Bl♠ckj♠ck AI

Ryan Karch (karchr), Dominic Beyer (beyerd), Angelica Loshak (loshaa), Michael Dong (dongm2)

Fall 2023

1 Project Description

1.1 Problem Solved

Blackjack AI gives users a place where they can test their blackjack skills alongside artificial intelligence agents, including being able to upload their own artificial intelligence agents. The main objective of this project is to provide an entertaining platform that encourages skill development in blackjack. It enhances the strategic thinking of players by giving them a sense of competition from a challenging AI opponent. The website aims to create a user-friendly environment where players can interact with AI and gain experience with betting cryptocurrency in a safe and secure space. The project also addresses the different skill levels of players by allowing them to adjust the intelligence levels of the bot that they are playing with. This ensures that players from varying blackjack backgrounds can benefit and grow from Blackjack AI.

1.2 Background

Blackjack is one of the most popular card games in casinos. However as online casinos become a significant source of revenue for the gambling industry, more players are falling to gambling addiction. The accessibility and convenience that come with online gambling make it easier than ever before for players to make impulsive decisions and run into financial problems. Along with that, deception among online gambling platforms contributes to players falling victim to underage gambling, scams, and fraud.

Blackjack AI aims to bring users an alternative way to play blackjack with friends and AI players without losing large amounts of money or risking their safety and privacy. We prioritize safety and responsibility by having users bet in Wei instead of full Eth denominations. Users are authenticated through their MetaMask wallet which ensures the credibility of every individual player. A smart contract is used to securely transfer and receive cryptocurrency during betting and payouts. These transactions are recorded in the blockchain, so any fraudulent activity will be recorded and caught. The platform also ensures fairness by allowing players to adjust the AI to play at their skill level. The AI component brings a fair and challenging component to the game. Blackjack AI is a platform that focuses on entertainment and education for the players while allowing them to interact with AI players and bet cryptocurrency.

1.3 Motivating User Story

Online games today are full of microtransactions and advertisements. Small developer-run games normally are full of bugs and not fun to use. Now, with a bit of Testnet Sepolia Ethereum and a MetaMask account, anybody can play a fun blackjack game right alongside a real artificial intelligence agent. Motivated users can even create programs that interface directly with our website to play as well!

Blackjack AI is motivated to bring the best user experience to the customers. It prioritizes safety and entertainment over all else. Users who are struggling with a gambling addiction can use Blackjack AI to get the thrill of betting on blackjack while playing with friends, without financial consequences.

2 Implementation Details

2.1 AI Aspects

We used the reinforcement learning algorithm "Q-learning" to train our AI model. We chose this algorithm because it lends itself to AI exploration of game environments, where there is a reward that is easily modeled, as the purpose of Q-learning is to maximize that reward. Due to the nature of reinforcement learning, an agent makes decisions over time in an environment, and those decisions affect its knowledge of the environmental states, which impacts a decision's probability of being chosen. We refer to a Q-learning AI agent as a QAgent from here on. A QAgent's knowledge is stored in a so-called "Q-table", which is a mapping of environment states to reward values, each associated with a decision. Over time, these reward values are adjusted, and the one that becomes maximized is the decision that the agent will learn to choose when provided its associated state. We performed an initial training session of 5,000,000 blackjack games to create an effective Q-table with optimal strategy. This allowed our QAgent to maintain a level of "smartness", which we could control by providing the probability of the QAgent selecting a decision from the Q-table, thus being the probability of the QAgent making the optimal decision. Our QAgent model came from an open-source Jupyter Notebook [1], written by Till Zemann, which can be found in the bibliography.
We provided an AI template for users to upload custom agents that contain a fixed set of methods that are executed within the program. The abstract Agent class acts as the skeleton for all agents used in the Blackjack AI system, including the QAgent. For an agent to interact with the system it must have certain methods defined, so they can be executed and the behavior can be guaranteed by the system. We provided an example of a custom agent that either hits or stands via a "coin flip", or a 50% chance of making either choice.

