BlackJack Player – A Deep Reinforcement Learning Approach

Nehmiya Shikur
Faculty of Computer Science and Business Information Systems
University of Applied Sciences Wurzburg-Schweinfurt
Wurzburg, Germany
nehmiya.shikur@student.fhws.de

Abstract— **Reinforcement learning methods along with deep learning have been and are still dominant in groundbreaking breakthroughs which have been recorded in recent times. With the help of these advancements different agents are able to learn and pickup patterns from their environments without a human assistance. Mainly, these self-learning agents are game agents that absorb and realize their corresponding surroundings which helps them play different complex games optimally. Said that, this paper outlines the implementation of a self-learning Blackjack player or agent using deep and reinforcement learning methods. In addition it demonstrates all the major steps which were taken in the realization starting from the environment setup of the game up to the evaluation of the accuracy of the game play with the self learning agent. The practical implementation of the gameplay have been implemented using Python programming language along with some of the resourceful packages it offers like Matplotlib , Numpy and Pytorch.**

I. INTRODUCTION

Deep reinforcement learning, as its name implies, combines the concepts deep learning and reinforcement learning. It describes the use of deep learning models to assist agents in learning about different environments from the states and actions they take, simulating how people learn and adjust their paths as a result of their experiences. Since a person's capacity to learn from past acts strongly correlates with their ability to play most games, this idea and knowledge have had a profoundly revolutionary impact on the scope of game play.

Moreover, It's nearly hard to find an interactive game today that doesn't include a learning model that generates a distinctive response by analyzing and learning the gameplay surroundings and patterns. These models, which are used and tested on many gameplays, are transferable since they have shown researchers and computer scientists how to teach and test them to address real-world issues affecting human beings.[1]

Specifically card and board games take the wider share of the many AI researches in gameplay.[3] Out of these card games Blackjack which goes by the nickname of "twenty one" is the one that is widely used that evolves on having a card count of more than the dealer while not exceeding the number 21. A thorough explanation of the gameplay is found on the next chapter.

II . BLACKJACK GAME RULE

Blackjack is a card game which is played between a dealer and a player. The game can be played in a single or in a multiplayer settings. The dealer is responsible for managing and running the complete gameplay. The main goal of the game is to have cards that sum up to 21 or get as close to 21 without surpassing it. If the sum of the player's or dealer's cards surpass 21 it is called a "Bust" which automatically stops the gameplay.[2]

The game starts by the dealer handing out two face up cards to the player and two cards to himself which one of the cards will be faced down. Then the player have a chance to "hit" which notifies the dealer to add him a card or a "stand" which means he is satisfied with the sum of cards he currently owe. The player is allowed to hit as many times he wishes. After the player signals a stand the dealer flips his face downed card and if the sum of his cards is below 17 , he is obliged to hit on the contrary if his cards sum is above 17 is required to stand.

Once both the player and dealer decide to stand, the dealer proceeds to the point counting. All the cards that are between 2 and 10 have a weight that is equal to their face number while number one or Ace can be counted as a 1 or an 11 depending on the players or dealers preferences. Kings , Queens and Jacks have a weight of 10.

After the point counting of the cards is completed who ever records a higher point count while maintaining less than or equal to 21 wins the round. If both the player and the dealer records equal points then the game will end without any winner.

III. IMPLEMENTATION OF THE SYSTEM

A . Initial Environment

The implementation of the basic gameplay environment and rules starts by creating a 1D array that holds all the 52 deck of cards for the game (excluding the 2 jokers). Afterwards a function named "Hit" that randomly select and removes a card from the 1D array is implemented. Consecutively a function called "Game_start" that simulates the initial distribution of the cards to the dealer and player is implemented . Whenever this function is called it starts by creating 2 empty 1D arrays called "player" and "dealer" that accommodate all the cards the player and the dealer takes throughout the game round. Additionally this function is responsible for appending the first 2 cards to the player and 1 cards to the dealer for the start of the game.

