## List of Figures

# Table of Contents

# CHAPTER - 1

## 1. INTRODUCTION

In video games, a **bot** is a type of artificial intelligence (AI)–based expert system software that plays a video game in the place of a human. Bots are used in a variety of video game genres for a variety of tasks. The latter may be used to automate a repetitive and tedious task like farming.

Bots written for first-person shooters usually try to mimic how a human would play a game. Advanced bots feature machine learning for dynamic learning of patterns of the opponent as well as dynamic learning of previously unknown maps – whereas more trivial bots may rely completely on lists of waypoints created for each map by the developer, limiting the bot to play only maps with said waypoints. Bots may be static, dynamic, or both. Static bots are designed to follow pre-made waypoints for each level or map. These bots need a unique waypoint file for each map.

## 1.1 Problem Definition

- Model-free: In this type of ML algorithm, the RL create a model of the training steps.
- Policy-Based: A policy is the Bot's (Agent) strategy to take the next steps. It's basically represented as a probability distribution of the available actions of the Bot. The action which has the maximum probability becomes the next step's action for the Bot.
- On-Policy approach: This means that the agent requires fresh data to be obtained from the maze environment. The algorithm doesn't dwell on old historic data.

## 1.2 Project Overview

This project is a Q-learning application that solves a puzzle using Deep Learning, which is a part of a broader family of Machine Learning and also a subset of Artificial Intelligence and the Backtracking Algorithm, which is a popular recursive algorithm for solving the puzzle.

## 1.3 HARDWARE SPECIFICATION:

| | | |
|---|---|---|
| Processor | : | Intel i3 (or above) |
| Processor Speed | : | 1000MHz to 2000MHz |
| RAM | : | 8 GB to 16 GB |
| Hard Disk | : | 2GB to 30GB |

## 1.4 SOFTWARE SPECIFICATION:

| | | |
|---|---|---|
| Language | : | Python |
| Algorithms and Tools<br>Cuda | : | Machine Learning (Spyder tool), Anaconda, |

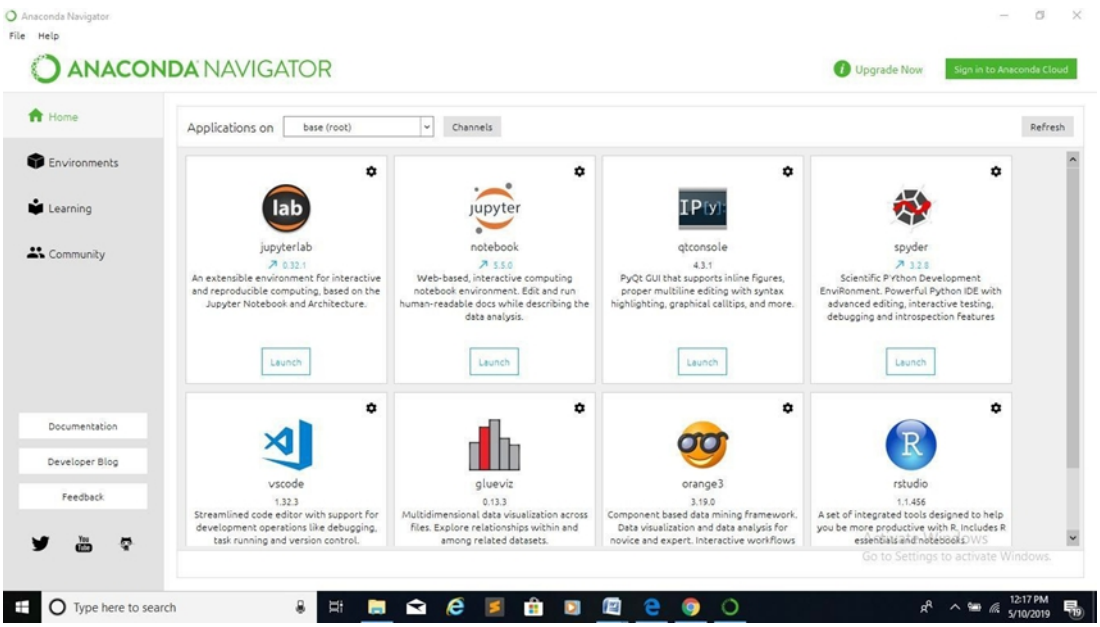



**Fig. 1.1 Anaconda Navigator**

| | | |
|---|---|---|
| Operating System | : | MS Windows XP and above |
| RAM | : | 8 GB |

### 1.4.1 Anaconda Navigator

Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda® distribution that allows you to launch applications and easily manage conda packages, environments and channels without using command-line commands. Navigator can search for packages on Anaconda Cloud or in a local Anaconda Repository. It is available for Windows, macOS, and Linux.

### 1.4.2 Spyder

Spyder is an open source cross-platform (IDE) for scientific programming in the Python language. Spyder integrates with a number of prominent packages in the scientific including NumPy, SciPy, Matplotlib, pandas, IPython, SymPy and Cython, as well as other open source software.[ It is released under the MIT license.

Initially created and developed by Pierre Raybaut in 2009, since 2012 Spyder has been maintained and continuously improved by a team of scientific Python developers and the community.

### 1.4.3 Cuda Toolkit

**CUDA** (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia.[1] It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

# CHAPTER-2
# LITERATURE SURVEY

## 2.1 EXISTING SYSTEM

Very few systems use the available advance deep learning purposes and even if they do, they are restricted by the large number of association rules that apply. Training of the bot solely depends on the algorithms and hardware for deep learning.

### 2.1.1 DRAWBACKS IN EXISTING SYSTEM:

- Detection is not possible at an earlier stage.
- In the existing system, practical use of various collected data is time consuming.
- Big Data applications are featured with autonomous sources and decentralized Controls, aggregating distributed data sources to a centralized site for mining is systematically prohibitive due to the potential transmission cost and privacy concerns.

## 2.2 PROPOSED SYSTEM

The proposed system acts as a decision support system and will prove to be an aid for the automated devices. The algorithm Markov Decision Process, Decision Tress and Q-Learning uses to train the agent to explore the environment and gain experience through exploration and exploitation.

### 2.2.1 ADVANTAGES OF PROPOSED SYSTEM:

- Very fast and accurate.
- No need of any extra manual effort.
- Reinforcement learning and Q-Learning are very flexible and is widely used in various domains with high rate of success.

