value function at every time-step and use bootstrapping to guess what lies ahead from the current state onwards (also known as on-line update). Temporal Difference methods have low variance and bias. My chosen TD algorithms are:
    a. Sarsa
    b. Sarsa Lambda
    c. Q-Learning
    d. Watkins

Regarding which policy agents try to improve, algorithms can be classified in:

1. On-policy: agents improve the policy they use to make decisions. My on-policy algorithms are:
    a. Every visit Montecarlo
    b. Sarsa
    c. Sarsa Lambda
2. Off-policy: agents improve a policy different from the one used to make decisions. My off-policy algorithms are:
    a. Off-policy Montecarlo
    b. Q-Learning
    c. Watkins

Finally, it is worth mentioning that I have devoted /01_Montecarlo.ipynb and /02_TemporalDifference.ipynb to provide theoretical explanation of each implemented algorithm.

Blackjack – The game

Blackjack is a casino card game, which is usually played with the French 52-card deck. Unlike other games such as poker, players compete against nobody but the dealer (Wikipedia, 2021).

The goal of the game is to achieve card totals greater than the dealer's but not exceed 21 points – automatic loss befalls a player in such situation. A player can ask for more cards as long as she/he does not exceed that score; the dealer must stop handing cards out to himself if she/he hits or exceeds 17 points. The values for each card are:

- Ace: 1 or 11 points at player's convenience.
- Number cards: their own value.

- Face cards (jack, queen and king): 10 points.

The following concepts are part of the frequent terminology of the game:

- Hit: ask the dealer for another card.
- Stand: stop asking the dealer for more cards.
- Double down: double the initial bet and just ask for one more card.
- Split: create two hands from an initial one in which a player's cards have the same value.
- Surrender: redeem half of the bet and end the hand.
- House edge: it is the inherent statistical advantage that the dealer enjoys. Every version of the game entails such advantage, and its value completely hinges on the enforced set of rules. **So, one early conclusion is that it is impossible to beat the dealer in the long run and the main target is to minimise the expected loss as much as possible.**
- Go bust exceed 21 points.
- Blackjack: a combination of an ace and a 10-point card. It is unbeatable and adding up to 21 points with any other combination does not level the hand up.
- Soft hand/usable ace hand: a hand in which an ace is worth 21 points.
- Soft 17: the dealer has a soft hand that is worth 17 points.

There are more than a hundred variations of blackjack (Wikipedia, 2021), and every casino sets its rules at its own discretion. Some of the aspects of the game affected by those changes in rules are: the number of decks (from 1 to 8, in general), whether or not the dealer must stand on soft 17, how much pay-out is offered when obtaining a blackjack, whether or not splitting or/and double down are allowed, whether or not the dealer beats a player when a draw occurs, and whether or not insurance is offered to players.

That myriad of possibilities makes the house edge a very fluctuating value. However, it is considered that the house edge is around 1% on average; that is, players lose 1% of their actions in the long run (Wikipedia, 2021).

I have used a simple version of the game to undertake the project and the set of rules is as follows:

1. Players are initially handed out two cards.
2. Dealer hands one card out to himself, and that card is observable for players.
3. 6 decks are used with full reshuffling after each hand.

4. Dealer stands on soft 17.
5. No split.
6. No insurance offered.
7. No surrender.
8. Blackjack pays 3:2.
9. A draw can take place.

Theoretically, the house edge for such version of the game is in the region of 2.35% (Blackjack House Edge - No splitting/No Doubling discussed in Math/Questions and Answers at Wizard of Vegas - Page 2, 2021)**.**

Architecture

Although this project does not have an external source of data and a feature engineering thereof, it is necessary to discuss how data is internally generated.

Requirements and technical details

1. A Miniconda/Anaconda package manager.
2. The project has been developed on an Ubuntu 20.04.2 LTS operating system, but it is intended to be cross-platform for most of its operations, except for some of them that are mentioned in the point right below.
3. Some operations carried out in some notebooks are multiprocessed and do not work on Windows. Anyways, the notebooks are presented after execution and results can be perfectly read regardless of the OS.
4. The project is written in Python, except for the web-based front-end that has been written in HTML5, CSS, and Vanilla JavaScript powered by the D3 library (for plotting).
5. The Python version and third-party packages used are listed in hist-environment.yml file in the Github repository.
6. To create an environment with those libraries, execute on a terminal: *conda env create –f hist-environment.yml* in the root folder of the project.
7. To make sure that the project properly runs, notebooks and folders must be at the same filesystem level. In other words, the Github repository structure must be exactly replicated on the local filesystem. Otherwise, notebooks cannot load the RL libraries. This requirement includes renaming neither folders nor files.
8. Terminal commands must be executed from the root folder of the project.

General overview

The architecture that I have designed aims to gain some operational leverage. That is, I have built abstractions for those operations that are repeatedly executed over the project. From such perspective, I identified two scopes of abstraction:

1. **Project-wide abstractions:** initializing a blackjack game as a RL environment and executing multiple hands, executing RL algorithms, persisting results, keeping a results history, consolidating results, and presenting results using a graphical user interface. For such purpose, I have deployed four hand-made modules: games, environments, interfaces, and agents. Additionally, I have included three folders to stow persisted information: results, stored_agents and tables.
2. **Notebook-wide abstractions:** plotting information and running multiple RL experiments in a parametric fashion. Some those functionalities could have also had a project-wide scope. Nonetheless, every notebook builds upon the previous and keeping those operations at notebook level helps set down the gradually evolving nature of the project on an easily assessable instrument like Jupyter Notebooks.

"games" module

First, it is not necessary to understand this module to merely execute this project, since it is only used by the "environments" module to convert the game into an RL environment. There is no direct interaction with it.

As any Python module, it has an "__init__" file.

It is a hand-made Python module that encapsulates the blackjack game logic. The following classes have been developed:

- **Decks:** it represents one or more decks of cards. The deck is built by executing the cartesian product of the card values and suits (itertools). The number of decks wanted is passed as an argument when instantiating the class.
- **BlackjackDecks:** a subclass of Decks. It adds the blackjack card values for each type of card (combination of suite and number/letter). It also allows picking cards from the decks and know how many cards are left in the decks for each type.
- **BlackjackPlayer:** it represents a player, so that it implements methods such as: requesting cards, placing bets, and compute cash status (gains and money left).
- **Hand:** it represents a blackjack hand, so that it implements: whether a hand is owned by a player or the dealer ("croupier"); appending cards to it; computing values such as

minimum value, maximum value and best value; whether a hand has gone bust; whether a hand is a blackjack; and whether a hand is over 17.

- **Blackjack:** it contains the game logic itself. The following methods and properties are implemented: the number of players around the table, dealing with players' actions (hit and stand), deciding who wins a hand, handing out cards to the dealer as well.
- **Exceptions:** there are a string of exception classes that allow ensuring that the course of the game is properly implemented. For example, it warns that a bet placement has been made after receiving some cards (correctly, bet is placed before a player is handed a pair of cards out).

If this module is of interest beyond this project, a terminal executable implementation is stored in /interfaces/bash_interface.py. This file showcases how to deploy the whole module for a multi-player blackjack.

"environments" module

It turns the blackjack game supplied by game module into a RL environment.

In terms of environment design, I have used the observation representation described by Widrow, Gupta and Maitra (1973). That representation entails a three-element tuple containing for each time step in a hand:

1. The players' total at a given time step.
2. The dealer's initial card value at a given time step.
3. Whether or not the agent has a usable ace at a given time step.

Some parts of the 01_Montecarlo.ipynb are devoted to introducing users to the environment management (instantiation and usage). They also show the reward function applied and how actions must be passed to the environment.

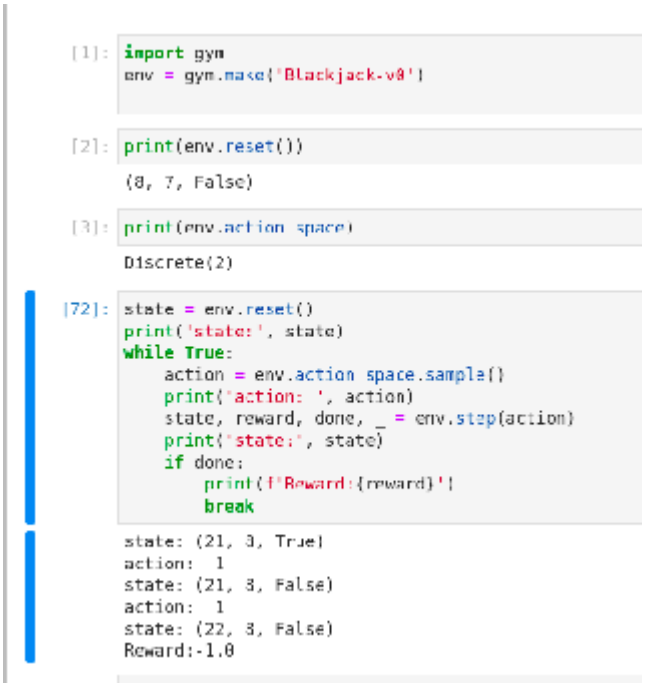
As any Python module, it has an "__init__" file. Additionally, it consists of two more files:

- **base.py:** it only contains one method that retrieves an instance of the desired environment, and it is called "make". There is only one environment thus far ("hitstand"). Nevertheless, it looks like a gentle way to instance environments, especially if more new environments are added in the future - hopefully.
- **blackjack.py:** it contains three classes:
    - **BaseEnviroment:** it is an empty class that indicates the methods to implement for any subsequent environment. It imitates the methods of a Gym environment.

- **HitStand:** it overrides three methods of the BaseEnvironment class:
  - render: it allows printing hands on the notebook cells by setting the property verbose to True.
  - reset: it starts off a new Blackjack instance from the games module and sends an agent the first observation. The initial bet is always one monetary unit, so that results can have a quick translation into percentage terms. It returns a four-element tuple:
    - Observation: the state representation.
    - Reward: the reward given for falling into the observation
    - Terminal: whether or not the state ends the hand.
    - Description: the cards' names handed out by the Blackjack instance for that hand.
  - Step: it allows sending the environment an action and retrieve the following time step in the shape of a four-element tuple as described above.
- **ResetEnvironmentError:** it warns that step method has been called before resetting the environment and thus having a Blackjack instance available.