2.2 Blockchain Aspects

Our blockchain implementation is written and compiled in Solidity and deployed on the Sepolia ETH test network through RemixIDE. It contains private member variables that hold a mapping of addresses to balances, as well as a list of all users that have participated. The constructor saves the address of the sender and declares it as the owner. The contract is initialized as "payable", and can receive ETH/Wei at any time, however, the main payable function that will be used is called "deposit". This function not only accepts ETH/Wei into the contract but also saves the amount deposited to the user's account in the mapping. If the user has a balance of 0, their address is also added to the user list. The cashOut function is used to withdraw money from a user's account. It has a modifier that requires the requested value to cash out to be within the range of their current balance. The function sends the user the requested amount of ETH/Wei and then updates their balance. The changeBalance function is meant to be an administrator-only function, as it directly modifies the user's balance without receiving or sending any currency. This function is used to adjust balances during the betting/winning processes. To safeguard this function, it requires an input of the owner's wallet address to proceed. This function also protects against negative integers by ensuring that if the input value is larger than the user's balance, it sets their balance to 0. The contract also contains two view functions: getBalance and contractBalance. The function getBalance is used to return the balance of the user that is stored in the blockchain, and contractBalance returns the full amount of ETH/Wei that the contract has. The final function is meant to be a development-only function, called evacuateFunds. It loops across all users that have deposited into the contract and sends them any amount of money that they have in the contract. The leftover money is then returned to the owner of the contract. The main use of this function is to allow the deployer and users to not lose their money if a new version of the contract is deployed.

2.3 AI and Blockchain Integration

Both the AI and Blockchain aspects of our project are integrated through the Python backend and JavaScript frontend. The AI's Python implementation was specifically defined in a manner that allows it to be wrapped by a custom-created object called LocalPlayer. This object is used to wrap all of the possible agents that can play the game, from the AI player to the WebUser (which is controlled almost exclusively through the frontend). This wrapper acts as an interface between the backend game logic and the AI agent. This

guarantees that the data given to the Agent by the game cannot be modified, as the LocalPlayer object will keep track of the state of play and ultimately control when a player busts, preventing the user from modifying their game state. The frontend communicates with the backend game logic (and therefore the AI Agent) through a series of post requests that echo the events that the user triggers on the web page. For example, when the user hits the deal button, a post request is triggered that asks the backend to deal the cards. The backend deals the cards and informs the LocalPlayers that it stores, and then sends the cards to the frontend to be displayed. The backend game is structured in such a way that allows it to execute parts of a blackjack game as instructed by the frontend. As for the blockchain, both the frontend and the backend can directly communicate with it. The Python backend uses the web3.py library and an Infura API to connect to the Sepolia network. The interface is contained within a custom BlockchainInterface class that handles storing the web3 object, as well as abstracting a few functions. This interface is used for view only functions, as signed transactions would need to be handled by the frontend where a user can authenticate. This object can request a user's balance from the contract, and watch a pending transaction until it is completed. It also contains a function that uses web3's keccak encoding functionality to correctly obtain the encoded name of the contract's functions, which will be used by the frontend. The JavaScript frontend uses the official MetaMaskSDK to connect the user's MetaMask account. The functionality that we used here was able to return the address of the user, as well as send signed transactions through the MetaMask extension (visually identical to transactions through RemixIDE). When communicating with the smart contract, the frontend uses a function called "ethereum.request", which uses a method called "eth_sendTransaction". This method allows currency to be transferred from the user's wallet to any other address (another user or a contract). This method also takes in a field called data. This data field can take a function selector, which allows a specific function of the contract to be called instead of the money being sent to the generic receive function. This data field also takes any function arguments that the function may require. The function selector is made by performing a keccak encoding on the declaration of the function, which is in the form: func(argtype1,argtype2,...). For example, the blockchain's change balance function would look like this: changeBalance(address,uint256,bool). This encodes to a full 64-byte (256-bit) value: 0x30faf3287258e1386879c7d08ad722e5c13a09bd618c41b8336d92ee9e101100. The function selector is the first 8 bytes of this encoded value: 0x30faf328. Continuing with the changeBalance example, the parameters then have to be appended to the function selector. Every parameter must be in full 64-byte form. This includes the address parameter, even though addresses have a set length. The uint 256 parameter is self-explanatory. The bool gets encoded to a 1 for true and a 0 for false. These values must then also be converted to a full 64-byte form. A full data parameter for a changeBalance transaction may look something like this: 0x30faf32800000000000000000000000005233862f7245cb0d76af46716631abfb389163c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000c8000000000000000000000000000000000000000000000000000000000000000000000000000001. This value essentially tells the smart contract to increase the balance of the sender by 200. The address given is the owner's for authentication purposes, c8 is 200 in decimal, and the bool is 1, therefore the balance will be increased. Transactions to the blockchain happen whenever a user needs to deposit or withdraw money when they place their bet, and when a round finishes and they receive their payout.