## 2.3 FEASIBILITY STUDY

Once the problem is clearly understood, the next step is to conduct feasibility study, which is high level capsule version of the entered systems and design process. The objective is to determine whether or not the proposed system is feasible. The three tests of feasibility have been carried out.

- Technical Feasibility
- Economical Feasibility
- Operational Feasibility

### 2.3.1 TECHNICAL FEASIBILITY

In Technical Feasibility study, one must test whether the proposed system can be developed using existing technology or not. It is planned to implement the proposed system using java technology. It is evident that the necessary hardware and software are available for development and implementation of the proposed system. Hence, the solution is technically feasible.

### 2.3.2 ECONOMICAL FEASIBILITY

As part of this, the costs and benefits associated with the proposed system compared and the project is economically feasible only if tangible or intangible benefits outweigh costs. The system development costs will be significant. So, the proposed system is economically feasible.

### 2.3.3 OPERATIONAL FEASIBILITY

It is a standard that ensures interoperability without stifling competition and innovation among users, to the benefit of the public both in terms of cost and service quality. The proposed system is acceptable to users. So, the proposed system is operationally feasible.

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages.

The main properties of the Python, which made it so popular in Data Science, are as follows:

A simple and easy to learn language which achieves result in fewer lines of code than other similar languages like R. Its simplicity also makes it robust to handle complex scenarios with minimal code and much less confusion on the general flow of the program.

It is cross platform, so the same code works in multiple environments without needing any change. That makes it perfect to be used in a multi-environment setup easily.

It executes faster than other similar languages used for data analysis like R and MATLAB.

Its excellent memory management capability, especially garbage collection makes it versatile in gracefully managing very large volume of data transformation, slicing, dicing and visualization.

Most importantly Python has got a very large collection of libraries which serve as special purpose analysis tools. For example – the NumPy package deals with scientific computing and its array needs much less memory than the conventional python list for managing numeric data. And the number of such packages is continuously growing.

Python has packages which can directly use the code from other languages like Java or C. This helps in optimizing the code performance by using existing code of other languages, whenever it gives a better result.

Data science is the process of deriving knowledge and insights from a huge and diverse set of data through organizing, processing and analyzing the data. It involves many different disciplines like mathematical and statistical modelling, extracting data from it source and applying data visualization techniques. Often it also involves handling big data technologies to gather both structured and unstructured data.

## Recommendation systems

As online shopping becomes more prevalent, the e-commerce platforms are able to capture

users shopping preferences as well as the performance of various products in the market. This leads to creation of recommendation systems which create models predicting the shoppers needs and show the products the shopper is most likely to buy.

## Financial Risk management

The financial risk involving loans and credits are better analyzed by using the customers past spend habits, past defaults, other financial commitments and many socio-economic indicators. These data is gathered from various sources in different formats. Organizing them together and getting insight into customers profile needs the help of Data science. The outcome is minimizing loss for the financial organization by avoiding bad debt.

## Improvement in Health Care services

The health care industry deals with a variety of data which can be classified into technical data, financial data, patient information, drug information and legal rules. All this data need to be analyzed in a coordinated manner to produce insights that will save cost both for the health care provider and care receiver while remaining legally compliant.

## Computer Vision

The advancement in recognizing an image by a computer involves processing large sets of image data from multiple objects of same category. For example, face recognition. These data sets are modelled, and algorithms are created to apply the model to newer images to get a satisfactory result. Processing of these huge data sets and creation of models need various tools used in Data science.

## Efficient Management of Energy

As the demand for energy consumption soars, the energy producing companies need to manage the various phases of the energy production and distribution more efficiently. This involves optimizing the production methods, the storage and distribution mechanisms as well as studying the customers consumption patterns. Linking the data from all these sources and deriving insight seems a daunting task. This is made easier by using the tools of Machine Learning.

## Machine learning

Machine Learning is an idea to learn from examples and experience, without being explicitly programmed. Instead of writing code, you feed data to the generic algorithm, and it builds logic based on the data given. For example, one kind of algorithm is a classification algorithm. It can put data into different groups. The classification algorithm used to detect handwritten alphabets could also be used to classify emails into spam and not-spam.

–A computer program is said to learn from experience E with some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.‖ -Tom M. Mitchell.

## Consider playing checkers.

E = the experience of playing many games of checkers T = the

task of playing checkers.

P = the probability that the program will win the next game.

## Examples of Machine Learning

There are many examples of machine learning. Here are a few examples of classification problems where the goal is to categorize objects into a fixed set of categories.

**Face detection**: Identify faces in images (or indicate if a face is present).

**Email filtering**: Classify emails into spam and not-spam.

**Medical diagnosis**: Diagnose a patient as a sufferer or non-sufferer of some disease.

**Weather prediction**: Predict, for instance, whether or not it will rain tomorrow.

## NEED OF MACHINE LEARNING

Machine Learning is a field which is raised out of Artificial Intelligence (AI). Applying AI, we wanted to build better and intelligent machines. But except for few mere tasks such as finding the shortest path between point A and B, we were unable to program more complex and constantly evolving challenges. There was a realisation that the only way to be able to achieve this task was to let machine learn from itself. This sounds similar to a child learning from itself. So, machine learning was developed as a new capability for computers. And now machine learning is present in so many segments of technology, that we don‘t even realize it while using it.
Finding patterns in data on planet earth is possible only for human brains. The data being very massive, the time taken to compute is increased, and this is where Machine Learning comes into action, to help people with large data in minimum time.

If big data and cloud computing are gaining importance for their contributions, machine learning as technology helps analyze those big chunks of data, easing the task of data scientists in an automated process and gaining equal importance and recognition.

The techniques we use for data mining have been around for many years, but they were not effective as they did not have the competitive power to run the algorithms. If you run deep learning with access to better data, the output we get will lead to dramatic breakthroughs which is machine learning.

**Kinds of Machine Learning**

There are three kinds of Machine Learning Algorithms.

a. Supervised Learning

b. Unsupervised Learning

c. Reinforcement Learning

## Supervised Learning

A majority of practical machine learning uses supervised learning. In supervised learning, the system tries to learn from the previous examples that are given. (On the other hand, in unsupervised learning, the system attempts to find the patterns directly from the example given.)

Speaking mathematically, supervised learning is where you have both input variables (x) and output variables(Y) and can use an algorithm to derive the mapping function from the input to the output.