It is worth having a quick word on why I have not directly used the blackjack environment provided by Gym library. After examining it, I found three quirks:

1. It allows picking cards once a player hits 21 points. That situation is not realistic at all, since it is not contemplated in any version of blackjack (see screenshot below).
2. Card reshuffling/replacement must be avoided over a hand. So that the probability to pick a card that has already been handed out must be lowered. The fulfilment of such requirement does not look clear either.
3. Observations provided are just numbers, so no text encodement of cards is provided (e.g.: AS for a spade ace). I thought that having such a handy feature from the very start could reduce time for rendering the environment in a graphical user interface at more advanced stages of the project.



"agents" module

This module abstracts the algorithms and provide agents with some other persistence-oriented capabilities.

As any Python module, it has an "__init__" file.

The "agents.py" file exposes the following classes:

- **Agent:** it implements all those methods that are common for all types of agents. Not so self-explanatory methods are:
    - table: it contains the v/q-table of an agent. It is a Numpy array. There are two types of table initialization: null (all 0s) and random. I always use null initializations for all deployed agents, since the latter entails adding more stochasticity to agent evaluation and affect convergence to an optimal policy.
    - time_step_counter: it is a Numpy array that tracks how many times a state has been visited.
    - table_look_up: Numpy's array indices start at value 0. However, it may not be the case for environments' observation spaces. This method helps translate values of the latter into those of the former. It is usually used every time an agent receives an observation.
    - evaluate_state: this method must be implemented in subsequent subclasses, so that a NotImplementedError is raised if called from the Agent class. It must

implement the algorithm itself. Montecarlo algorithms must also implement the update_table method, since the table update takes place offline and the whole hand course must be swept back to compute values properly.

- follow_policy: this method must be implemented in subsequent subclasses in the notebooks. It must define how an agent plays given an observation and return an action within the environment's action space.

Herewith find enclose the relationship between subclasses of the Agent class and the algorithms implemented:

Algorithm	Subclass
Every visit Montecarlo	MontecarloController
Off-Policy Montecarlo	OffPolicyMontecarlo
QLearning	QLearning
Sarsa	Sarsa
Sarsa Lambda	SarsaLambda
Watkins Lambda	WatkinsLambda

To deploy an agent at its fullest, the above-mentioned subclasses must be subclassed again in a notebook and implement the follow_policy method.

Additionally, the same document offers the following methods:

folderpath_search: it recursively searches for a given folder. This method is only used by other methods in the module and helps find persistence-related folders regardless of the computer executing the project – avoiding the project becoming directory-tree-dependant.

find_results: it retrieves the binary file paths containing results of a given agent.

consolidate_results: it merges the binary files containing the results of a given agent into a single binary file, whose name ends by "_CON".

list_saved_agents: if no value is given to the filter parameter, it lists all files stored in "stored_agents" table. If 'unique' value is given to the filter parameter, it just returns unique values of agents stored in that folder – same agent might have been persisted many times at different points in time; in such case, the retrieved values have the following format: "A_" + agent's class name + _ + agent's id.

get_agent: it retrieves the instance of a persisted agent. To do so, third-party library **Dill** is brought to bear. It accepts a parameter called criterion: when 'most_trained' is specified, it searches for the most trained version of the agent thas has been persisted: when 'most_recent'

is specified, it searches for the latest timestamp of the persisted versions of the agent. In both cases, filename parameter just needs the following string: "A_" + agent's class name + _ + agent's id. Finally, if no value is given to that parameter, the whole name of the searched binary file must be given as filename parameter. This function is generally executer after the list_saved_agents method.

After having read 01_Montecarlo.ipynb and 02_Temporal_Difference.ipynb, the reader should be familiar with deploying agents and their general capabilities.

"interfaces" module

The web folder contains all necessary files to run the final data product as a web page, which is available at http://34.77.255.37/. Anyways, it can be locally run by executing the following command on terminal: *python3 –m interfaces.web*. The documents are organised as follows:

- "__main__.py": it runs the server-side of the web page. It is just a small flask application serving the following calls:
    - "main": it sends the initial page.
    - "start": it receives an agent id and sends its main features to be displayed on the interface.
    - "play": it sends fully terminated blackjack hands to be displayed on the interface.
    - "results": it sends the reward time series of the chosen agent to be plotted on the interface.
    - "policy": it sends the policy matrix of the chosen agent to be plotted on the interface. It only sends the most trained matrix available.
- "templates" folder: it only contains an HTML file defining the web page layout. It is slightly powered by Jinja library (Python) to fill in the dropdown wherein available agents are listed.
- "static" folder: it contains the static resources of the web page:
    - "img": images such as the blackjack table and the cards themselves.
    - "css": the CSS code to prettify the web page layout.
        - "main.css": style applied to the web page in general.
        - "plot.css": plot-related prettifiers.
    - "js": code to improve web page interactivity written in Vanilla Javascript language.

- "main.js": it runs the general interactivity of the webpage and defines the AJAX calls to execute the methods served by the web server document.
- "plot.js": it generates the web page's plot and is powered by third-party library d3.
- "d3.v7.js": it contains the third-party library to power plots up.
- "formatter.js": some general JS functions to format numbers and used by both "plot.js" and "main.js".

"results" folder

It is used to store agents' results after their training. Every binary file in the folder is named with this structure: "results_" + agent's class name + _ + agent's id + _ + an additional identifying term. If they end by "_CON", it means that that file contains a consolidated version of all results stemming from an agent at different points in time. They are the ones retrieved by the web interface.

Python's **pickle** library has been chosen to perform those persistence operations, as the underlying objects are just dictionaries.

"stored_agents" folder

It is used to persist agents themselves. Every binary file in the folder is named with this structure: "A_" + agent's class name + _ + agent's id + _ + the total number of trained episodes up to that point.

It was impossible to persist agents with Python's pickle library, so that third-party D**ill** library is brought to bear instead.

"tables" folder

It is used to persist agents' q/v-tables. Every binary file in the folder is named with this structure: "T_" + agent's class name + _ + agent's id + _ + the total number of trained episodes up to that point.

Python's **pickle** library has been chosen to perform those persistence operations, as the underlying objects are just Numpy arrays.

The notebooks

Structure

Every notebook attached to this project follows the same structure:

1. Goals: it is stated what the purpose of the notebook is and what should be achieved at the end of it.
2. Library importations: every library used throughout the notebook is declared beforehand.
3. Plot utilities: it exposes methods that help draw recurrent plots.
4. Experiment definition: it contains the methods that allow interaction between agents and environments.
5. Notebook-specific code: the necessary code execution is carried out to achieve the goals stated at the beginning of the notebook.

That structure may sometimes look repetitive, but it intends to build some narrative throughout the project and showcase the gradually evolving nature of it; methods are tweaked and expanded from one notebook to another to fulfil new requirements. Even more so, some pieces of code that look notebook-specific are permanently adopted into subsequent notebooks to boost performance and readability.

Finally, code execution has not always worked at a first attempt and some design decisions have shifted during the project. To keep notebooks as clean and tidy as possible, I have pruned them back.

Notebooks' workload parallelization

From 03_Train_Test.ipynb onwards, I have used Python's **multiprocessing** library to speed up execution, since I foresaw that training a large number of agents for a large number of episodes would be computationally expensive.

Multiprocessed operations are only brought to bear when it comes to evaluating a policy. Then, a group of agents can be given the same q-table, play independently for a while, and finally average all their results.

Methodology and constraints

Preparatory work

As stated before, 01_Montecarlo.ipynb and 02_Temporal_Difference.ipynb aim to make sure that algorithms are properly implemented, which is a necessary condition for carrying out RL experimentation.

I have developed 04_Exploration.ipynb to introduce a last concept, the exploitation and exploration dilemma, which is also important for subsequent experimentation.

Defining a metric

To gauge agent performance, I have picked the following metric:

Reward per betted m.u. = total reward over n episodes ÷ total cash betted

m.u stands for monetary unit.

As the amount betted per hand is 1 monetary unit, total cash betted can be easily replaced by the number of episodes played.

Using a win rate (number of hands won/total hands played) could have been another potential metric, but I have discarded it for three reasons:

1. Blackjack wins are worthier than "normal" wins, and that must be somehow taken into account.
2. The agent's reward function would have been fully detached from the metric and that adds complexity. That is, I could not straight use output signals of the environment to calculate the metric.
3. The complementary value of a win rate contains two more values: the loss rate and the draw rate. Thus, it is not fully meaningful by itself, and the other two values must also be presented for completeness.

Train and test split (or learn and evaluate split)

Inspired by the Gransville's (n.d.) work and unlike other works, I do not use the rewards generated by training hands to evaluate agent performance after a given number of hands. Instead, I consider that it is better to keep some separation between training and testing (as if it was a supervised problem). In RL terms, I let an agent train for a while, stop training, and generate test hands with the policy achieved up to that point in time.

However, generating blackjack hands entails some stochasticity and both an agent or a human can have either a good or a bad streak. Would it be appropriate to claim a performance level after a very good streak?

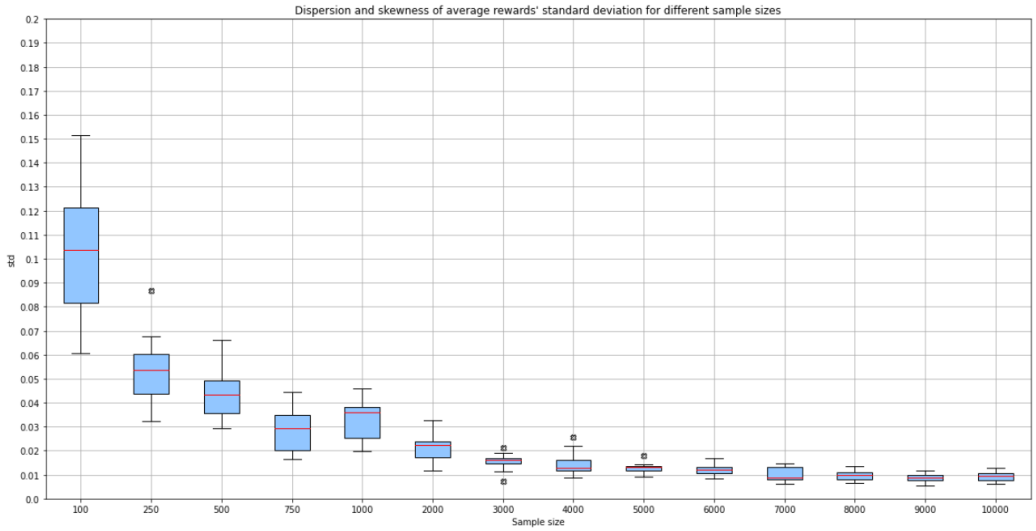
Therefore, two questions arise at evaluation stage:

1. **How many test hands must be averaged to obtain a good estimate of an agent's performance?**

The bigger the sample size, the lower the standard deviation. However, since I did not know either the population's statistics (mean and variance) or what the probability

distribution is, I decided to run an empirical experiment in 03_Train_Test.ipynb. The experiment design is as follows:

1. Pick some possible sample sizes.
2. For each of them, generate 10 samples of that size.
3. Compute average reward and standard deviation.
4. Execute 15 runs (steps 2 & 3).
5. Pick the sample size whose standard deviation is around 0.01.