3 Project Demonstration

To use the Blackjack AI platform, the user first needs to log in with their MetaMask account, as shown in Figure 1. Upon opening the website for the first time, they will be automatically prompted to link their account. After the first login, the user will only ever be prompted to log in if they need to unlock their wallet, as MetaMask saves a list of connected sites. The user has the option to deposit or withdraw Sepholia ETH into and from their balance at will. Next, the user has the option to upload an agent to play alongside them. This agent undergoes a testing process that verifies its validity and if rejected, the user will be notified. A description of how to create an agent that fits this validity is listed in the README. The user can also use a slider to adjust the level of our provided pre-trained AI. The slider runs from 1-100, where 100 is the smartest. The AI's display name on the next screen depends on this "smartness" value. The smartest bot is named 'Lord Blackjack", at 100% smartness. Anywhere below 100 but above 70 grants the AI the name "Master Mind AI". If the smartness is between 40 and 70, the AI will be named "Ninja AI". A smartness value between 10 and 40 warrants the name "NPC AI", and a level below 10 grants the AI the name "Fresh

Off the Compiler AI".
After logging in, the user is taken to the game's main page. The buttons "DEAL", "BET", STAND", and "HIT", allow the user to play the game and bet the Sepolia ETH in their balance. The main user is displayed in the middle of the screen so that their cards are perfectly vertical for easy reading. On the right of the player will always be the site-provided AI. If the user provided an agent file, their agent will be displayed on the left side of the table, with the username "¡User¿'s Agent", where "User" is the user's username.
The "BET" button becomes active at the beginning of each game, as shown in Figure 2. To place a bet, users can adjust the desired amount of Sepholia Ethereum (in Wei) by pressing the "+" or "-" buttons. The buttons will not allow a user to bet higher than their balance, nor can they bet negatively. When the user is satisfied with their amount, they must press the "BET" button to confirm the bet. This will trigger a MetaMask transaction that removes the bet amount from the user's wallet. This transaction will pop up again if rejected, as the user must confirm the transaction to continue.
Once the bet is confirmed, the "DEAL" button becomes active. Two cards are dealt to each player and the dealer. The players can see their cards, or they can click on the cards to flip them over and hide them from view. According to the rules of blackjack, only one of the dealer's cards is visible.
After the cards are dealt, the "STAND" and "HIT" buttons become active. The player can hit when they want to be dealt another card from the deck, and they can stand when they are done. This portion of the game will be automatically ended on a user stand, or if the user hits and subsequently busts.
As seen in Figure 3, the user will then be prompted with another MetaMask transaction to confirm the results of the game. If the player wins, they will be paid back twice their bet. If the player was dealt blackjack, they will be paid back thrice their bet. If the player and the dealer push, they will be paid back their bet, and if the player loses, they will not be paid anything. As with the bet transaction, this transaction will continue to pop up if the user rejects it, as they cannot proceed without updating their balance.
The player can play as many rounds as they want, or until they run out of funds in their balance. In that case, the player should use the "EXIT" button to return to the login page and deposit more funds into their account.
As a note about the persistent transactions: Users can avoid these transactions by refreshing the page, as the site will not remember the game that they were in. However, this either means that the user was able to cancel their bet and must bet again, or they have canceled a payout and lost some of their balance. Because these two cases do not encourage dodging transactions, we did not feel the need to implement data that persists beyond refreshing.