The mapping function is expressed as Y = f(X).

## Unsupervised Learning

In unsupervised learning, the algorithms are left to themselves to discover interesting structures in the data.

Mathematically, unsupervised learning is when you only have input data (X) and no corresponding output variables. This is called unsupervised learning because unlike supervised learning above, there are no given correct answers and the machine itself finds the answers.

Unsupervised learning problems can be further divided into association and clustering problems.

## Reinforcement Learning

A computer program will interact with a dynamic environment in which it must perform a goal (such as playing a game with an opponent or driving a car). The program is provided feedback in terms of rewards and punishments as it navigates its problem space.

Using this algorithm, the machine is trained to make specific decisions. It works this way: the machine is exposed to an environment where it continuously trains itself using trial and error method.

# CHAPTER-3
# LIBRARIES DESCRIPTION

## 3.1 PANDAS

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data. In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data. Prior to Pandas, Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyse. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

### 3.1.1 Key Features of Pandas

1. Fast and efficient Data Frame object with default and customized indexing.
2. Tools for loading data into in-memory data objects from different file formats.
3. Data alignment and integrated handling of missing data.
4. Reshaping and pivoting of date sets.
5. Label-based slicing, indexing and sub setting of large data sets.
6. Columns from a data structure can be deleted or inserted.
7. Group by data for aggregation and transformations.
8. High performance merging and joining of data.
9. Time Series functionality.

## 3.2 NUMPY

NumPy is a Python package. It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

**Numeric**, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open source project.

### 3.2.1 Operations using NumPy

Using NumPy, a developer can perform the following operations −

1. Mathematical and logical operations on arrays.
2. Fourier transforms and routines for shape manipulation.
3. Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

## 3.3 DATA VISULIZATION WITH SEABORN AND MATPLOTLIB

In the world of Analytics, the best way to get insights is by visualizing the data. Data can be visualized by representing it as plots which is easy to understand, explore and grasp. Such data helps in drawing the attention of key elements. To analyse a set of data using Python, we make use of Matplotlib, a widely implemented 2D plotting library. Likewise, Seaborn is a visualization library in Python. It is built on top of Matplotlib.

### 3.3.1 Seaborn Vs Matplotlib

It is summarized that if Matplotlib ―tries to make easy things easy and hard things possible‖, Seaborn tries to make a well-defined set of hard things easy too.‖

Seaborn helps resolve the two major problems faced by Matplotlib; the problems are –

- Default Matplotlib parameters
- Working with data frames

### 3.3.2 Important Features of Seaborn

Seaborn is built on top of Python's core visualization library Matplotlib. It is meant to serve as a complement, and not a replacement. However, Seaborn comes with some very important features. Let us see a few of them here. The features help in −

1. Built in themes for styling matplotlib graphics
2. Visualizing univariate and bivariate data
3. Fitting in and visualizing linear regression models
4. Plotting statistical time series data
5. Seaborn works well with NumPy and Pandas data structures
6. It comes with built in themes for styling

Matplotlib graphics This is how a plot looks with the

defaults Matplotlib –

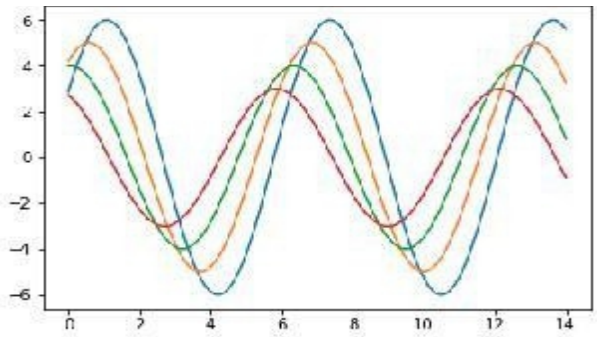



**Fig. 3.1: - Matplotlib plot view**

To change the same plot to Seaborn defaults, use the **set ()** function –

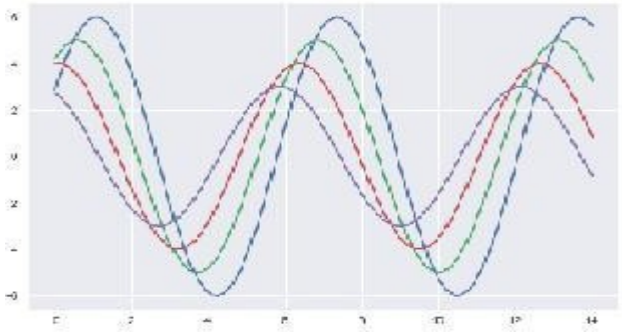


**Fig. 3.2: - Seaborn plot view**

## 3.4 Scikit-Learn IN MACHINE LEARNING

Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python.

It is licensed under a permissive simplified BSD license and is distributed under many Linux distributions, encouraging academic and commercial use.

The library is built upon the SciPy (Scientific Python) that must be installed before you can use scikit-learn. This stack that includes:

**NumPy**: Base n-dimensional array package

**SciPy**: Fundamental library for scientific computing

**Matplotlib**: Comprehensive 2D/3D plotting

**I Python**: Enhanced interactive console

**Sympy**: Symbolic mathematics

**Pandas**: Data structures and analysis

Some popular groups of models provided by scikit-learn include:

**Clustering**: for grouping unlabelled data such as KMeans.

**Cross Validation**: for estimating the performance of supervised models on unseen

data.

**Datasets**: for test datasets and for generating datasets with specific properties for investigating model behaviour.

**Dimensionality Reduction**: for reducing the number of attributes in data for summarization, visualization and feature selection such as Principal component analysis.

**Ensemble methods**: for combining the predictions of multiple supervised models.

**Feature extraction**: for defining attributes in image and text data.

**Feature selection**: for identifying meaningful attributes from which to create

Supervised models.

**Parameter Tuning**: for getting the most out of supervised models.

**Manifold Learning**: For summarizing and depicting complex multi-dimensional data.

## 3.5 DIFFERENCE BETWEEN MACHINE LEARNING AND DEEP LEARNING

Deep learning is a specialized form of machine learning. A machine learning workflow starts with relevant features being manually extracted from images. The features are then used to create a model that categorizes the objects in the image. With a deep learning workflow, relevant features are automatically extracted from images. In addition, deep learning performs "end-to-end learning" – where a network is given raw data and a task to perform, such as classification, and it learns how to do this automatically.

