Blackjack

This project's objective is to train a model to play Blackjack using reinforcement learning. Blackjack is a "solved" game, as in there are mathematically most sound moves for each case. We can thus verify the accuracy by comparing the model with the theoretical results.

Manual

Blackjack Rules

In a game of Blackjack, players play against the dealer to see who can get closest to 21 without going over it.

Cards 2-10 are worth their number values. Face cards are worth 10. Aces are worth either 1 or 11, whichever is more advantageous.

The round begins with each player being dealt 2 cards face up. The dealer also receives 2 cards, but the second is kept face down. In this implementation of the game, each player can choose between 3 actions: standing, hitting, or doubling.

- Stand: End the player's turn without taking a new card
- Hit: Receive an additional card. The player can keep hitting until he either reaches 21 or goes over.
- Double: Double the player's bet. Hit once. End the player's turn.

In casinos, players usually also have the option to split or surrender, but that goes beyond the scope and purpose of this project. (Learn more)

After all the players have ended their turns, the dealer reveals his downcard. The dealer keeps hitting until his count is at or above 17, then ends his turn. The dealer hits on soft-17 (Ace + cards that add up to 6) in this version.

If a player is dealt a Blackjack, an Ace and a 10 value card, they are paid 3:2 and end their turn. This case is ignored in the model since it does not require any decisions from the player.
If a player's count goes above 21, called a bust, they lose their bet regardless of the dealer's cards.
If the dealer busts, all players who have not busted or Blackjacked are paid 1:1.
Elsewise, the player's sum is compared to the dealer's. If the player's count is greater, they win and are paid 1:1. If the dealer's bet is higher, the player loses their bet. If the sums are equal, the bets are returned (called a push).

Installation

This project uses some external libraries. To install the dependencies, run `pip install -r requirements.txt` in the shell.

Usage

This project can be used in multiple ways.
First, you can use the pre-trained model to determine which moves it would play. The best iteration is not completely accurate, but it comes really close to the theoretical optimal actions.
Included is a program that generates charts of moves for all pertinent scenarios.
Second, you can tweak the training settings and try to reproduce the theoretical best moves.
Lastly, the model can be extended to try out different strategies. One possibility is the idea of Gambits from chess. In a multi-hand game, would it be sometimes better to play suboptimally on an early hand to get an advantage on a later one?

For most users, the usage of this project is just to see which move should be played for a specific case.

To do so, simply launch, predict.py, select a weights file (I included some), then enter a player count (4-20), if the player count is soft (0,1), and the dealer's upcard (2-11). and it will tell you which move is optimal according to it. I recommend using either weights file b or h.

Design Guide

This project has two main parts: the environment (blackjack_env.py) and the training (training.py). The project also includes graphic.py for visualizations and predict.py to try the model. These are more like add-ons, so they won't be covered.

Environment

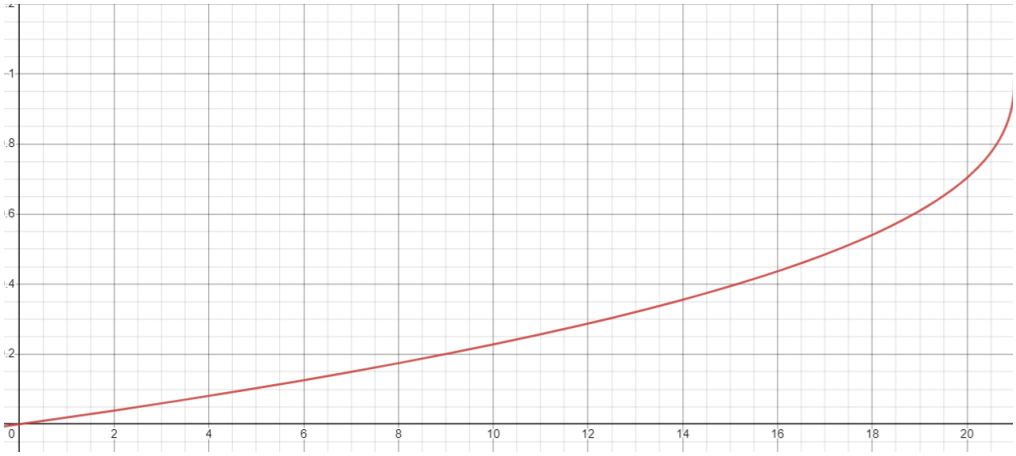
The environment is the implementation of the game itself. The current version is intentionally rudimentary, as the goal was to match the theoretical moves using the 3 basic actions of standing, hitting, and doubling down. This version also only looks at isolated rounds, which is why the shoe is an incomplete deck; it only contains one set of each possible card. In future iterations where more complex mechanics like card counting are to be explored, the shoe can be extended to contain more decks and to persist through rounds. For our purposes, though, choosing a random card from the possibilities suffices.