According to those results, I decided to pick an **8,000-hand sample size**.

2. **How confident should one be about that estimate?**

As the underlying probability distribution of the mean is not known beforehand, I have generated it empirically by applying some sort of bootstrapping. In other words, I have taken 1,000 samples of size 8,000 and generated the distribution for those sample. Finally, I have found what values the two-tail 95% confidence interval lies on. Such experiment definition is also introduced in 03_Train_Test.ipynb.

In conclusion, the following pseudocode applies when training agents and generating its expected reward time series thereof:

1. Train agent for n episodes
2. Evaluate agent's performance after n episodes by generating 1,000 samples of 8,000-hand size.

3. Store average reward with its lower and upper bounds.
4. Repeat steps 1, 2 & 3.

Such approach is used from 03_Train_Test.ipynb onwards.

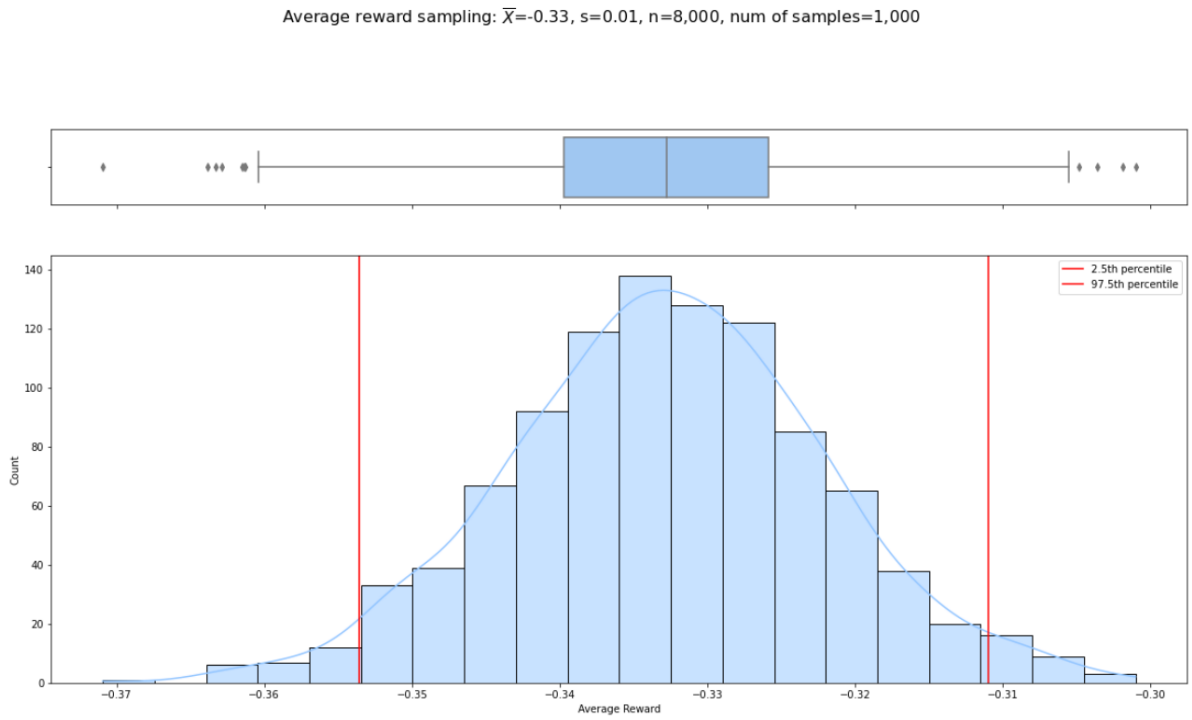
Defining a baseline

In 05_Train_Test.ipynb and before starting off any training of any kind, I have run an agent with a simple deterministic policy to gauge how much improvement is obtained by training agents and let them guess an optimal policy by themselves.

The enforced policy is:

- If the agent's card total is over 19 points, the agent always stands,
- Else hit.

After obtaining 1,000 samples of size 8,000 hands, the expected reward per betted unit distribution is:



It can be safely stated that expected reward per m.u. is **between –0.3536 and –0.3109** with a 95% confidence interval.

Thus far, there are two backdrops against which agent performance can be assessed: the baseline (lower bound) and the house edge (upper bound).

GridSearch Training

In 05_Train_Test.ipynb, I have designed some sort of hand-made gridsearch. In other words, I have created instances of agents with different parametrization and exploration strategies for each algorithm.

These are the parameters evaluated:

- For agents using a constant level of exploration (FixEpsilon):
    - Minimum epsilon
    - Learning rate
    - Discount rate
- For agents using a decaying exploration rate based on the number of visits to a state (VisitsDecay):
    - Minimum epsilon
    - Learning rate
    - Discount rate
- For agents using a constant to decay exploration (Decay rate):
    - Epsilon decay
    - Learning rate
    - Discount rate
- For agents using upper bound confidence for exploration (UCB):
    - C
    - Learning rate
    - Discount rate

Every exploration strategy has been deployed for each algorithm.

For lambda-based algorithms (Watkins Lambda and Sarsa Lambda), some possible values for lambda parameter have been also tabled. As adding new parameters has a multiplicative effect, I have cut down on possible values for c parameter in agents using those algorithms and an upper bound confidence exploration strategy. The only reason to do so is that I was not very confident in the sensitiveness of that parameter in comparison with others before training agents. Additionally, traces parameter remains constant for all agents.

A total of 1,183 RL agents have been created.

Initially, a 1M-episode training session with 10 evaluation points (every 100K episodes) has been executed for every agent.

However, the project has experienced **some limitations** at this point. First, I have had to cut down on the number of bootstrap samples at each evaluation point, from 1,000 to 10. Generating full bootstrapping for each evaluation point makes a training session last up to 80 minutes. Training for 1M episodes with 10 evaluation points and those generating 10 samples of 8,000-hand size takes around 7 minutes. For some every-visit Montecarlo agents, I had already generated the full bootstrapping when I made that decision and I have used those results. Secondly, I had planned to execute two more training sessions subsequently: one to stretch the training per agent up to 10M episodes with an evaluation point every 1M episodes, and another to stretch the training per agent up to 100M episodes with an evaluation point every 10M episodes. Nonetheless, carrying out such plan has been physically impossible due to a time constraint, since the first stretch takes up to 50 minutes per agent and the second one, up to 12 hours per agent. At the point of giving up that plan, I had already trained some agents up to 10M episodes, and 2 agents up to 100M episodes. I have not brought the results thereof up when it comes to comparing all agents for the sake of fairness, but I have taken advantage of that additional training if some of those agents happen to be amongst the best at 1M episode evaluation point.

Finally, it is worth pointing out that the effective number of training episodes is always lesser than the theoretical one. That is due to the fact that straight blackjack hands are not taken into account for learning. Thus, for example, if an agent is trained for 100 episodes, it uses around 95 for training. It is shown in 01_Montecarlo.ipynb.

Results evaluation

Having experienced the constraints explained in the last section, I have had to come up with a methodology to evaluate results, winnow down the list of agents, and pick the best without enforcing further training (if necessary) for the whole bunch of them. Consequently, I have been forced to neglect the fact that some agents may learn later but better and make up for it somehow. Having said so, the strategy is as follows (detailed in 06_Leadeboard.ipynb):

1. Rank agent performances after 1M episode training by the expected value of the metric.

```
[29]: clean_leaderboard.sort_values('mean', ascending=False)
```

[29]:

	algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.05	N/A	N/A	952828	[-0.0055, -0.0350625, -0.010125, -0.0149375, -...	-0.014766	-0.032383	-0.001633
A_QVisitsDecay_a08b71ff7c07496180276315O7a49bbf	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.005	N/A	N/A	952340	[-0.033375, -0.01925, -0.023875, -0.0025, -0.0...	-0.01725	-0.031767	-0.003452
A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.1	N/A	N/A	952506	[-0.0285625, -0.0121875, -0.027625, -0.0255, -...	-0.017961	-0.028398	0.000586
A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a	QLearning	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.005	0.99	N/A	952926	[-0.02725, -0.0185625, -0.0061875, -0.012625, ...	-0.018695	-0.029931	-0.007314
A_UCB_649a110f417048f69014d025ebca6a80	Every visit Montecarlo	Upper bound confidence	1/visits	0.8	N/A	N/A	N/A	N/A	0.5	952646	[-0.0255625, -0.0231875, -0.0315, -0.01725, -0...	-0.018898	-0.030461	-0.011764
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
A_GreedyOffPolicyMC_f2579ef5c7904eef9c88fc53a3aecab4	OffPolicyMontecarlo	E-Greedy	1/visits	0.95	N/A	N/A	N/A	N/A	N/A	952903	[-0.1031875, -0.088625, -0.0688125, -0.0641875...	-0.084109	-0.101263	-0.064997
A_WatkinsFixEpsilon_ead63069fd6c4da3bf201c04283e8421	WatkinsLambda	E-Greedy	1/visits	0.95	0.95	accumulating	0.001	N/A	N/A	952684	[-0.08025, -0.09075, -0.09825, -0.0749375, -0....	-0.088508	-0.100261	-0.075867
A_WatkinsFixEpsilon_aa5c78e6a85a4f129439fbf74383630e	WatkinsLambda	E-Greedy	1/visits	0.8	0.75	accumulating	0.001	N/A	N/A	952618	[-0.0986875, -0.0920625, -0.10025, -0.08725, -...	-0.090445	-0.099977	-0.070297
A_GreedyOffPolicyMC_5df7241d68174995aed48b6aa8a67bea	OffPolicyMontecarlo	E-Greedy	1/visits	0.8	N/A	N/A	N/A	N/A	N/A	952472	[-0.077625, -0.097, -0.1106875, -0.0861875, -0...	-0.096562	-0.110217	-0.079058
A_GreedyOffPolicyMC_46a6e102b8964ab295d74e4fa403c921	OffPolicyMontecarlo	E-Greedy	1/visits	1	N/A	N/A	N/A	N/A	N/A	952463	[-0.11925, -0.138625, -0.1280625, -0.122125, -...	-0.132609	-0.144916	-0.119753

1183 rows × 14 columns

Please zoom in!