Another key difference is deep learning algorithms scale with data, whereas shallow learning converges. Shallow learning refers to machine learning methods that plateau at a certain level of performance when you add more examples and training data to the network.

A key advantage of deep learning networks is that they often continue to improve as the size of your data increases.

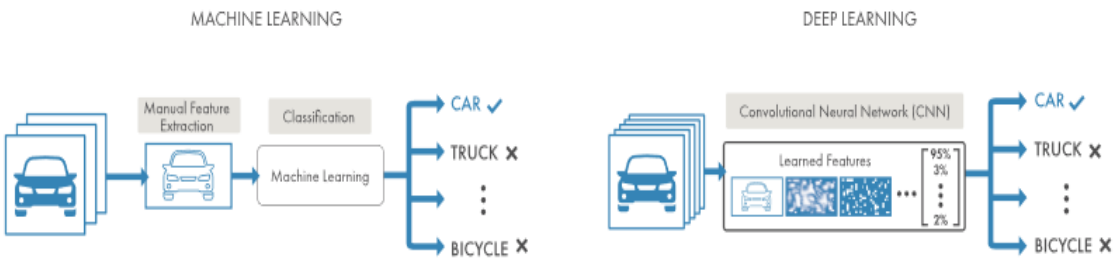



Figure 3. Comparing a machine learning approach to categorizing vehicles (left) with deep learning (right).

In machine learning, you manually choose features and a classifier to sort images.
With deep learning, feature extraction and modeling steps are automatic.

# CHAPTER - 4

# METHODOLOGY USED

## 4.1 OpenAI Gym

So, as mentioned by using Python and OpenAI Gym to develop our reinforcement learning algorithm. The Gym library is a collection of environments that we can use with the reinforcement learning algorithms we develop.

Gym has a ton of environments ranging from simple text based games to Atari games like Breakout and Space Invaders. The library is intuitive to use and simple to install. Just run pip install gym.


We'll be making use of Gym to provide us with an environment for a simple game called Frozen Lake. We'll then train an agent to play the game using Q-learning, and we'll get a playback of how the agent does after being trained.

## 4.2 FROZEN LAKE

In winter you and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend. The surface is described using a grid like the following:

```
SFFF
FHFH
FFFH
HFFG
```

This grid is our environment where S is the agent's starting point, and it's safe. F represents the frozen surface and is also safe. H represents a hole, and if our agent steps in a hole in the middle of a frozen lake, well, that's not good. Finally, G represents the goal, which is the space on the grid where the prized frisbee is located.

The agent can navigate left, right, up, and down, and the episode ends when the agent reaches the goal or falls in a hole. It receives a reward of one if it reaches the goal, and zero otherwise.

| State | Description | Reward |
|---|---|---|
| S | Agent's starting point - safe | 0 |
| F | Frozen surface - safe | 0 |
| H | Hole - game over | 0 |
| G | Goal - game over | 1 |

Our agent has to navigate the grid by staying on the frozen surface without falling into any holes until it reaches the frisbee. If it reaches the frisbee, it wins with a reward of plus one. If it falls in a hole, it loses and receives no points for the entire episode.

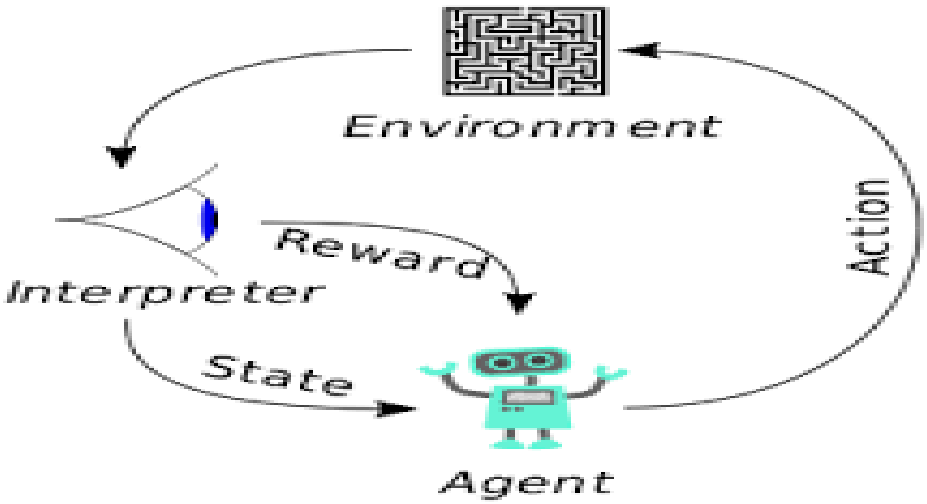
## 4.3 ENVIRONMENT

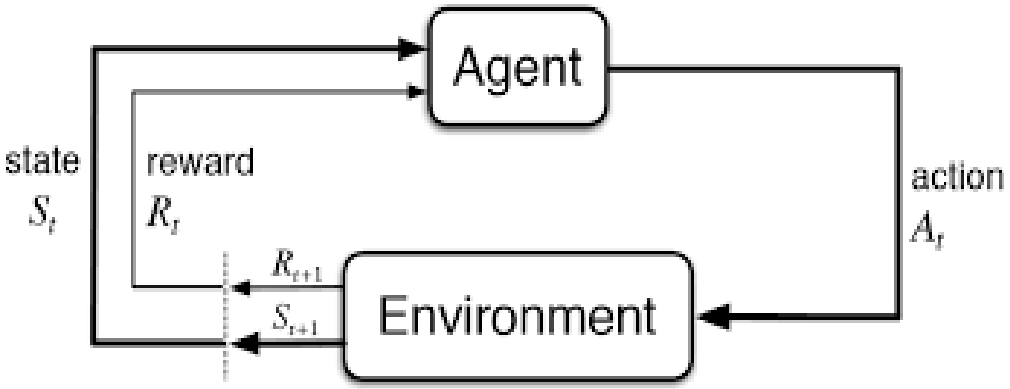
Next, to create our environment, we just call gym.make() and pass a string of the name of the environment we want to set up. We'll be using the environment FrozenLake-v0. All the environments with their corresponding names you can use here are available on Gym's website.

env = gym.make("FrozenLake-v0")

With this env object, we're able to query for information about the environment, sample states and actions, retrieve rewards, and have our agent navigate the frozen lake. That's all made available to us conveniently with Gym.