The motivation behind implementing Players and Dealers as subclasses of the abstract Agent class was to future proof the project. Instead of hardcoding 1 player and 1 dealer, which is very one dimensional, using Agents allows for a lot of expansion room. For example, the split action (splitting a pair of identical cards into two hands, more information) is essentially the same thing

as adding another player. Player agents also allow for multiplayer games. Possible multihand strategies like the gambit mentioned earlier can also be implemented using Player agents. Having multiple hands is the same as having many players.

Currently, the three observations that are taken into account at each step are the player's count, if the player's count is soft, and the dealer's upcard. In isolated rounds, these are the only possible observations, and all of them can affect which action should be taken. If a larger shoe that persists through rounds is added, then it would also be pertinent to use the card count (Hi-Lo method) as an observation, as it signals whether the shoe is favorable or not to the player and could affect bet sizes.

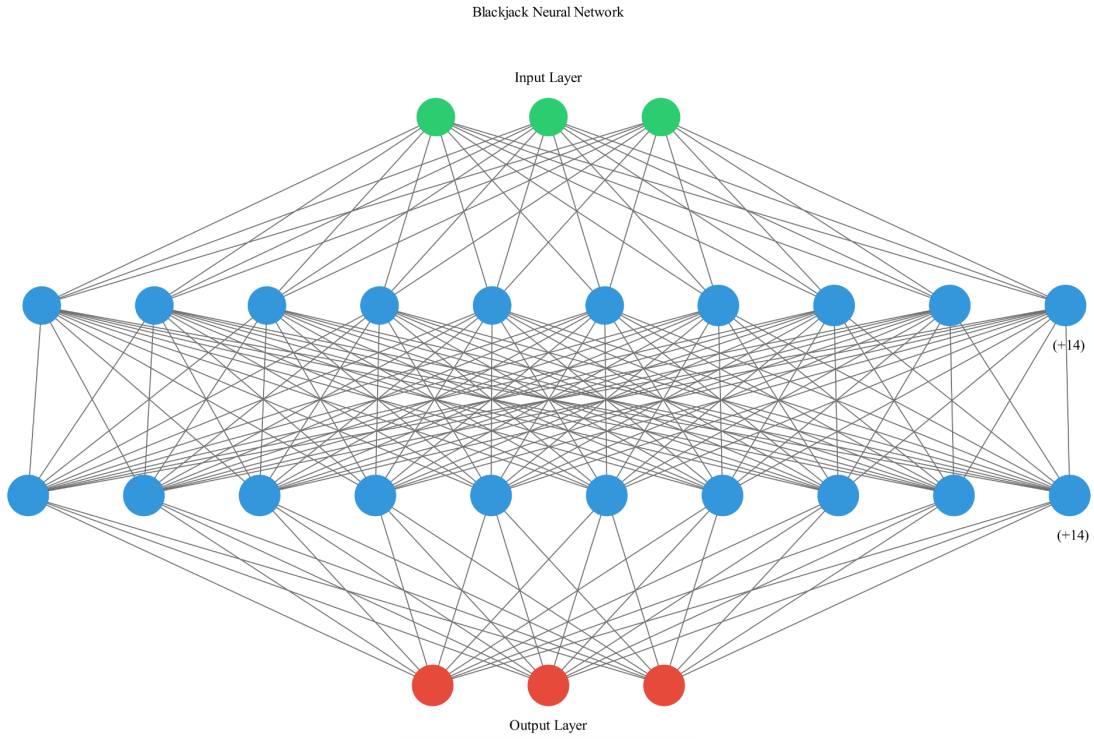
Obviously, the goal of the model is to beat the dealer and win the round, so the rewards are set up in a way to incentive the agent to win, and discourage it from losing. However, only giving a reward, be it positive or negative, at the end of rounds is a generally bad idea. Rounds usually consist of multiple actions (ex 2 hits and a stand), so it is difficult for the agent to figure out what it did right or wrong if it is only rewarded at the end. This is called sparse rewards. To make the agent's learning process easier, we shape the rewards and give feedback at each step. The reward function is $r = 1 - ((21 - player\ count) / 21)\ **\ 0.4$ for steps that do not end the turn. This function was taken from Writing Great Reward Functions - Bonsai.



