L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Artificial Intelligence

Reinforcement Learning Documentation

Isaac Muscat* (0265203L), Matthias Bartolo* (0436103L), Jerome Agius* (0353803L)

*B.Sc. (Hons) Artificial Intelligence (2nd Years)

Study-unit: **Reinforcement Learning**

Code: **ARI2204**

Lecturer: **Dr Josef Bajada**

Table of Contents

Introduction

Blackjack's simplicity makes it an ideal choice for implementing and utilising reinforcement learning. Furthermore, this assignment involved representing a simplified **Blackjack** environment and training an agent via four Reinforcement Learning algorithms.

In Blackjack, a standard deck of 52 cards is used. Additionally, the game is comprised of two game objects, the player, and the dealer. In the start of our game implementation, the player is given two cards whilst the dealer is given one. Moreover, the goal of both the player and the dealer is to get as close to 21 as possible without going over. Both the player and the dealer can choose whether to **stand**, meaning stop taking cards, or to **hit**, meaning take a card from the deck. When both the dealer and the player stand, the player with the largest **total value** wins, given that they do not exceed 21. If one of the players exceeds 21 points, they will lose the game and the win is awarded to the other player [1].

Note: The **total value** of the hand is determined from the cards which comprise it.

Note: Additionally, the **Double Q-Learning** algorithm is not explained thoroughly in this documentation, this was due to the word count limit. However, its evaluation was still provided to be able to compare its results with the other three algorithms.

To train the agent, four algorithms were implemented:

- **Monte Carlo On-Policy Control**
- **SARSA On-Policy Control**
- **Q-Learning (SARSAMAX) Off-Policy Control**
- **Double Q-Learning Off-Policy Control (Extra)**

The programming language chosen to implement such a problem was **Python** as it provided a variety of in-built functions, as well as offering a great degree of convenience which was deemed useful for such an assignment.

Blackjack Environment Representation

Firstly, prior to implementing any of the algorithms mentioned, a standard deck of cards was created.

```
# Defining the point cost of each card value
PointsLookup={ "2":2, "3": 3, "4":4,"5":5,"6":6,"7":7,"8":8,"9":9,"10":10,"Q":10,"J":10,"K":10, "A":11}

# Defining the card components that compose a deck
cardsValue= {"A", "2","3", "4", "5", "6", "7", "8", "9", "10","J","Q","K"}
cardsSuit={"Diamonds", "Spades","Clubs","Hearts"}
```

Figure 1 - Deck of Cards

GetTotalPoints Function

Following this, the GetTotalPoints function was implemented to get the total sum of the cards' values. Due to the Ace having two possible values, initially all aces are worth 11 points, excluding when said ace will cause the player to exceed 21 points, in such cases the ace is assigned a value of 1. This also applies when the player has more than one ace. When the player receives their first ace, the **hasAce** flag is set to true.

Note: The card values are assigned in accordance with the **PointsLookup** dictionary seen in Figure 1.

Method which calculates the total points given a number of cards

```
def GetTotalPoints(player):
    totalPoints=0

    # Looping through the cards in the player hand
    for cards in player.hand:
        # Summing up the total points
        totalPoints+=PointsLookup[cards[1]]
        # If the player has more than 1 ace all other aces are assigned the value of 1 according to black jack rules
        if cards[1] == "A" and player.hasAce:
            totalPoints-=10
        # If the player has only 1 ace the hasAce flag is set to true to activate the above if statement
        elif cards[1] == "A" and not(player.hasAce):
            player.hasAce=True

    # If the player has an ace and more than 21 points the value of said ace
    # is set to 1 so as to not set him over the point limit and hasAce flag is set to false
    if(player.hasAce and totalPoints>21):
        totalPoints-=10
        player.hasAce=False
    return totalPoints
```

Figure 2 - GetTotalPoints Function

Hit & Stand Functions

Moreover, the **hit** and **stand** actions were implemented via a function of the same name. The **Hit** function pops a card from the deck and adds to the player / dealer's hand. The **Stand** function sets the **stand** flag to true.

Method denoting a Hit Action

```
def Hit(player,cardDeck):
    # Removing the top card from the deck
    card=cardDeck.pop(0)
    # Adding said card to the player hand
    player.hand.append(card)
```

Method denoting a Stand Action

```
def Stand(player):
    # Setting the stand flag to denote that the player has stopped playing
    player.stand=True
```

Figure 3 - Hit & Stand Functions

CheckWin Function

To determine the winner of an episode, the **CheckWin** function was implemented which returns **0** if a draw has occurred, a **1** if the player has won and a **-1** if the player has lost. Otherwise, if the game is still ongoing, **None** is returned.

Method used to check if the Player / Dealer has won

```
def CheckWin(player,dealer):
    # Checking if the player has equal points to the dealer and the dealer has stood resulting in a draw
    if(dealer.stand and player.totalPoints==dealer.totalPoints):
        return 0
    # Checking if the player has exactly 21 points resulting in a win
    elif(player.totalPoints==21):
        return 1
    # Checking if the dealer has exactly 21 points resulting in a loss
    elif(dealer.totalPoints==21):
        return -1
    # Checking if the player has more than 21 points resulting in a loss
    elif(player.totalPoints>21):
        return -1
    # Checking if the dealer has more than 21 points resulting in a win
    elif(dealer.totalPoints>21):
        return 1
    # Checking if the dealer has stood and has less points than the player resulting in a win
    elif(dealer.stand and player.totalPoints>dealer.totalPoints):
        return 1
    # Checking if the dealer has stood and has more points than the player resulting in loss
    elif(dealer.stand and player.totalPoints<dealer.totalPoints):
        return -1
    # Returning None if game is still on going
    return None
```

Figure 4 - CheckWin Function

Player Class

The player class is used to create instances of player and dealer objects. Furthermore, this class is composed of the following components:

- **hand** – List of cards composing the player's hand.
- **totalPoints** – Sum of the hand's values.
- **stand** – Flag to denote whether the player has stood.
- **hasAce** – Flag to denote whether the player has an Ace.

Class denoting the Player Object

```
class Player:
    # Method to initialise the player object, each player starts of with:
    # an empty hand, 0 points and both flags set to false
    def __init__(self):
        self.hand=list()
        self.totalPoints=0
        self.stand=False
        self.hasAce=False
```

Figure 5 - Player Class

InitiateGame Function

The **InitiateGame** function was created to instantiate the player and dealer game objects alongside the base game by creating and shuffling a card deck and assigning each player the appropriate number of cards.

Note: The player starts with two cards whilst the dealer with one.

Method used to initialise the starting state of the game

```
def InitiateGame():
    # Initialising an empty deck
    cardDeck=list()
    # Initialising two players
    player=Player()
    dealer=Player()

    # Assigning all the cards to the deck
    for suit in cardsSuit:
        for value in cardsValue:
            cardDeck.append((suit,value))
    # Shuffling the cards in the deck
    random.shuffle(cardDeck)

    # Assigning two cards to the player and one card to the dealer
    Hit(player,cardDeck)
    Hit(player,cardDeck)
    Hit(dealer,cardDeck)

    # Assigning the appropriate points to the player and the dealer
    player.totalPoints=GetTotalPoints(player)
    dealer.totalPoints=GetTotalPoints(dealer)
    return player,dealer,cardDeck
```

Figure 6 - InitiateGame Function

InitialiseQTable Function

The **InitialiseQTable** function constructs a data structure that stores all the possible states present within the environment. Each state consists of the following properties:

- The number of times an action was selected in a state.
- The current estimated value of performing an action in a state.

Method to initialise the QTable (Data structure containing all possible states and relevant information)

```
def InitialiseQTable():
    Qtable=dict()

    # Looping if the player has between 12 and 21 points
    for playerPoints in range(12,22):
        # Looping if the dealer has between 2 and 11 points (meaning they only have 1 card)
        for dealerCard in range(2,12):
            # Looping twice, once for each possible state given that the player is in possesion of an ace
            for AceFlag in [True,False]:
                # Constructing the state which consists of the player and dealer points
                # as well as whether an ace is in the player hand
                state=str(playerPoints)+","+str(dealerCard)+","+str(AceFlag)
                Qtable[state]={}
                # Assigning the Q-value and times visited to each state, both initialised to 0
                for action in ["Hit","Stand"]:
                    Qtable[state][action]={"Values":0,"Count":0}
    return Qtable
```

Figure 7 - InitialiseQTable Function

ArgMax Function

Similarly, the **ArgMax** function returns a “Hit” or a “Stand” depending on which action leads to the greatest reward depending on the value in the **QTable**.

Method used to retrieve the action according to the max Q-value for the specified state

```
def ArgMax(Qtable, state):
    # Checking if hit has a greater value than stand and returning the appropriate action
    if(Qtable[state]["Hit"]["Values"]>=Qtable[state]["Stand"]["Values"]):
        return "Hit"
    else:
        return "Stand"
```

Figure 8 - ArgMax Function

This section utilises the following references: [2,3].

Monte Carlo On-Policy Algorithm

Overview

The **Monte Carlo** algorithm is a Reinforcement Learning algorithm which works best with episodic tasks as it looks at the complete episode to determine the best course of action. This algorithm determines the mean return (reward) of each episode. As can be seen in the code below, this algorithm uses the values in the QTable to conclude the best action to execute [4, 5].

Method used to outline the step by step happenings throughout the game

```
def GameMonteCarlo(configBit, discountFactor, episodes=100000):
    episodesDict = {"wins": [], "draws": [], "losses": [], "episodes": [], "stateAction": {}, "qEstimatedValue": {}}
    qTableLookupList=list()
    Qtable=InitialiseQTable()
    playerWins,dealerWins,draws,reward=0,0,0,0
    rewardEpisode=0

    # Looping through all the episodes
    for episode in range(1,episodes+1):
        player,dealer,cardDeck=InitiateGame()

        # Simulating a single black jack round
        checkWin,stateActionList=BlackJackRoundMonteCarlo(configBit,player, dealer,cardDeck,episode,Qtable)

        # Adding the points accordingly based on the results of the game
        if(checkWin==1):
            playerWins+=1
            reward+=1
            rewardEpisode=1
        elif(checkWin==-1):
            dealerWins+=1
            reward-=1
            rewardEpisode=-1
        else:
            draws+=1
        futureReward=0
        # Looping through all the state actions in a single episode and updating "count" and "values" of the QTable accordingly
        for stateaction in reversed(stateActionList):
            if(stateaction[0] is not None):
                futureReward+=rewardEpisode
                Qtable[stateaction[0]][stateaction[1]]["Count"]+=1
                Qtable[stateaction[0]][stateaction[1]]["Values"]= Qtable[stateaction[0]][stateaction[1]]["Values"]
                *discountFactor + float(1/Qtable[stateaction[0]][stateaction[1]]["Count"]
                *(futureReward-Qtable[stateaction[0]][stateaction[1]]["Values"]))

            key=str(stateaction[0])+","+str(stateaction[1])

            # Incrementing the visited counter for each state action pair
            if(key not in episodesDict["stateAction"].keys() and stateaction[0] is not None):
                episodesDict["stateAction"][key] = 1
                qTableLookupList.append((stateaction[0],stateaction[1]))
            elif(stateaction[0] is not None):
                episodesDict["stateAction"][key] += 1

        # Storing the relevant information in 1000 episode intervals
        if(episode%1000==0):
            # Printing the requested results
            print("Episode: ",episode," Player Wins:",playerWins," Dealer Wins: ",dealerWins," Draws: ",draws, " Reward: ",
                  reward)
            episodesDict["wins"].append(playerWins)
            episodesDict["draws"].append(draws)
            episodesDict["losses"].append(dealerWins)
            episodesDict["episodes"].append(episode)

            # Resetting counts to zero
            playerWins,dealerWins,draws,reward=0,0,0,0

    # Updating the qEstimatedValue for the respective state action pair and returning the corresponding dictionary
    for qKey in qTableLookupList:
        if(qKey[0] not in episodesDict["qEstimatedValue"].keys()):
            episodesDict["qEstimatedValue"][qKey[0]]={qKey[1]:Qtable[qKey[0]][qKey[1]]["Values"]}
        else:
            episodesDict["qEstimatedValue"][qKey[0]][qKey[1]]=Qtable[qKey[0]][qKey[1]]["Values"]

    # Sorting the state action pairs by descending order
    episodesDict["stateAction"] = {k: v for k, v in sorted(episodesDict["stateAction"].items(), key=lambda item: item[1])}
    return episodesDict
```

Figure 9 - GameMonteCarlo Function

GameMonteCarlo Function

The function seen in Figure 9 above, first loops in accordance with the **episode** parameter. For each **episode**, a round is carried out via the **BlackJackRoundMonteCarlo** function, this is than followed up by setting the appropriate rewards given that the game has terminated. A loop is then performed on the state actions list so that the **count** and **values** of the **QTable** are updated accordingly. For every **1000** episodes, the wins, losses, and draws are recorded for evaluation later. Moreover, the **qEstimatedValue** for each state action pair is updated and finally, the state action pairs are sorted in descending order.

Note: The pseudocode presented in Figure 10 inspired the code discussed above.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow \text{average}(Returns(S_t))$

Source: Sutton and Barto (2018)

Figure 10 - MonteCarlo Pseudocode [6]

BlackJackRoundMonteCarlo Function

The **BlackJackRoundMonteCarlo** function first checks whether the game has concluded, if the latter is the case it will proceed to repeatedly call the **PlayerPolicy** function to determine a winner. The functionality of the aforementioned function differs in accordance with the **configBit**. The **configBit** determines whether exploration will be carried out and the **epsilon** (exploration probability) to be used. In this case, the code utilises four different configurations:

- $\varepsilon = \frac{1}{k} +$ exploring
- $\varepsilon = \frac{1}{k}$ without exploring
- $\varepsilon = e^{(\frac{-k}{1000})}$ without exploring
- $\varepsilon = e^{(\frac{-k}{10000})}$ without exploring

After this, the resultant state action pair is appended to the **stateActions** list. Finally, if the player has stood, the dealer continues to hit until they reach a total point value greater than or equal to 17, at which point the game will conclude and a winner is determined.

Method defining the rules of a Black Jack Round

1. configBit - Value denoting the chosen configuration type
2. player - Holds the player object
3. dealer - Holds the dealer object
4. cardDeck - Holds the card deck object
5. episode - Denotes the current episode (An episode consists of a full sequnce of actions leading to a terminal state)
6. Qtable - Data structure holding the possible states along with their value and number of times visited

```
def BlackJackRoundMonteCarlo(configBit,player,dealer,cardDeck,episode,Qtable):
    # Checking if the player has won/lost
    checkWin=CheckWin(player,dealer)
    # Initialising the stepNumber
    stepNumber=1
    # Initialising the stateAction list
    stateAction=list()
    # Looping until a winner is determined
    while True:

        # Utilising the intended policy according to the configBit
        if(configBit==1):
            # Policy with exploration turned on with epsilon being 1/episode
            resultantStateAction=PlayerPolicy(player,dealer,cardDeck,stepNumber,1/episode,Qtable, True)
        elif(configBit==2):
            # Policy with exploration turned off with epsilon being 1/episode
            resultantStateAction=PlayerPolicy(player,dealer,cardDeck,stepNumber,1/episode,Qtable, False)
        elif(configBit==3):
            # Policy with exploration turned off with epsilon being e^-episode/1000
            resultantStateAction=PlayerPolicy(player,dealer,cardDeck,stepNumber,math.e**(-episode/1000),Qtable, False)
        elif(configBit==4):
            # Policy with exploration turned off with epsilon being e^-episode/10000
            resultantStateAction=PlayerPolicy(player,dealer,cardDeck,stepNumber,math.e**(-episode/10000),Qtable, False)
        # Appending the resultant action in accordance with the policy
        stateAction.append(resultantStateAction)

        # Checking if the player has stood and has less than 22 points, if this is the case
        # the dealer will proceed to hit until they have more than 17 points
        if player.stand is True and player.totalPoints<=21:
            while dealer.totalPoints<17:
                Hit(dealer,cardDeck)
                # Updating the dealer points accordingly
                dealer.totalPoints=GetTotalPoints(dealer)
            Stand(dealer)

        checkWin=CheckWin(player,dealer)

        # If a winner is chosen, exit from the loop
        if(checkWin is not None):
            break
        # Incrementing the stepNumber
        stepNumber+=1

    # Returning the winning condition and resultant stateAction
    return checkWin,stateAction
```

Figure 11 - BlackJackRoundMonteCarlo Function

PlayerPolicy Function

The **PlayerPolicy** function determines which action the player is going to choose. The following takes effect given that the player has a value between 11 and 21. This is the case as it always makes sense to hit if the player has less than 11 points whilst it makes sense to stand if they have 21. Otherwise, if the **stepNumber** is 1 and the **exploreFlag** is enabled, a random action is chosen with equal probability, this is used for the $\varepsilon = \frac{1}{k} + \text{exploring}$ configuration. If this is not the case, based on a probability **epsilon**, a random or optimal action is selected. The latter is determined by calling the **ArgMax** function described earlier. The state action pair is then returned.

Method defining the Player Policy (Technique determining which actions the player will choose)

1. player - Holds the player object
2. dealer - Holds the dealer object
3. cardDeck - Holds the card deck object
4. stepNumber - Value denoting which step we are currently in
5. epsilon - Value denoting the exploration probability
6. Qtable - Data structure holding the possible states along with their value and number of times visited
7. exploreFlag - Flag denoting if random exploration should occur as the initial step

```
def PlayerPolicy(player,dealer,cardDeck,stepNumber,epsilon,Qtable,exploreFlag):

    action=None
    # Constructing the current state
    state=str(player.totalPoints)+","+str(PointsLookup[dealer.hand[0][1]])+","+str(player.hasAce)

    # Checking if the player has exactly 21 points, if so the player stands as further playing would lead to a less
    if player.totalPoints==21:
        Stand(player)
        action="Stand"
    # Checking if the player has between 12 and 20 points, if so executing an action and transferring to a successor state
    elif player.totalPoints>11 and player.totalPoints<21:
        # Checking if it's the first step and the explore flag is set to true, if so the player
        # will randomly choose to hit or stand with equal probability
        if(stepNumber==1 and exploreFlag == True):
            if(random.randint(0,1)==1):
                action="Hit"
                Hit(player,cardDeck)
                # Updating the player points accordingly
                player.totalPoints=GetTotalPoints(player)
            else:
                action="Stand"
                Stand(player)
        # If the above doesn't hold, the player will choose to either hit/stand
        # with equal probability (Exploration) or choose the best possible action (Exploitation) in accordance with epsilon
        else:
            if(random.random()<epsilon): # Probability of epsilon
                if(random.randint(0,1)==1):
                    action="Hit"
                    Hit(player,cardDeck)
                    # Updating the player points accordingly
                    player.totalPoints=GetTotalPoints(player)
                else:
                    action="Stand"
                    Stand(player)
            else:# Probability of 1-epsilon
                bestAction=ArgMax(Qtable, state)
                if(bestAction=="Hit"):
                    action="Hit"
                    Hit(player,cardDeck)
                    # Updating the player points accordingly
                    player.totalPoints=GetTotalPoints(player)
                else:
                    action="Stand"
                    Stand(player)
    # If the above doesn't hold, the player will be forced to hit as this action has no
    # downside given that the player has less than 12 points
    else:
        Hit(player,cardDeck)
        player.totalPoints=GetTotalPoints(player)
        action="Hit"
        # This action maps to no state in the Q-table given that the player has less than 12 points
        state=None

    # Returning the old state and action carried out
    return [state,action]
```

Figure 12 - PlayerPolicy Function

SARSA On-Policy and Q-Learning (SARSAMAX) Off-Policy Algorithms

Overview

SARSA and **SARSAMAX** (also known as Q-Learning) are two other types of reinforcement learning algorithms which were implemented in this assignment. The main difference between **SARSA** and **SARSAMAX** is that the former is an on-policy based learning algorithm whilst the latter is an off-policy learning algorithm. **On-policy** means that the algorithm uses the same policy to learn and improve. On the other hand, **off-policy** means that the algorithm uses a behavioural policy to determine how it will explore and exploit whilst improving a different target policy. In the **SARSA** implementation, an epsilon greedy policy is used whereas in **SARSAMAX** an epsilon greedy policy is used for exploring and a greedy policy is used for updating [7-9].