## 4.4 Flowchart/DFDs/EFDs:



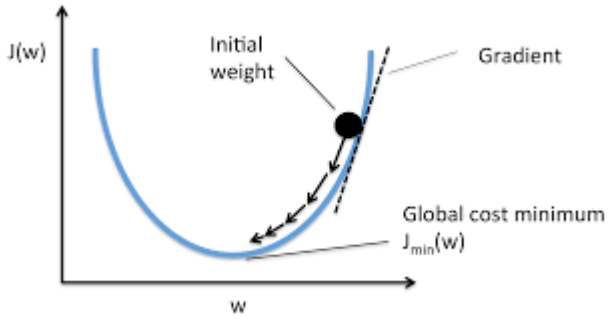




## 4.5 ALGORITHMS

Algorithms that may come in handy for the research and development of this project.

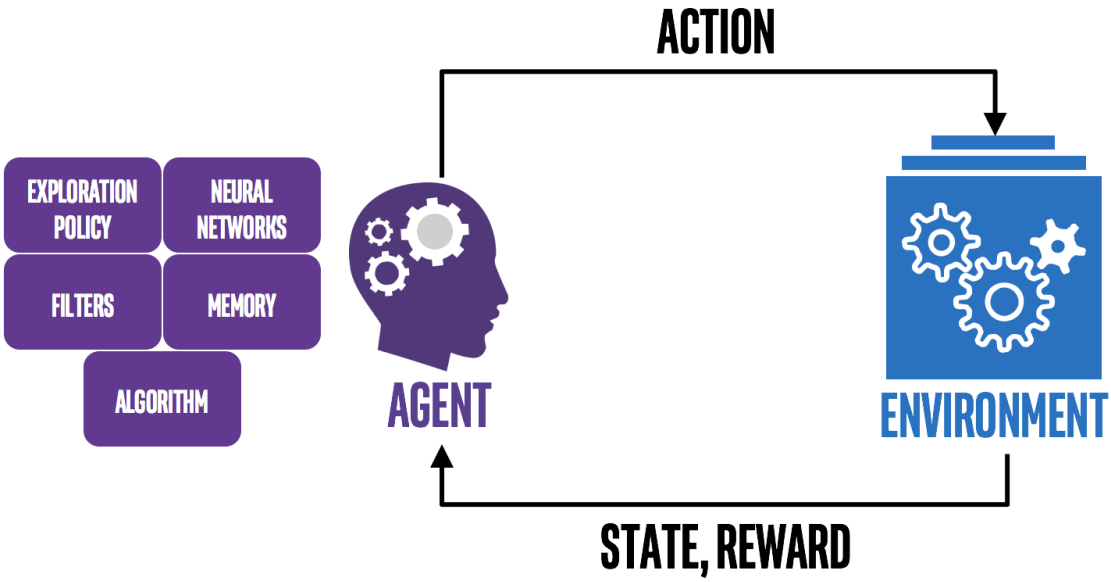
1. **Stochastic Gradient Descent**

**Stochastic gradient descent** (often abbreviated **SGD**) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub differentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in big data applications this reduces the computational burden, achieving faster iterations in trade for a slightly lower convergence rate.



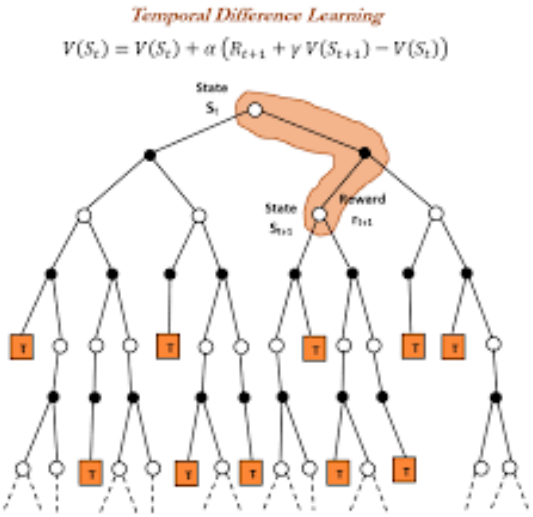

## 2. Reinforcement Learning

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward.
Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.



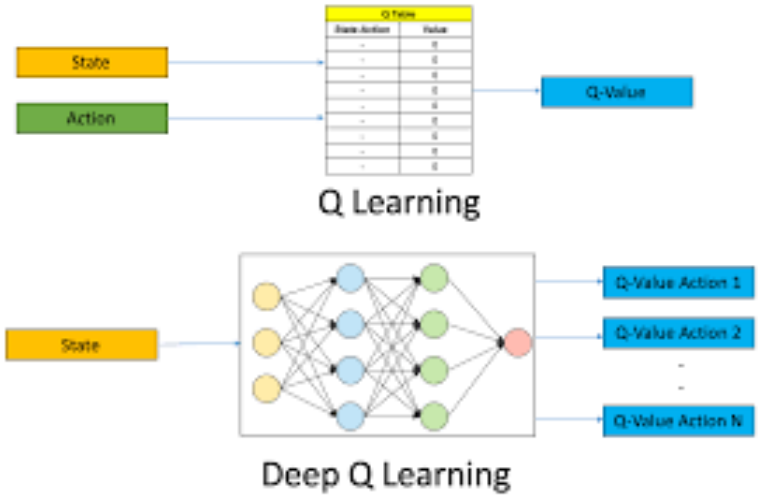

### 3. Temporal Difference Learning

It is an approach to learning how to predict a quantity that depends on future values of a given signal. The name TD derives from its use of changes, or differences, in predictions over successive time steps to drive the learning process. The prediction at any given time step is updated to bring it closer to the prediction of the same quantity at the next time step. It is a supervised learning process in which the training signal for a prediction is a future prediction. TD algorithms are often used in reinforcement learning to predict a measure of the total amount of reward expected over the future, but they can be used to predict other quantities as well. Continuous-time TD algorithms have also been developed.



