# Suit Collector

In this tutorial we are going to train an agent to play ‘Suit Collector’ game using Reinforcement Learning. We will also learn how to create a gymnasium environment for the agent. The problem is solved using Reinforcement Learning techniques such as Deep Learning, Temporal Difference Q-learning algorithm, Double Q-Networks (DQN), Experience Replay, Epsilon-Greedy. The tutorial covers how to implement these techniques to train an agent. The tutorial also covers the background on what Suit Collector game is and how it’s played.

# Objectives

The objectives of this tutorial are:

* Understand the Suit Collector game, including the rules and objective of the game.
* Understand the concept of Deep Learning, Q-learning algorithm, Double Q-Networks (DQN), Experience Replay, Epsilon-Greedy, smart rewards and how it can improve the training of a reinforcement learning agent.
* Understand the architecture of the Neural Network used in the Suit Collector game and its training process, including the input and output layers, the hidden layers, and the activation functions.
* Be able to implement a basic version of the Suit Collector game using a reinforcement learning framework, gymnasium.
* Be able to modify the hyperparameters of the training process, like the learning rate, discount factor, and epsilon-greedy exploration rate, replay buffer size, and evaluate the agent's performance.

# Level

This tutorial is meant for graduate students and is a Hard level tutorial.

# Background

## Suit Collector - Game Background:

|  |  |
| --- | --- |
| Suit Collector is a two player which requires 16 cards from a deck of playing cards which are Ace, King, Queen, and Jack from each of the four suits’ namely, Clubs, Hearts, Spades and Diamonds. These 16 cards are shuffled and arranged in a 4x4 grid. See the image on right for an example. |  |

Then the two players take turns playing the game such that:

* On the first turn Player 1 picks a suit and then Player 2 picks a different suit.
* In the following turns Player 1 first swaps two of the neighboring cards such that neither of the cards do belong to the Player 2 suit, Followed by Player 2 swapping two neighboring cards such that they do belong to player 1s suit.
* The game ends with player 1 winning if after his turn all the cards of his suit are arranged with Ace, King, Queen, and Jack in that order (or reversed) in a single(same) row, column, or diagonal or by player 2 winning if after his turn all the cards of his suit are arranged with Ace, King, Queen, and Jack are in that order (or reversed) in a single(same) row, column, or diagonal.
* After ‘N’ number of turns if the game is still undecided then it can be considered as a draw.

## Game Analysis:

### State Space:

Since there are 16 cards and each player can pick a suit in the game, and to represent the state of the game at any given time ‘t’, we need to know the board position and the suits picked by each player so therefore in total we have *, “271 trillion 996 billion 268 million 544 thousand”* number of states. Which can be represented by 18 Positions, where the first 16 represents the card value at each position in a 4x4 grid, and the next two represent the chosen suit of the players. The card values require 4 bits of memory to store efficiently, as there are 16 different cards. The chosen suit requires 3 bits of memory, as there are 4 suits (which require 2 bits to represent) and 1 more option to for the states where a player has not yet chosen a suit (Since the agent can go first at which point the chosen suit for player 2 is undecided or go second at which point the opponent suit is given). Therefore, a state can be represented efficiently in 3 = 67 bits of memory.

### Action Space:

At the beginning of the game, each player picks a suit, and there are 4 suits, therefore, the action space can include 4 actions one for picking each of the suit, namely Clubs, Hearts, Spades, and Diamonds.