Method used to outline the step by step happenings throughout the game

```
def GameSarsa(configBit,sarsaConfigBit,episodes, maxSteps,  discountFactor):
    episodesDict = {"wins": [], "draws": [], "losses": [], "episodes": [], "stateAction": {}, "qEstimatedValue": {}}
    qTableLookupList=list()
    Qtable=InitialiseQTable()
    playerWins,dealerWins,draws,reward=0,0,0,0

    # Looping through all the episodes
    for episode in range(1,episodes+1):
        player,dealer,cardDeck=InitiateGame()

        # Simulating a single black jack round
        checkWin,stateActionList=BlackJackRoundSarsa(configBit,sarsaConfigBit,player, dealer,cardDeck,episode,Qtable,
                                                     maxSteps,discountFactor)

        # Adding the points accordingly based on the results of the game
        if(checkWin==1):
            playerWins+=1
            reward+=1
        elif(checkWin==-1):
            dealerWins+=1
            reward-=1
        else:
            draws+=1

        # Looping through all the state actions in a single episode and updating the stateAction counts
        for stateaction in stateActionList:
            key=str(stateaction[0])+","+str(stateaction[1])
            if(key not in episodesDict["stateAction"].keys() and stateaction[0] is not None):
                episodesDict["stateAction"][key]=1
                qTableLookupList.append((stateaction[0],stateaction[1]))
            elif(stateaction[0] is not None):
                episodesDict["stateAction"][key]+=1
        # Storing the relevant information in 1000 episode intervals
        if(episode%1000==0):

            # Printing the requested results
            print("Episode: ",episode," Player Wins:",playerWins," Dealer Wins: ",dealerWins," Draws: ",draws,
                  " Reward: ",reward)
            episodesDict["wins"].append(playerWins)
            episodesDict["draws"].append(draws)
            episodesDict["losses"].append(dealerWins)
            episodesDict["episodes"].append(episode)

            # Resetting counts to zero
            playerWins,dealerWins,draws,reward=0,0,0,0

    # Updating the qEstimatedValue for the respective state action pair and returning the corresponding dictionary
    for q in qTableLookupList:
        if(q[0] not in episodesDict["qEstimatedValue"].keys()):
            episodesDict["qEstimatedValue"][q[0]]={q[1]:Qtable[q[0]][q[1]]["Values"]}
        else:
            episodesDict["qEstimatedValue"][q[0]][q[1]]=Qtable[q[0]][q[1]]["Values"]

    # Sorting the state action pairs by descending order
    episodesDict["stateAction"] = {k: v for k, v in sorted(episodesDict["stateAction"].items(), key=lambda item: item[1])}
    return episodesDict
```

Figure 13 - GameSarsa Function

GameSarsa Function

The above **GameSarsa** function first loops in accordance with the episode parameter. For each episode, the **BlackJackRoundSarsa** function is called, and if the game has terminated, the appropriate reward is set. A loop is then performed on the state actions list so that the counts of the state actions are updated appropriately. For every **1000** episodes, the wins, losses, and draws are recorded for evaluation later. Moreover, the **qEstimatedValue** for each state action pair is updated and finally these are sorted in descending order.

Note: This function works the same for both **SARSA** and **SARSAMAX**.

The code shown in Figure 13 above was inspired by the following pseudocode:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Source: Sutton and Barto (2018)

Figure 14 - SARSA Pseudocode [10]

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Source: Sutton and Barto (2018)

Figure 15 - SARSAMAX Pseudocode [11]

BlackJackRoundSarsa Function

The **BlackJackRoundSarsa** function first checks whether the game has finished or not. Following this, the epsilon was determined based on the configuration bit passed. In this case, the code utilises four different configurations of epsilon:

- $\varepsilon = 0.1$
- $\varepsilon = \frac{1}{k}$
- $\varepsilon = e^{\left(\frac{-k}{1000}\right)}$
- $\varepsilon = e^{\left(\frac{-k}{10000}\right)}$

The state was then constructed, and the action was chosen via the **TakeEpsilonGreedyPolicy** function. A loop is then performed for **maxSteps** times, in which the appropriate action is performed first. Following this, if the player has stood, the dealer continues to hit until they reach a total point value greater than or equal to 17, at which point the game will conclude and a winner is determined. The reward is then assigned based on the result of the game, the successor state is constructed and the loop restarts. The **QTables** count is incremented, and **alpha** is calculated given that the state action pair has been visited at least once. This is then followed by updating the **QTable** values in accordance with the policy being used.

Method defining the rules of a Black Jack Round

1. configBit - Value denoting the chosen configuration type
2. sarsaConfigBit - Value denoting the chosen SARSA or SARSA Max
3. player - Holds the player object
4. dealer - Holds the dealer object
5. cardDeck - Holds the card deck object
6. episode - Denotes the current episode (An episode consists of a full sequnce of actions leading to a terminal state)
7. Qtable - Data structure holding the possible states along with their value and number of times visited
8. maxSteps - Steps in an episode (Set to a large value)
9. discountFactor - Value used to denote the importance of future rewards

```
def BlackJackRoundSarsa(configBit,sarsaConfigBit,player,dealer,cardDeck,episode,Qtable,maxSteps,discountFactor):
    epsilon=0
    # Checking if the player has won/lost
    checkWin=CheckWin(player,dealer)
    stateAction=list()

    # Utilising the intended policy according to the configBit
    if(configBit==1):
        # Setting epsilon to 0.1
        epsilon=0.1
    elif(configBit==2):
        # Setting epsilon to 1 divided by the episode count
        epsilon=1/episode
    elif(configBit==3):
        # Setting epsilon to e ^ -episode count / 1000
        epsilon=math.e**(-episode/1000)
    elif(configBit==4):
        # Setting epsilon to e ^ -episode count / 10000
        epsilon=math.e**(-episode/10000)
```

Figure 16 - BlackJackRoundSarsa Function Part 1

```
    stepNumber=1

    # Constructing the current state
    state=str(player.totalPoints)+","+str(PointsLookup[dealer.hand[0][1]])+","+str(player.hasAce)

    # Getting the next action based on epsilon greedy policy
    action=TakeEpsilonGreedyPolicy(state, Qtable, epsilon)

    # Loop to maxSteps
    while stepNumber<maxSteps:

        # Playing the appropriate action
        if(action=="Hit"):
            Hit(player,cardDeck)
            player.totalPoints=GetTotalPoints(player)
        else:
            Stand(player)

        # Checking if the player has stood and has less than 22 points, if this is the case
        # the dealer will proceed to hit until they have more than 17 points
        if player.stand is True and player.totalPoints<=21:
            while dealer.totalPoints<17:
                Hit(dealer,cardDeck)
                # Updating the dealer points accordingly
                dealer.totalPoints=GetTotalPoints(dealer)
            Stand(dealer)

        # Checking if the player has won/lost
        checkWin=CheckWin(player,dealer)
        reward=0
        # Updating the reward appropriately
        if(checkWin==1):
            reward=1
        elif(checkWin==-1):
            reward=-1
        elif(checkWin==0):
            reward=0

        # Constructing the new current state
        newstate= str(player.totalPoints)+","+str(PointsLookup[dealer.hand[0][1]])+","+str(player.hasAce)

        # Getting the next action based on epsilon greedy policy
        newaction=TakeEpsilonGreedyPolicy(newstate, Qtable, epsilon)

        # Adding the state action pair, updating the count of the QTable
        # and calculating alpha if the state is present in it
        if(state in Qtable.keys()):
            stateAction.append([state,action])
            Qtable[state][action]["Count"]+=1
            alpha=float(1/(Qtable[state][action]["Count"]+1))

        # Updating the corresponding QTable according to the sarsaConfigBit for Non-Terminal States
        # 1 is SARSA
        # 2 is SARSAMAX
        if(checkWin is None and state in Qtable.keys() and newstate in Qtable.keys()):
            if(sarsaConfigBit==1):
                Qtable[state][action]["Values"]+=float(alpha*(reward+discountFactor*Qtable[newstate][newaction]["Values"]
                                                              -Qtable[state][action]["Values"]))
            else:
                Qtable[state][action]["Values"]+=float(alpha*(reward+discountFactor*Qtable[newstate]
                                                              [TakeGreedyPolicy(newstate, Qtable)]["Values"]
                                                              -Qtable[state][action]["Values"]))
        # Updating for Terminal States
        elif(state in Qtable.keys()):
            Qtable[state][action]["Values"]+=float(alpha*(reward-Qtable[state][action]["Values"]))

        action=newaction
        state=newstate
        stepNumber+=1
        # Checking if a winning condition has occurred
        if(checkWin is not None):
            break

    return checkWin,stateAction
```

Figure 17 - BlackJackRoundSarsa Function Part 2

TakeEpsilonGreedyPolicy & TakeGreedyPolicy Functions

For these algorithms, two policies were implemented, the **Epsilon Greedy Policy** and the **Greedy Policy**. These policies only take effect given that the player has a value greater than 11 and less than 21. This is the case as it always makes sense to hit if the player has less than 11 points whilst it makes sense to stand if they have 21. In the **TakeGreedyPolicy**, the action is determined by calling the **ArgMax** function. The **TakeEpsilonGreedyPolicy** function builds upon the **TakeGreedyPolicy** function, executing a random or optimal action in accordance with probability **epsilon**. The successor state action pair is then returned.

Method to choose the best action according to the Epsilon Greedy Policy

```
def TakeEpsilonGreedyPolicy(state, Qtable, epsilon):

    # Constructing the current state
    stateParams=state.split(",")

    # Checking if the player has exactly 21 points, if so the player stands as further playing would lead to a less
    if int(stateParams[0])==21:
        action="Stand"

    # Checking if the player has between 12 and 20 points, if so executing an action and transferring to a successor state
    elif int(stateParams[0])>11 and int(stateParams[0])<21:
        # Choosing to either hit/stand with equal probability (Exploration) or
        # choosing the best possible action (Exploitation) in accordance with epsilon
        if(random.random()<epsilon): # Probability of epsilon
            if(random.randint(0,1)==1):
                action="Hit"
            else:
                action="Stand"
        else: # Probability of 1-epsilon
            # Choosing the best action via the ArgMax function
            bestAction=ArgMax(Qtable, state)
            if(bestAction=="Hit"):
                action="Hit"
            else:
                action="Stand"
    else:
        action="Hit"

    return action
```

Figure 18 - TakeEpsilonGreedyPolicy Function

Method to choose the best action according to the Greedy Policy

```
def TakeGreedyPolicy(state, Qtable):

    # Constructing the current state
    stateParams=state.split(",")

    # Checking if the player has exactly 21 points, if so the player stands as further playing would lead to a less
    if int(stateParams[0])==21:
        action="Stand"

    # Checking if the player has between 12 and 20 points, if so executing an action and transferring to a successor state
    elif int(stateParams[0])>11 and int(stateParams[0])<21:

        # Choosing the best action via the ArgMax function
        bestAction=ArgMax(Qtable, state)
        if(bestAction=="Hit"):
            action="Hit"
        else:
            action="Stand"
    else:
        action="Hit"

    return action
```

Figure 19 - TakeGreedyPolicy Function

Evaluation

The main goal in **Reinforcement Learning** is finding a balance between **Exploration** and **Exploitation**. In continuation, **Exploration** involves increasing the current knowledge by executing different actions randomly which may not necessarily be the most optimal. On the other hand, **Exploitation** refers to executing the most optimal action currently known.

Note: The evaluation metrics provided in this documentation were done via functions within the notebook which create bar charts and line graphs. Moreover, a main menu was created to increase user-friendliness.

Monte Carlo On-Policy Algorithm

The following graphs and charts below, were generated when computing the **Monte Carlo On-Policy** algorithm:

Configuration 1: $\varepsilon = \frac{1}{k} +$ exploring

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	H	S	S	S	S	S	S	S
18	S	S	H	S	S	S	S	S	S	S
17	S	S	S	H	S	S	S	S	S	S
16	S	S	S	H	H	S	S	S	S	H
15	S	S	H	S	S	S	S	S	S	H
14	S	H	H	S	H	S	S	S	S	S
13	H	H	S	H	H	H	S	S	S	H
12	H	S	H	H	S	S	H	S	S	S

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	H	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	H	S	S	S	S	S
17	H	H	S	H	H	S	S	S	S	S
16	H	H	S	H	H	S	S	S	S	S
15	S	H	S	H	H	S	S	S	S	S
14	S	S	S	H	H	S	H	S	S	S
13	H	H	S	H	S	H	S	S	S	S
12	S	H	H	S	S	S	S	H	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	S	H	S
15	S	S	S	S	H	H	H	S	S	S
14	S	S	S	S	S	H	S	S	S	S
13	S	S	S	S	S	H	H	H	S	S
12	S	H	S	S	S	H	H	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	H	H	S	S
16	S	S	S	S	S	H	H	H	S	S
15	S	S	S	S	S	H	H	H	S	H
14	S	S	H	S	S	H	S	S	S	H
13	S	S	S	S	H	H	H	H	S	H
12	H	S	S	H	S	H	H	H	S	S

Figure 20 - BlackJack Strategy Table (Monte Carlo Config 1)

Configuration 2: $\varepsilon = \frac{1}{k}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	S	S	H	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	H	S	S	S	S	S
16	S	S	S	S	H	S	S	S	S	S
15	S	S	H	S	H	S	S	S	S	S
14	S	S	H	S	H	H	S	S	S	H
13	S	H	S	S	H	S	S	S	S	H
12	H	H	S	H	H	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	H	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	H	S	S	H	S	S	S	S
17	S	S	S	S	S	H	S	H	S	S
16	S	S	H	H	S	S	S	H	S	H
15	S	S	S	S	S	S	H	S	S	S
14	S	S	S	S	S	S	S	S	S	S
13	S	S	S	H	S	S	S	S	H	S
12	S	S	S	H	S	S	S	S	S	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	H	S	H	S	S	S	S	S	S
19	S	S	S	S	H	S	S	H	S	H
18	S	S	S	H	S	S	S	S	H	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	S	S	S	S	S
15	S	H	S	S	H	S	S	S	S	S
14	S	S	S	S	H	S	S	S	S	S
13	H	S	H	S	S	S	S	S	S	S
12	H	S	H	S	S	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	H	H	S	S	H	S
19	S	S	H	H	S	S	S	S	S	S
18	S	S	H	S	S	S	H	S	S	S
17	S	H	S	S	S	S	H	S	S	S
16	H	H	H	S	H	H	H	H	S	S
15	S	S	S	S	S	S	H	S	S	S
14	H	S	H	S	S	S	H	H	S	S
13	H	S	S	S	H	H	H	S	S	S
12	H	S	S	H	H	H	S	S	S	S

Figure 21 - BlackJack Strategy Table (Monte Carlo Config 2)

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	H	S	S	H	S
19	S	S	S	S	S	S	S	S	S	S
18	H	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	H	S	S	H
16	S	S	S	S	S	S	S	S	S	S
15	S	S	S	S	S	H	H	S	S	S
14	H	S	H	S	S	S	H	S	S	S
13	S	S	S	S	S	S	H	S	S	S
12	H	S	H	S	S	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	H	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	H	S	S	S	S	S	S	S	S
16	S	S	S	S	S	S	S	H	S	H
15	S	S	S	S	S	H	S	S	S	S
14	S	H	S	S	H	H	S	S	S	S
13	S	S	S	S	H	H	S	S	S	H
12	S	H	S	S	H	S	S	H	S	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	H	S	S	S	S	S	S	S	H
18	S	S	S	H	S	S	S	S	S	S
17	H	S	S	S	S	S	S	S	S	S
16	H	S	H	S	S	S	S	S	S	H
15	S	S	S	S	S	H	S	S	H	S
14	S	S	S	S	S	S	S	H	S	S
13	H	H	S	S	S	S	H	S	S	S
12	S	S	S	S	S	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	H	S	S	S	H	S
17	H	S	H	S	H	H	S	S	S	S
16	S	S	S	S	S	S	H	S	S	S
15	H	H	S	S	S	S	S	S	S	H
14	H	H	S	S	S	S	H	S	S	S
13	S	S	S	S	S	S	H	S	S	H
12	S	S	S	S	S	S	H	S	S	H

Figure 22 - BlackJack Strategy Table (Monte Carlo Config 3)

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	H	S	S	S	S	S	S
17	S	H	S	S	S	S	S	S	S	S
16	S	H	S	H	S	S	S	S	S	S
15	S	H	S	S	H	S	S	S	S	S
14	H	S	H	H	S	H	S	S	S	S
13	S	H	S	S	S	S	S	S	S	H
12	S	S	S	H	S	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	H	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	H	S	S	S	S	S	S	S	S	S
16	H	S	S	H	S	S	S	S	S	S
15	H	S	S	S	S	S	S	S	S	S
14	S	H	S	S	S	S	S	S	S	S
13	S	S	S	S	S	S	S	S	S	S
12	H	S	H	S	S	H	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	S	S	S
15	S	S	S	S	S	S	S	S	S	S
14	S	S	S	S	S	S	H	S	S	S
13	S	S	S	S	S	S	S	S	S	S
12	S	S	S	S	S	S	S	S	S	S

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	S	S	H	S	S
15	S	S	S	S	S	H	S	S	S	S
14	S	S	S	S	S	H	H	S	S	S
13	S	S	S	S	S	S	H	S	S	H
12	S	S	H	S	S	S	S	S	S	S

Figure 23 - BlackJack Strategy Table (Monte Carlo Config 4)

The above strategy tables seen in Figures 20-23 were generated after running the algorithm twice with each respective configuration. From the figures above, it can be concluded that given the large number of episodes for which the algorithms were allowed to execute they converged onto the same strategy which is clearly visible via the strategy tables. As can be seen most strategy tables are composed of Stands in the upper half whilst the lower half tends to be composed of a mixture of Stands and Hits. This shows how an agent trained on the Monte Carlo algorithm is bound to play more cautious rather than take risks to ensure a win.