Given after the game have started by calling the "Game_start" function, we now need the player to decide whether he wants to hit or stand after observing the player and dealer arrays. For this we need a function with conditional statements that decide the actions of both the player and the dealer. For this we have created a function called play that accommodates two conditional while loops that that makes the player hit if his card sum is below some arbitrary number which will later get modelled using our deep learning model's predictions. And the other loop is responsible for informing the dealer call the hit function while his card sum is below 17.

Once the play function decides the actions of both the player and dealer, the environment proceeds to call the winner function which announces the winner of the game round based on the sum the cards on the 2 arrays.

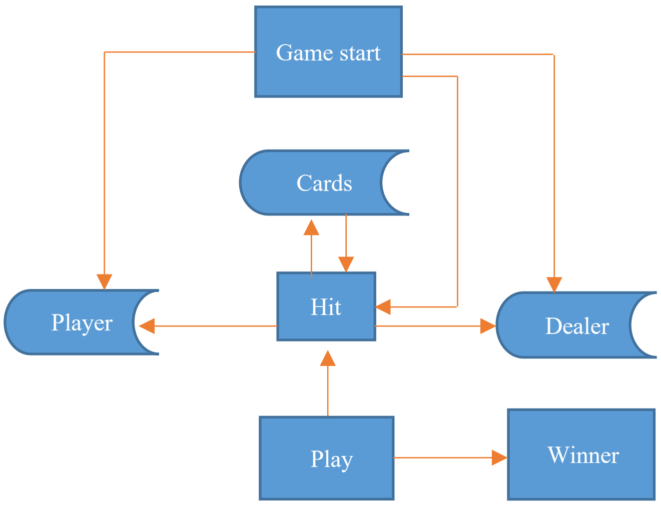


Fig. 1: Initial set up of the game Environment.

The initial game environment which can be seen in Figure 1 have been designed in a confined manner with no learning policy or model added to it. The decision of the player have been directly added in the while loop of the play function which orders it to hit for a card while it's sum is less than 18.This can be demonstrated in the pseudocode below.

Algorithm 1: pseudocode for the initial game environment

```
1   Set cards = [1, … , 10] x 4          -> stores 52 cards
2   function Hit(cards)
3       Set value = random.choice(cards)
4       Remove value from cards deck
5       Return value
6   function Game_start
7       Set player  = Hit (cards) x 2     -> player ∈ $R^{1 x n}$
8       Set dealer  = Hit (cards)         -> dealer ∈ $R^{1 x n}$
9   function play()
10      While sum(player) < 18 do
11          player.insert(Hit(cards))   end for
12  While sum(dealer) < 18 do
13          player.insert(Hit(cards))   end for
14  function winner(player,dealer)
15      Decide winner from sum(player) and sum(dealer)
16      Set Output = decision
17      Return Output
```

Given that we have implemented the basic framework environment of the game now we should add more layers to it so that the agent (player) can learn and play the game dynamically.

B . Modified Environment with more layers

Since our main goal is to design an agent that learns how to play dynamically by learning and improving from the reward it gets while playing the game. For this the first thing we should do is remove the condition inside the while loop on line 10 of Algorithm 1 since we no longer decide on behalf of the player or agent.

On top of our previous environment a reward generating function have been added which helps our agent learn from every action it takes in different states. This function gets called every time the player takes an action weather it is a hit or a stand.

So the agent will be a multi layer perceptron (MLP) neural network model that learns from the rewards it gets while playing the game. This model consists of 2 linear layers with a ReLu non-linearity activation function together with a dropout and a sigmoid as a final layer. The model have an input size of 15 and spits out a single output which is an action 0 or 1.in our case 0 translates into a stand and 1 translates into a Hit.