|  |  |  |  |
| --- | --- | --- | --- |
| After picking the suit, the card can be swapped by any of its neighboring cards, meaning, it can be swapped with any of the card that it is surrounded by, in this game a card can be surrounded by a maximum of 8 cards. However, the card on the corner of the 4x4 grid is surrounded by 3 cards, and the card on the edge of the board is surrounded by 5 cards, as shown in the diagram on the right. | | | A picture containing table  Description automatically generated |
| However, we know that if cards ‘A’ and ‘B’ are neighbors then swapping ‘A’ with ‘B’ or ‘B’ with ‘A’ will result in the same state, therefore we can reduce the total number of actions to 42 by removing redundant actions from the left corner of the grid to the right which is shown on the right. Each action is a floating-point number which requires 32 bits for accuracy. | | | Table  Description automatically generated |
| Therefore, in total we have 46 actions, 42 for swapping two cards, and 4 for selecting a suit. The action space for this game is decided as the table on the right:  The 4x4 grid is numbered from top-left to bottom-right starting with ‘0’ in the top left cell and ending with ‘15’ in the bottom right cell. |  | | |

## Solution Background:

### Challenges with Q-Learning with Q tables:

To store the values of all states and their corresponding Q values for each action we require a storage space of (271996268544000\*67)\*(46\*32) bits = 26,825,359,988,883,456,000 bits = 3122882916.23 Gigabytes, 312 billion 288 million 289 thousand and 6.23 Gigabytes of storage space and a fast computational processor (Like a Quantum parallel processor maybe?) which can process and calculate Q values for this Stored Q-table data is not easily available in the year 2023. Therefore, we have chosen to go with training a Neural Network model to get a function approximation for the game with Temporal difference Q-Learning method instead.

### Breaking the Game into two Phases:

Since the storage and computational space for the game is too high, we have logically divided the game into two phases, Phase 1, and Phase 2.

#### Phase 1:

In the first phase, we create a sub-game, where the aim of the agent is to pick a suit given a board position. Thus, reducing the number of actions to 4. A Q table for this would require a storage space of (271996268544000\*67) \* (4\*32) bits =27 billion (Approx.) which is big enough however, far lesser than what we started off. Below is a brief overview on the ideas we have for solving phase 1:

##### Idea 1: Q-Learning with smart rewards

The main approach taken at this point to solve the phase 1 problem is Q Learning. The phase 1 problem is training the agent through Q learning to pick an optimal suit for a given random placement of cards on the 4x4 grid. We are keeping track of the Q values for each suit to be picked and using them to train the agent.

Phase 1 of the game is to pick a suit when a new game is started. If the agent has given the chance to pick the suit first, then the agent has 4 choices – Spades, Hearts, Clubs, and Diamonds. In this scenario, picking the optimal suit is important because it will increase the chances of winning the game for the agent if the cards are placed closer and if it requires fewer moves to create a winning position formation.

|  |  |  |
| --- | --- | --- |
| In the current iteration/model, we are considering the winning positions, i.e., 4 positions where the cards are aligned horizontally, 4 positions where the cards are aligned vertically, and 2 positions where the cards are aligned diagonally. The winning positions are displayed on the right:  Here we are not considering the sequence of the cards that are positioned in these alignments. We are only concerned with seeing the chosen cards of the suit in one row (horizontal, vertical, or diagonal). | |  |
|  |  | |

###### Q Learning Approach

Q Learning is a very well-suited approach for this problem. To use Q learning here, we are combining it with the epsilon-greedy approach to select an action. The action is picking a suit. After the agent picks a suit, the status of the board along with the picked suit is passed to the Q learning function to calculate the reward of the agent’s action. Since this is Q learning, we are concerned with the next state and action as well. This next state and action will be the updated state of the board which marks the cards picked by the agent as negative values and 3 choices given to the opponent of the agent. As there are 3 possible next states, the rewards for all these states are calculated and the case where the opponent will choose the worst possible suit is selected (looking at the best case right now).

After this, the reward of the current state is calculated by using the Q learning formula but with a slight modification. Instead of adding the reward of the next state, we will be subtracting it since it is the reward of the opponent (After multiplying it with the Gamma).

These calculated rewards will modify the Q table and then till the policy converges, this procedure is repeated. A policy is considered converged when the agent picks the same suit for the board 5 times in a row. After the policy is converged, the chosen optimal action is displayed along with the board positions.