2. Check out how many agents are above the house edge (keeping in mind that no full bootstrapping has been employed and results may be somewhat volatile):

```
[32]: clean_leaderboard[clean_leaderboard['mean'] > -0.0235].sort_values('mean', ascending=False)
```

[32]:

	algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.05	N/A	N/A	952828	[-0.0055, -0.0350625, -0.010125, -0.0149375, -...	-0.014766	-0.032383	-0.001633
A_QVisitsDecay_a08b71ff7c07496180276315O7a49bbf	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.005	N/A	N/A	952340	[-0.033375, -0.01925, -0.023875, -0.0025, -0.0...	-0.01725	-0.031767	-0.003452
A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.1	N/A	N/A	952506	[-0.0285625, -0.0121875, -0.027625, -0.0255, -...	-0.017961	-0.028398	0.000586
A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a	QLearning	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.005	0.99	N/A	952926	[-0.02725, -0.0185625, -0.0061875, -0.012625, ...	-0.018695	-0.029931	-0.007314
A_UCB_649a110f417048f69014d025ebca6a80	Every visit Montecarlo	Upper bound confidence	1/visits	0.8	N/A	N/A	N/A	N/A	0.5	952646	[-0.0255625, -0.0231875, -0.0315, -0.01725, -0...	-0.018898	-0.030461	-0.011764
A_DecayRate_a81ffac6ceef4e7abbedab434b11dca6	Every visit Montecarlo	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.05	0.995	N/A	952598	[-0.0173125, -0.0016875, -0.0251875, -0.020687...	-0.01907	-0.027559	-0.004247
A_QVisitsDecay_c688b215e42e40daa54c2249040a9a39	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.005	N/A	N/A	952375	[-0.0244375, -0.013625, -0.010375, -0.032625, ...	-0.019508	-0.035977	-0.000142
A_FixEpsilon_e36595ac2b9a405eb447769717a131bb	Every visit Montecarlo	E-Greedy	1/visits	0.95	N/A	N/A	0.01	N/A	N/A	952792	[-0.0196875, -0.0218125, -0.0055625, -0.015875...	-0.020594	-0.031678	-0.006591
A_UCB_5fa7fc532a824bcdbfcc998fa6e29e29	Every visit Montecarlo	Upper bound confidence	1/visits	0.95	N/A	N/A	N/A	N/A	2	952817	[-0.0313125, -0.021, -0.0128125, -0.013375, -0...	-0.020703	-0.03518	-0.004717
A_QDecayRate_b798d2c5acd64e658f52d0be9169d01e	QLearning	E-Greedy with decay rate	0.01	0.8	N/A	N/A	0.005	0.995	N/A	952993	[-0.0295, -0.0213125, -0.017625, -0.01575, -0....	-0.020758	-0.029314	-0.011316
A_UCB_e1f04e8b5044425fbbf582b3e4930c66	Every visit Montecarlo	Upper bound confidence	1/visits	1	N/A	N/A	N/A	N/A	2	952611	[-0.0193125, -0.001875, -0.043125, -0.0208125,...	-0.020844	-0.039734	-0.004325
A_QVisitsDecay_05dd1207d70c42ba862422488182773f	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.1	N/A	N/A	952916	[-0.006, -0.023375, -0.0255, -0.0096875, -0.01...	-0.021055	-0.034369	-0.006645
A_DecayRate_8829def8ac3b4ebdaa265be9d235939b	Every visit Montecarlo	E-Greedy with decay rate	1/visits	0.8	N/A	N/A	0.05	0.99	N/A	952485	[-0.0061875, -0.0209375, -0.048, -0.0205, -0.0...	-0.021594	-0.044227	-0.004847
A_QFixEpsilon_e492fde058bb4571ae5af01ceccdb62a	QLearning	E-Greedy	0.01	0.95	N/A	N/A	0.05	N/A	N/A	952535	[-0.0190625, -0.0206875, -0.0170625, -0.004625...	-0.021602	-0.039103	-0.006408
A_QDecayRate_394b5b8691cc482db09d69526e04ff55	QLearning	E-Greedy with decay rate	0.01	0.95	N/A	N/A	0.005	0.99	N/A	952715	[-0.009375, -0.03675, -0.0315625, -0.03325, -0...	-0.021953	-0.036137	0.002433
A_QFixEpsilon_e8bd3696dd9644289b994a6ea842ad3d	QLearning	E-Greedy	1/visits	1	N/A	N/A	0.05	N/A	N/A	952643	[-0.028625, -0.0241875, -0.0169375, -0.02575, -...	-0.022828	-0.028122	-0.017561
A_QVisitsDecay_4dad800d2d9949dab130392e31b343df	QLearning	E-Greedy 1/visits decay	0.01	0.8	N/A	N/A	0.1	N/A	N/A	952517	[-0.0160625, -0.010625, -0.0434375, -0.016625,...	-0.02293	-0.042366	-0.010213
A_QDecayRate_f5fd752dff664284837112S6ba526fd7	QLearning	E-Greedy with decay rate	0.01	0.8	N/A	N/A	0.005	0.99	N/A	952524	[-0.02825, -0.0131875, -0.0149375, -0.03375, -...	-0.023055	-0.037256	-0.013494
A_QUCB_b034ef1cdb8a4ff3a343735129c80a6c	QLearning	Upper bound confidence	1/visits	1	N/A	N/A	N/A	N/A	2	952808	[-0.0133125, -0.033625, -0.0146875, -0.0189375...	-0.023156	-0.033728	-0.013553
A_QVisitsDecay_64a3435ea16045e4aa3a76643c262e2e	QLearning	E-Greedy 1/visits decay	0.01	1	N/A	N/A	0.1	N/A	N/A	952857	[-0.0220625, -0.0194375, -0.0335, -0.026, -0.0...	-0.023172	-0.032428	-0.015117
A_UCB_dc93faabf62d4dd7b5f1596ff2e99bcb	Every visit Montecarlo	Upper bound confidence	1/visits	0.95	N/A	N/A	N/A	N/A	1	953002	[-0.0245625, -0.039125, -0.01825, -0.0221875, ...	-0.023344	-0.036577	-0.017889

```
[33]: #how many agents are there?
      len(clean_leaderboard[clean_leaderboard['mean'] > -0.0235])
```

[33]: 21

Please, zoom in!

3. See if some types of algorithms are prominent in that selected group:

```
[34]: #Which algorithms are predominant in the top group?
      clean_leaderboard[clean_leaderboard['mean'] > -0.0235].groupby('algorithm').size().sort_values(ascending=False)
```

[34]:
```
algorithm
QLearning                14
Every visit Montecarlo    7
dtype: int64
```

4. Check out if the predominance of QLearning and Every visit Montecarlo stretches further down the ranking by counting types of algorithm per mean quantiles:

algorithm	Top_10%	Top_20%	Top_30%	Top_40%	Top_50%	Top_60%	Top_70%	Top_80%	Top_90%	Top_100%
Every visit Montecarlo	35	52	62	63	64	65	65	65	66	67
OffPolicyMontecarlo	3	3	3	4	4	6	7	8	9	11
QLearning	81	164	183	188	190	192	192	192	192	192
Sarsa	0	10	33	59	85	109	130	154	175	192
SarsaLambda	0	3	35	74	124	173	216	261	313	360
WatkinsLambda	0	5	39	85	124	165	218	265	309	360

5. Apply a min-max criterion to select some agents to retake and improve their performance. In other words, I have picked the top 5 performers for each of the 4 worst algorithms (Off-Policy Montecarlo, Sarsa, Sarsa Lambda and Watkins Lambda):

[43]:
```
retakers_algorithms = ['OffPolicyMontecarlo', 'Sarsa', 'SarsaLambda', 'WatkinsLambda' ]
def top(df, n=5):
    return df.sort_values('mean', ascending=False).head(5)

retakers = clean_leaderboard[clean_leaderboard['algorithm'].isin(retakers_algorithms)].groupby('algorithm').apply(top)
retakers
```

[43]:

algorithm		algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
OffPolicyMontecarlo	A_RandomOffPolicyMC_832a12bccf6142b6a5b533947642481	OffPolicyMontecarlo	Random play	1/visits	0.95	N/A	N/A	N/A	N/A	N/A	952635	[-0.0421875, -0.03325, -0.036625, -0.022125, -...	-0.029359	-0.041214	-0.019959
	A_RandomOffPolicyMC_df727aa114854b10a746c89c3ccc8e40	OffPolicyMontecarlo	Random play	1/visits	1	N/A	N/A	N/A	N/A	N/A	952317	[-0.03025, -0.042, -0.0195, -0.0175, -0.03925,...	-0.030102	-0.041978	-0.01785
	A_RandomOffPolicyMC_89f2c667bde94e12b3a9ef54b7161536	OffPolicyMontecarlo	Random play	1/visits	0.8	N/A	N/A	N/A	N/A	N/A	952674	[-0.0485, -0.03825, -0.03025, -0.027375, -0.02...	-0.031797	-0.047778	-0.01295
	A_GreedyOffPolicyMC_a6a5886faf054e4585f67fd3123d50e6	OffPolicyMontecarlo	E-Greedy	1/visits	0.8	N/A	N/A	0.05	N/A	N/A	952879	[-0.0409375, -0.0798125, -0.035125, -0.0718125,...	-0.053023	-0.078412	-0.029298
	A_GreedyOffPolicyMC_c43a8276ff0948659450 7cacbf9ff197	OffPolicyMontecarlo	E-Greedy	1/visits	0.8	N/A	N/A	0.01	N/A	N/A	952747	[-0.0696875, -0.0595625, -0.04325, -0.0584375,...	-0.0575	-0.068025	-0.044902
Sarsa	A_SarsaUCB_172450fda31d43ea9f67fdb101d001a8	Sarsa	Upper bound confidence	0.1	1	N/A	N/A	N/A	N/A	2	952882	[-0.044625, -0.0404375, -0.036875, -0.0435625,...	-0.037586	-0.050761	-0.019425
	A_SarsaVisitsDecay_95c0af8bfbaf4f68a0fc5ed7f564de45	Sarsa	E-Greedy 1/visits decay	0.1	1	N/A	N/A	0.01	N/A	N/A	952623	[-0.050875, -0.0271875, -0.032625, -0.031875, ...	-0.040703	-0.050569	-0.028008
	A_SarsaVisitsDecay_dfa9f0ecad5d41b182c94732c44e85a8	Sarsa	E-Greedy 1/visits decay	0.05	0.8	N/A	N/A	0.1	N/A	N/A	952542	[-0.03725, -0.042, -0.0425625, -0.0374375, -0....	-0.042578	-0.063806	-0.034878
	A_SarsaUCB_3d8f02d644ac4ef88df9ff908eaa5fda	Sarsa	Upper bound confidence	0.1	0.8	N/A	N/A	N/A	N/A	2	952489	[-0.018375, -0.05225, -0.0499375, -0.0645, -0....	-0.04325	-0.062892	-0.019655
	A_SarsaDecayRate_41d042e8be5c49668459eda4ecb0bb97	Sarsa	E-Greedy with decay rate	0.01	1	N/A	N/A	0.005	0.99	N/A	952530	[-0.0436875, -0.0469375, -0.0346875, -0.056, -...	-0.043461	-0.058784	-0.033422
SarsaLambda	A_SarsaLambdaUCB_e81436c59f96469aaa70d20ce6f1da34	SarsaLambda	Upper bound confidence	0.1	1	0.5	accumulating	N/A	N/A	1	952962	[-0.0288125, -0.0245625, -0.057125, -0.0339375...	-0.042125	-0.056841	-0.025306
	A_SarsaLambdaDecayRate_7d1d830597fc441aaca49c1b86c996aa	SarsaLambda	E-Greedy with decay rate	0.05	0.8	0.5	accumulating	0.005	0.95	N/A	952629	[-0.043125, -0.04875, -0.0468125, -0.0475, -0....	-0.043883	-0.048652	-0.034611
	A_SarsaLambdaVisitsDecay_2395be0d2855405bb14aa337a87ff442	SarsaLambda	E-Greedy 1/visits decay	0.05	0.8	0.5	accumulating	0.005	N/A	N/A	952677	[-0.0519375, -0.0403125, -0.04075, -0.055625, ...	-0.044641	-0.05498	-0.035878
	A_SarsaLambdaDecayRate_6affcf31e6354c5e9025a95b84dc6aec	SarsaLambda	E-Greedy with decay rate	0.01	0.95	0.5	accumulating	0.005	0.99	N/A	952635	[-0.0679375, -0.0128125, -0.04125, -0.054625, ...	-0.044867	-0.067314	-0.016006
	A_SarsaLambdaDecayRate_f106347d9c5a4a05bc9a3023bad9ad3c	SarsaLambda	E-Greedy with decay rate	0.05	0.8	0.75	accumulating	0.005	0.95	N/A	952607	[-0.038, -0.045625, -0.0398125, -0.0538125, -0....	-0.045148	-0.062681	-0.037055
WatkinsLambda	A_WatkinsDecayRate_f12f582698b449f6820e746dcbfd86e2	WatkinsLambda	E-Greedy with decay rate	0.05	0.8	0.75	accumulating	0.005	0.95	N/A	952572	[-0.036, -0.0536875, -0.0360625, -0.0218125, -...	-0.037039	-0.053791	-0.023092
	A_WatkinsDecayRate_d1d51e73325b4d468099dc4c524153ed	WatkinsLambda	E-Greedy with decay rate	0.05	0.8	0.75	accumulating	0.005	0.99	N/A	952602	[-0.02225, -0.0465, -0.03, -0.03175, -0.048375...	-0.038578	-0.050437	-0.023606
	A_WatkinsVisitsDecay_ca68f5ff6d1e43fbbcc061057bfa00b1	WatkinsLambda	E-Greedy 1/visits decay	0.05	1	0.5	accumulating	0.01	N/A	N/A	952758	[-0.0358125, -0.04275, -0.0410625, -0.0373125,...	-0.041609	-0.048125	-0.036075
	A_WatkinsUCB_05c676ea47e84540841e5787b329017e	WatkinsLambda	Upper bound confidence	0.05	0.95	0.5	accumulating	N/A	N/A	1	952483	[-0.0324375, -0.0463125, -0.0521875, -0.039625...	-0.0445	-0.054302	-0.033652
	A_WatkinsFixEpsilon_fe5de47d59ce4966a0f8e26351a24f80	WatkinsLambda	E-Greedy	0.05	1	0.5	accumulating	0.005	N/A	N/A	952673	[-0.0408125, -0.0390625, -0.0478125, -0.038187...	-0.044672	-0.066194	-0.032258

Please, zoom in!

6. Train them for 1M more episodes and generate 1 evaluation point at the end of it.

7. Include the new information and repeat steps 1 to 5.

8. As results did not improve, I allowed that group of sluggish agents to train for 1M episodes. Results did not improve either.

9. Pick the top 10 performers.

```
[84]: top_10 = clean_leaderboard[clean_leaderboard['mean'] > -0.0235].sort_values('mean', ascending=False).head(10)
```

```
[121]: top_10
```

[121]:

	algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.05	N/A	N/A	952828	[-0.0055, -0.0350625, -0.010125, -0.0149375, -...	-0.014766	-0.032383	-0.001633
A_QVisitsDecay_a08b71ff7c0749618027631507a49bbf	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.005	N/A	N/A	952340	[-0.033375, -0.01925, -0.023875, -0.0025, -0.0...	-0.01725	-0.031767	-0.003452
A_FixEpsilon_31bad91b04084908852b5c283cb2cf08	Every visit Montecarlo	E-Greedy	0.01	1	N/A	N/A	0.01	N/A	N/A	2858286	[-0.024375, 0.0086875, -0.0294375, -0.0304375,...	-0.017672	-0.030262	0.005942
A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.1	N/A	N/A	952506	[-0.0285625, -0.0121875, -0.027625, -0.0255, -...	-0.017961	-0.028398	0.000586
A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a	QLearning	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.005	0.99	N/A	952926	[-0.02725, -0.0185625, -0.0061875, -0.012625, ...	-0.018695	-0.029931	-0.007314
A_DecayRate_a81ffac6ceef4e7abbedab434b11dca6	Every visit Montecarlo	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.05	0.995	N/A	2857896	[-0.012, -0.0079375, -0.0425625, -0.0140625, -...	-0.018859	-0.040473	-0.008648
A_QVisitsDecay_c688b215e42e40daa54c2249040a9a39	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.005	N/A	N/A	952375	[-0.0244375, -0.013625, -0.010375, -0.032625, ...	-0.019508	-0.035977	-0.000142
A_FixEpsilon_1b720ec52e6640e298a3b41eeb9043f4	Every visit Montecarlo	E-Greedy	0.01	0.8	N/A	N/A	0.1	N/A	N/A	2858875	[-0.0105, -0.0226875, -0.0309375, -0.0308125, ...	-0.020047	-0.030916	-0.010795
A_FixEpsilon_1694735483da4ee5aa0b4e0c020a5225	Every visit Montecarlo	E-Greedy	1/visits	1	N/A	N/A	0.05	N/A	N/A	2857329	[-0.0253125, -0.0288125, -0.016, -0.02675, 0.0...	-0.020242	-0.033092	0.002816
A_QDecayRate_b798d2c5acd64e658f52d0be9169d01e	QLearning	E-Greedy with decay rate	0.01	0.8	N/A	N/A	0.005	0.995	N/A	952993	[-0.0295, -0.0213125, -0.017625, -0.01575, -0....	-0.020758	-0.029314	-0.011316

Please, zoom in!

10. At this point, the top 10 had an uneven number of total training episodes: 6 QLearning agents had only enjoyed a 1M-episode session, 3 every-visit Montecarlo had trained for 10M episodes, and 1 every-visit Montecarlo had trained for 100M episodes. So, I decided to train QLearning agents up to 25M episodes; their performances boded well from the very first results and executing such additional amount of training just takes up to 12 hours for all agents.

11. Execute full bootstrapping (1,000 samples) for top performers to gauge performance confidently.

Summary of findings

I draw the most important findings in this document. Additionally, some secondary findings are detailed in 06_Leadeboard.ipynb.

General findings after 1M episode training

- All agents regardless of their features can beat the Baseline, since worst-of-all performance – on the lowest side of CI tails - is at –0.144916 um per betted unit.

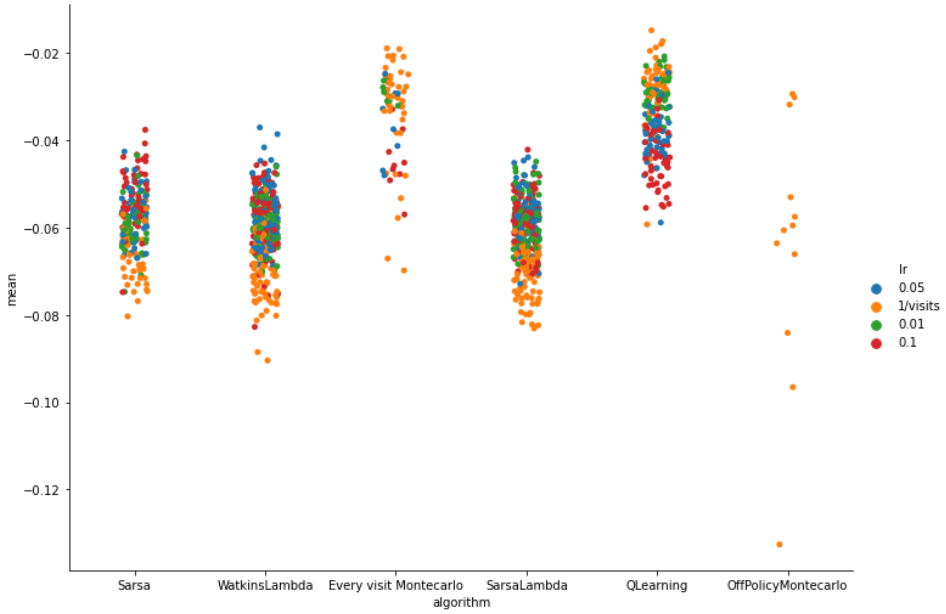
```
[31]: clean_leaderboard['lower_bound'].astype('float64').describe()
```

```
[31]: count    1183.000000
      mean       -0.069749
      std         0.014212
      min        -0.144916
      25%        -0.079155
      50%        -0.071698
      75%        -0.062886
      max        -0.027559
      Name: lower_bound, dtype: float64
```

- Both QLearning and every-visit Montecarlo sweep the top of the board, outperforming the other types of algorithms.

algorithm	Top_10%	Top_20%	Top_30%	Top_40%	Top_50%	Top_60%	Top_70%	Top_80%	Top_90%	Top_100%
Every visit Montecarlo	35	52	62	63	64	65	65	65	66	67
OffPolicyMontecarlo	3	3	3	4	4	6	7	8	9	11
QLearning	81	164	183	188	190	192	192	192	192	192
Sarsa	0	10	33	59	85	109	130	154	175	192
SarsaLambda	0	3	35	74	124	173	216	261	313	360
WatkinsLambda	0	5	39	85	124	165	218	265	309	360

- Decaying learning rates boost performance for both QLearning and every-visit Montecarlo; on the other hand, they hinder it for Sarsa, Sarsa Lambda and Watkins Lambda:



- No other conclusive performance driver has been found at this stage.