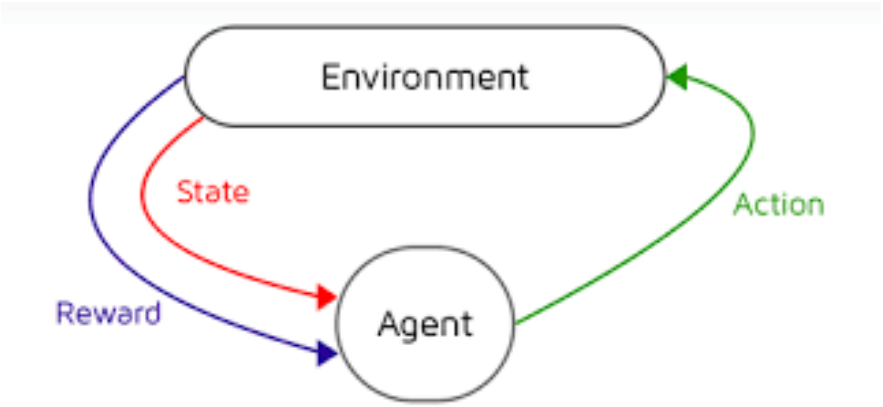

## 4. Q-Learning

It is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.



## 5. Markov Decision Process

A **Markov decision process** (**MDP**) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying optimization problems solved via dynamic programming and reinforcement learning. MDPs were known at least as early as the 1950s;[1] a core body of research on Markov decision processes resulted from Ronald Howard's 1960 book, *Dynamic Programming and Markov Processes*.[2] They are used in many disciplines, including robotics, automatic control, economics and manufacturing. The name of MDPs comes from the Russian mathematician Andrey Markov as they are an extension of the Markov chains.



## 4.6 Design and Test process:

## Setting up Frozen Lake in code

## Libraries

First we're importing all the libraries we'll be using. Not many, really... Numpy, gym, random, time, and clear_output from Ipython's display.

```
import numpy as np
import gym
import random
import time
from IPython.display import clear_output
```

## Creating the environment

Next, to create our environment, we just call gym.make() and pass a string of the name of the environment we want to set up. We'll be using the environment FrozenLake-v0. All the environments with their corresponding names you can use here are available on Gym's website.

```
env = gym.make("FrozenLake-v0")
```

With this env object, we're able to query for information about the environment, sample states and actions, retrieve rewards, and have our agent navigate the frozen lake. That's all made available to us conveniently with Gym.

## Creating the Q-table

To construct our Q-table, and initialize all the Q-values to zero for each state-action pair.

Remember, the number of rows in the table is equivalent to the size of the state space in the environment, and the number of columns is equivalent to the size of the action space. We can get this information using using env.observation_space.n and env.action_space.n, as shown below. We can then use this information to build the Q-table and fill it with zeros.

```
action_space_size = env.action_space.n
state_space_size = env.observation_space.n

q_table = np.zeros((state_space_size, action_space_size))
```

Q-table!

print(q_table)

```
[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]
```

## Initializing Q-learning parameters

Now, we're going to create and initialize all the parameters needed to implement the Q-learning algorithm.

```
num_episodes = 10000
max_steps_per_episode = 100

learning_rate = 0.1
discount_rate = 0.99

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.01
```

Let's step through each of these.

First, with num_episodes, we define the total number of episodes we want the agent to play during training. Then, with max_steps_per_episode, we define a maximum number of steps that our agent is allowed to take within a single episode. So, if by the one-hundredth step, the agent hasn't reached the frisbee or fallen through a hole, then the episode will terminate with the agent receiving zero points.

Next, we set our learning_rate, which was mathematically shown using the symbol $\alpha$

We also set our discount_rate, as well, which was represented with the symbol $\gamma$

previously.

Now, the last four parameters are all for related to the exploration-exploitation trade-off we talked about last time in regards to the epsilon-greedy policy. We're initializing our exploration_rate to 1 and setting the max_exploration_rate to 1 and a min_exploration_rate to 0.01. The max and min are just bounds to how large or small our exploration rate can be. Remember, the exploration rate was represented with the symbol $\epsilon$

Lastly, we set the exploration_decay_rate to 0.01 to determine the rate at which the exploration_rate will decay.

Now, all of these parameters can change. These are the parameters you'll want to play with and tune yourself to see how they influence and change the performance of the algorithm.

## Coding the Q-learning algorithm training loop

First, we create this list to hold all of the rewards we'll get from each episode. This will be so we can see how our game score changes over time. We'll discuss this more in a bit.

```
rewards_all_episodes = []
```

In the following block of code, we'll implement the entire Q-learning algorithm we discussed in detail in a couple posts back. When this code is executed, this is exactly where the training will take place. This first for-loop contains everything that happens within a single episode. This second nested loop contains everything that happens for a single time-step.

```
# Q-learning algorithm
for episode in range(num_episodes):
    # initialize new episode params

    for step in range(max_steps_per_episode):
        # Exploration-exploitation trade-off
```

```
        # Take new action
        # Update Q-table
        # Set new state
        # Add new reward

    # Exploration rate decay
    # Add current episode reward to total rewards list
```

## FOR EACH EPISODE

Let's get inside of our first loop. For each episode, we're going to first reset the state of the environment back to the starting state.

```
for episode in range(num_episodes):
    state = env.reset()
    done = False
    rewards_current_episode = 0

    for step in range(max_steps_per_episode):
    ...
```

The done variable just keeps track of whether or not our episode is finished, so we initialize it to False when we first start the episode, and we'll see later where it will get updated to notify us when the episode is over.

Then, we need to keep track of the rewards within the current episode as well, so we set rewards_current_episode to 0 since we start out with no rewards at the beginning of each episode.

## FOR EACH TIME-STEP

Now we're entering into the nested loop, which runs for each time-step within an episode. The remaining steps, until we say otherwise, will occur for each time-step.

### Exploration vs. exploitation

```
for step in range(max_steps_per_episode):

    # Exploration-exploitation trade-off
    exploration_rate_threshold = random.uniform(0, 1)
    if exploration_rate_threshold > exploration_rate:
        action = np.argmax(q_table[state,:])
```

```
    else:
        action = env.action_space.sample()
    ...
```

For each time-step within an episode, we set our exploration_rate_threshold to a random number between 0 and 1. This will be used to determine whether our agent will explore or exploit the environment in this time-step.

If the threshold is greater than the exploration_rate, which remember, is initially set to 1, then our agent will exploit the environment and choose the action that has the highest Q-value in the Q-table for the current state. If, on the other hand, the threshold is less than or equal to the exploration_rate, then the agent will explore the environment, and sample an action randomly.