Configuration 1: $\varepsilon = \frac{1}{k} +$ exploring

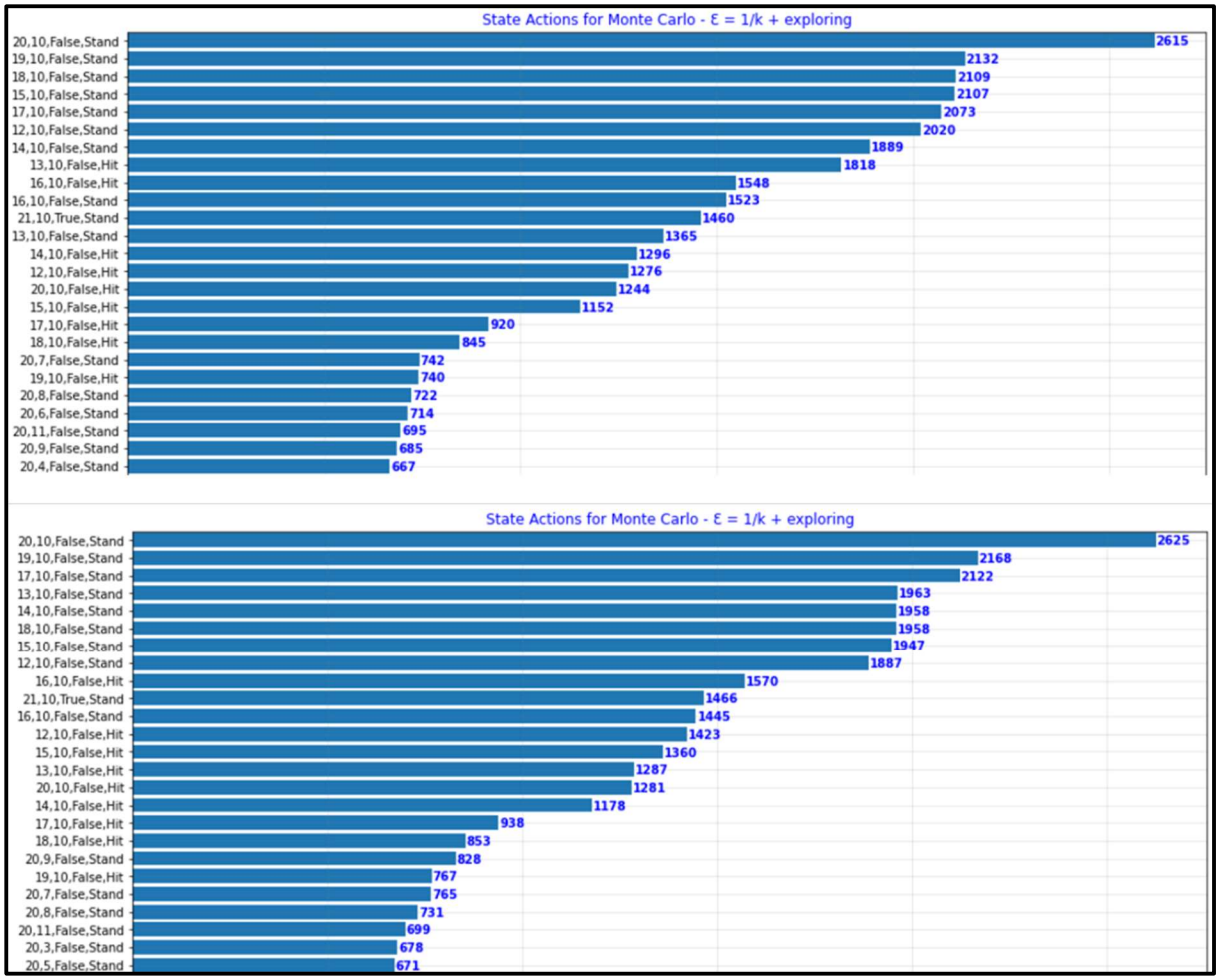


Figure 24 - Monte Carlo State Actions List Config 1 Part 1

The above bar charts (Figure 24) were generated when running the algorithm twice. This configuration's most executed action was (20, 10, False, Stand) thus being the most exploited. Similarly, it was noted that all state actions were chosen more than once, as can be seen in the graphs below (Figure 25). This occurs mainly due to the exploring functionality being enabled; hence it is more likely to choose an action which is not necessarily the most optimal. This is extremely important when it comes to reinforcement learning algorithms as by running each state more than once we are able to accurately estimate its appropriate award and thus the algorithm is able to make more informed decisions.

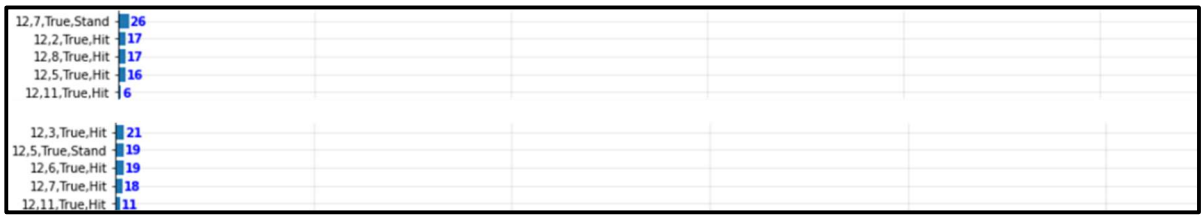


Figure 25 - Monte Carlo State Actions List Config 1 Part 2

The next three configurations (Figures 26-28), contrary to the previous one, do not utilise exploration in relation to the first action. However, with a probability of Ɛ they are still capable of choosing a random action after the first one. Considering this, it was made apparent that with respect to second configuration, the number of episodes passed is directly proportional to the exploitation rate (1 - Ɛ) given $\mathcal{E} = \frac{1}{k}$. The following two configurations, $\mathcal{E} = e^{(\frac{-k}{1000})}$ and $\mathcal{E} = e^{(\frac{-k}{10000})}$, provide a more gradual variation between exploration and exploitation given that the probability changes in accordance with the exponential implementation of **k**. Moreover, it can be noted how from the last three configurations, the least explored states are explored only a few times, in most cases only once. This is a major negative because these states were denoted as being unappealing states based on minimal executions which may not necessarily reflect the overall reward of executing such a state action pair.

Configuration 2: $\mathcal{E} = \frac{1}{k}$

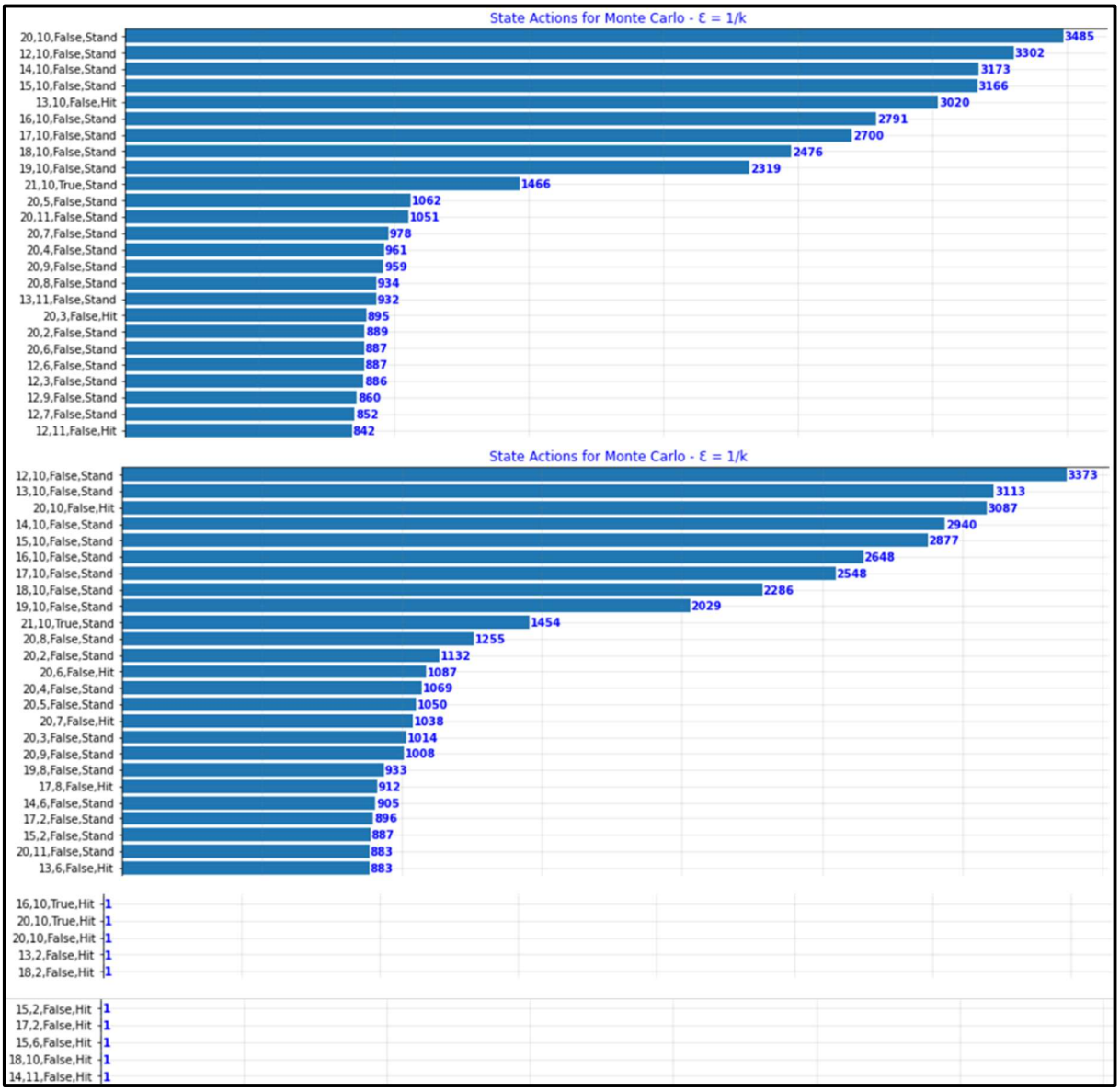


Figure 26 - Monte Carlo State Actions List Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

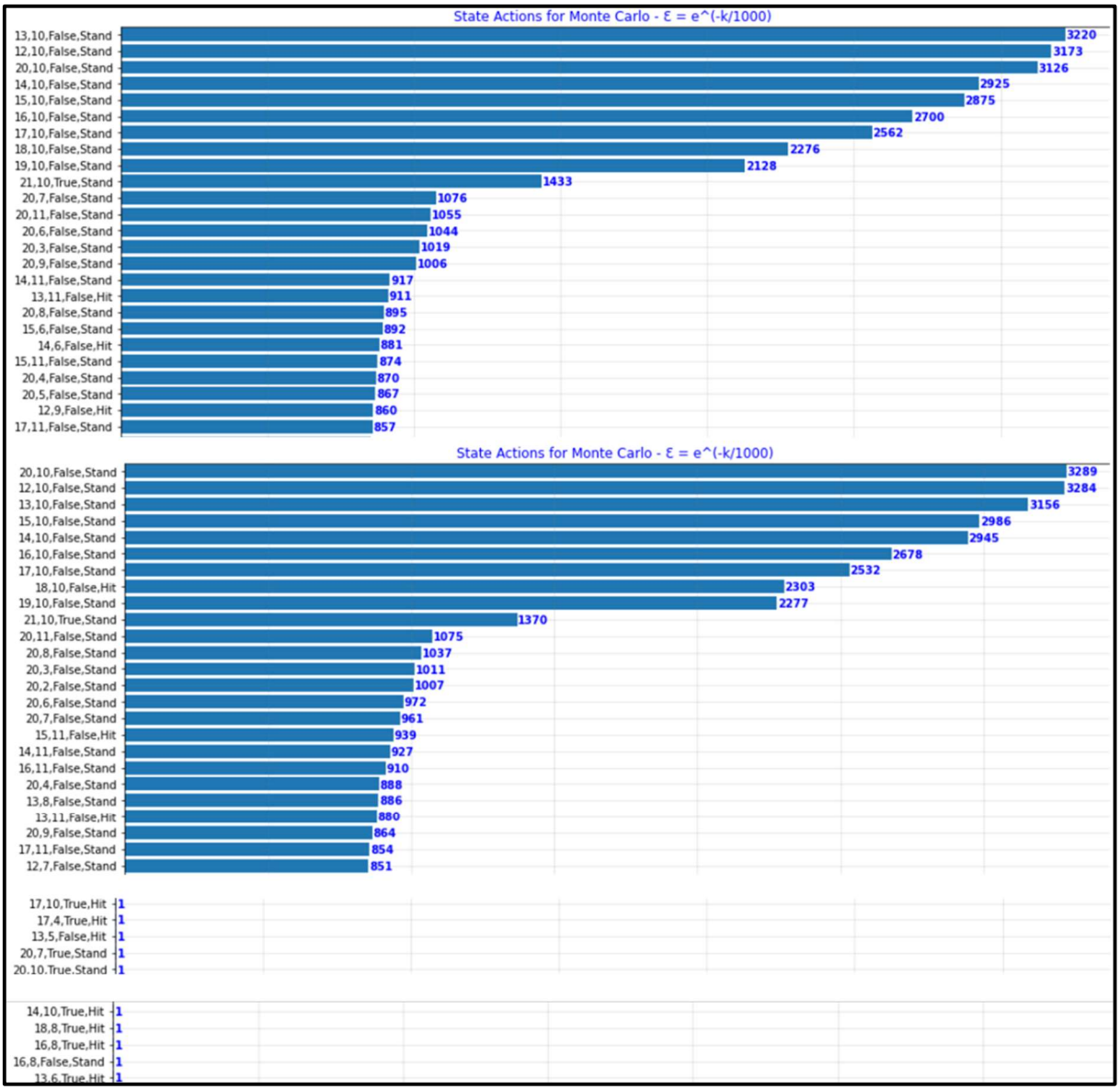


Figure 27 - Monte Carlo State Actions List Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

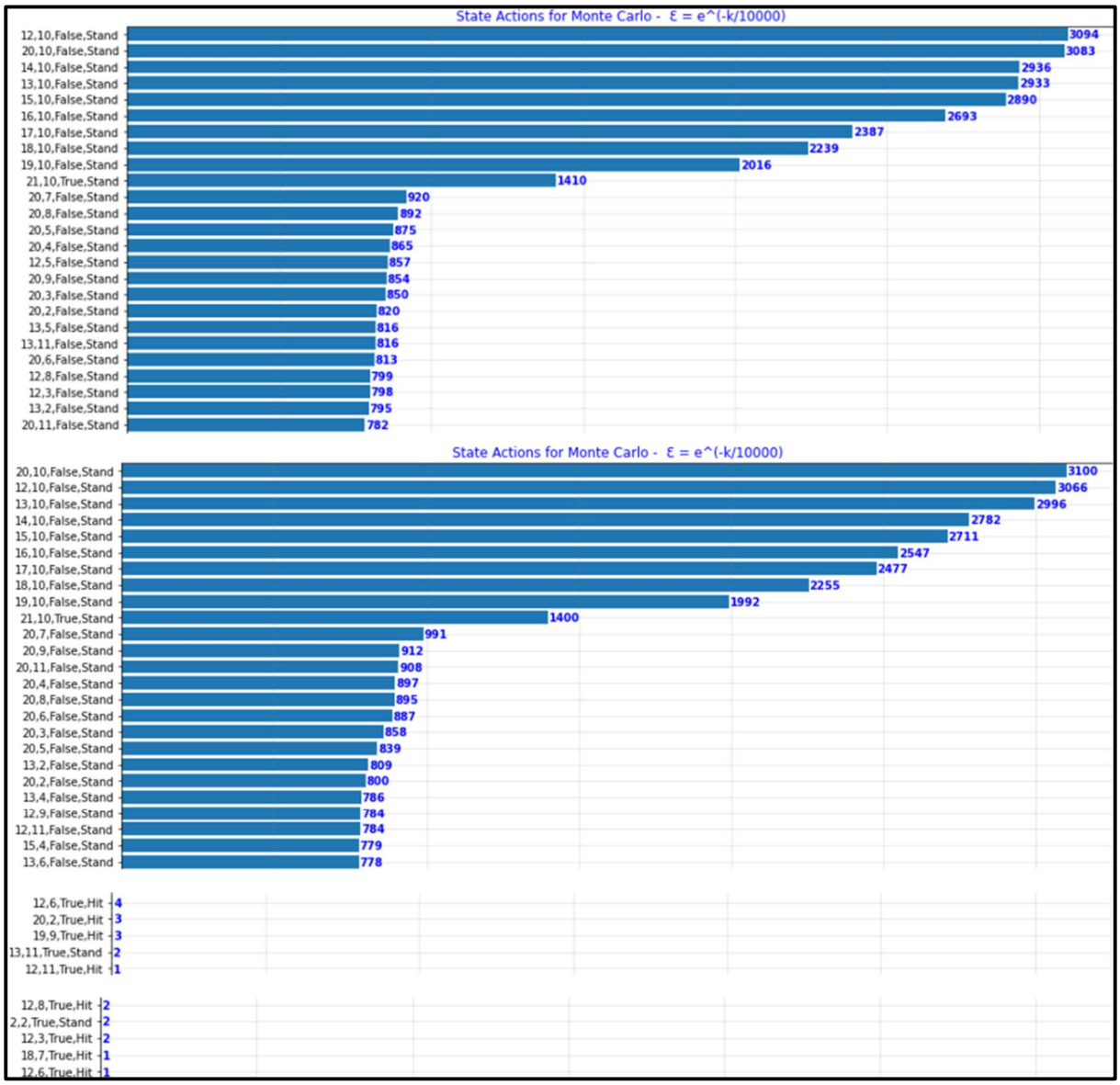


Figure 28 - Monte Carlo State Actions List Config 4

The graphs below (Figure 29-32) depict the average number of wins, draws and losses for every 1000 episodes over the course of the total 100000 episodes.