4 Evaluation

Our program's main goal was to have an AI agent that played blackjack optimally, that a user could play alongside. We plotted the results of the AI training over 5,000,000 games, as seen in Figure 5. Over time, the rewards were maximized, and the training error curve flattened out until our AI was playing mostly optimal blackjack. We also plotted the strategy generated by the AI, which can be seen in Figures 6 and 7. We can observe that the AI's strategy was more conservative when was not holding a usable ace, as it was more likely to bust, versus its strategy when it was holding a usable ace, as it could afford to be more risky.
To test the blockchain aspect of our code, we made a spreadsheet that aimed to test corner cases of the functionality of our contract. We deployed a test version of the contract that was used exclusively to conduct the tests. These tests are reflected in Figure , and the spreadsheet itself is linked in the resources section. These tests ensured that the fail-safes that we intended to program for did work. This included ensuring that users could not withdraw more money than what was in their balance, ensuring that users could not call the balance modification function easily, and ensuring that only the owner could remove all of the money from the contract. This document also tested the standard functionality of the contract, including deposits, standard withdrawals, adjusting balances after pretend wins and losses, and removing all funds.
As for the frontend and backend that ran and interacted with both the AI agent(s) and the smart contract, we covered as many edge cases as we could consider. As we were developing the website we ran into a few errors here and there and were able to correct them, but we also thought of some exploits that we needed to protect against. First off, both the backend and the frontend do not allow the execution of functions that are not expected. For example, a user cannot deal themselves cards, hit, stand, and receive payment

without first placing a bet and confirming it in the frontend and subsequently the backend. The backend game logic uses a state enumeration to proceed to the next state, only allowing functions to run if the active state matches the required state of the function. Likewise, the frontend uses boolean values to keep track of if a button should be able to be pressed if a button was pressed at the wrong time, the function would simply exit. We took this one step further and disabled buttons that were not meant to be clicked, fully preventing any wrong interaction. As discussed in the demo section, we also ensured that users cannot dodge signing their MetaMask transactions without fully refreshing the webpage, ensuring that they never gain currency from this exploit. We also do not allow users to access the blackjack page unless they go through the login page, as any refresh attempts redirect the user to the login page. Finally, we created a redirect from any parent directory URL to the main login screen, allowing users to enter a generic URL and be immediately directed to the login page.

5 Conclusion

5.1 Summary

Our project allows users to play blackjack games alongside artificial intelligence agents and experience a fully functional blockchain that acts as a bank to keep track of balances. Users can upload custom Agent files, which play alongside them at the blackjack table.

5.2 Ethical Use

Our project is purely simulative, so there are no concerns regarding real financial systems or currencies. As our program allows users to upload custom artificial intelligence agents, we made sure to have some protective measures against system abuse put in place. The uploaded agent code is checked against a series of verifiers to ensure that the specified methods are defined within the class. This allows our program to guarantee that no errors will be thrown when executing the custom agent code, which would throttle our system. Another protective measure that was implemented is the fact that only certain predefined class methods are defined within the program. This guarantees that our system can interpret the behavior of the agent. The use of the agent code is wrapped in a try-except block, so if any methods fail we ignore the custom class, which protects against users who do not follow the template and function behavior exactly. If we were to deploy this system, the end user would not know anything about the system's code structure, apart from the custom artificial intelligence agent template. We would also implement a timeout checker, by wrapping each custom-declared function in a timing decorator, which would allow our system to be protected against code that either runs for too long or is maliciously intending to slow down the system. Lastly, the MetaMask integration itself has preventative measures against abuse, which we use within our system. The maximum amount a user can bet is their MetaMask Testnet Sepolia Ethereum balance. This verification is performed by MetaMask and the "Testnet" so no abuse of this fact is possible, as this case is handled externally.