## Taking Action

```
new_state, reward, done, info = env.step(action)
```

After our action is chosen, we then take that action by calling step() on our env object and passing our action to it. The function step() returns a tuple containing the new state, the reward for the action we took, whether or not the action ended our episode, and diagnostic information regarding our environment, which may be helpful for us if we end up needing to do any debugging.

## Update the Q-value

After we observe the reward we obtained from taking the action from the previous state, we can then update the Q-value for that state-action pair in the Q-table.
Here is the forumla:

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q\left(s',a'\right)\right)}^{\text{learned value}}$$

And here is the same formula in code:

```
# Update Q-table for Q(s,a)
q_table[state, action] = q_table[state, action] * (1 - learning_rate) + \
```

```
    learning_rate * (reward + discount_rate * np.max(q_table[new_state, :]))
```

So, remember, the new Q-value for this state-action pair is a weighted sum of our old value and the "learned value." So we have our new Q-value equal to the old Q-value times one minus the learning rate plus the learning rate times the "learned value," which is the reward we just received from our last action plus the discounted estimate of the optimal future Q-value for the next state action pair.

## Transition to the next state

```
state = new_state
rewards_current_episode += reward
```

Next, we set our current state to the new_state that was returned to us once we took our last action, and we then update the rewards from our current episode by adding the reward we received for our previous action.

```
if done == True:
    break
```

We then check to see if our last action ended the episode for us, meaning, did our agent step in a hole or reach the goal? If the action did end the episode, then we jump out of this loop and move on to the next episode. Otherwise, we transition to the next time-step.

## Exploration rate decay

```
# Exploration rate decay
exploration_rate = min_exploration_rate + \
    (max_exploration_rate - min_exploration_rate) * np.exp(-exploration_decay_rate*episode)
```

Once an episode is finished, we need to update our exploration_rate using exponential decay, which just means that the exploration rate decreases or decays at a rate proportional to its current value. We can decay the exploration_rate using the formula above, which makes use of all the exploration rate parameters.

```
rewards_all_episodes.append(rewards_current_episode)
```

We then just append the rewards from the current episode to the list of rewards from all episodes.

## After all episodes complete

After all episodes are finished, we now just calculate the average reward per thousand episodes from our list that contains the rewards for all episodes so that we can print it out and see how the rewards changed over time.

```
# Calculate and print the average reward per thousand episodes
rewards_per_thousand_episodes =
np.split(np.array(rewards_all_episodes),num_episodes/1000)
count = 1000

print("********Average reward per thousand episodes********\n")
for r in rewards_per_thousand_episodes:
    print(count, ": ", str(sum(r/1000)))
    count += 1000
```

```
********Average reward per thousand episodes********

1000 :  0.16800000000000012
2000 :  0.32800000000000024
3000 :  0.46900000000000036
4000 :  0.5350000000000004
5000 :  0.6580000000000005
6000 :  0.6910000000000005
7000 :  0.6470000000000005
8000 :  0.6550000000000005
9000 :  0.6980000000000005
10000 :  0.7000000000000005
```

From this printout, we can see our average reward per thousand episodes did indeed progress over time. When the algorithm first started training, the first thousand episodes only averaged a reward of 0.16, but by the time it got to its last thousand episodes, the reward drastically improved to 0.7.

## Interpreting the training results

Let's take a second to understand how we can interpret these results. Our agent played 10,000 episodes. At each time step within an episode, the agent received a reward of 1 if it reached

the frisbee, otherwise, it received a reward of 0. If the agent did indeed reach the frisbee, then the episode finished at that time-step.

So, that means for each episode, the total reward received by the agent for the entire episode is either 1 or 0. So, for the first thousand episodes, we can interpret this score as meaning that 16%

of the time, the agent received a reward of 1 and won the episode. And by the last thousand episodes from a total of 10,000, the agent was winning 70%

of the time.

By analyzing the grid of the game, we can see it's a lot more likely that the agent would fall in a hole or perhaps reach the max time steps than it is to reach the frisbee, so reaching the frisbee 70%

of the time isn't too shabby, especially since the agent had no explicit instructions to reach the frisbee. It *learned* that this is the correct thing to do.

SFFF

FHFH

FFFH

HFFG

Lastly, we print out our updated Q-table to see how that has transitioned from its initial state of all zeros.

```
# Print updated Q-table
print("\n\n********Q-table********\n")
print(q_table)
```

```
********Q-table********

[[0.57804676 0.51767675 0.50499139 0.47330103]
[0.07903519 0.16544989 0.16052137 0.45023559]
[0.37592905 0.18333739 0.18905787 0.17227745]
[0.01504804 0.         0.         0.        ]
[0.59422496 0.42787803 0.43837162 0.45604075]
[0.         0.         0.         0.        ]
```

```
[0.1814022  0.13794979 0.31651935 0.09308381]
[0.         0.         0.         0.        ]
[0.43529839 0.32298132 0.36007182 0.64475741]
[0.3369853  0.75303211 0.42246585 0.50627733]
[0.65743421 0.48185693 0.32179817 0.35823251]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.53127669 0.63965638 0.86112718 0.53141807]
[0.68753949 0.94078659 0.76545158 0.71566071]
[0.         0.         0.         0.        ]]
```

## The code to watch the agent play the game

This block of code is going to allow us to watch our trained agent play Frozen Lake using the knowledge it's gained from the training we completed.

```
# Watch our agent play Frozen Lake by playing the best action
# from each state according to the Q-table

for episode in range(3):
    # initialize new episode params

    for step in range(max_steps_per_episode):
        # Show current state of environment on screen
        # Choose action with highest Q-value for current state
        # Take new action

        if done:
            if reward == 1:
                # Agent reached the goal and won episode
            else:
                # Agent stepped in a hole and lost episode

        # Set new state
env.close()
```

## For each episode

```
for episode in range(3):
    state = env.reset()
    done = False
    print("*****EPISODE ", episode+1, "*****\n\n\n\n")
    time.sleep(1)
    ...
```

For each of the three episodes, we first reset the state of our environment, and set done to False. This variable is used for the same purpose as we saw in our training loop last time. It just keeps track whether or not our last action ended the episode.

We then just print to the console what episode we're starting, and we sleep for one second so that we have time to actually read that printout before it disappears from the screen.

Now, we'll move on to the inner loop.