Findings after winnowing down the list of agents and enforce more learning

- A total of 6 agents (4 QLearning + 2 every visit Montecarlo) are, on average, 0.0011 u.m. per betted unit away from the house edge (0.0235):

	algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
A_FixEpsilon_1694735483da4ee5aa0b4e0c020a5225	Every visit Montecarlo	E-Greedy	1/visits	1	N/A	N/A	0.05	N/A	N/A	95,265,681	[-0.0089375, -0.0270625, -0.043, -0.004125, -0...	-0.023746	-0.046128	-0.003187
A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a	QLearning	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.005	0.99	N/A	23,816,286	[-0.0109375, -0.006125, -0.0331875, -0.0243125,...	-0.023957	-0.046939	-0.002247
A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.1	N/A	N/A	23,815,582	[-0.043875, -0.005875, -0.025875, -0.0280625, ...	-0.024177	-0.044942	-0.002625
A_DecayRate_a81ffac6ceef4e7abbedab434b11dca6	Every visit Montecarlo	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.05	0.995	N/A	9,526,672	[-0.0211875, -0.019, -0.0213125, -0.0259375, -...	-0.024241	-0.045378	-0.002998
A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.05	N/A	N/A	23,818,797	[-0.01925, -0.0125, -0.0200625, -0.0111875, -0...	-0.024536	-0.045137	-0.003059
A_QVisitsDecay_c688b215e42e40daa54c2249040a9a39	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.005	N/A	N/A	23,816,316	[-0.0440625, -0.0084375, -0.027, -0.0195625, -...	-0.024632	-0.045439	-0.0015

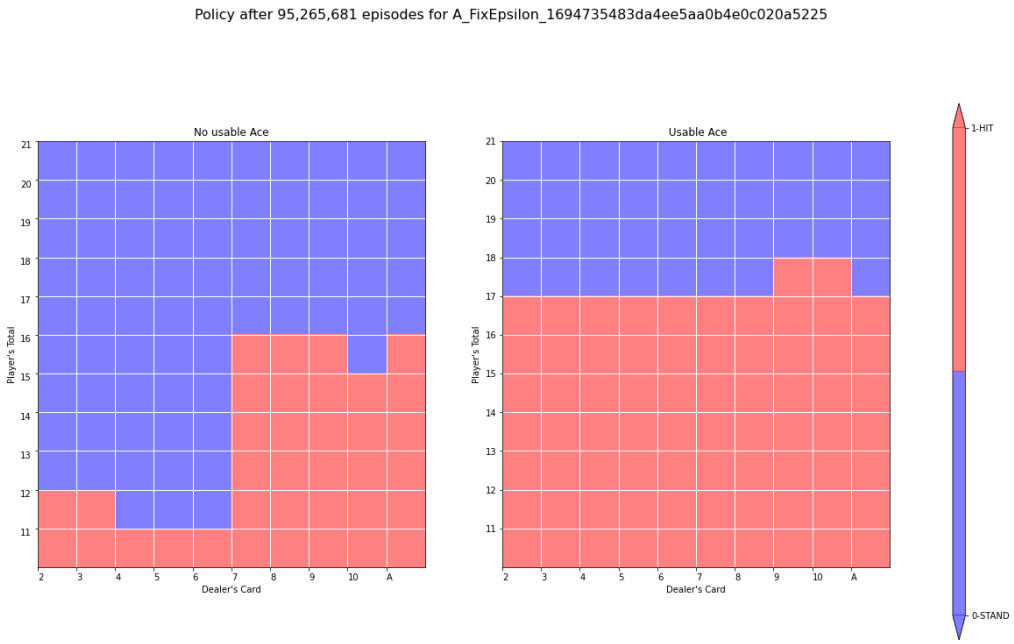
Please, zoom in!

- All top 10 performers find the house edge well within their 95% Confidence Interval.

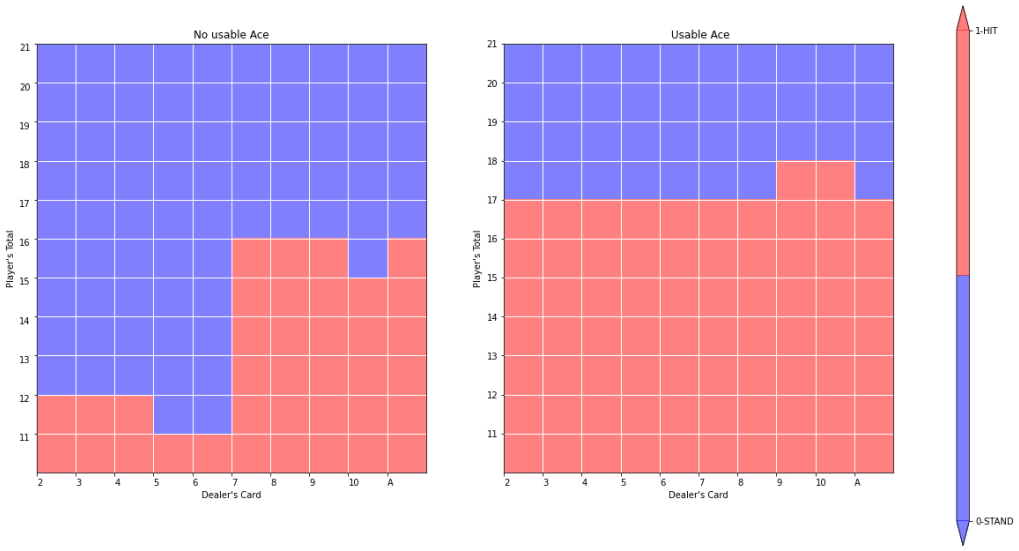
	algorithm	strategy	lr	dr	lambda	traces	ep_min	ep_decay	ucb	Last_episode	results	mean	lower_bound	upper_bound
A_FixEpsilon_1694735483da4ee5aa0b4e0c020a5225	Every visit Montecarlo	E-Greedy	1/visits	1	N/A	N/A	0.05	N/A	N/A	95,265,681	[-0.0089375, -0.0270625, -0.043, -0.004125, -0...	-0.023746	-0.046128	-0.003187
A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a	QLearning	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.005	0.99	N/A	23,816,286	[-0.0109375, -0.006125, -0.0331875, -0.0243125...	-0.023957	-0.046939	-0.002247
A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.1	N/A	N/A	23,815,582	[-0.043875, -0.005875, -0.025875, -0.0280625, ...	-0.024177	-0.044942	-0.002625
A_DecayRate_a81ffac6ceef4e7abbedab434b11dca6	Every visit Montecarlo	E-Greedy with decay rate	1/visits	1	N/A	N/A	0.05	0.995	N/A	9,526,672	[-0.0211875, -0.019, -0.0213125, -0.0259375, -...	-0.024241	-0.045378	-0.002998
A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.05	N/A	N/A	23,818,797	[-0.01925, -0.0125, -0.0200625, -0.0111875, -0...	-0.024536	-0.045137	-0.003059
A_QVisitsDecay_c688b215e42e40daa54c2249040a9a39	QLearning	E-Greedy 1/visits decay	1/visits	1	N/A	N/A	0.005	N/A	N/A	23,816,316	[-0.0440625, -0.0084375, -0.027, -0.0195625, -...	-0.024632	-0.045439	-0.0015
A_QVisitsDecay_a08b71ff7c0749618027631507a49bbf	QLearning	E-Greedy 1/visits decay	1/visits	0.8	N/A	N/A	0.005	N/A	N/A	23,818,173	[-0.0300625, -0.0304375, -0.0221875, -0.029, -...	-0.025197	-0.046133	-0.003427
A_QDecayRate_b798d2c5acd64e658f52d0be9169d01e	QLearning	E-Greedy with decay rate	0.01	0.8	N/A	N/A	0.005	0.995	N/A	23,816,340	[-0.013125, -0.030875, -0.0230625, -0.038, -0....	-0.026211	-0.046634	-0.005812
A_FixEpsilon_31bad91b04084908852b5c283cb2cf08	Every visit Montecarlo	E-Greedy	0.01	1	N/A	N/A	0.01	N/A	N/A	9,527,478	[-0.02075, -0.0165, -0.0464375, -0.03475, -0.0...	-0.027991	-0.048817	-0.00587
A_FixEpsilon_1b720ec52e6640e298a3b41eeb9043f4	Every visit Montecarlo	E-Greedy	0.01	0.8	N/A	N/A	0.1	N/A	N/A	9,527,776	[-0.0334375, -0.0206875, -0.0071875, -0.032375...	-0.029118	-0.051081	-0.006687

Please, zoom in!