This function gives more reward as the count approaches 21, so the agent is encouraged to increase its count to 21, which is the goal of the game. These step rewards (max 1) are small compared to end-of-round rewards (max +/- 10) because, ultimately, it is the outcome that matters most. The step reward is only there to help the learning process.

Training

The neural network used is fairly simple. There are 3 input nodes for the 3 observations, a total of 48 nodes across 2 hidden layers, and 3 output nodes for standing, hitting, and doubling. The model was built using Tensorflow and Keras.



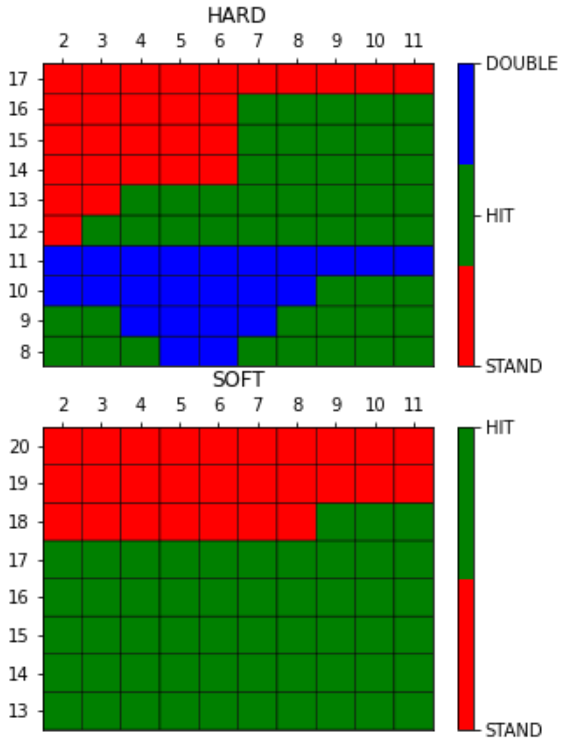
The interesting part is the agent that interacts with the environment. It is through the agent's actions that the model can be optimized through reinforcement learning. The agent is built using Keras-RL.

HARD TOTALS

	DEALER UPCARD									
	2	3	4	5	6	7	8	9	10	A
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	H	S	S	S	H	H	H	H	H
11	D	D	D	D	D	D	D	D	D	D
10	D	D	D	D	D	D	D	D	H	H
9	H	D	D	D	D	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H

SOFT TOTALS

	DEALER UPCARD									
	2	3	4	5	6	7	8	9	10	A
A,9	S	S	S	S	S	S	S	S	S	S
A,8	S	S	S	S	Ds	S	S	S	S	S
A,7	Ds	Ds	Ds	Ds	Ds	S	S	H	H	H
A,6	H	D	D	D	D	H	H	H	H	H
A,5	H	H	D	D	D	H	H	H	H	H
A,4	H	H	D	D	D	H	H	H	H	H
A,3	H	H	H	D	D	H	H	H	H	H
A,2	H	H	H	D	D	H	H	H	H	H