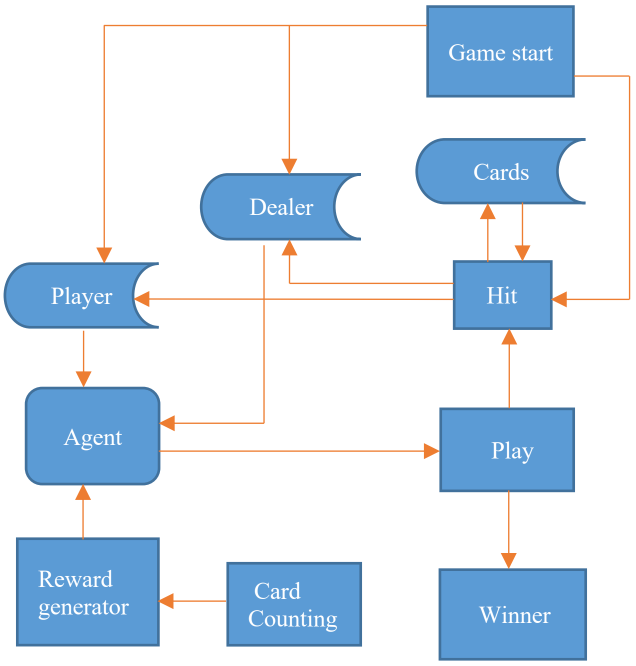


Fig. 2: Modified Environment with more layers

Figure 2 illustrates how our agent model which takes the dealer's and player's cards together with action rewards plays the game iteratively. In addition to Figure 2 there exists one function called "fill_zeros" which is responsible for matching the players and the dealers cards to the input size of our model which is 15. So if the current state of the player is 4 cards this function will append zeros into the players array till the size reaches 15.

Algorithm 2: pseudocode of the modified Enviroment + gameplay

```
1   Set cards = [1, … , 10] x 4        -> stores 52 cards
2   function Hit(cards)
3       Set value = random.choice(cards)
4       Remove value from cards deck
5       Return value
6   function Game_start
7       Set player  = Hit (cards) x 2    -> player ∈ $R^{1 x n}$
8       Set dealer  = Hit (cards)        -> dealer ∈ $R^{1 x n}$
9   Class NeuralNetwork
10      Input: player , dealer , rewards
11      Set linear_layers , Dropout,ReLu , Sigmoid(Σ)
12  function fill_zeros(player,dealer)
13      Set player.insert(0) till len(player) == 15
14      Set dealer.insert(0) till len(player) == 15
15  function rewards_generator(player)
16      Set  value = 0
17      Set  value = 1
18  function play()
19      While Neural_Network == 1 do
20          player.insert(Hit(cards))    end for
21      While sum(dealer) < 17 do
22          player.insert(Hit(cards))    end for
23  function winner(player,dealer)
24      Decide winner from sum(player) and sum(dealer)
25      Set Output = decision
29      Return Output
```

The above pseudocode may seem similar to algorithm 1 but the whole gameplay and decision making have been shifted and changed. Now our agent makes and takes an action freely from the experience it picks up while playing the game which sums up the concept of deep reinforcement learning from a very high level. Line 19 on the above algorithm is the place where the agent makes a decision depending on , as depicted in the condition if it's prediction is 1 , the Hit function will get called which appends a card into the player array.

As it can be seen on line 11 of algorithm 2 our model is made up of an initial linear layer that takes a 1 x 15 sequence and it have a hidden size of 128 , later the data we feed into this network is passed through a ReLu activation function along with a dropout layer which prevents overfitting. After that we have a final linear layer which outputs a single prediction and this prediction is passed through a sigmoid function to retrieve a binary value.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

Formula 1 denotes the sigmoid function which is found in the output layer of our neural network's model. The main reason we have used this activation function is because it gives us a value of true or false (1 or 0). Which is very essential in our setting since it tells the player to hit or stand.

C . Card Counting

Given that we have set up and defined all the required frameworks for our agent to play the game, One additional improvement would be to add a card counting method to it so that it can keep track of the game play and evaluate it's decision based on the cards which have been removed from the deck of cards.