5.3 Limitations

Blackjack AI has some limitations that we need to address before releasing the game to the public. The blackjack game does not implement doubling down or splitting. This is mainly due to the limitations of the artificial intelligence model that we used. The Q-Table is only capable of generating hit and stand as decisions. Furthermore, the site is not correctly configured to host multiple users at one time, it can only serve a single user. This would need to be ratified before full deployment.

5.4 Future Work

We do not plan to maintain or update the project. Ultimately there are plenty of ways for anyone to play Blackjack, and while we have a cool concept, we do not believe it is worth the time, effort, and money to deploy to a live website and continue to maintain the project.

5.5 Project Resources

Github
Contract address and code are also stored in the Github repository.
Blockchain Testing Document
Demo

References

[1] *Gymnasium documentation.* Solving Blackjack with Q-Learning - Gymnasium Documentation. (n.d.). https://gymnasium.farama.org/tutorials/training_agents/blackjack_tutorial/

6 Figures

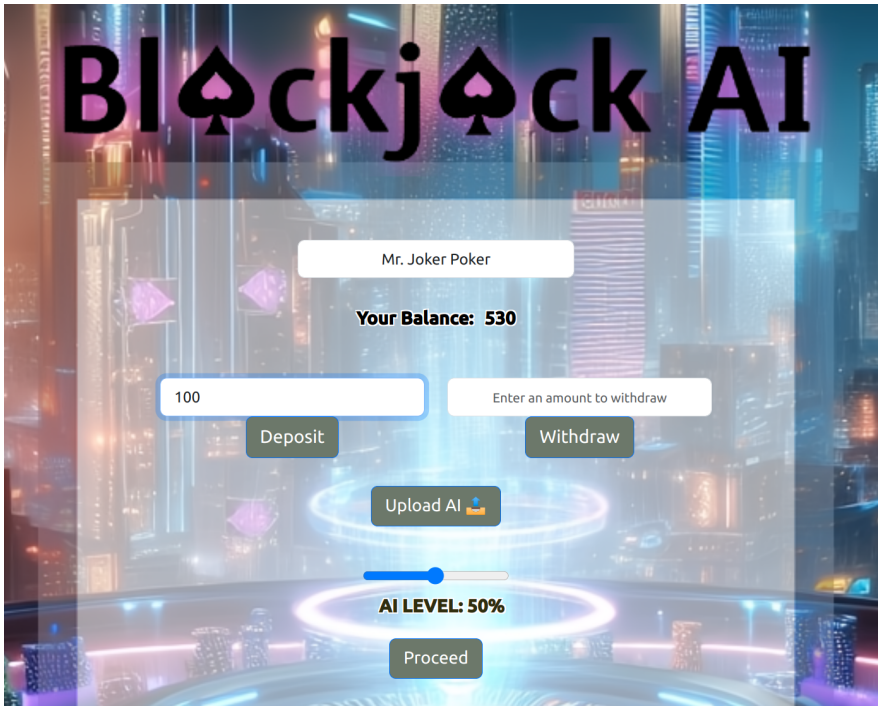


Figure 1: The Blackjack AI Login page

Figure 2: Game Page: The "BET" button is active at the beginning the game

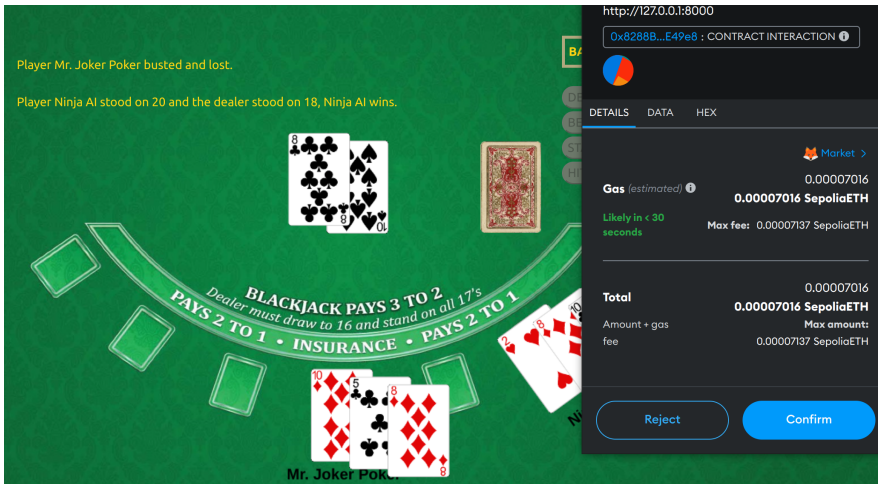
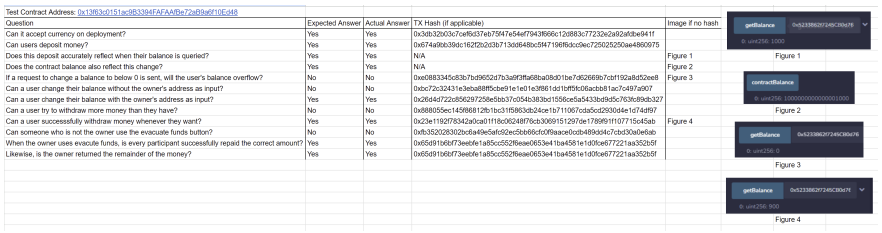


Figure 3: Transaction at the end of each round



Test Contract Address: 0x13f83c0151ac9B3394FAFAAfBe72aB9a6f10Ed48

Question	Expected Answer	Actual Answer	TX Hash (if applicable)	Image if no hash