###### Brief Overview of the Techniques Used in the Training:

1. Q Learning [1]

###### Brief Overview of the Training:

1. The above-described approach in Idea 1 section for phase 1 is iterated till the policy is converged. This is done for 10 random boards for now.
2. Q table for phase 1 is small and contains only 4 entries, 1 entry per suit.
3. Right now, it is trained according to the proximity of the positions of the cards chosen to the winning positions available. The reward function depends on the closeness of the chosen cards to a winning position.

##### Idea 2: First Solve Phase 2 .

This Idea requires that we have an agent which has mastered the next phase (Phase 2) of the game (which is not yet discussed, However), assume that we have an agent, which can play the game well given its suit and the opponents suits at any position of the board. If we have such an agent which can give us an appropriate Q-Values for any given State. We can then choose a suit for any given position by considering the Q-Values for all the valid suits in that position and picking the one which has the maximum Q-Value, since the higher the Q-Value the better the next state is for us to win.

#### Phase 2:

Phase 2 of the game is where we assume the suits for the two players have been decided and learn to play the game with the given suits. Since the suits for both the player has been decided we can reduce the game state space at any given time ‘t’, we only care about the agent’s cards and the opponents’ cards on the 4x4 grid which is sufficient for us to calculate the move for the next state as part of the Markov Decision Process. Therefore, we can mask the board to have only 9 different values, 4+4 for the opponents and agent’s cards and 1 to represent all the other cards. Which decrease the overall complexity of the game for a neural network as it has fewer unique numbers of states to deal with, this also decreases the overall size of the Q-Table to the following:

.

This is much lesser than what we had before and solving this would also give us Phase 1 (using idea 2). Below are the brief overviews on the Ideas we have for solving phase 2:

##### Idea 1: The Min-Max Approach

Since this is a two-player game, The idea here is to create a Min-Max agent first using the classic Min-Max Algorithm and train the Neural Network model to win against the Min-Max agent. However, this requires us to develop a solution first and then use the existing solution to solve the current one at hand. Which we were reluctant to do, and wanting to build intelligence from scratch iteratively as though the solution to this problem did not exist therefore, we did not pursue this solution. However, the Min-Max agent can be useful for testing and optimizing the neural network agent, and a combination of the Neural Network agent and Min-Max agent to decrease the overall computation time to find the optimal best solution is something that we are interested to investigate, however, we are parking this idea for the time being for idea 2.

##### Idea 2: The Iterative Ground Up Approach

The idea here is to iteratively build up an agent by letting it play against and beat a previously trained agent such that the current agent is better than the last agent and repeat this process. We start off with the base agent, in Iteration 0, which is named as ‘Agent Rock’, which plays a random valid move in any given state. From there in Iteration 1, we build a new agent ‘Agent Iron’ which would beat the previously trained agent and repeat this process. Following is the list of Agents in each Iteration and the name that we have used in the code and project. Iteration 0: Rock, Iteration 1: Iron, Iteration 2: Gold, Iteration 3: Diamond, Iteration 4: Obsidian, Iteration 5: Quartz.

###### Brief Overview of the Techniques Used in the Training:

1. Deep Learning Using Neural Network [2]
2. Temporal Difference, Q-Learning [3][4][5]
3. Experience Replay [6]
4. Double Q Learning (DQN) [7]
5. Epsilon Greedy Method [8]

###### Brief Overview of the Training:

1. Data is collected for 10 Games using the epsilon-greedy method.
2. The neural network is trained in batches of 32 using the data from the 10 Games in step 1, plus random pool of moves from experience replay buffer such that the total number of are 32\*200.
3. Step 2 is repeated 12 times.
4. Steps 1, 2, and 3 are repeated 30 times, after which the target Q Model is updated.
5. Steps 1, 2, 3, and 4 are repeated until the agent wins ten games sequentially, a thousand times.

A lot of inspiration and information was collected from Atari Breakout paper/code Implementation [9] [10] for this project to put all the above techniques together and to further optimize and update the code as per the requirements of this project.