Algorithm 3: Condensed pseudocode for card Counting

```
1   Set cards = [1, … , 10] x 4        -> stores 52 cards
2   Set memory = [ ]
3   function Hit(cards)
4       Set value = random.choice(cards)
5       Remove value from cards deck
6       Add value into memory
7       Return value
8   function Card_counting(memory):
10      Set memory_count = memory[0,..,n]++ / len(memory)
11      Set estimate = "high" if_memory_count < 6.5
12                  else "low"
13      Return estimate
```

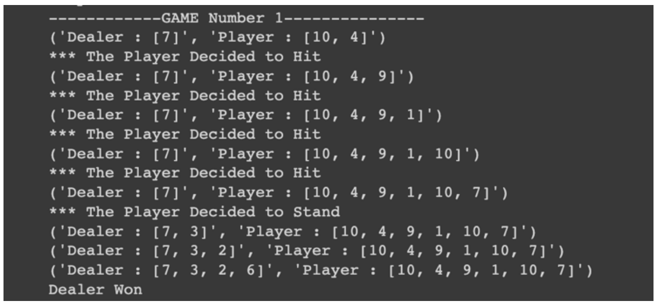
For the card counting we have created a separate 1D array called "memory" that keeps track of all the removed cards from the deck. Every time the player or dealer hits for a card it will be added into the memory array. Based on the prior information that states a full deck consisting 52 cards have a mean value of 6.5, we use the card counting function which is responsible for evaluating weather the cards in the memory array have a high or low mean value when compared to the current deck of cards. If the memory array's mean value is less than 6.5 it denotes that the cards in the deck are mostly cards with a high face value which signals the agent to hit less and vice versa.

Even though the card counting method we used seems good in theory and executable in our computer, It is almost impossible to use this method in real life scenario since keeping track of the mean value of a stochastic array is rather puzzling.

IV EVALUATIONS

Given the main goal of this paper is to implement an agent that learns the game rules and patterns while playing it, we would expect the agent to perform very bad at the start of the game but after some time we anticipate it to learn and play the game well. At the first few iterations of the game a bad policy and actions are expected from the agent since it will be new to the game environment we set up for it.

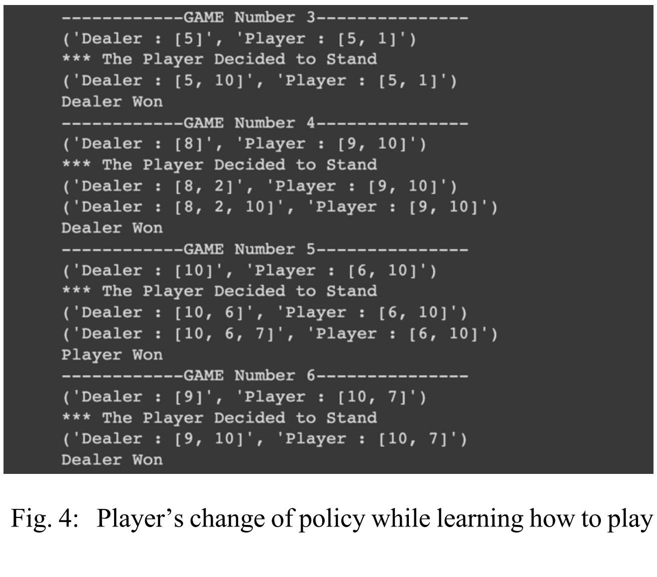
Since we want to demonstrate the progress the agent records while playing the game, we haven't implemented any training function since that would eradicate the novelty and make our model already fit to play the game efficiently which is not our goal.



```
------------GAME Number 1---------------
('Dealer : [7]', 'Player : [10, 4]')
*** The Player Decided to Hit
('Dealer : [7]', 'Player : [10, 4, 9]')
*** The Player Decided to Hit
('Dealer : [7]', 'Player : [10, 4, 9, 1]')
*** The Player Decided to Hit
('Dealer : [7]', 'Player : [10, 4, 9, 1, 10]')
*** The Player Decided to Hit
('Dealer : [7]', 'Player : [10, 4, 9, 1, 10, 7]')
*** The Player Decided to Stand
('Dealer : [7, 3]', 'Player : [10, 4, 9, 1, 10, 7]')
('Dealer : [7, 3, 2]', 'Player : [10, 4, 9, 1, 10, 7]')
('Dealer : [7, 3, 2, 6]', 'Player : [10, 4, 9, 1, 10, 7]')
Dealer Won
```

Fig. 3: Initial gameplay actions of our agent.