Configuration 1: $\varepsilon = \frac{1}{k} + \text{exploring}$

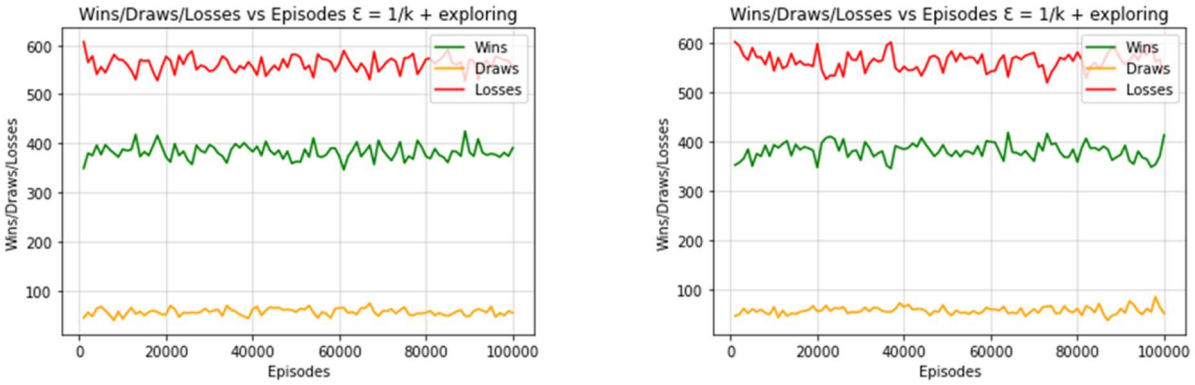


Figure 29 - Monte Carlo Wins / Draws / Losses per 1000 Episodes Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

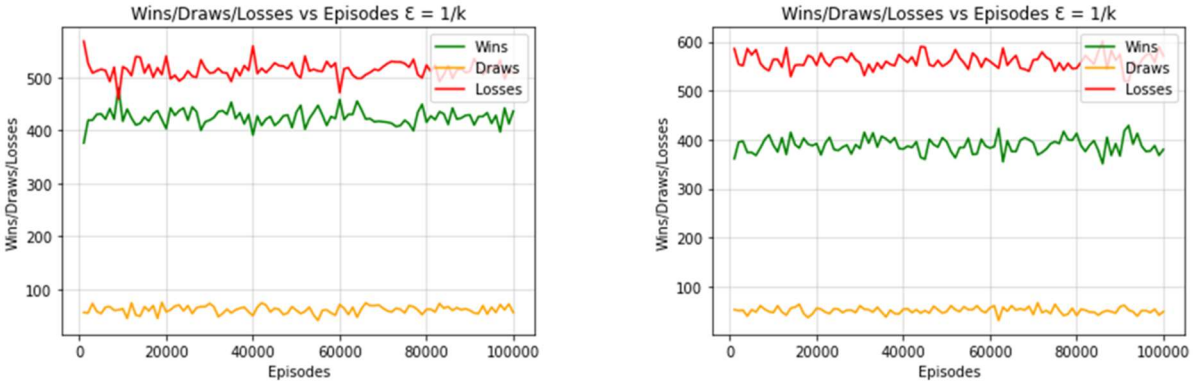


Figure 30 - Monte Carlo Wins / Draws / Losses per 1000 Episodes Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

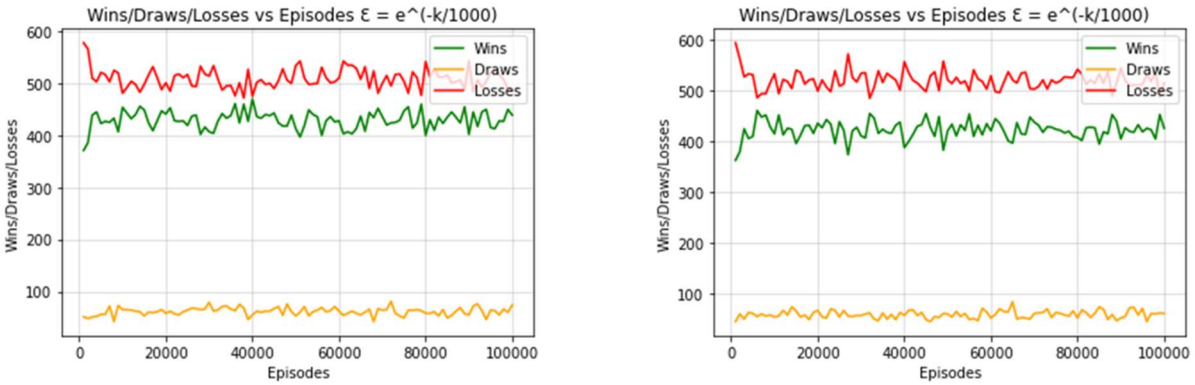


Figure 31 - Monte Carlo Wins / Draws / Losses per 1000 Episodes Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

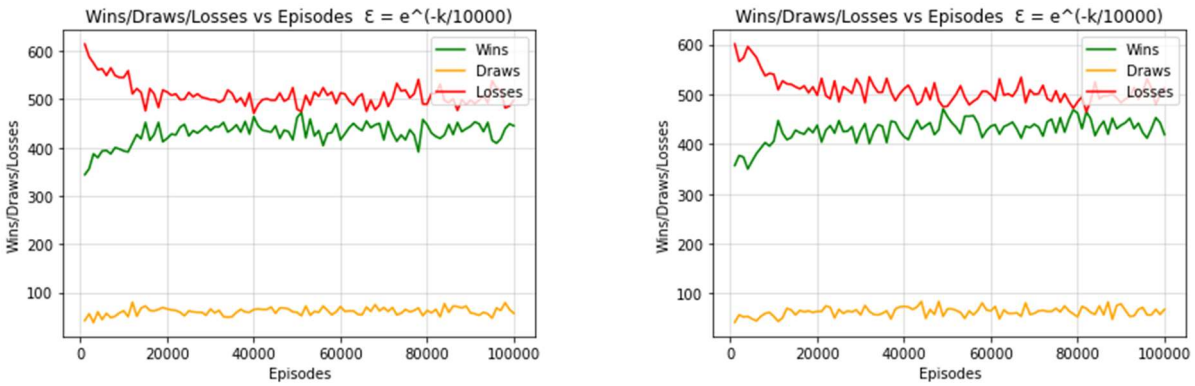


Figure 32 - Monte Carlo Wins / Draws / Losses per 1000 Episodes Config 4

The gradual change from exploration to exploitation is also evident in the above graphs (Figure 29-32). The first two algorithms provide a sharp sudden change in the win and loss rate, whereas in the last two configurations a much more gradual convergence was noted. It was observed that the fourth configuration takes the most episodes to converge. One thing of note is that there was no instance where the agent started to win more than they lost. This can be seen in the dealer advantage graph seen in Figure 34.

The following three bar graphs (Figure 33-35) are being used to compare all the four configurations implemented for the Monte Carlo Off-Policy Algorithm.

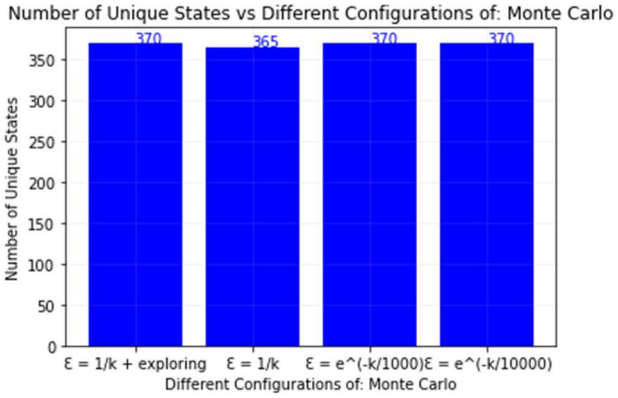


Figure 33 - Monte Carlo Unique States per Configuration

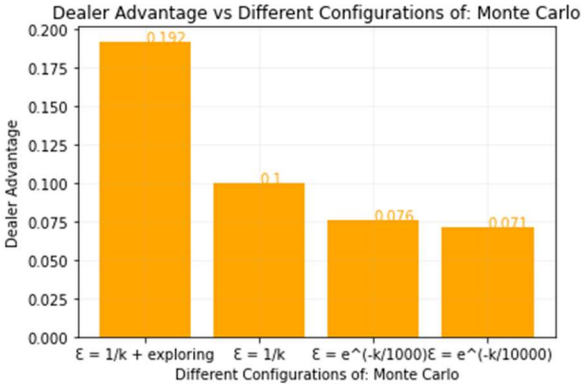


Figure 34 - Monte Carlo Dealer Advantage per Configuration

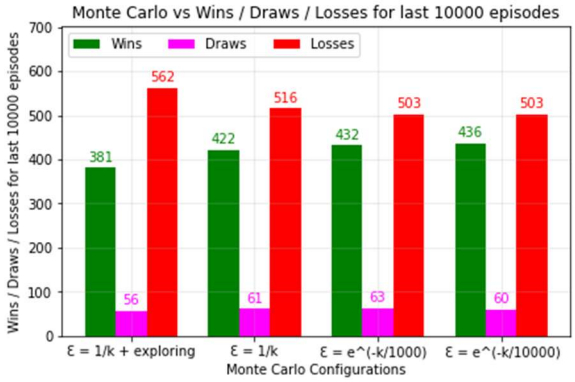


Figure 35 - Monte Carlo Wins / Draws / Losses per Configuration

Finally, from the above graphs it was noted that the configuration with the highest win rate was the $\varepsilon = e^{(\frac{-k}{10000})}$ configuration, followed closely by the $\varepsilon = e^{(\frac{-k}{1000})}$ configuration. The configuration with the highest dealer advantage was the configuration $\varepsilon = \frac{1}{k} + \boldsymbol{exploring}$. Thus, it can be concluded that the fourth configuration is the best from those implemented, providing the most gradual convergence. Notice the lowest dealer advantage of Monte Carlo is 0.071, as seen in the above results. Similarly, observe the current number of wins experienced a gradual increase over the configurations, whilst the losses experienced a reduction. Something that was also noted was that in Figure 33, configuration two failed to explore every state action. This is most likely because the configuration transitions into its greedy counterpart before it is able to try every state action pair once. Contrarily, the other configurations transition more smoothly towards an exploitative variant which allows them to try every possible state action pair at least once.

SARSA On-Policy Algorithm

The following graphs and charts were generated when computing the SARSA On-Policy Algorithm:

Configuration 1: ε = 0.1

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	H	H	S	S	S	H	H	H	H	H
17	H	H	H	S	H	S	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	H	S
18	H	S	H	S	S	H	S	H	H	H
17	H	H	H	H	H	H	S	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	S	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	S	H	S	H	S
15	S	S	S	S	S	H	S	S	H	S
14	S	S	S	S	S	S	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	S	H	S	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	S	H	S
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	S
13	S	H	S	S	S	H	H	H	H	S
12	S	S	S	S	S	H	H	H	H	H

Figure 36 - Strategy Table (SARSA Config 1)

Configuration 2: $\varepsilon = \frac{1}{k}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	H	H	S	H	S	S	S	S	S
19	H	H	H	H	H	H	S	H	H	H
18	H	H	S	H	H	S	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	H	S	S
17	H	S	S	S	S	S	S	H	H	H
16	S	S	S	S	S	H	H	H	H	S
15	H	S	H	H	H	H	H	H	S	S
14	S	S	H	S	S	S	S	H	H	S
13	H	H	S	S	S	H	H	H	H	S
12	H	S	H	H	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	S	H	S	H	H	S	S	S	S
19	H	S	H	H	H	S	H	S	H	S
18	H	H	H	S	H	H	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	H	S	S	H	H	S	H	H
15	S	S	S	S	S	H	H	S	S	H
14	H	H	S	S	H	H	S	S	H	H
13	H	H	H	S	S	H	H	H	H	S
12	S	S	H	S	S	H	S	H	H	S

Figure 37 - Strategy Table (SARSA Config 2)

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	H	H	S	S	S	S	S	S	S
19	H	S	H	H	H	S	S	S	S	S
18	H	S	H	H	S	S	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	H	S	S
17	S	S	S	S	S	S	S	S	S	S
16	H	H	S	S	S	S	H	H	S	H
15	S	S	S	H	H	S	H	H	S	S
14	H	S	H	S	S	H	H	H	H	S
13	H	S	H	S	S	H	S	H	H	H
12	H	H	S	S	H	S	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	H	S	S	H	S	S	H	S	H
19	H	S	S	S	H	H	S	S	H	S
18	H	H	H	H	S	H	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	S	H	H	H	H	H	H	H	H
15	H	H	H	S	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	H	H	S	S	S	H	S	S	S
16	S	S	S	H	S	H	S	H	S	H
15	S	S	S	S	S	S	H	S	H	H
14	S	S	H	H	S	H	H	H	H	S
13	S	S	H	S	S	H	S	H	H	S
12	H	S	S	H	S	H	H	H	H	H

Figure 38 - Strategy Table (SARSA Config 3)

Configuration 4: $\mathcal{E} = e^{(\frac{-k}{10000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	H	S	S	H	S	S	S	H	H	S
17	H	H	H	H	H	H	H	H	H	H
16	H	S	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	S	H	S	S	S
15	S	S	S	S	S	S	H	H	H	S
14	S	S	S	S	S	H	H	H	H	S
13	S	S	S	S	S	H	H	H	H	H
12	S	S	S	H	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	H	S	S	S	H	S	S	S	S	S
18	H	H	S	S	S	S	S	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	H
16	S	S	S	S	S	S	S	S	H	S
15	S	H	S	S	S	H	H	S	S	H
14	S	S	S	S	S	H	H	S	H	S
13	S	H	S	S	S	H	H	H	H	H
12	S	S	H	S	S	H	H	H	H	H

Figure 39 - Strategy Table (SARSA Config 4)

The above strategy tables seen in Figures 36-39 were generated after running the algorithm twice with each respective configuration. Similar to the strategy tables generated by Monte Carlo, the SARSA algorithm produces tables mostly composed of Stand actions and minimal Hits denoting that an agent trained on the SARSA algorithm is bound to play more cautious rather than take risks to ensure a win. This further emphasises the point that given a significant number of episodes, both Monte Carlo and SARSA will eventually adopt a greedy approach since the Ɛ value is based on the episode value.

Most of the Monte Carlo's graphs are similar to those of SARSA excluding the first configuration as both functions given enough episodes will converge and become greedy functions meaning that the unappealing states are scarcely visited. However, one thing of note is that with respect to $\varepsilon = e^{(\frac{-k}{10000})}$, there was a minor increase in exploration as the unappealing states were explored more than once. This similarity with the Monte Carlo figures can be seen in Figures 25-28 and Figures 40-43.

Configuration 1: $\varepsilon = 0.1$

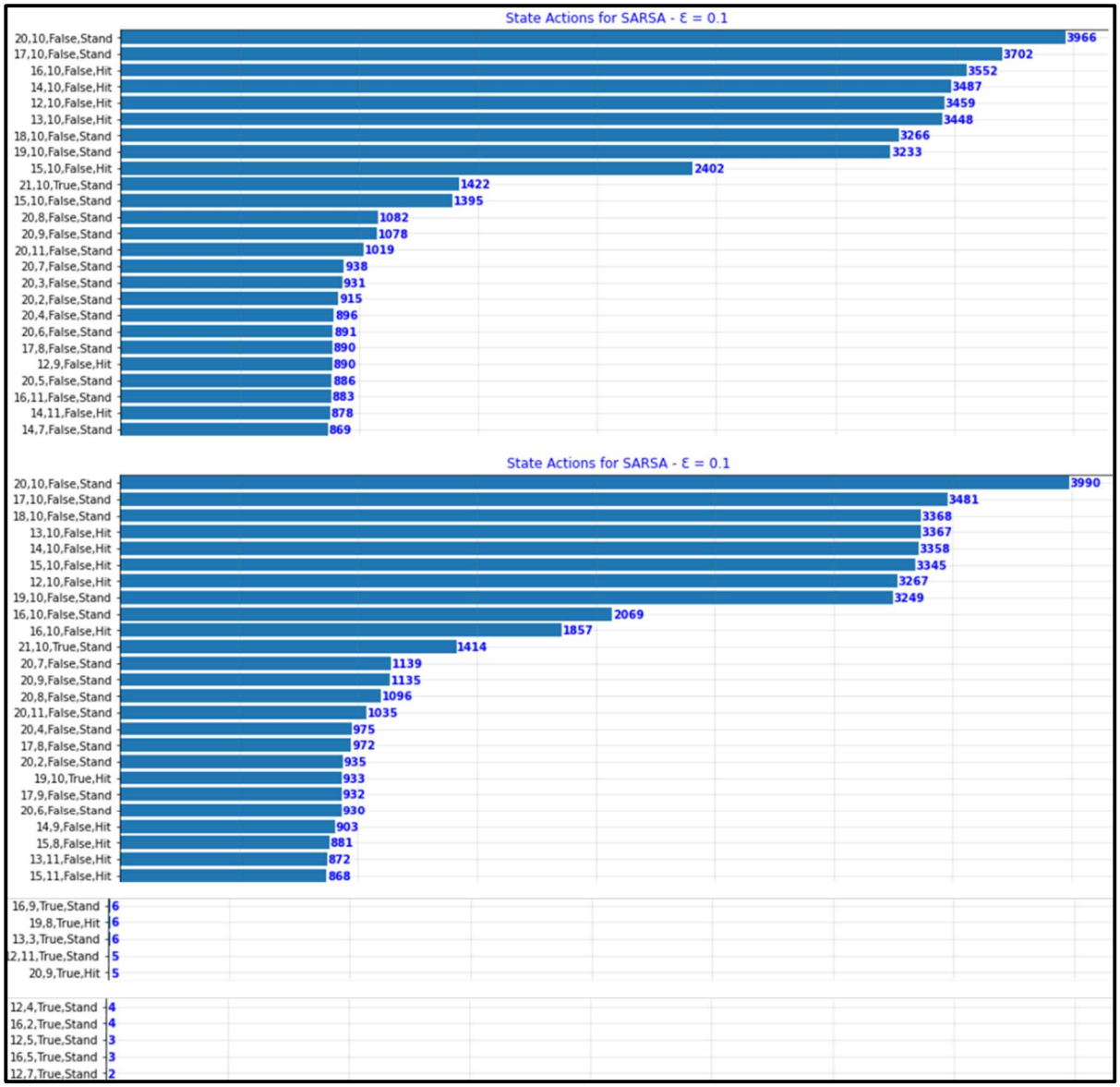


Figure 40 – SARSA State Actions List Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

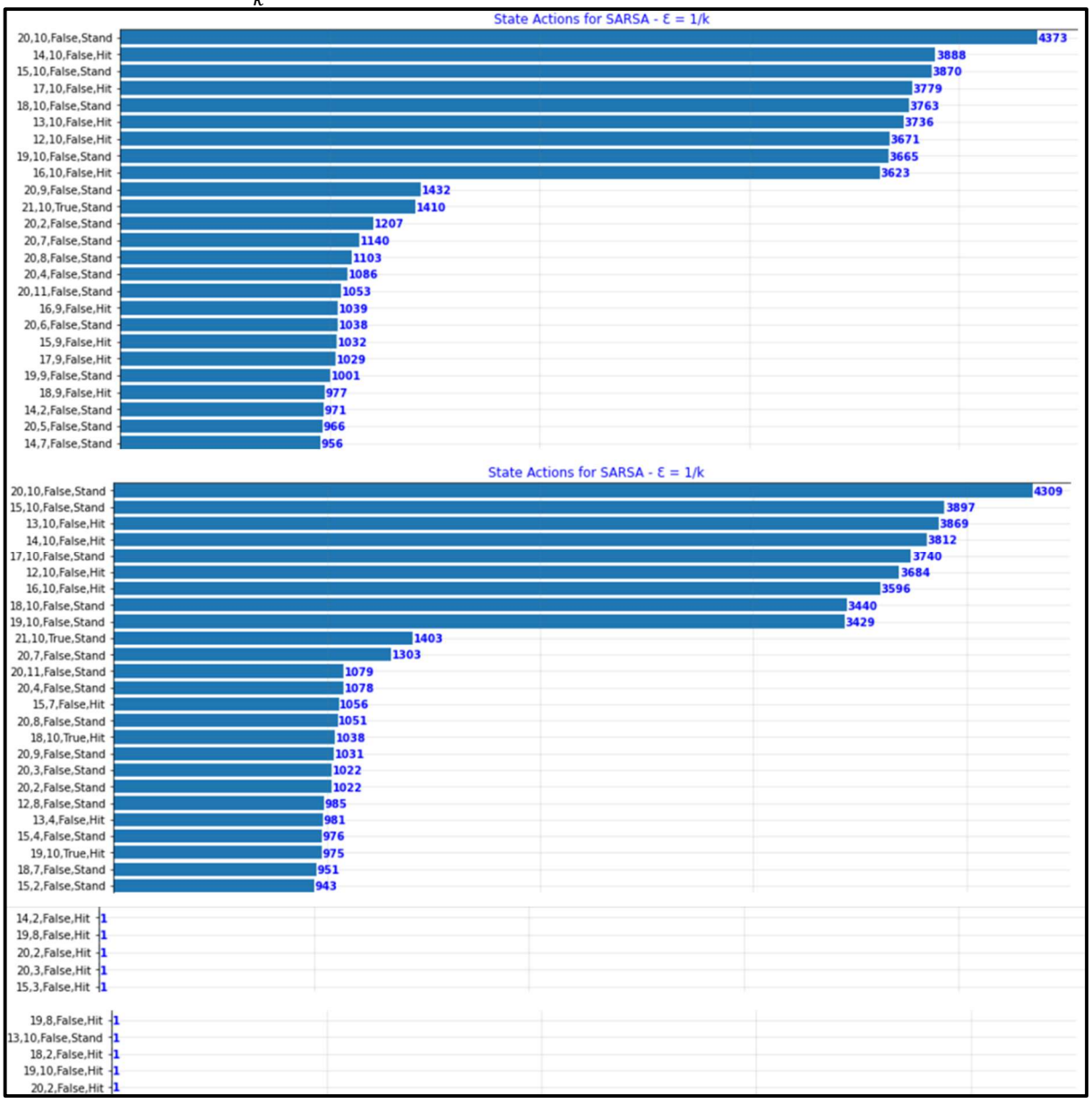


Figure 41 – SARSA State Actions List Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