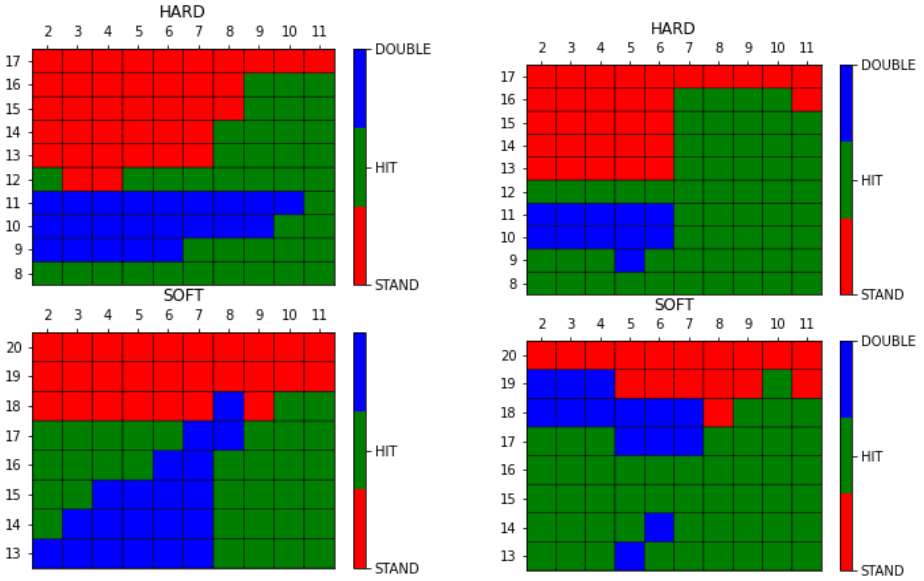


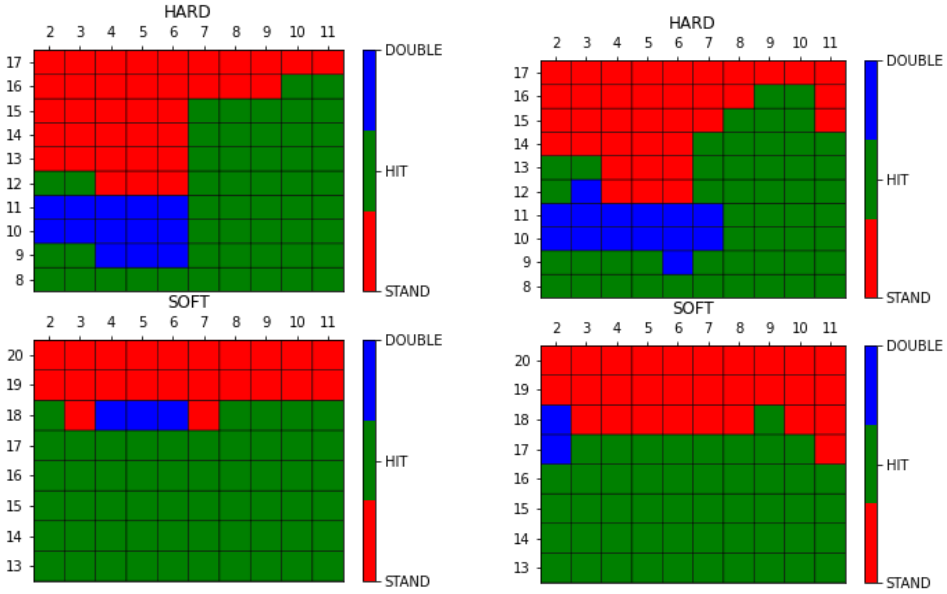
Left graph from https://www.blackjackapprenticeship.com/blackjack-strategy-charts/