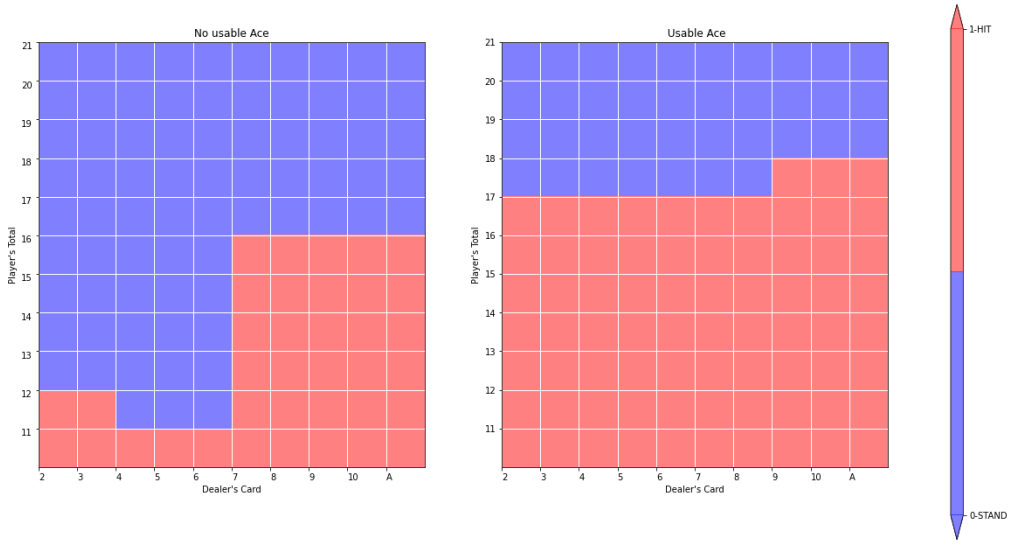
- Decaying both learning rates and exploration strategy improves time to convergence, since they help achieve similar results with a lesser number of training episodes (see table above).
- When comparing the baseline with the first findings and with these last findings, it seems clear that the most part of learning takes place in the first 1M episodes. Thereafter, gains are marginal.
- Top 6 performers showcased very similar results, but their inferred policies do not converge into the same policy (they are very similar though):



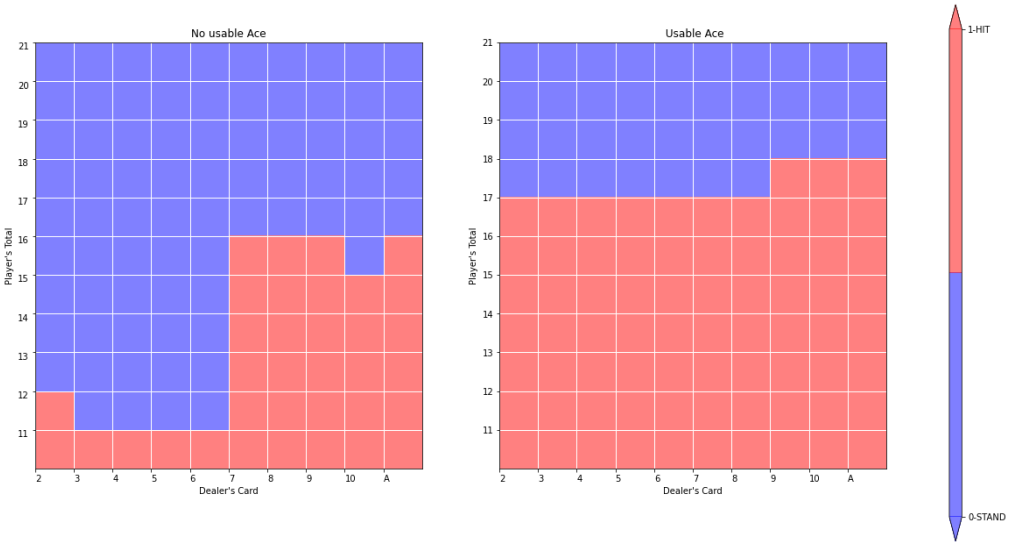
Policy after 23,816,286 episodes for A_QDecayRate_41eb1831b3fd43d5aab3e5b38422208a



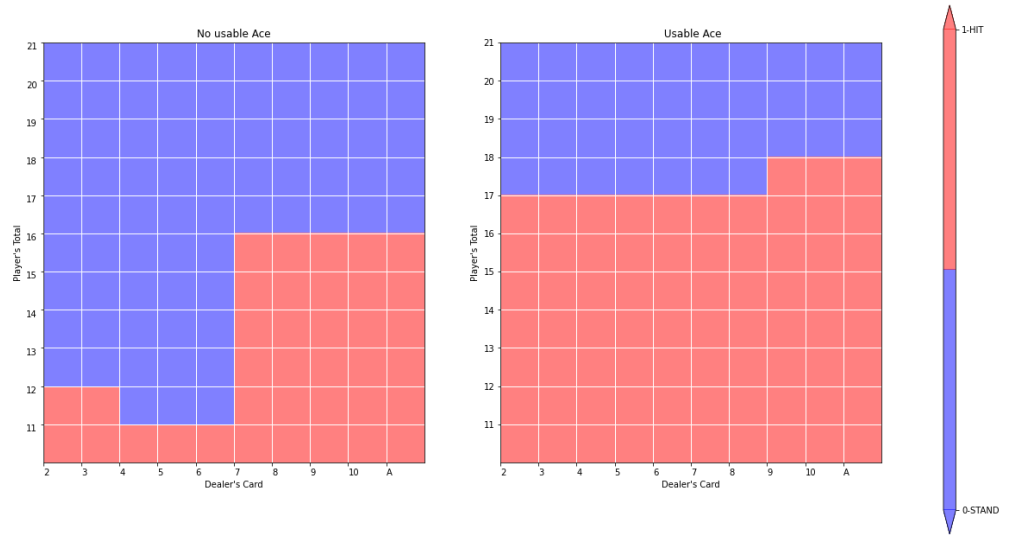
Policy after 23,815,582 episodes for A_QVisitsDecay_a448d79b4b5b45258103e111d1c1f490

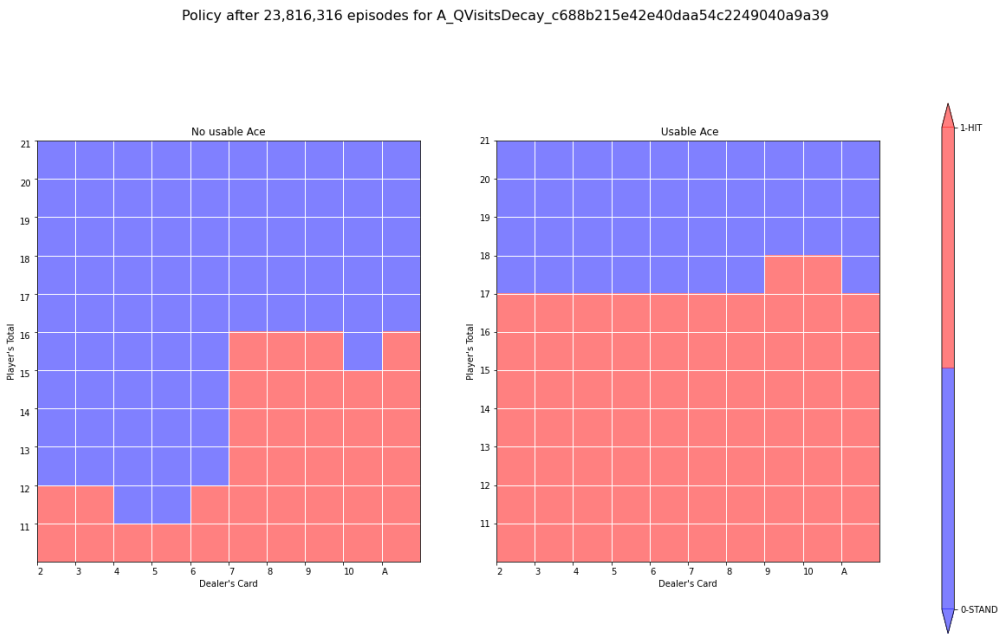


Policy after 9,526,672 episodes for A_DecayRate_a81ffac6ceef4e7abbedab434b11dca6



Policy after 23,818,797 episodes for A_QVisitsDecay_6cfb7d8b6df64593a88c8de054bebedd





Conclusions

Even though this project is just an introduction to RL and the amount of experimentation has found some time constraints, what has been discussed in the present document lets me go back to the questions framing the problem and give some brief answers.

First, it is not possible to get a positive outcome from blackjack in the long run. That is due to how the game rules are set. Therefore, that money-squandering quality of the game must be considered intrinsic.

However, it has been shown that RL gets very close to the performance claimed by combinatorial theory (house edge) and that a game strategy can be inferred as well.

Finally, albeit difficult to quantify, some scalability has been achieved. I think it is observable when comparing the length and readability of notebook 05_Train_Test.ipynb to the number of underlyingly executed operations to hit 1M-episode training sessions for 1,183 agents.

Future developments

From a theoretical standpoint, it must be said that, while this project has presented 6 algorithms, there are more sophisticated ones (especially, the neural network-based ones) and implementing them may be a new good learning opportunity. Moreover, there are also more sophisticated exploration strategies (especially those applying Bayesian approach) that may yield better results for the algorithms described.

From an operational standpoint, it might be worth exploring the parallelization of the training sessions with a many-agents-one-shared-table approach to speed up execution and overcome the time constraints.

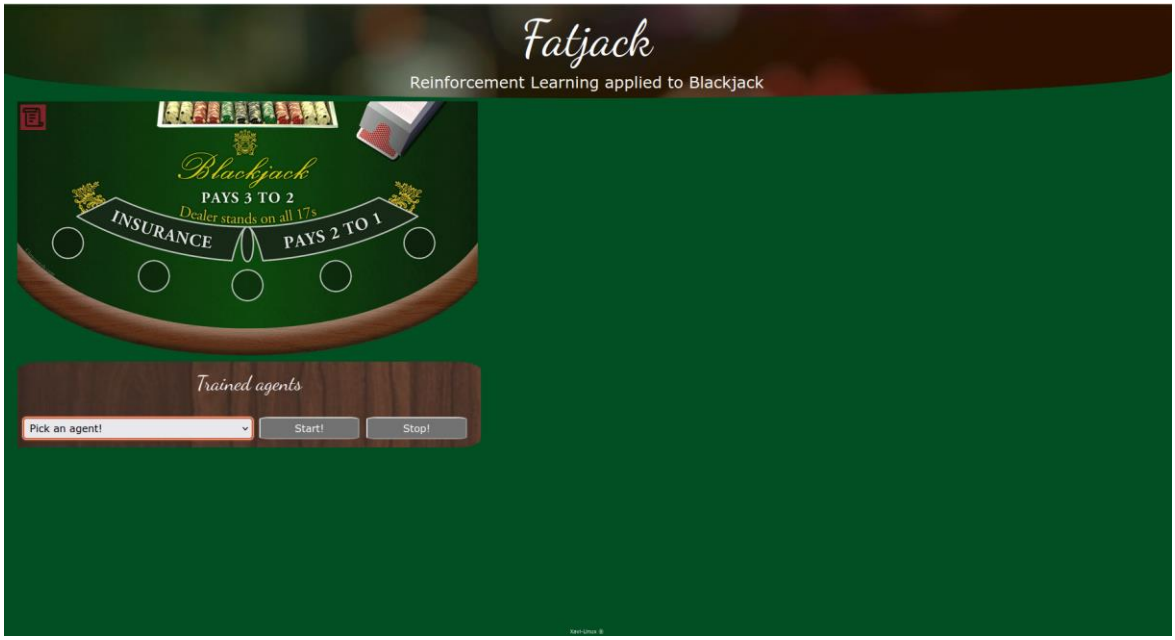
Finally, the Python framework developed for this project is intended to be easily reusable for different tasks. The best way to prove it is to try new tasks in the forthcoming future.