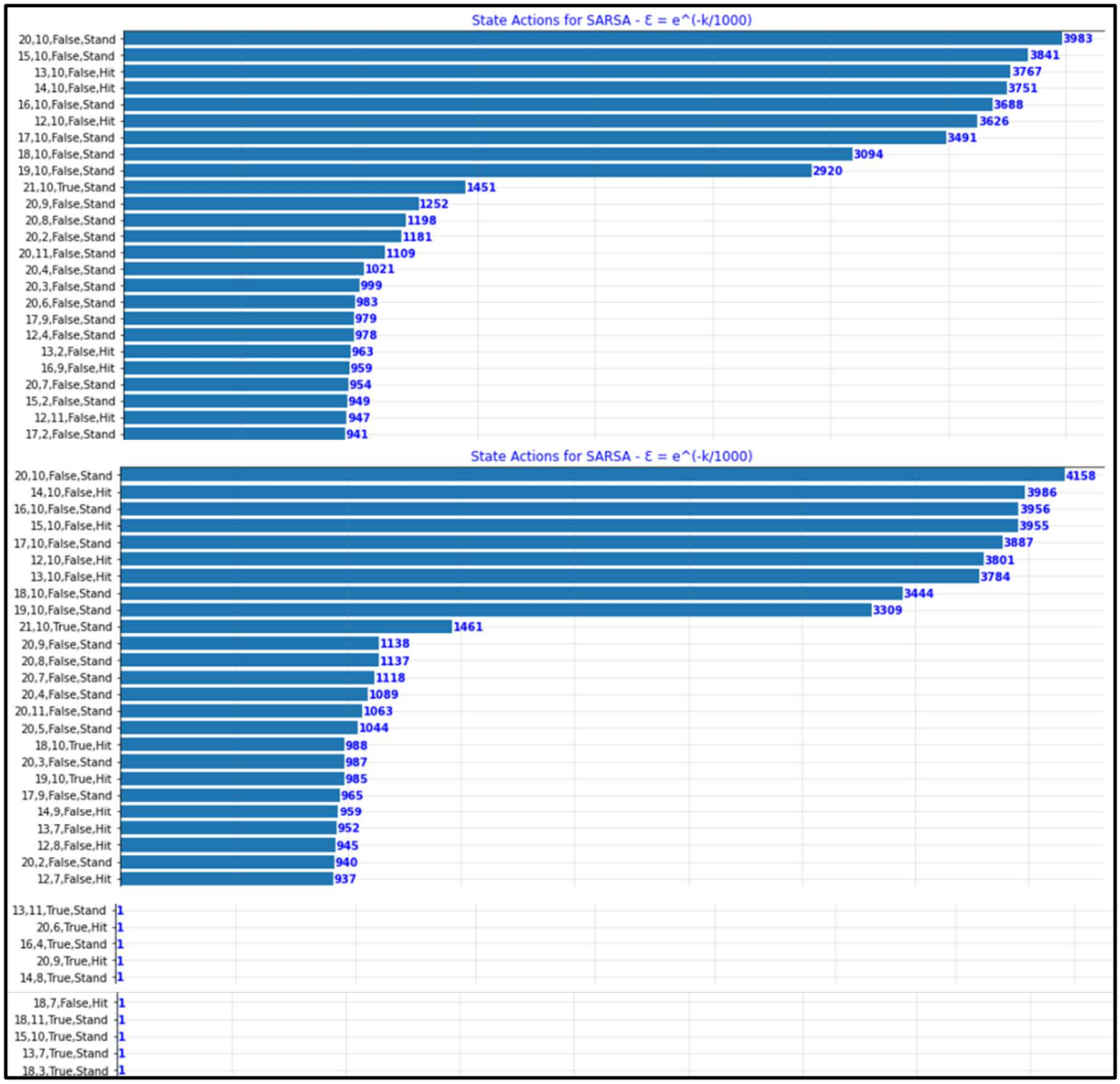


Figure 42 – SARSA State Actions List Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

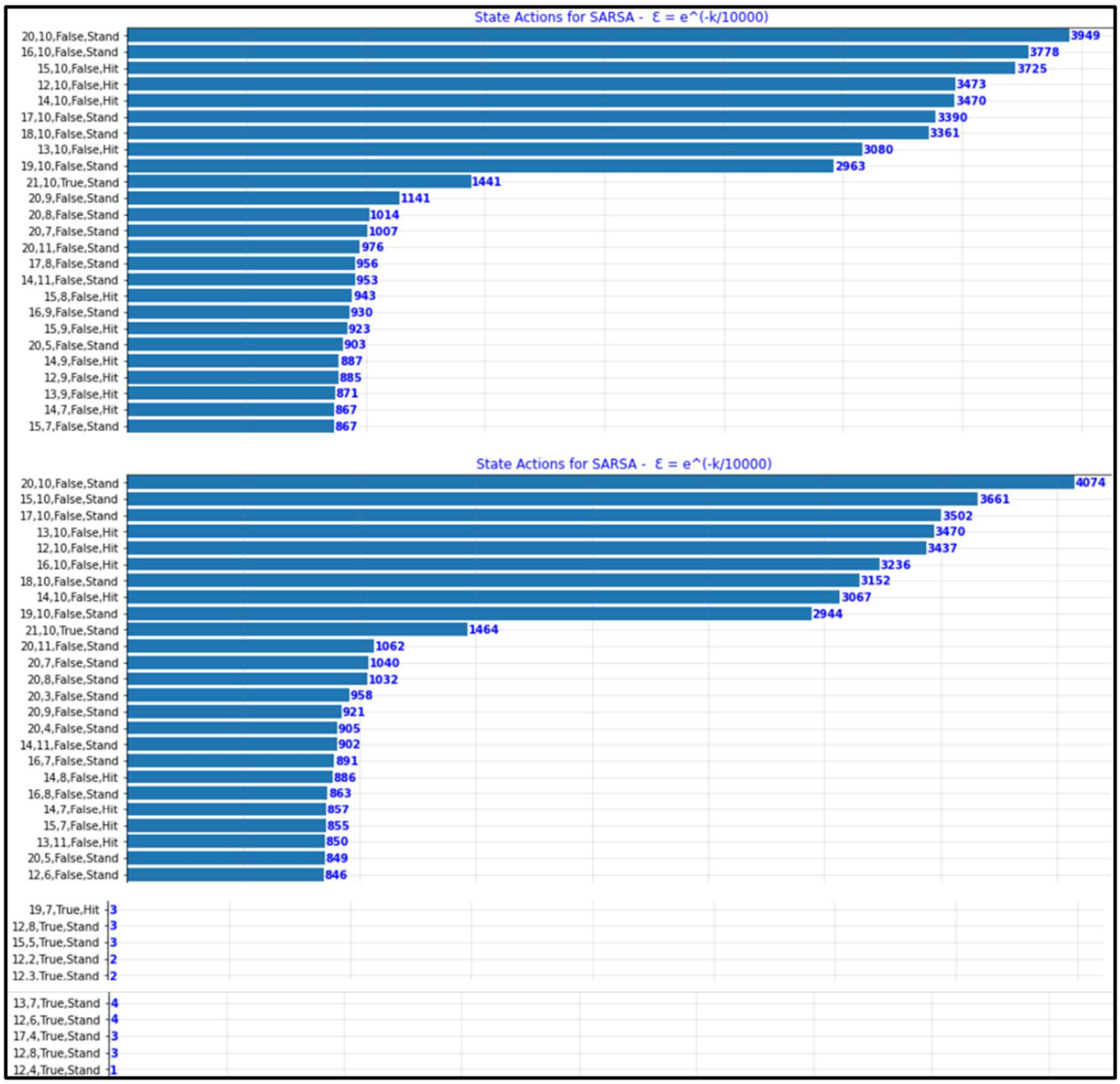


Figure 43 – SARSA State Actions List Config 4

The graphs below depict the average number of wins, draws and losses for every 1000 episodes over the course of the total 100000 episodes.

Configuration 1: $\varepsilon = 0.1$

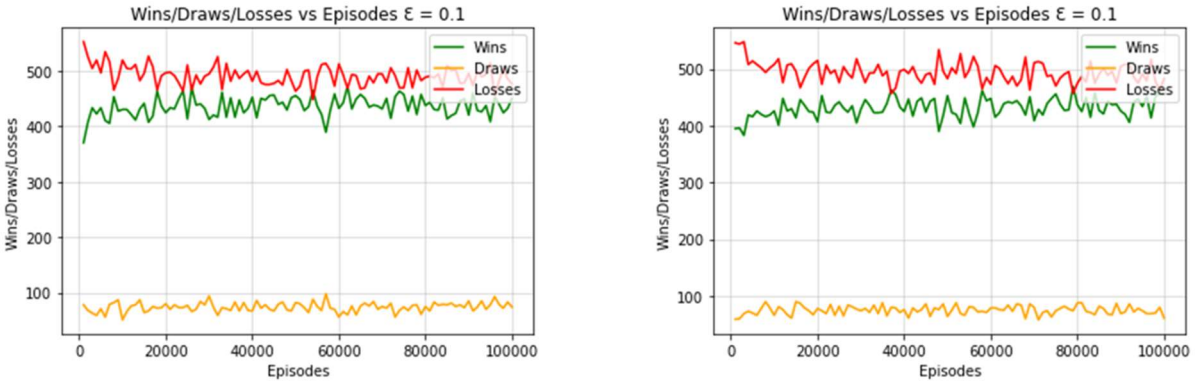


Figure 44 - SARSA Wins / Draws / Losses per 1000 Episodes Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

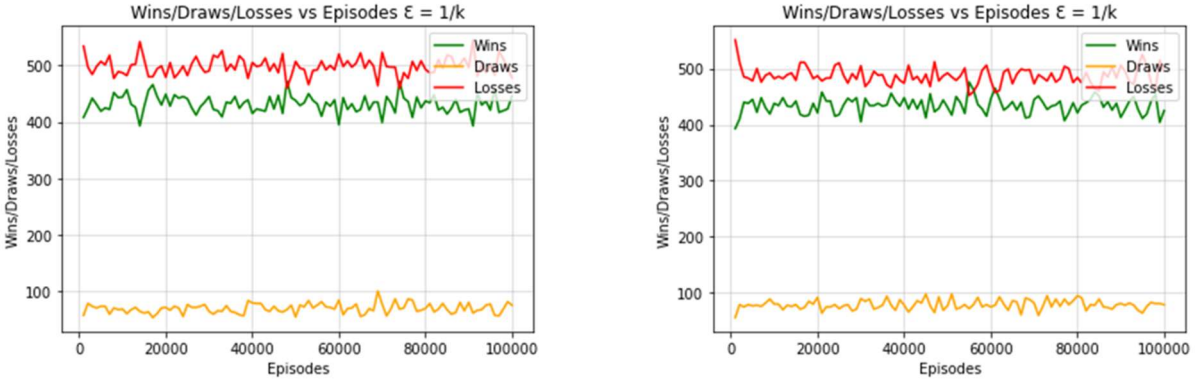


Figure 45 - SARSA Wins / Draws / Losses per 1000 Episodes Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

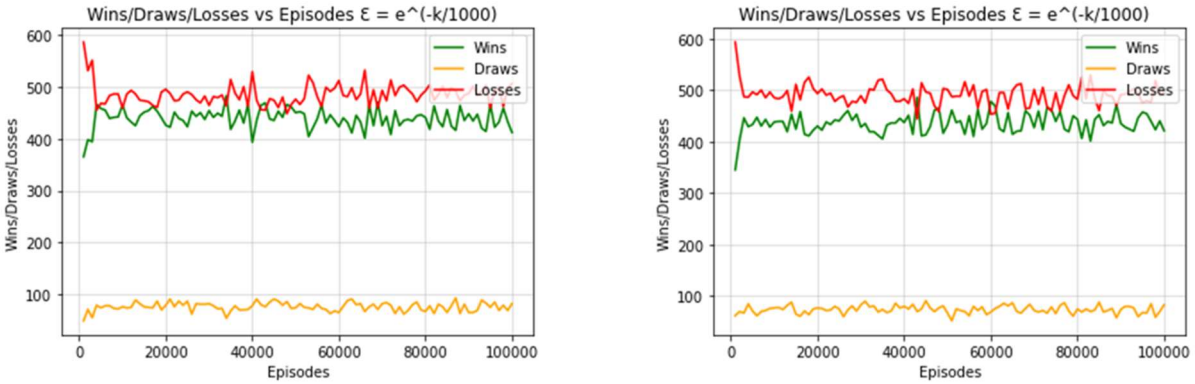


Figure 46 - SARSA Wins / Draws / Losses per 1000 Episodes Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

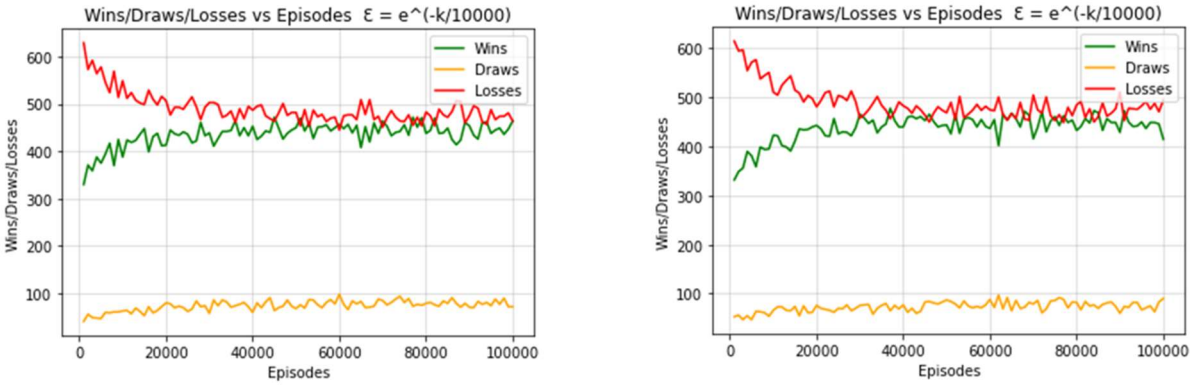


Figure 47 - SARSA Wins / Draws / Losses per 1000 Episodes Config 4

The claims made for the Monte Carlo algorithm similarly apply to the SARSA line graphs displayed above (Figure 44-47) where the last configuration is noted to have the slowest convergence due to the exploration rate gradually decreasing whereas the other configurations seem to converge suddenly due to the sharp switch to a high exploitation rate. Furthermore, when using SARSA, unlike Monte Carlo, in some configurations, notably 3 and 4, in some cases the agent wins more games than they lose. This can be seen in some instances in the above line graphs where the green line is higher than the red line.

The following three bar graphs are being used to compare all the four configurations implemented for the SARSA Off-Policy Algorithm.

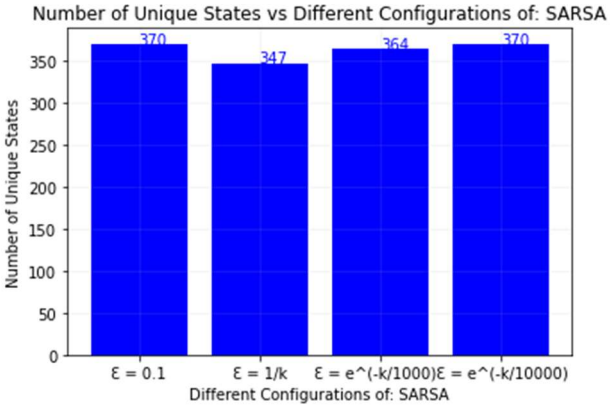


Figure 48 - SARSA Unique States per Configuration

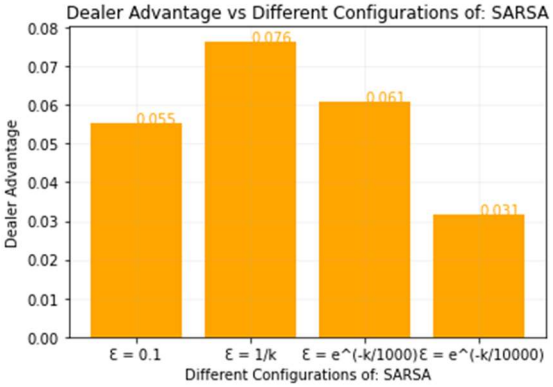


Figure 49 - SARSA Dealer Advantage per Configuration

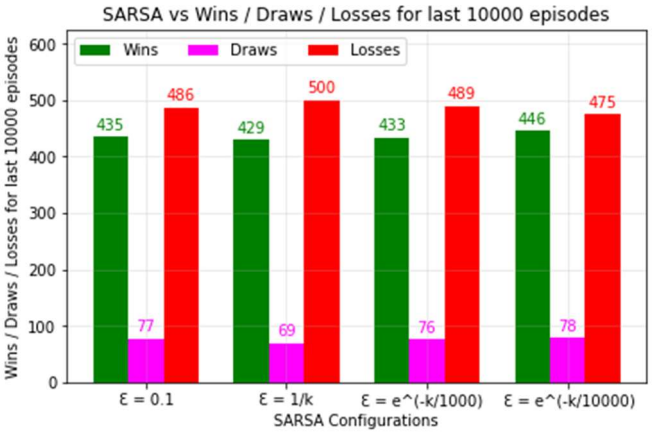


Figure 50 - SARSA Wins / Draws / Losses per Configuration

Finally, from the above graphs it was noted that although configuration four boasted the highest win rate the discrepancy between all four configurations was minimal. However, the dealer advantages shown in Figure 49 make this discrepancy more apart as configuration four has the least advantage with a value of 0.031 whereas the remaining configuration have a value larger or equal to 0.055. Thus, it can be concluded that the fourth configuration is the best from those implemented, providing the most gradual convergence. Another thing of note is how only two of the four configurations managed to fully explore the entire state action set, this can be seen in Figure 48. This is most likely because the algorithms transition into greedy variants prior to visiting every state action pair once. Contrarily, the configuration one stays consistent throughout as it utilises a constant $\varepsilon = 0.1$ whereas configuration four transitions smoothly towards its greedy counterpart giving it sufficient time to visit all the states at least once.