Can it accept currency on deployment?	Yes	Yes	0x3db32b03c7cef6d37eb75f47a54ef7943f666c12d883c77232e2a92af0be941f	
Can users deposit money?	Yes	Yes	0x674d9bb39dc162f2b2d3b713dd648bc5ff47196ffdcd4ec725025250ae4860975	
Does this deposit accurately reflect when their balance is queried?	Yes	Yes	N/A	Figure 1
Does the contract balance also reflect this change?	Yes	Yes	N/A	Figure 2
If a request to change a balance to below 0 is sent, will the user's balance overflow?	No	No	0xe0883345c83b7bd9652d7b3a9f3ffa68ba08d01be7d626697b7cbf192a8d52ee8	Figure 3
Can a user change their balance without the owner's address as input?	No	No	0xbc72c32431e3eba88ff5cbe91e1e01e3f661dd1bff5fc06acbb81ac7c497a907	
Can a user change their balance with the owner's address as input?	Yes	Yes	0x26d4d722c856297258e5bb37c054b383bd1556ce5a5433bd9d5c763fc89db327	
Can a user try to withdraw more money than they have?	No	No	0x888055ec145f86812fb1bc31f5863db24ce1b711067cda5cd2930d4e1d74df97	
Can a user successfully withdraw money whenever they want?	Yes	Yes	0x23e1192f78342a0ca01f18c06248f76cb3069151297de1789f91f10771 5c45ab	Figure 4
Can someone who is not the owner use the evacuate funds button?	No	No	0xfb352028302bc6a49e9afc92ec5bb66cfc0f9aace0cdb489d94c7cbd30a8e6ab	
When the owner uses evacuate funds, is every participant successfully repaid the correct amount?	Yes	Yes	0x65d91b6bf73eebfe1a85cc552f6eae0653e41ba4581e1d0fce677221aa352b5f	
Likewise, is the owner returned the remainder of the money?	Yes	Yes	0x65d91b6bf73eebfe1a85cc552f6eae0653e41ba4581e1d0fce677221aa352b5f	