The front-end

I would like to give a quick piece of guidance on how to deal with the graphical user interface:

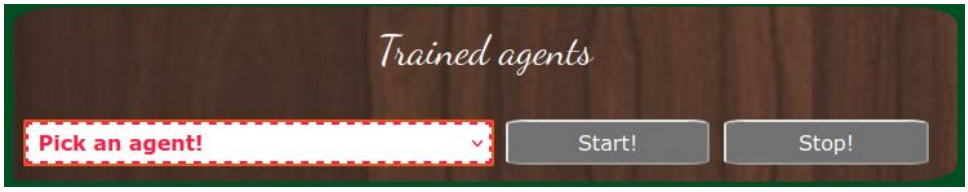
- It is online available at: http://34.77.255.37/.
- The main goal of the webpage is to behold how agents play blackjack by themselves.
- Someone visiting the webpage cannot play blackjack, just the agents.
- Some additional information about the agents is presented to enrich the experience.
- The web page has been explicitly adapted for smartphones.

The initial layout is as follows:

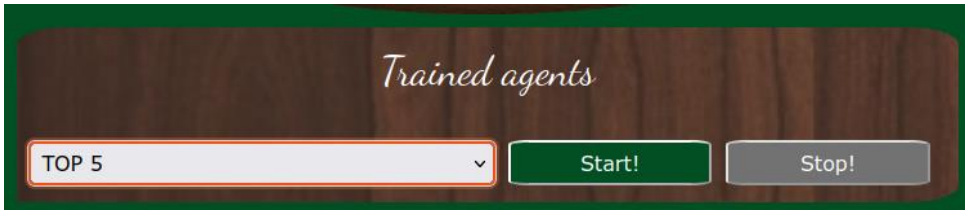


You might first desire to check the rules. Hover over the icon on the top-left corner of the table:

Pick an agent by using the dropdown menu in the “trained agents” section. Top 10 performers appear at the top of the list. The other listed agents do not follow any order.

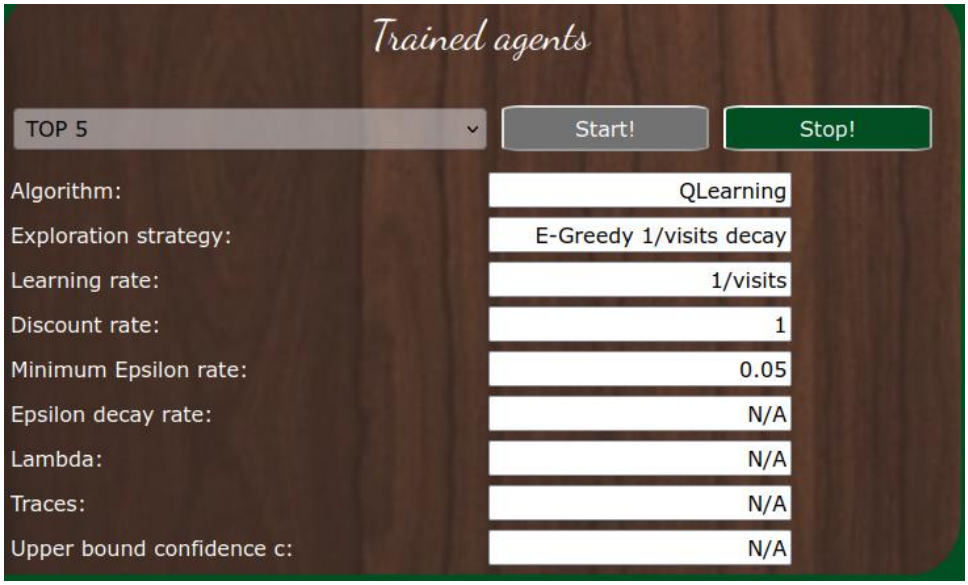


Press “Start!” to get the agent started:



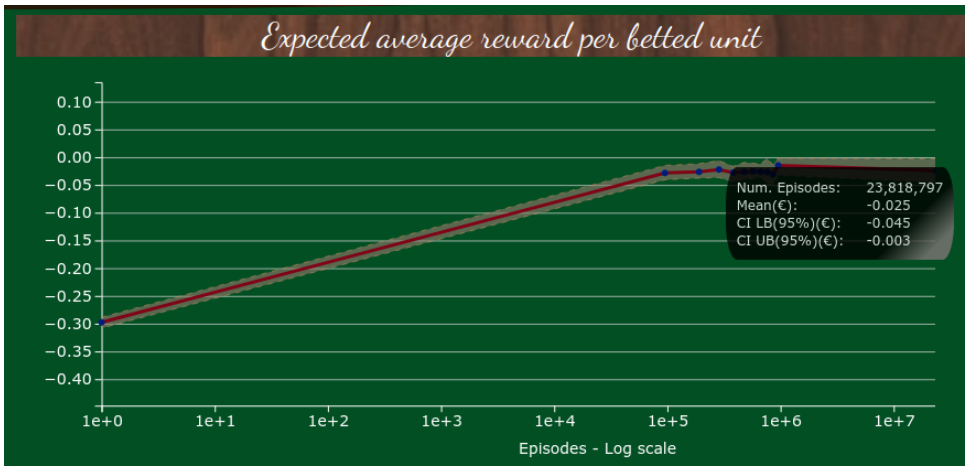
At this point, apart from watching the agent playing, some additional information will be displayed:

- The parameters chosen to train the agent:

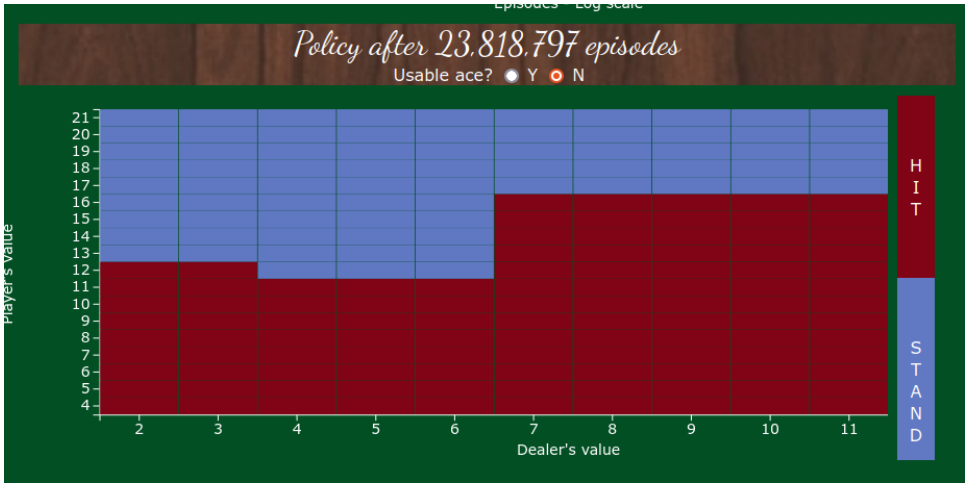


- The expected reward time series of the training process:

The x-axis has been logscaled. Hover over the blue dots to get to know the exact values at each evaluation point:

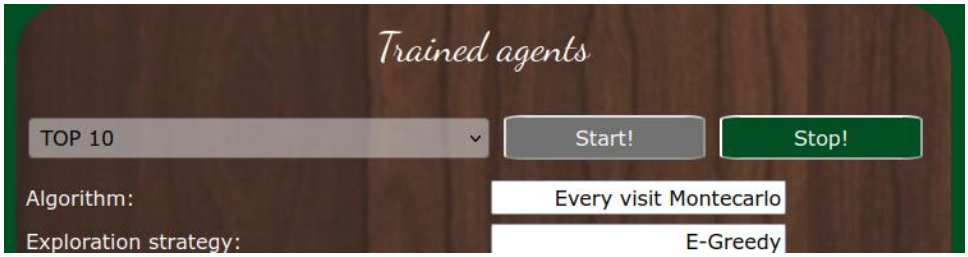


- The last policy inferred by the agent:



By default, you see the non-usable-ace policy. Switch to the usable-ace table by using the radio buttons below the plot's title:

Once you get tired of observing one agent, press "Stop" in the "Trained agents" panel and start over:



References

Brownlee, J., 2021. *How to Calculate Bootstrap Confidence Intervals For Machine Learning Results in Python*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/calculate-bootstrap-confidence-intervals-machine-learning-results-python/> [Accessed 21 July 2021].

Cohen, J., 1990. Things I have learned (so far). *American Psychologist*, 45(12), pp.1304-1312.

de Grandville, C., n.d. *Applying Reinforcement Learning to Blackjack using Q-Learning*. unspecified. University of Oklahoma.

En.wikipedia.org. 2021. *Blackjack - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Blackjack> [Accessed 19 July 2021].

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. and Merger, D., 2019. *Deep Reinforcement Learning that matters*. unknown. McGill University.

Isbell, C., 1992. *Explorations of the Practical Issues of Learning Prediction-Control Tasks Using Temporal Difference Learning Methods*. Master of Science. MIT.

AlphaGo - The movie. 2017. [film] Directed by G. Kohs. Unknown: Moxie Pictures.

Lorica, B., 2021. *Practical applications of reinforcement learning in industry*. [online] O'Reilly Media. Available at: <https://www.oreilly.com/radar/practical-applications-of-reinforcement-learning-in-industry/> [Accessed 18 July 2021].

Mahajan, P., 2021. *Playing Blackjack using Model-free Reinforcement Learning in Google Colab!*. [online] Medium. Available at: <https://towardsdatascience.com/playing-blackjack-using-model-free-reinforcement-learning-in-google-colab-aa2041a2c13d> [Accessed 21 July 2021].

Silver, D., 2015. *RL Course. London, University College.*

Sutton, R., n.d. *Introduction to reinforcement learning*. Cambridge, Mass: MIT Press.

Sutton, R., Bach, F. and Barto, A., 2018. *Reinforcement Learning – An Introduction*. Massachusetts: MIT Press Ltd.

Widrow, B., Gupta, N. and Maitra, S., 1973. Punish/Reward: Learning with a Critic in Adaptive Threshold Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(5), pp.455-465.

Wizardofvegas.com. 2021. *Blackjack House Edge - No splitting/No Doubling discussed in Math/Questions and Answers at Wizard of Vegas - Page 2*. [online] Available at: <https://wizardofvegas.com/forum/questions-and-answers/math/10964-blackjack-house-edge-no-splitting-no-doubling/2/> [Accessed 17 July 2021].