SARSAMAX (Q-Learning) Off-Policy Algorithm

The following graphs and charts were generated when computing the SARSAMAX (Q-Learning) Off-Policy Algorithm:

Configuration 1: $\varepsilon = 0.1$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	H	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	H	S	H	S	H	S	S	H	H	S
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	S	S
15	S	S	S	S	S	H	S	S	S	S
14	H	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	S
12	H	S	H	S	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	H	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	H	H	S	S	S	H	H	S
17	S	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	S
15	S	S	H	S	S	S	H	H	H	S
14	S	H	S	S	S	H	H	H	H	H
13	S	H	H	S	S	H	H	H	H	H
12	H	H	S	H	S	H	H	H	H	H

Figure 51 - Strategy Table (SARSAMAX Config 1)

Configuration 2: $\varepsilon = \frac{1}{k}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	H	S	S	S	H	H	S	S	S
19	H	H	H	H	S	S	H	S	H	H
18	H	H	S	S	H	S	H	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	H	S	S
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	H	S	H	H	S	S	H	H
14	H	H	S	S	S	H	H	S	H	H
13	S	H	S	S	H	H	S	H	H	H
12	H	H	H	S	H	H	S	S	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	H	S	H	H	S	H	S	S	S
19	S	H	S	H	H	S	S	S	H	S
18	H	H	H	H	H	S	S	H	H	H
17	H	H	H	S	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	S	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	H	S	H	S	S	S	S
16	S	S	H	S	H	S	H	S	H	H
15	H	S	S	S	S	H	H	H	H	H
14	S	S	H	S	S	S	H	S	H	H
13	S	H	H	H	S	H	H	H	H	H
12	H	S	H	H	H	S	H	H	H	H

Figure 52 - Strategy Table (SARSAMAX Config 2)

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	H	S	H	S	S	H	S	S	S	S
19	H	S	S	S	H	S	H	S	H	S
18	H	H	H	H	H	S	S	H	H	H
17	H	H	H	H	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	S
16	S	S	S	S	H	H	S	H	H	H
15	H	H	H	H	S	S	S	H	H	H
14	S	H	H	S	S	S	H	H	H	H
13	H	S	S	S	H	H	H	H	S	S
12	H	S	S	H	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	H	S	H	H	H	S	H
19	S	S	H	H	S	S	H	S	H	S
18	S	S	H	H	H	S	H	H	H	H
17	H	H	H	H	S	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	S
16	S	S	S	S	S	S	H	S	H	S
15	S	H	S	S	S	H	S	H	S	S
14	S	S	H	H	H	H	H	H	H	H
13	H	H	H	S	S	H	S	H	H	H
12	H	S	S	H	S	H	H	H	H	H

Figure 52 - Strategy Table (SARSAMAX Config 3)

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	H	H	H	S	S	H	H	H	H	S
17	H	H	H	S	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	H	S	S	S	S	S
19	S	S	H	S	S	S	S	S	S	S
18	H	H	S	H	S	S	S	H	H	S
17	H	H	H	H	S	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	S
15	S	S	S	S	S	H	H	H	H	S
14	S	S	S	S	S	H	H	H	H	H
13	S	H	S	S	S	H	H	H	H	S
12	H	H	H	S	S	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	S
15	S	S	S	S	S	S	H	H	S	H
14	H	S	S	S	S	S	H	H	H	S
13	S	S	S	S	S	H	H	H	H	H
12	S	H	S	S	S	H	H	H	H	H

Figure 53 - Strategy Table (SARSAMAX Config 4)

The above strategy tables seen in Figures 50-53 were generated after running the algorithm twice with each respective configuration. A difference was noted in these strategy tables compared to those shown in Monte Carlo and SARSA. However, when the Ace is considered as a 1, the policy opts to play riskier moves than those in Monte Carlo and SARSA. The safer policy is still upheld when the Ace is an 11.

Since SARSAMAX is a variation of SARSA, their graphs are quite similar. This can be seen when comparing Figures 40-43 to Figures 54-57. Due to SARSAMAX expanding upon SARSA the former function also converges to a greedy function given an adequate number of episodes as well as an epsilon based on said number of episodes.

Configuration 1: $\varepsilon = 0.1$

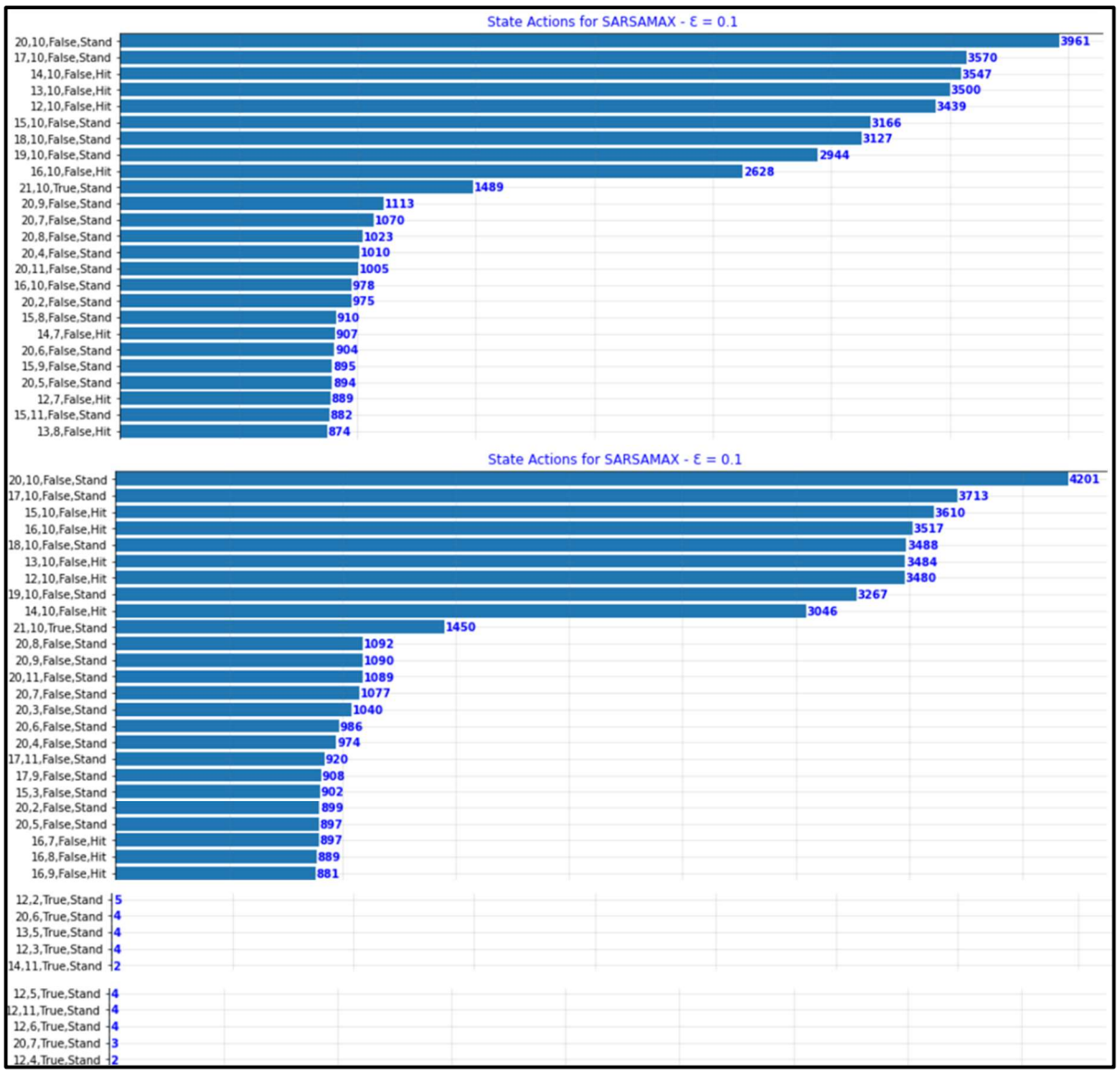


Figure 54 - SARSAMAX State Actions List Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

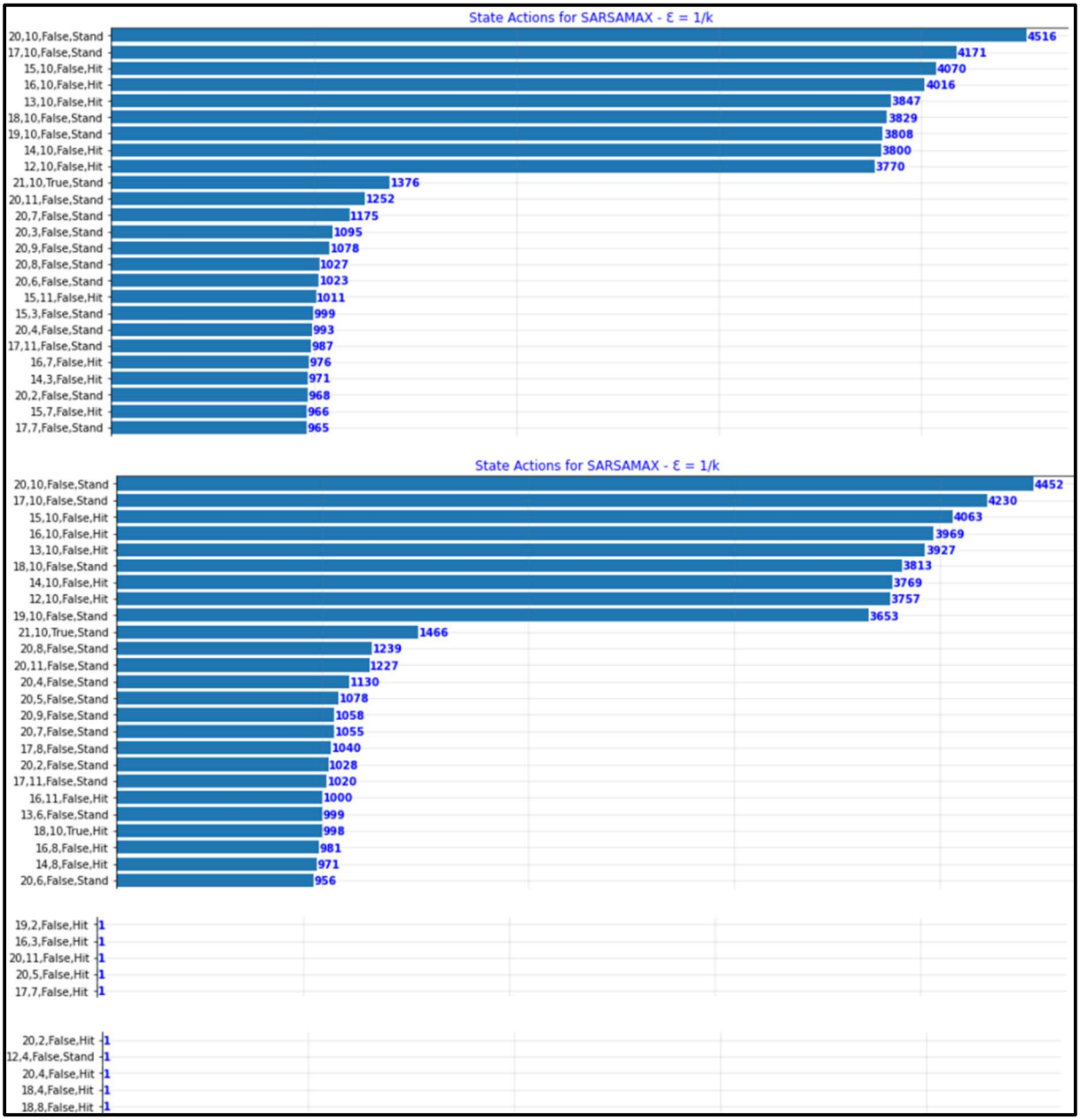


Figure 55 - SARSAMAX State Actions List Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

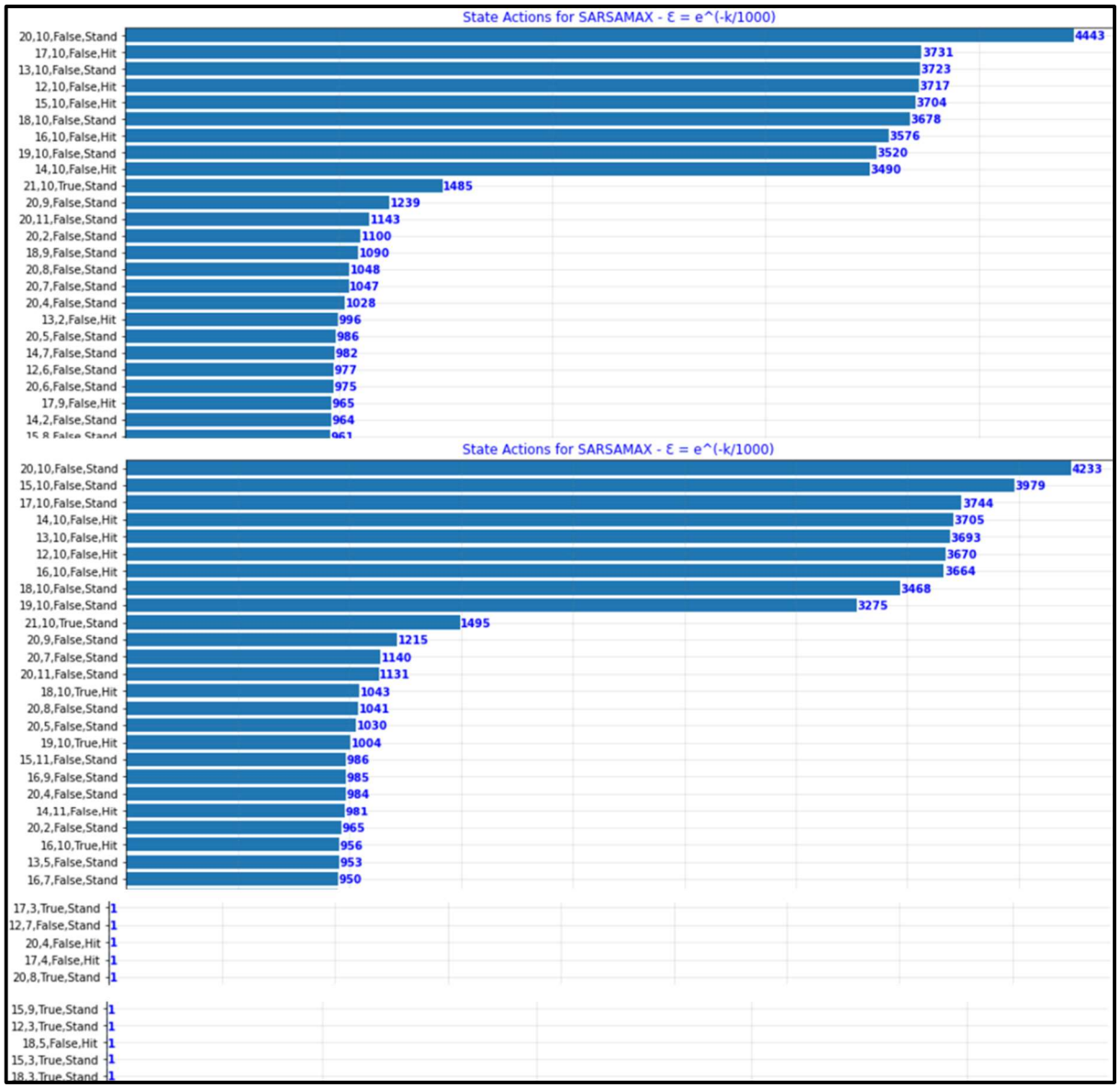


Figure 56 - SARSAMAX State Actions List Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

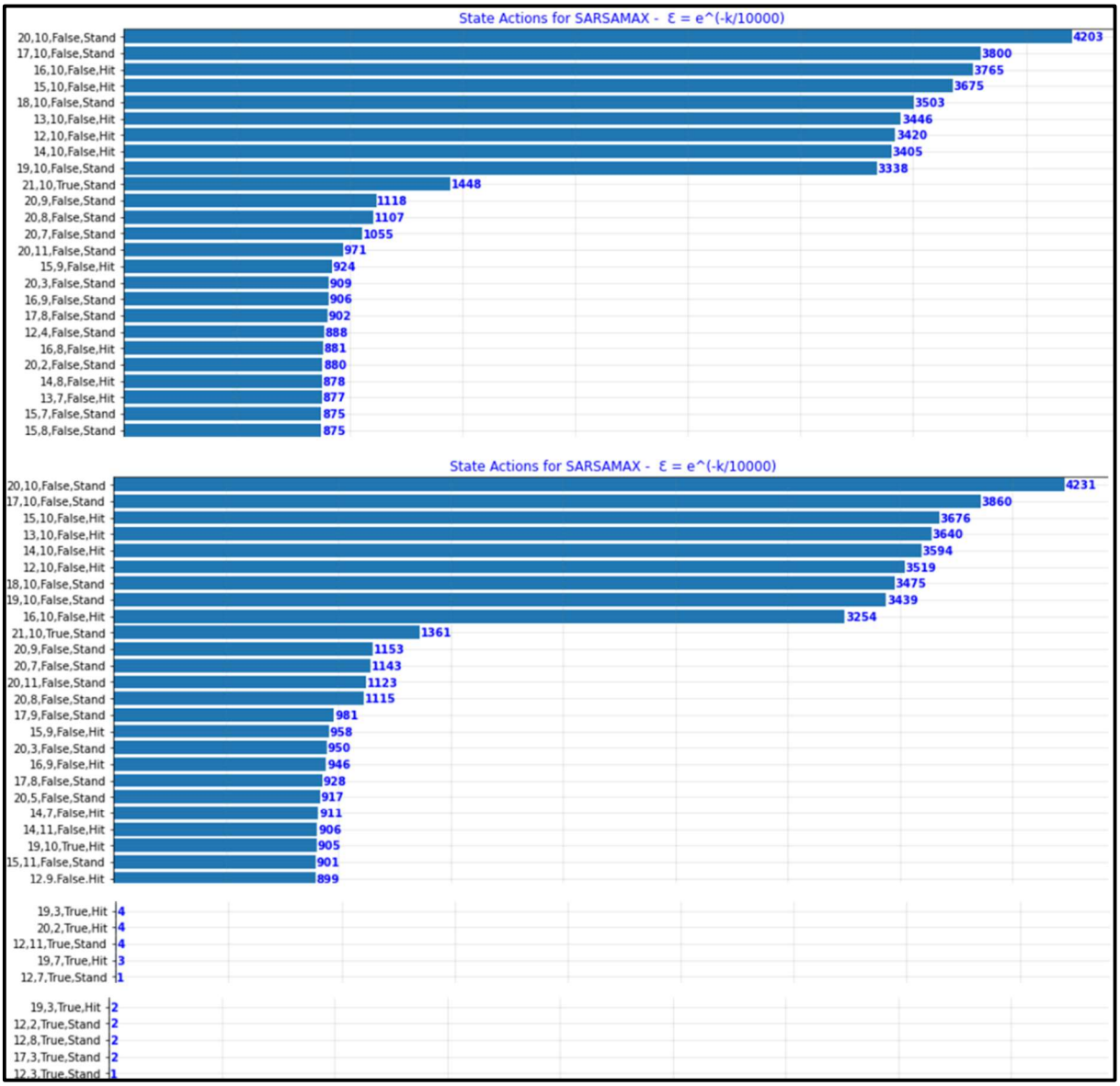


Figure 57 - SARSAMAX State Actions List Config 4

The graphs below depict the average number of wins, draws and losses for every 1000 episodes over the course of the total 100000 episodes.