# Requirements

1. Knowledge of Python programming language, including concepts such as variables, functions, loops, and conditional statements.
2. Experience with using VS code for coding and debugging.
3. Installation of Anaconda and the creation of a Python 3.9 Anaconda environment.
4. Installation of the required packages: NumPy, Gymnasium, TensorFlow and Keras.
5. Knowledge of the game Suit Collector, including the rules and the actions that the agent can take.
6. Understanding the reinforcement learning concepts, including Markov Decision Processes, reward functions, and policy optimization.
7. The knowledge of concepts adapted in our project such as Neural Networks, Deep learning, Epsilon Greedy, DQN, Experience Replay.

# Tasks

## Project Setup and training the model:

You can execute the following steps to set up the environment. It is assumed that you already have the requirements specified above.

1. Install Anaconda from the official Anaconda website.
2. Open the terminal and run the following command to create a new conda environment: *> conda create -n gputest python=3.9*
3. Activate the conda environment using the following command: *> conda activate gputest*
4. Install gymnasium: *> pip install gymnasium*
5. Install TensorFlow using: *> conda install tensorflow*
6. Next, we need to install Keras: *> conda install tensorflow.keras*
7. Finally, unzip Raider\_Squad\_prototype1\_V1.zip and extract all the files to get started.
8. To Train Phase 1, navigate to phase1.py file and run *> python phase1.py*
9. Training in Phase 2: it is assumed that conda environment is set using *> conda activate gputest* and the command prompt is pointing to ‘*<extract\_directory>/Raider\_Squad\_prototype1\_V1/Train/Phase2>*’ directory.
   * To train Phase 2 agent Iron navigate to Iteration2 *>cd ./Iteration1*, Run train.py using the command *>python train.py*
   * To train Phase 2 agent Gold navigate to Iteration3 *>cd ./Iteration2*, Run train.py using the command *>python train.py*
10. To Play against agent trained in phase 2: it is assumed that conda environment is set using *> conda activate gputest* and the command prompt is pointing to ‘*<extract\_directory>/Raider\_Squad\_prototype1\_V1/Train/Phase2>*’ directory.
    * To play with agent Rock navigate to Iteration1 using *>cd ./Iteration1*, then Run game.py using the command *>python game.py*
    * To play with agent Iron navigate to Iteration 2 using *>cd ./Iteration2*, then Run game.py using the command *>python game.py*
    * To play with agent Gold navigate to Iteration 3 using *>cd ./Iteration3*, then Run game.py using the command *>python game.py*

## Fill in the Code Snippets: (Phase 1)

value\_opponent **=** **[]**

next\_action\_set **=** **[**0**,** 1**,** 2**]** # Assuming agent has chosen action 3...

reward**,** state **=** new\_reward**(**agent\_ID**,** state\_revived**,** action**,** agent\_suit\_cards**)**

agent\_ID **=** 1

# Setting agent ID as 1 which is agent has picked the suit, but now we will consider what happens after the agent has picked this suit... The opponent will have 3 choices... Computing the reward for each choice the opponent has, and appending those to the list called value\_opponent declared above...

reward\_previous**,** reward\_index **=** q\_function**(**agent\_ID**,** state\_copy\_1**,** next\_action\_set**[**0**],** q\_table**,** gamma**,** agent\_suit\_cards**)**

# TO DO

# Fill in the code here...

# Append the reward calculation in the value\_opponent list.

# Code similarly for action 1 and 2... Print the maximum out of those three...

# Chosen suit is Hearts...

Card\_IDs **=** **[**4**,** 5**,** 6**,** 7**]**

Board **=** **[[**11**,** 12**,** 5**,** 9**],** **[**0**,** 10**,** 2**,** 8**],** **[**3**,** 1**,** 13**,** 6**],** **[**14**,** 15**,** 7**,** 4**]]**

Current\_sparse **=** **[]** # current sparse matrix is empty

# TO DO

# Fill in the code here...

# Create a sparse matrix out of the given input of the card\_IDs and board.

# Your output should look like this:

# [[0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]

**def** epsilon\_greedy**(**q\_table**,** epsilon**=**0.5**):**

# Epsilon greedy policy to pick the action according to the random function and the epsilon...

policy **=** random**.**uniform**(**0**,** 1**)** # Random number generator in the range of 0 to 1...