On the left are the optimal moves for each case, and on the right is the first iteration of the RL model. This iteration was trained over 5 000 000 steps, around 3 500 000 rounds of Blackjack, using the Boltzmann Q Policy. BQP usually picks the best action for the current q values. This means it will try to make the best move according to the current model at every step. The results show that this policy works really well and produces fairly accurate results. However, there is a lack of doubles for soft counts. Using BQP, if the agent has past success hitting on A-6 to 3, for example, then it will do it, which is likely to succeed, which then updates the q values to further want to hit on A-6 to 3. The agent is content with its actions and has no reason to change.

In an attempt to encourage the agent to double more and play more greedily, the bet multiplier for doubling was increased from 2 to 4 and the policy was changed to Epsilon Greedy. By increasing the multiplier, the agent will receive more reward when it doubles down successfully,

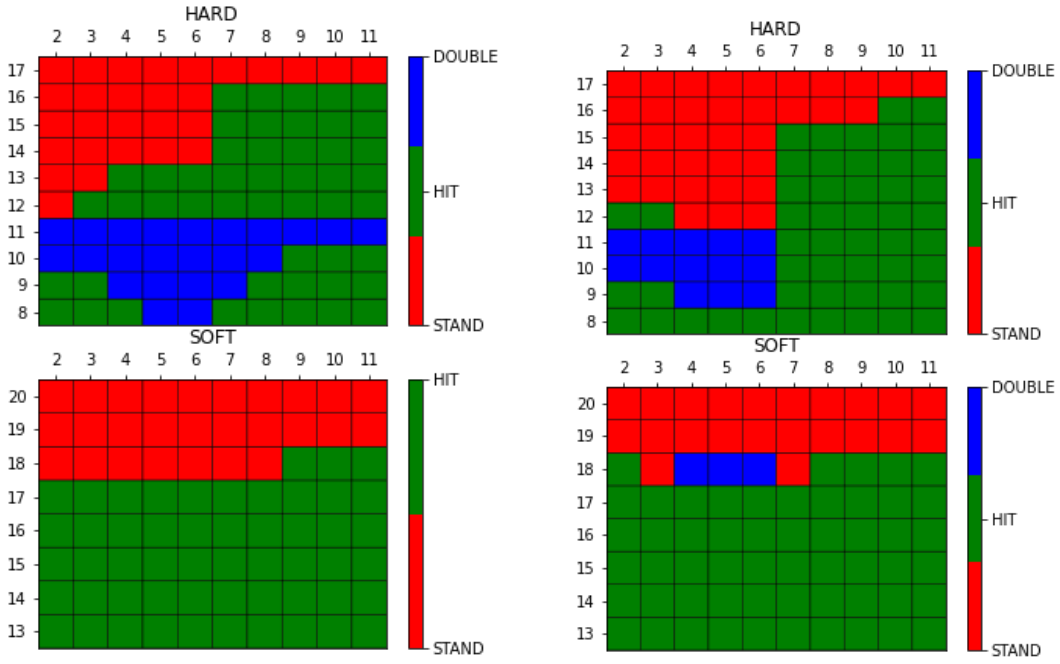
so it will be more likely to double in future rounds. The Eps policy may also encourage more doubling. In contrast to BQP, Eps will not always pick the best current action. By allowing for suboptimal actions, Eps favors more exploration than BQP. We want the agent to try different moves from what it already knows works.





The results show that the agent did indeed start to double more during soft counts, but accuracy is not always preserved. Although the change to the multiplier did have the intended effect, it changes the rules of the game, and will thus produce different results. The multiplier was returned to 2.

All in all. The best iterations appear to be b (left) and h (right)



B has more correct doubles, while H has more correct stands. Having a hit instead of a double is not bad at all, as a double is just a single hit. A normal hit just doesn't reward you as much. Mixing stands with hit/double is bad, though, as it is the difference between getting new cards or not.