Configuration 1: $\varepsilon = 0.1$

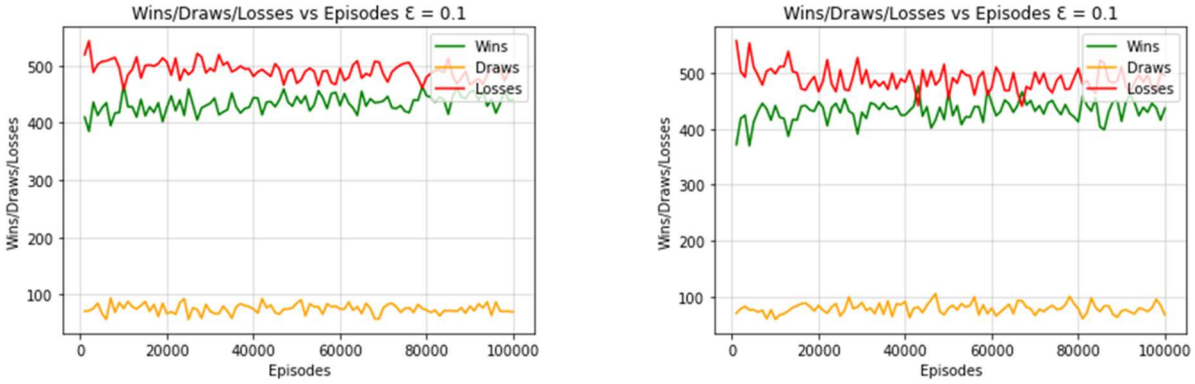


Figure 58 - SARSAMAX Wins / Draws / Losses per 1000 Episodes Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

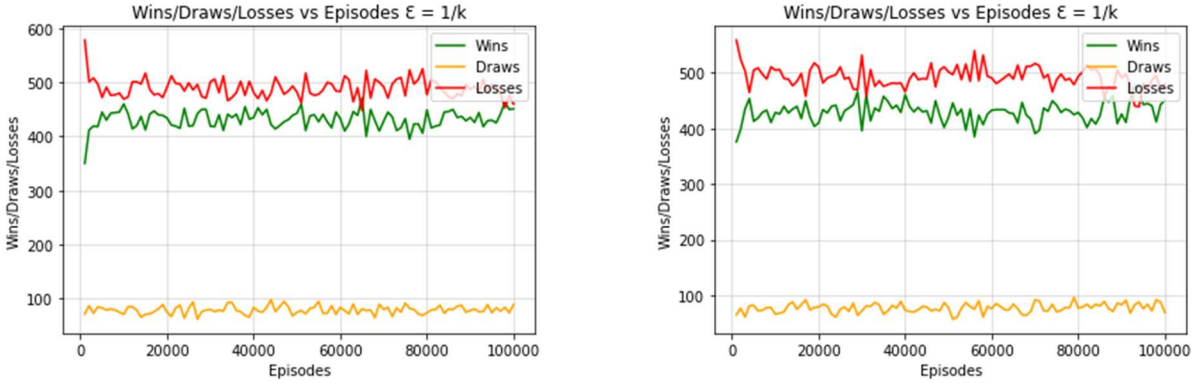


Figure 59 - SARSAMAX Wins / Draws / Losses per 1000 Episodes Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

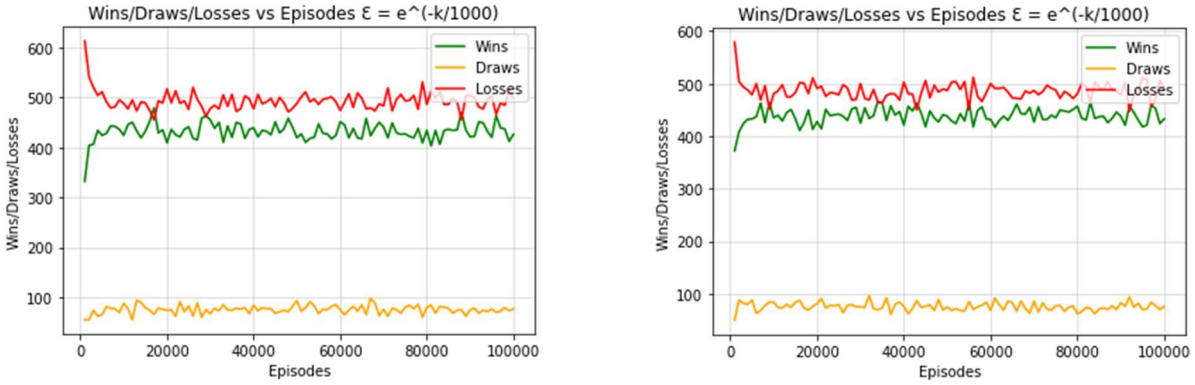


Figure 60 - SARSAMAX Wins / Draws / Losses per 1000 Episodes Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

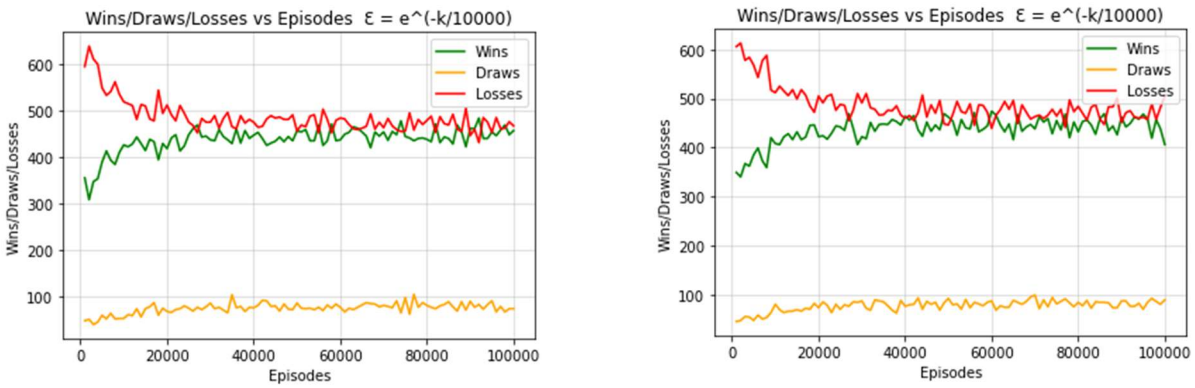


Figure 61 - SARSAMAX Wins / Draws / Losses per 1000 Episodes Config 4

The claims made for the SARSA algorithm similarly apply to SARSAMAX line graphs displayed above (Figures 58-61) where the last configuration is noted to have the slowest convergence due to the exploration rate gradually decreasing whereas the other configurations seem to converge suddenly due to the sharp switch to a high exploitation rate. Moreover, unlike SARSA, all configurations had at least one occurrence where they won more rounds than they lost. This can be seen especially in the fourth configuration.

The following three bar graphs are being used to compare all the four configurations implemented for the SARSAMAX (Q-Learning) On-Policy Algorithm.

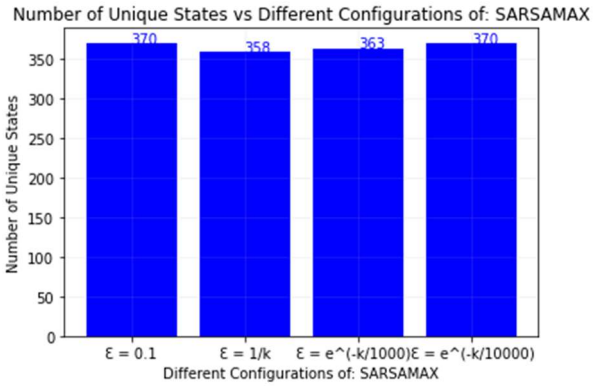


Figure 62 - SARSAMAX Unique States per Configuration

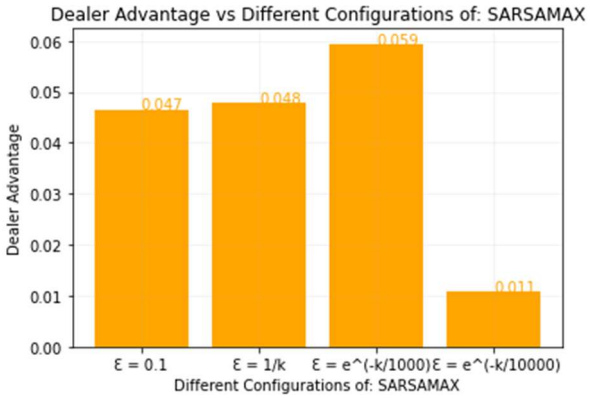


Figure 63 - SARSAMAX Dealer Advantage per Configuration

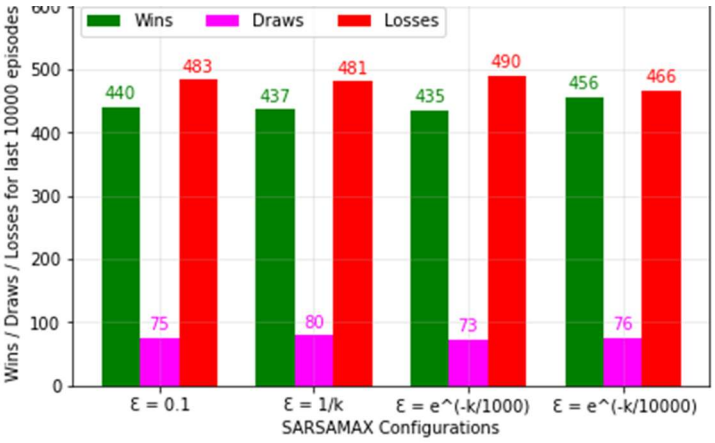


Figure 64 - SARSAMAX Wins / Draws / Losses per Configuration

Similar to SARSA's graphs, it was noted that for SARSAMAX although configuration four boasted the highest win rate the discrepancy between all four configurations was minimal. The dealer advantages shown in Figure 63 make this discrepancy more apparent as configuration four has the least advantage with a value of 0.011 whereas the remaining configurations have a value larger or equal to 0.047. Therefore, similar to SARSA, the fourth configuration serves to be the best out of the four implemented. SARSAMAX continues to mirror SARSA as similarly only the first and fourth configurations manage to explore the full state action set for the same reasons as those discussed with respect to SARSA.

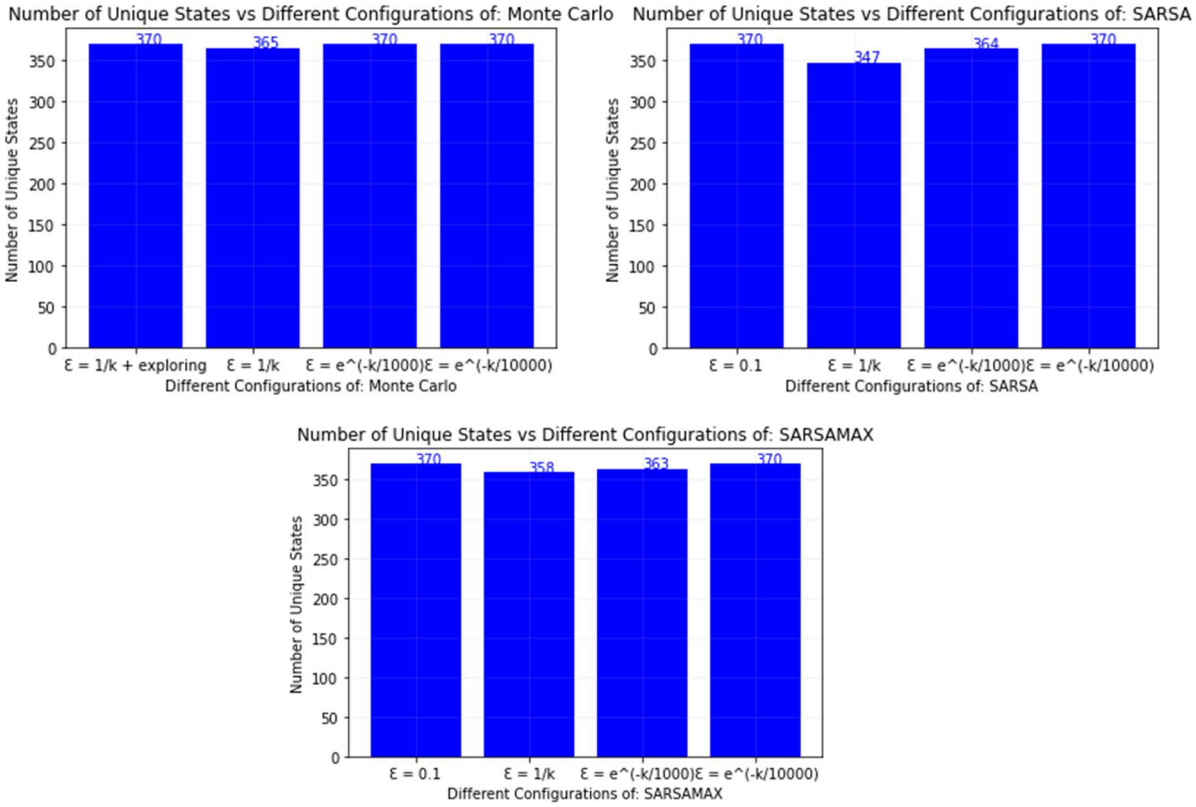


Figure 65 – Unique States for each Algorithm Configuration

In the above diagrams it is apparent that in relation to Monte Carlo the different configurations did not affect the results of the algorithm as excluding the outlier all configurations explored 370 unique states. This observation also holds true in relation to SARSA and SARSAMAX although in the latter the discrepancy in the unique states present in the second configuration is easier to notice. Although the unique states explored do not vary much in relation to three algorithms and their configurations the dealer advantage does indeed change from one algorithm to the other as Monte Carlo proved to have the greatest dealer advantage of 0.071 whilst SARSAMAX had the lowest of 0.011. This coincides with the findings presented above in which the SARSAMAX leads to the greatest overall win rate for the player whilst the opposite holds for Monte Carlo.

Extra Implementation – Double Q-Learning Algorithm

Since this implementation was not requested in the brief, it was decided not to explain it in the documentation. The following graphs were generated when running this algorithm:

Configuration 1: $\varepsilon = 0.1$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	H	S	S	S	S	S	S	S
18	H	S	S	H	S	S	H	S	H	H
17	H	H	H	H	H	S	H	H	H	H
16	H	S	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	S	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	S	S	H	S
15	S	S	S	S	S	H	H	H	H	S
14	S	H	S	S	S	H	H	H	H	S
13	H	S	S	S	S	H	H	H	H	S
12	H	S	H	S	S	H	H	H	H	H

Figure 66 – Strategy Table (Double Q-Learning Config 1)

Configuration 2: $\varepsilon = \frac{1}{k}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	H	S	S	S	S	S	S
19	H	S	S	H	H	H	S	S	H	S
18	S	S	H	S	S	H	S	H	H	S
17	H	H	H	S	H	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	S	S
16	S	S	S	S	H	S	H	H	H	S
15	H	S	S	S	S	S	S	S	H	S
14	S	S	H	S	H	S	H	H	H	H
13	S	H	S	H	S	H	S	H	H	H
12	H	S	H	S	S	H	H	H	H	H

Figure 67 – Strategy Table (Double Q-Learning Config 2)

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	H	S	S	S	S
19	S	S	S	S	H	H	S	S	H	H
18	H	H	S	H	S	H	H	H	H	H
17	H	H	H	S	H	H	H	H	H	H
16	H	H	H	H	S	H	H	H	H	H
15	H	H	H	H	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	H	H	S	S
16	S	S	S	S	S	H	H	S	H	S
15	S	S	S	S	S	S	S	H	H	S
14	S	S	S	S	S	S	H	H	S	H
13	S	S	S	S	S	H	H	S	H	H
12	H	H	S	H	H	H	H	H	H	H

Figure 68 – Strategy Table (Double Q-Learning Config 3)

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

BlackJack Strategy Table when player is using an Ace as 1

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	H	S	S
18	S	S	S	H	H	S	S	H	H	H
17	H	H	S	S	S	H	H	H	H	H
16	H	H	H	H	H	H	H	H	H	H
15	H	H	H	S	H	H	H	H	H	H
14	H	H	H	H	H	H	H	H	H	H
13	H	H	H	H	S	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

BlackJack Strategy Table when player is using an Ace as 11

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	S
15	H	S	S	S	S	H	H	H	H	S
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	S
12	H	S	H	S	S	H	H	H	H	H

Figure 69 – Strategy Table (Double Q-Learning Config 4)

The above strategy tables seen in Figures 66-69 were generated after running the algorithm once with each respective configuration. In comparison with Q-Learning most of the strategy tables produced similar results however with respect to the configuration $\varepsilon = \frac{1}{k}$ although the improvement is minimal more Stand actions can be noticed in Double Q-Learning whereas in Q-Learning the action chosen was mainly a Hit.

Configuration 1: $\varepsilon = 0.1$

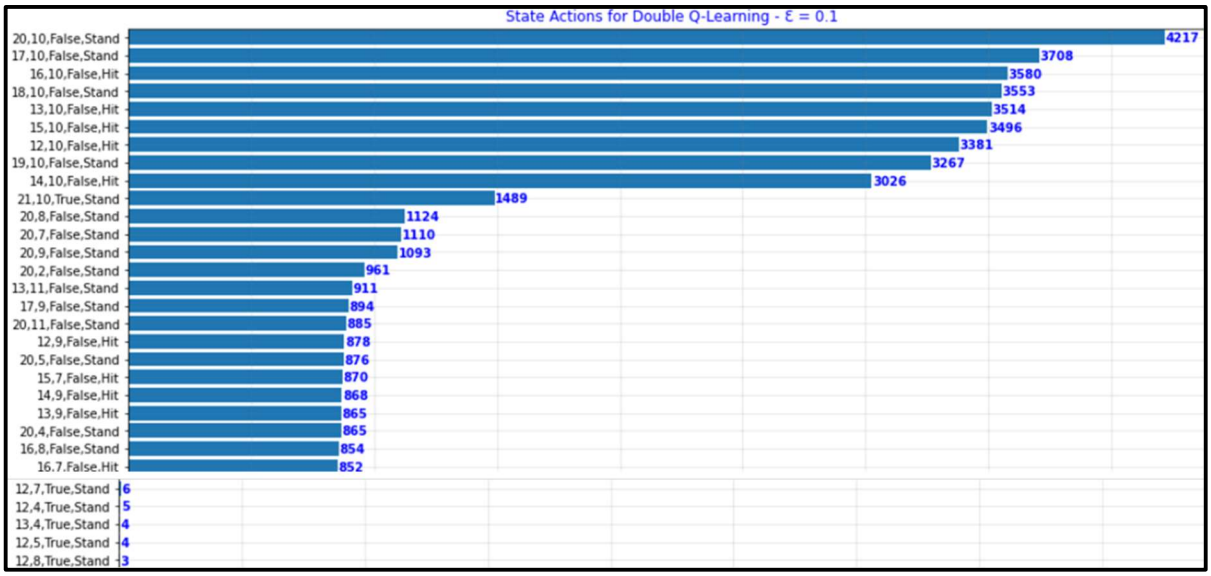


Figure 70 – Double Q-Learning State Actions List Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

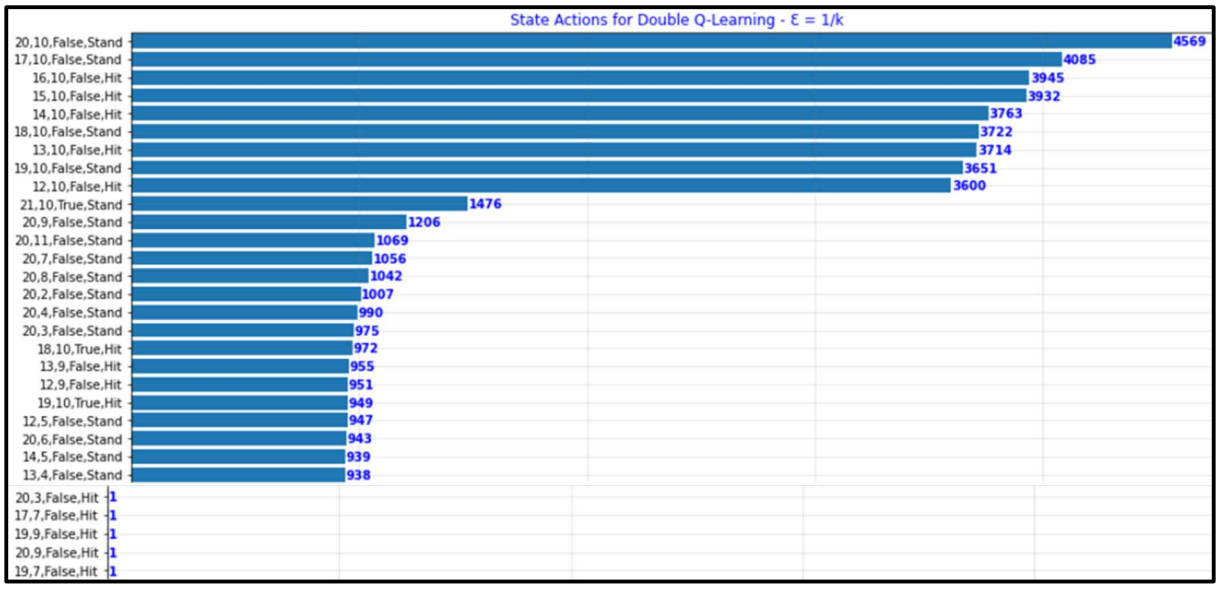


Figure 71 – Double Q-Learning State Actions List Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