## For each time-step

```
for step in range(max_steps_per_episode):
    clear_output(wait=True)
    env.render()
    time.sleep(0.3)
    ...
```

For each time-step within the episode, we're calling the iPython display function clear_output(), which clears the output from the current cell in the Jupyter notebook. With wait=True, it waits to clear the output until there is another printout to overwrite it. This is all just done so that the notebook and the screen display remain smooth as we watch our agent play.

We then call render() on our env object, which will render the current state of the environment to the display so that we can actually visually see the game grid and where our agent is on the grid. We then sleep again for 300 milliseconds to give us time to actually see the current state of the environment on screen before moving on to the next time step. Don't worry, this will all come together once we view the final product.

```
action = np.argmax(q_table[state,:])
new_state, reward, done, info = env.step(action)
```

We then set our action to be the action that has the highest Q-value from our Q-table for our current state, and then we take that action with env.step(), just like we saw during training.

This will update our new_state, the reward for our action, and whether or not the action completed the episode.

```
if done:
    clear_output(wait=True)
    env.render()
    if reward == 1:
        print("****You reached the goal!****")
        time.sleep(3)
    else:
        print("****You fell through a hole!****")
        time.sleep(3)
        clear_output(wait=True)
    break
```

If our action did end the episode, then we render the environment to see where the agent ended up from our last time-step. If the reward for that action was a 1, then we know that the episode ended because the agent reached the frisbee and won the game. So we print that info to the console. If the reward wasn't a 1, then we know it was alternatively a 0 and that the agent fell through a hole.

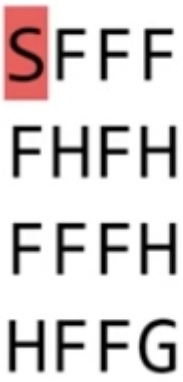
After seeing how the episode ended, we then start a new episode.

Now, if the last action didn't complete the episode, then we skip over the conditional, transition to the new state, and move on to the next time step.

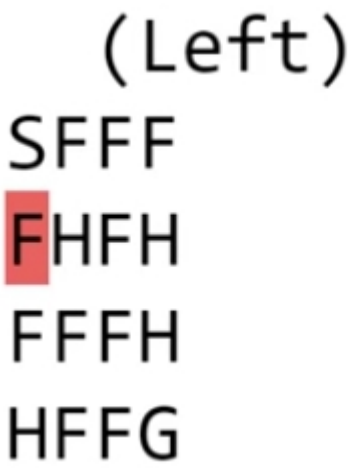
```
state = new_state
env.close()
```

## RESULTS / OUTPUTS

There are 10000 episodes and maximum epochs for each episode is 100. So, the bot will try to explore the environment and choose the path which leads to the final goal.
If the bot fell in hole (H), the episode will over and bot will lost the puzzle.




In above figure bot is at starting point (S) and it observe the environment which path to choose. The bot can move in all direction to choose the path which gives the reward. If it fell in the hole then there is no reward and will try to avoid the negative points.




When the bot takes the action then it move to the next state and update the value of the previous state in the Q-table.


The bot makes that decision again and again which action is better to reach the goal.

(Down)
SFFF
FHFH
FFFH
HFFG

(Left)
SFFF
FHFH
FFFH
HFFG

```
  (Down)
SFFF
FHFH
FFFH
HFFG
****You reached the goal!****
```

In above figure the bot reach the goal. The bot will continue for 9 more times and gain more experience about all the possible path which leads to the goal.

## CONCLUSIONS

In our project, to handle the deep learning and reinforcement learning approach on bot, we can allow the elite episodes to be kept in training iterations for a longer duration. Keeping a low learning rate which helps the agent to learn the optimal path.

This was the example of a simple Q-Learning technique which is an off-policy method. One can implement this method to train a model by unsupervised learning. Once the agent has learnt the environment, the model converges to a point where a random action is not required. We can trace the state at every iteration of every episode to see how the weights vary. The key in the off-policy method is the allowance of the greedy action and the move to a next state. Since we allow this action, the agent will converge faster at the local minima and learn the environment sooner than an on-policy method.

In the future, I plan to add more complex class describing the not-neighbor relationship. Such class will allow us to solve more challenging puzzles. I am extending my method to non-squared fragments with irregularities.

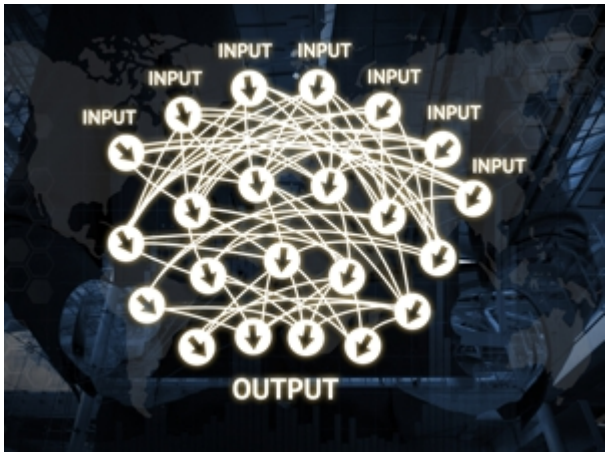
## Contributors

*Vikrant Chaudhary  and Sayem Akhtar*

## APPENDIX

This is an appendix were you will find some of the words, terms and names, found in *Puzzle solving Bot*.

Deep Learning, as a branch of Machine Learning, employs algorithms to process data and imitate the thinking process, or to develop *abstractions*. Deep Learning (DL) uses layers of algorithms to process data, understand human speech, and visually recognize objects. Information is passed through each layer, with the output of the previous layer providing input for the next layer.




The earliest efforts in developing deep learning algorithms date to 1965, when Alexey Grigoryevich Ivakhnenko and Valentin Grigor′evich Lapa used models with polynomial (complicated equations) activation functions, which were subsequently analysed statistically.

**Stochastic gradient descent** (often abbreviated **SGD**) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub differentiable).

TDL is an approach to learning how to predict a quantity that depends on future values of a given signal. The name TD derives from its use of changes, or differences, in predictions over successive time steps to drive the learning process.

The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances.

A **Markov decision process** (**MDP**) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

A majority of practical machine learning uses supervised learning. In supervised learning, the system tries to learn from the previous examples that are given.

In unsupervised learning, the algorithms are left to themselves to discover interesting structures in the data.

**Matplotlib**: Comprehensive 2D/3D plotting

**I Python**: Enhanced interactive console

**Sympy**: Symbolic mathematics

**Pandas**: Data structures and analysis