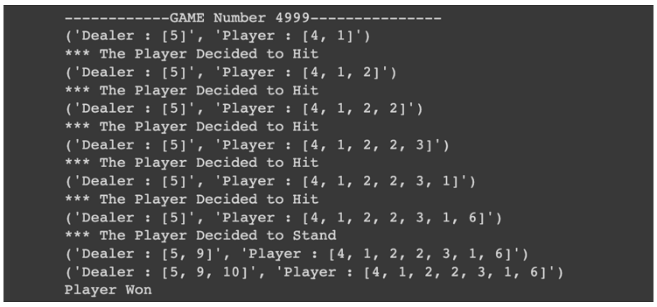
Our agent scored a total of 40 points on it's very first exposure to game. As it can be seen from Figure 3 the agent kept on calling a Hit even after it was in a Bust state. The player made 5 actions and out of these 5 actions the third and fourth ones were completely wrong which each received a reward and these rewards are meant to give the agent a pattern and lesson to come up with a better policy and actions to beat the dealer.



```
------------GAME Number 3---------------
('Dealer : [5]', 'Player : [5, 1]')
*** The Player Decided to Stand
('Dealer : [5, 10]', 'Player : [5, 1]')
Dealer Won
------------GAME Number 4---------------
('Dealer : [8]', 'Player : [9, 10]')
*** The Player Decided to Stand
('Dealer : [8, 2]', 'Player : [9, 10]')
('Dealer : [8, 2, 10]', 'Player : [9, 10]')
Dealer Won
------------GAME Number 5---------------
('Dealer : [10]', 'Player : [6, 10]')
*** The Player Decided to Stand
('Dealer : [10, 6]', 'Player : [6, 10]')
('Dealer : [10, 6, 7]', 'Player : [6, 10]')
Player Won
------------GAME Number 6---------------
('Dealer : [9]', 'Player : [10, 7]')
*** The Player Decided to Stand
('Dealer : [9, 10]', 'Player : [10, 7]')
Dealer Won
```

Fig. 4: Player's change of policy while learning how to play

As the player(agent) proceeds to play and learn the game it tries to change and come up with different policies to beat the dealer. Figure 4 illustrates how the player and model came up with a policy of a Stand despite its card being too low which makes it too easy to get beaten by the dealer. The model came up with this policy because of the negative reward it got for calling to many unnecessary Hits on the First round of the game which can be seen in Figure 3. In similar fashion, this reward based learning approach let's our agent learn and gasp the environment of the gameplay as it continues to play more rounds.



```
------------GAME Number 4999---------------
('Dealer : [5]', 'Player : [4, 1]')
*** The Player Decided to Hit
('Dealer : [5]', 'Player : [4, 1, 2]')
*** The Player Decided to Hit
('Dealer : [5]', 'Player : [4, 1, 2, 2]')
*** The Player Decided to Hit
('Dealer : [5]', 'Player : [4, 1, 2, 2, 3]')
*** The Player Decided to Hit
('Dealer : [5]', 'Player : [4, 1, 2, 2, 3, 1]')
*** The Player Decided to Hit
('Dealer : [5]', 'Player : [4, 1, 2, 2, 3, 1, 6]')
*** The Player Decided to Stand
('Dealer : [5, 9]', 'Player : [4, 1, 2, 2, 3, 1, 6]')
('Dealer : [5, 9, 10]', 'Player : [4, 1, 2, 2, 3, 1, 6]')
Player Won
```

Fig. 5: Agent playing the game proficiently after playing 5000 times.