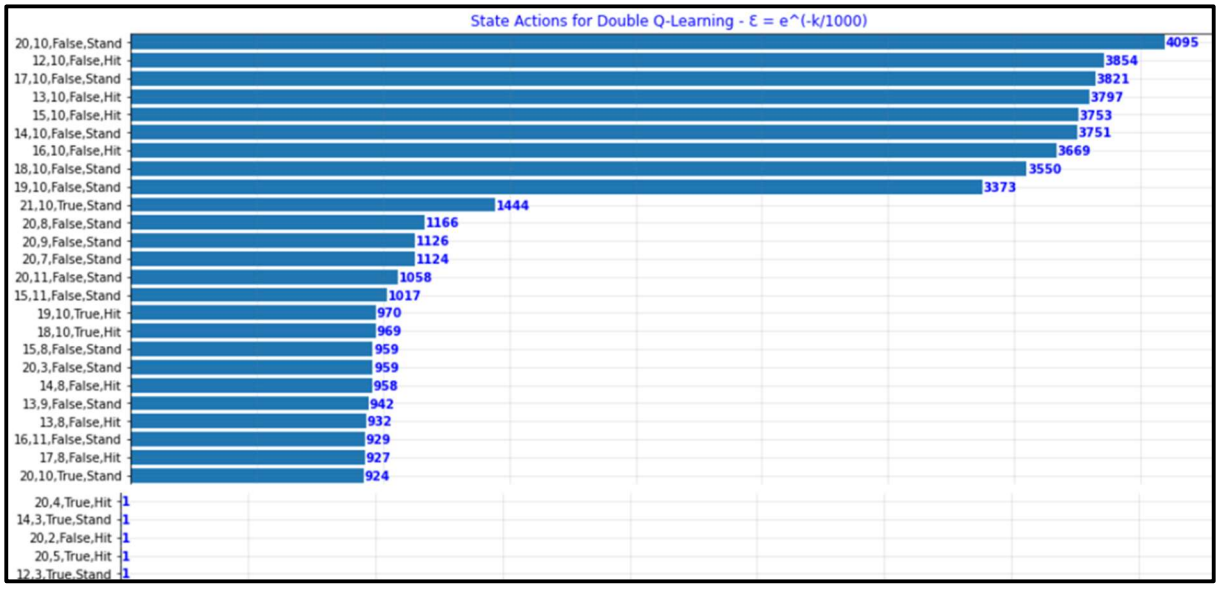


Figure 72 – Double Q-Learning State Actions List Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

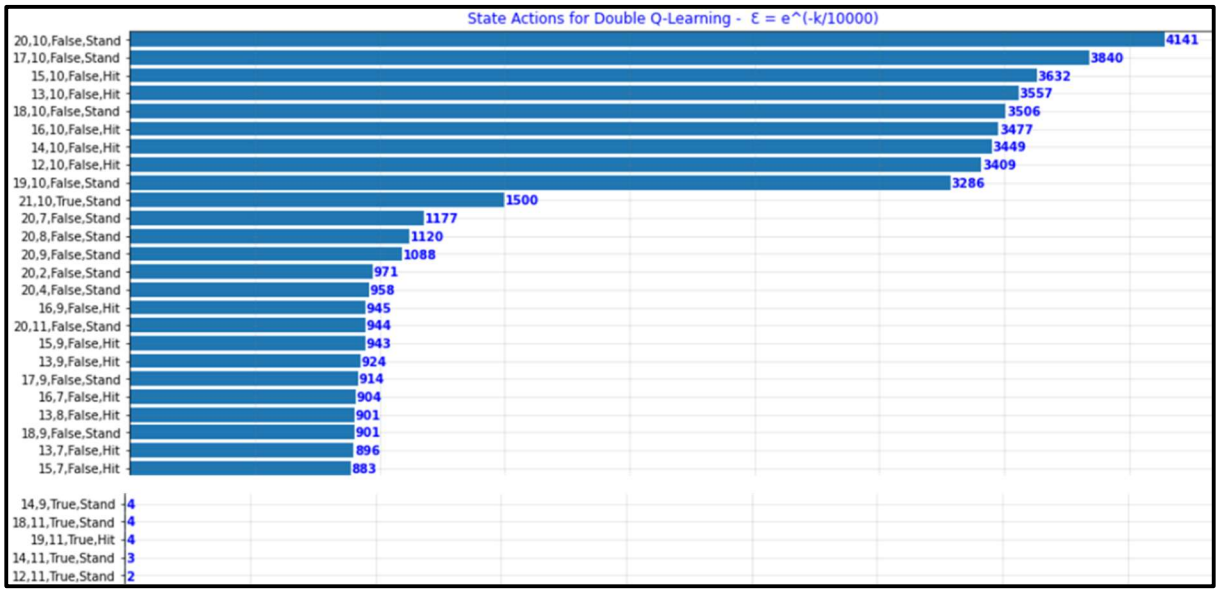


Figure 73 – Double Q-Learning State Actions List Config 4

The state-action pair graphs provided results very similar to those of Q-Learning. The only difference noted was that in Double Q-Learning, the third configuration's most exploited state action pair was selected for fewer times than that of the fourth configuration. However, it was also noted that the least chosen state action pairs in the fourth configuration were chosen for more times than those in the third configuration, thus the observation discussed earlier stating that the fourth configuration offers the most exploration still stands.

Configuration 1: $\varepsilon = 0.1$

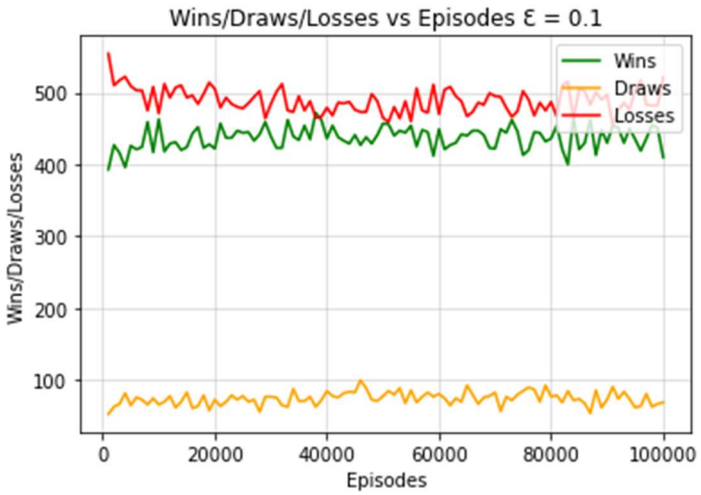


Figure 74 – Double Q-Learning Wins / Draws / Losses per 1000 Episodes Config 1

Configuration 2: $\varepsilon = \frac{1}{k}$

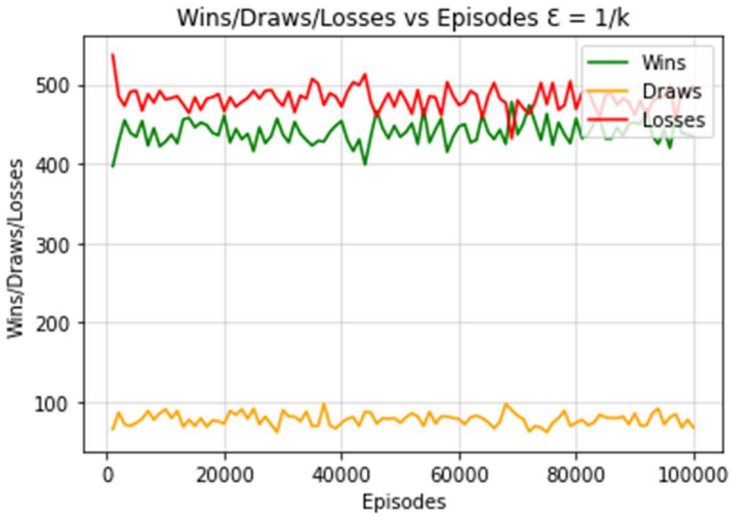


Figure 75 – Double Q-Learning Wins / Draws / Losses per 1000 Episodes Config 2

Configuration 3: $\varepsilon = e^{(\frac{-k}{1000})}$

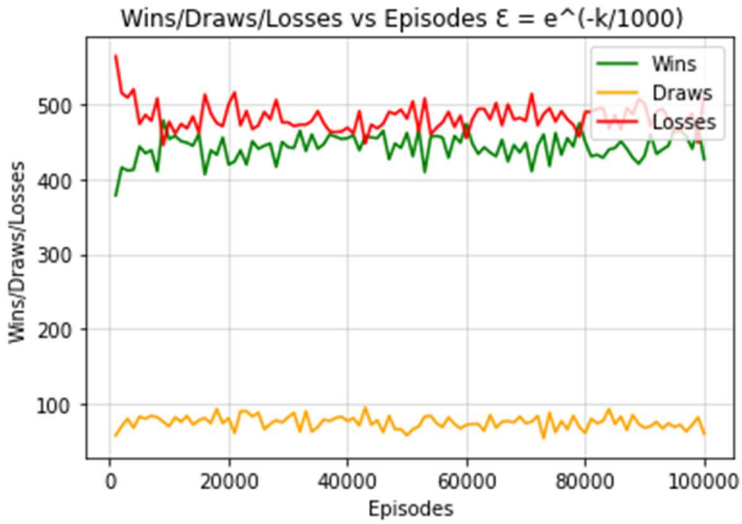


Figure 76 – Double Q-Learning Wins / Draws / Losses per 1000 Episodes Config 3

Configuration 4: $\varepsilon = e^{(\frac{-k}{10000})}$

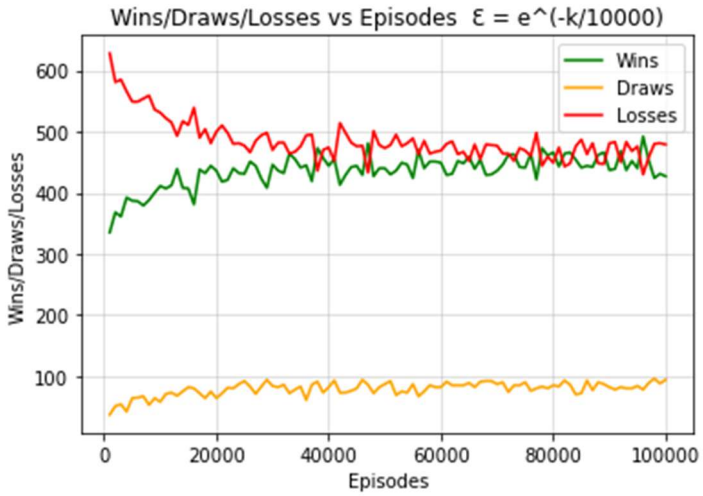


Figure 77 – Double Q-Learning Wins / Draws / Losses per 1000 Episodes Config 4

The claims made for Q-Learning apply very similarly to those of Double Q-Learning. Over multiple runs of each respective algorithm and configuration, it was noted that sometimes Q-Learning performed slightly better and sometimes Double Q-Learning performed better. What was also noted was that in general, Double Q-Learning tends to converge more smoothly when compared to Q-Learning as can be seen when comparing Figures 58-61 to Figures 74-77.

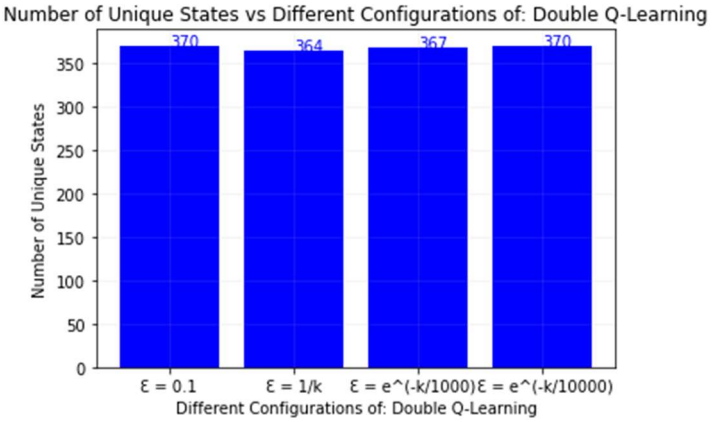


Figure 78 – Double Q-Learning Unique States per Configuration

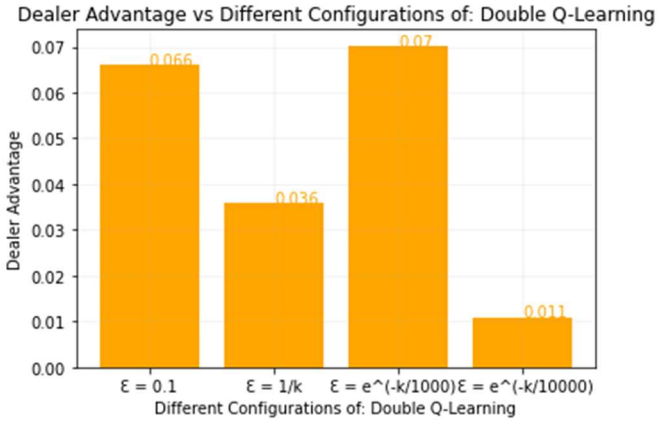


Figure 79 – Double Q-Learning Dealer Advantage per Configuration

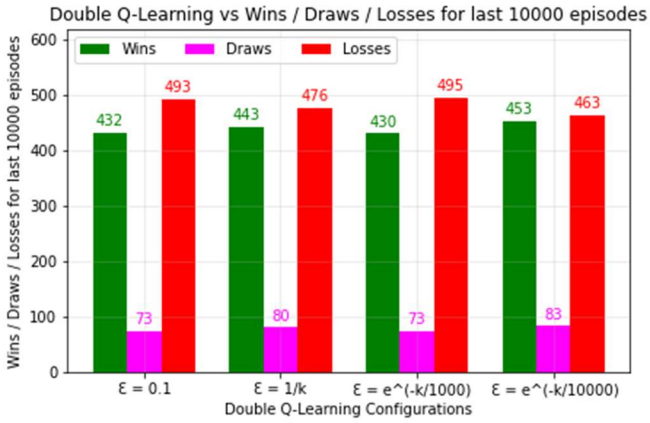


Figure 80 – Double Q-Learning Wins / Draws / Losses per Configuration

In continuation with the previous paragraph, it can be noted that Double Q-Learning performed slightly worse than Q-Learning as although the dealer advantage in this example is approximately the same, the results show that the agent lost more when compared to the Q-Learning results. In Figures 78-80 it can also be concluded that the exploration / exploitation rate of each configuration is similar to those in Q-Learning.

Double Q-Learning Conclusion

Double Q-Learning tends to perform better in environments with a high-level of noise, evidently the Blackjack environment has no noise and little uncertainty. Therefore, it is not necessarily practical to implement such an algorithm for this scenario. This can be seen in the above results as in this case, Double Q-Learning performed slightly worse than standard Q-Learning [12]. However, in general, both algorithms tend to produce very similar results because both converge to the same optimum. Due to the simplicity of the problem being solved, the benefits of Double Q-Learning could not be observed clearly however Double Q-Learning still converged slightly prior to Q-Learning. Additionally, there were instances where Double Q-Learning performs better than standard Q-Learning as can be seen in Figure 81, where a dealer advantage of 0.008 is generated compared to the 0.011 generated by Q-Learning. This result was more optimal than the previously generated results in Figure 63 and Figure 79.

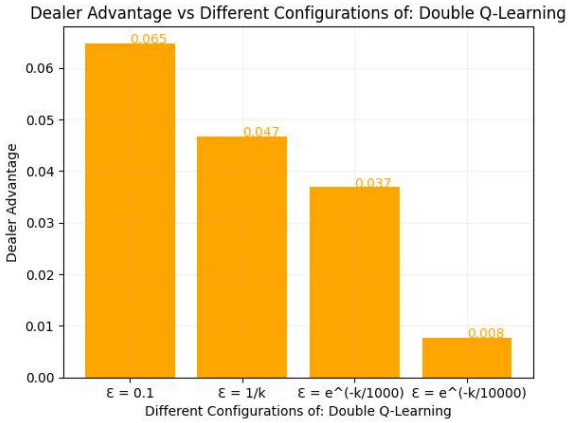


Figure 81 – Double Q-Learning Dealer Advantage per Configuration – Best Result

References

[1] "Blackjack," Wikispeedia. [Online]. Available: https://www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/wp/b/Blackjack.htm [Accessed: 21-Apr-2023].

[2] P. Bajaj, "Reinforcement Learning," GeeksforGeeks, 18-Apr-2021. [Online]. Available: https://www.geeksforgeeks.org/what-is-reinforcement-learning/ [Accessed: 21-Apr-2023].

[3] J. Zhang, "Reinforcement Learning: Solving Blackjack," Towards Data Science, 15-Jul-2019. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-solving-blackjack-5e31a7fb371f [Accessed: 21-Apr-2023].

[4] J. Bajada, ARI2204: "Monte Carlo Methods" [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1157997/mod_resource/content/3/Reinforcement%20Learning%20-%205%20-%20Monte%20Carlo%20Methods.pdf [Accessed: 21-Apr-2023].

[5] J. Liu, "On the convergence of reinforcement learning with Monte Carlo Exploring Starts," Automatica, vol. 129, pp. 109693, 2021. Available: https://www-sciencedirect-com.ejournals.um.edu.mt/science/article/pii/S0005109821002132 DOI: 10.1016/j.automatica.2021.109693. [Accessed: 21-Apr-2023].

[6] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. pp 114. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf [Accessed: 21-Apr-2023].

[7] J. Bajada, ARI2204: "Temporal Difference Methods" [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1157998/mod_resource/content/2/Reinforcement%20Learning%20-%206%20-%20Temporal%20Difference%20Methods.pdf [Accessed: 21-Apr-2023].

[8] A. Gupta, "SARSA Reinforcement Learning," GeeksforGeeks, 24-Jun-2021. [Online]. Available: https://www.geeksforgeeks.org/sarsa-reinforcement-learning/ [Accessed: 21-Apr-2023].

[9] R. S. Sutton, "Learning to predict by the methods of temporal differences," Machine Learning, vol. 3, no. 1, pp. 9-44, 1988. [Online]. Available: http://incompleteideas.net/papers/sutton-88-with-erratum.pdf [Accessed: 21-Apr-2023].

[10] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. pp 130. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf [Accessed: 21-Apr-2023].

[11] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. pp 131. [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf [Accessed: 21-Apr-2023].

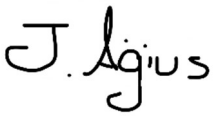
[12] H. Hasselt, "Double Q-learning," in Advances in Neural Information Processing Systems, Vancouver, Canada, Dec. 2010, pp. 2613-2621. [Online]. Available: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf. [Accessed: 23-Apr-2023].

Distribution of Work

Work was partitioned equally between all members.

Signatures

Matthias Bartolo 

Jerome Agius 

Isaac Muscat 

Plagiarism Declaration Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

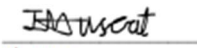
~~I~~/ We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is ~~my~~ / our* work, except where acknowledged and referenced.

~~I~~/ We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthias Bartolo	
Student Name	Signature
Jerome Agius	
Student Name	Signature
Isaac Muscat	
Student Name	Signature
Student Name	Signature

ARI2204	Reinforcement Learning Assignment
Course Code	Title of work submitted

23/04/2023
Date