Figure 1
Figure 2
Figure 3
Figure 4

Figure 4: Blockchain testing document

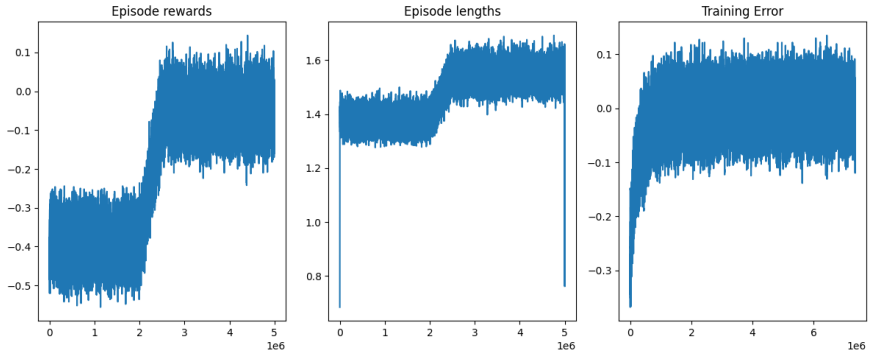


Figure 5: Q-Learning Training Results

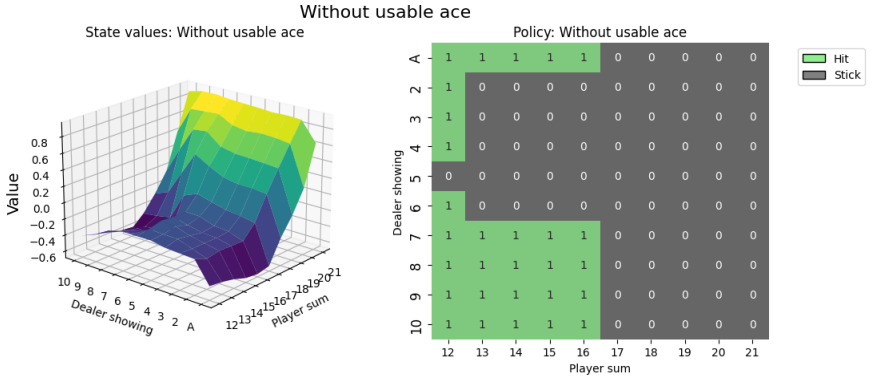


Figure 6: AI Strategy: No Usable Ace

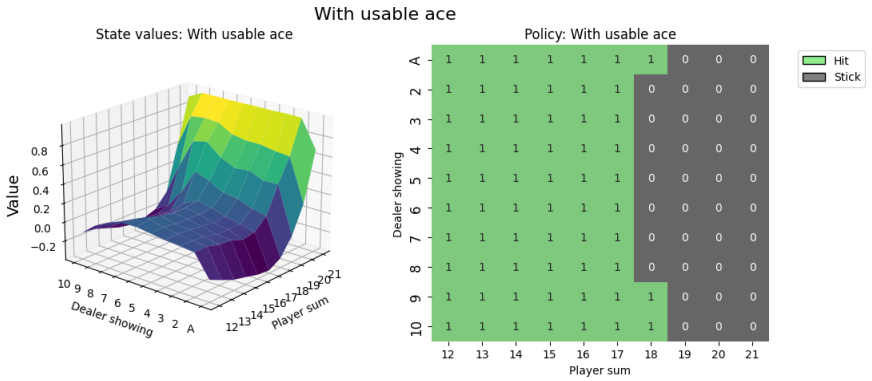


Figure 7: AI Strategy: With Usable Ace