The above figure shows how the agent is able to self-learn the environment and gameplay after playing it for multiple rounds. Now the agent is able to take actions that are reasonably sound under most states of the game. All the actions made by the player which are shown in Figure 5 are actions we would expect a human being who knows the basic gameplay rules of blackjack to make.

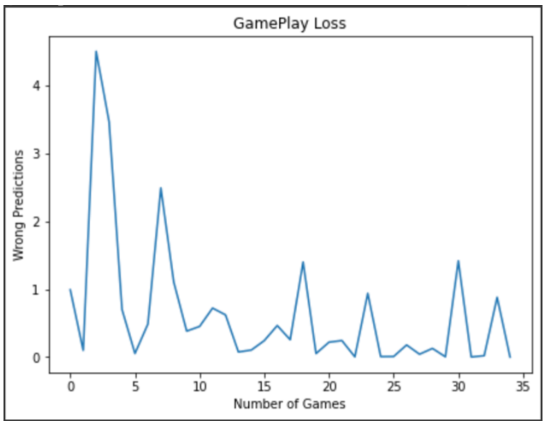


Fig. 6: Wrong actions taken for the first 35 games.

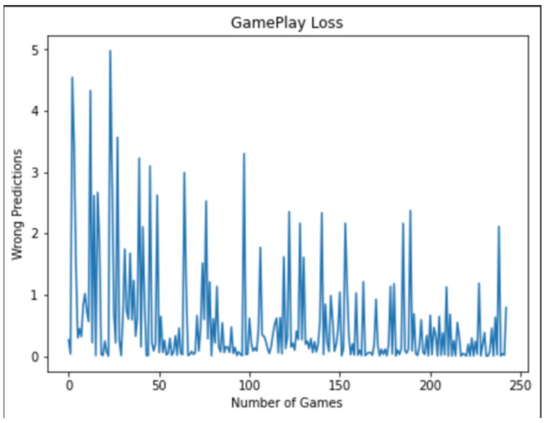


Fig. 7: A plot of the wrong actions taken for the first 250 games.

Both Figure 6 & 7 shows how the bad actions made by the player decreases as it progresses to play more rounds of the game. In most cases the Agent is able to record a winning rate of 30 to 40 percent which is a descent amount considering the restricted number of games the agent is playing on. As the number of games we are playing increases its accuracy also increases which in rare cases edges 50 percent.

Additionally, we have tried to change the rules after our model have started learning the environment and game rules. This rules changes include increasing and decreasing the minimum sum of cards the dealer should meet from 17 to 13. And our model reacts to these changes in a vague way meaning it takes an elongated time to learn and pick up this drastic changes

V. CONCLUSION

Our self learning blackjack player and the overall gameplay works well even though we haven't simulated the multiplayer setting of the game. Bases on the concepts we discussed on the previous sections, It wouldn't be too complicated to add another player into our environment since the basic foundations of the gameplay are established. So adding another player in this case mainly means adding another model and reconfiguring the reward mapping algorithm we implemented along with a little amendment in the functions that validates current state of the gameplay.

The implementation can be further developed and enhanced using time series deep learning models like Recurrent neural networks and LSTM that helps us model predictions of the upcoming game states together with monte carlo methods of reinforcement learning.[4]

REFERENCES

[1] Liu, Ziang, and Gabriel Spil. "Learning Explainable Policy For Playing Blackjack Using Deep Reinforcement Learning (Reinforcement Learning)."

[2] Edward O. Thorp. Beat the Dealer. Vintage, New York, 1966.

[3] Treanor, Mike, Alexander Zook, Mirjam P. Eladhari, Julian Togelius, Gillian Smith, Michael Cook, Tommy Thompson, Brian Magerko, John Levine, and Adam Smith. "AI-based game design patterns." (2015).

[4] Liu, Ziang, and Gabriel Spil. "Learning Explainable Policy For Playing Blackjack Using Deep Reinforcement Learning (Reinforcement Learning)."