**if** # TO DO Write the condition here:

**return** # TO DO Pick random action if epsilon > policy

**else:**

**return** # TO DO Else pick the max action from the q table.

## Fill in the Code Snippets: (Phase 2)

### Temporal Difference Q-Learning with Backpropagation for updating the Neural Network using DQN. [Line 289-315, Iteration 1, train.py]

# updated Q\_values = r + gamma \* max(Q(S',a'))

updated\_q\_values **=** **# Todo Fill in the code here...**

# Correct the Q(s,a) for s when s is the final state.

new\_update\_q\_values **=** updated\_q\_values**.**numpy**()**

**for** ildse **in** **range(len(**done\_sample**)):**

**# Todo Fill in the code to get new\_updated\_q\_values for final state.**

updated\_q\_values **=** tf**.**convert\_to\_tensor**(**new\_update\_q\_values**)**

# update the target network with new weights

**if(# Todo Fill in the code here...**

**):**

**# Todo Fill in the code here...**

### Epsilon-Greedy [Line 187-201, Iteration 1, train.py]

**if** epsilon **>** np**.**random**.**rand**(**1**)[**0**]:**

# Take random action

action **=** **# Todo(Fill in the code here to take a random action)**

**else:**

# Predict action Q-values

# From environment state

state\_tensor **=** tf**.**convert\_to\_tensor**(**state**)**

state\_tensor **=** tf**.**expand\_dims**(**state\_tensor**,** 0**)**

action\_probs **=** model**(**state\_tensor**,** training**=False)**

# Take best action

action **=** **# Todo(Fill in the code here)**

# epsilon greedy stuff...

epsilon **-=** **# Todo(Fill in the code here to reduce epsilon values)**

epsilon **=** **max(**min\_epsilon **,** epsilon**)**

### Experience Replay [Line 260-274, Iteration 1, train.py]

# get random moves from replay buffer so we have 'batch\_size' S,A,R,S' for training

indices\_replay\_buffer **=** **# Todo(Fill in the code here to take random indices from replay buffer)**

## Test Cases:

### Phase 1:

The test cases for phase 1 model are generated automatically every time we run the model since we are initializing the board randomly every time.

Following 2 are the examples of the output of the test cases generated randomly:

#### Test case 1:

|  |  |
| --- | --- |
|  | In this case, the agent has chosen Clubs which means cards = [8, 9, 10, 11]. The minimum moves required to form alignment of each suit are as follows: Spades = 1, Hearts = 2, Clubs = 1, Diamond = 3 According to this calculation, the agent picked the right suit! |

#### Test Case 2:

|  |  |
| --- | --- |
|  | In this case, the agent has chosen Diamonds which means cards = [12, 13, 14, 15]. The minimum moves required to form alignment of each suit are as follows: Spades = 1, Hearts = 2, Clubs = 3, Diamond = 3 According to this calculation, the agent picked the wrong suit! |

These test cases were taken from a single run time session of the model. In this session, 10 random boards were generated. Out of these 10, the agent picked the right suit in 6 cases (like test case 1), and it picked the wrong suit in 4 cases (like test case 2).

This result ratio may change in any other run time session. The agent may perform better or worse than the above example.

### Phase 2

In the below example, we play against agent Gold, where the aim of the agent is to get its cards namely be in that order or reversed in a winning position whereas the aim of the player(us) is to get the cards in that order or reversed in a winning position.

#### Test Case: Playing against Gold Agent.

|  |  |
| --- | --- |
| 1. Shows the result of the previous game where the player got a reward of -1 for making an invalid action (tried to swap opponents cards) and the game is now over with agent not needing to play its turn. 2. A new game is created and shows the board position. 3. User enters action 0 which swaps position 0(0) with position 4(0). 4. Agent Gold has chosen action 29, moving -2 from position. 12 to 9. 5. Shows the reward the player received for swapping useless/pointless indifferent cards -0.05. 6. Shows the state of the board after step 3 and step 4 actions. 7. Users perform the same action as shown in step 3 for the rest of the game for testing. 8. Agent Gold chooses to perform action 16 which swaps -3 and 0 from position 8 and position 5. 9. Shows the state of the board after the user and agent makes their moves. 10. After the user plays action 0, agent Gold performs action 6, swapping -4 and 0 from position 2 and 3. 11. After Agent Gold makes its moves, it has successfully moved -1, -2, -3 and -4 in column 2. Therefore, the agent won, and the user is rewarded with -1 concluding the game with a loss. 12. Shows the new board for the next game. | Note: Refer to ‘Game Analysis Action space’ section to understand what action corresponds to swapping which two cards. |

# Learning Outcomes

1. Proficiency in installing python and python packages like gymnasium, TensorFlow, Keras.
2. Ability to create Deep-learning models using Keras.
3. Ability to change the hyperparameters of the model for fine tuning performance.
4. Hands-on experience on Q-learning, Deep-learning, DQN, Epsilon greedy and Experimental Learning.

# Exercises

1. Change the hyper parameters in phase 1 and phase 2 training models to see how it affects the performance.
2. Try to change the rewards given for the actions taken and see how it affects the learning of agent.
3. Try to write a better reward function to increase the performance of the phase 1 model.
4. Try a unique way of adjusting the calculated reward instead of taking the reciprocal and subtraction in the phase 1 model.
5. Try to use a different approach instead of the winning positions in the phase 1 model.
6. Try to train agent Diamond in Phase 2 iteration 3.

# References

[1] Dr. Susan Mengel, CS-5392 Reinforcement Learning, Section – 001, Lecture PPT – Bellman\_Dynamic\_Programming.pdf, Page 3 - Slide 2 (page 6), Page 4 – Slide 1 & 2, Page 8 – Slide 1, Page 11 – Slide 2.

[2] Wang, M. (2020, November 17). *Hands on Tutorial: Deep Q-Learning Tutorial: MinDQN A Practical Guide to Deep Q-Networks*. Towards Data Science, Medium. <https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc>

[3] Q-learning. (2023, March 22). In *Wikipedia*. <https://en.wikipedia.org/wiki/Q-learning>

[4] Seif, G. (2019, May 20). *Understanding the 3 most common loss functions for Machine Learning Regression*. Towards Data Science, Medium. <https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3>

[5] Brownlee, J. (n.d.). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning>

[6] Data Science Stack Exchange (n.d.). *What is "experience replay" and what are its benefits?* <https://datascience.stackexchange.com/questions/20535/what-is-experience-replay-and-what-are-its-benefits>

[7] Q-learning. (2023, March 22). In *Wikipedia*. <https://en.wikipedia.org/wiki/Q-learning>

[8] Parkinson, A. (2019, December 2). *The Epsilon-Greedy Algorithm for Reinforcement Learning*. Analytics Vidhya, Medium. <https://medium.com/analytics-vidhya/the-epsilon-greedy-algorithm-for-reinforcement-learning-5fe6f96dc870>

[9] Patel, D., Hazan, H., Saunders, D. J., Siegelmann, H., & Kozma, R. (2019). Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to ATARI games. *ArXiv*. /abs/1903.11012 *Neural Networks*, *120*, 108–115. <https://doi.org/10.1016/j.neunet.2019.08.009>

[10] Chapman, J., & Lechner, M. (n.d.). *Deep Q-Learning for Atari Breakout*. Keras.io. <https://keras.io/examples/rl/deep_q_network_breakout/>