**Abstract**

Creating systems able to play classic games has been one of the highest challenges of artificial intelligence in the last few years. In this report, a Few elements of Reinforcement learning are discussed. Many inspirations have been used from various papers and related work during the implementation. This report mainly presents an AI-driven Snake game, in simple words, the goal of the project is to make the snake object eat the food object with the help of AI (Reinforcement Learning). A Deep Learning approach is implemented for playing the Snake game. Focus is given to the Neural Network hyperparameters tuning, which constitutes an essential step in the agent design process, to achieve a desired target level of performance.

**Introduction**

At its core, Reinforcement Learning (RL) is one of the machine learning techniques among the trio of fundamental machine learning paradigms, which include supervised and unsupervised learning as its companions. Certainly, one could delve into the realms of reinforcement learning by experimenting with distinct configurations of state spaces and reward mechanisms, coupled with a variety of network architectures, to then scrutinize the performance of the model at hand.

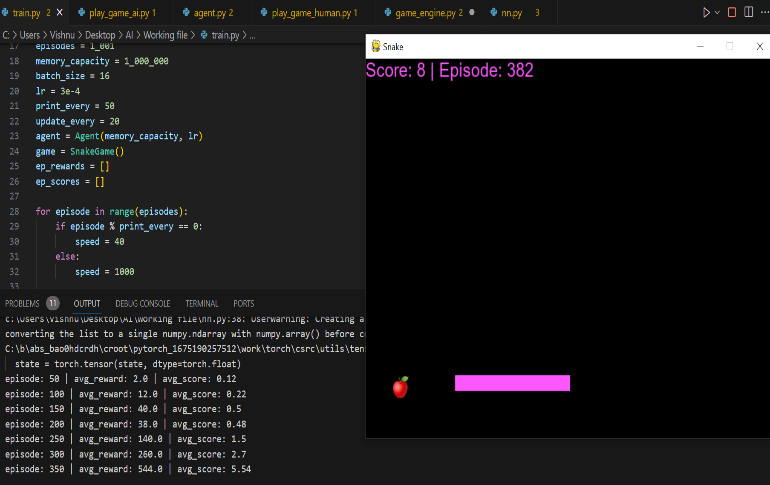
Approaching the environment with a Markov decision process (MDP) framework in mind, this technique doesn't necessitate a deep grasp of the underlying mechanisms controlling the scene. Mastering puzzle-like scenarios filled with decision-making layers and control issues falls within the reach of RL-based methods. This is fascinating because they manage to navigate these tough situations whether they involve solitary participants or numerous actors making choices to their actions, all while lacking detailed knowledge of the sophisticated rules that dictate their environments.

**Proposed System for the game**

Description of the Snake game's environment. The Snake game is composed of an environment, which is the game board, the fruit and the snake, a game points counter, and a total reward. A fruit is spawned any time is eaten, and the reward is given to the agent.

So far, we can describe the game board environment, which changes during each game performed. The first is composed of the game board, where the fruit is spawned, and the player (the agent) is a snake that moves in four possible directions, up down, left, and right. The snake’s body is composed of its head and an element of the snake’s tail. In general, a snake game includes a rule that if the snake collides with the wall, then the game ends. But in our project to exclude the complexity we have removed the feature of the game ending when the touches the wall. In our case, the game can end only if the snake touches itself. In the gaming environment, the game initially starts with a one-piece-long snake. The rules are pointed out next: whenever some fruit is eaten the game points counter is incremented and the tail of the snake last is now one piece longer.

Finally, it gets rewarded if the agent reaches the target position and performs the next actions respectively. We are not only talking about points and punishments, but instruments of learning. Points of rewards are teachable moments as given in DQN, through the neural instruction of agents, and scores are also not mere tallies but glamorous milestones in grading the agents’ algorithmic underpinning of the creature and the admirable artistry of the agent.



This depicts a screenshot during the training phase. In the above scenario, we see that at each interval of training the snake agent is being rewarded, and as the intervals of the episodes increase the reward also increases which means that the snake agent is being trained in the right way.

As the project works on states, The states in the current project are a vector of 8 binary values defined as follows,

**x(k)** = [moving left, moving right, moving up, moving down, food left, food right, food up, food down].

* The snake moves to the x direction and more precise to its right if the x-velocity is equal to 1 and moving to the right will be equal to 1 too.
* The snake moves to the x direction and more precise to its left if the x-velocity is equal to −1 and moving to the left will be equal to 1 too.
* The snake moves to the y direction and more precise downwards if the y-velocity is equal to 1 and moving downwards will be equal to 1 too.
* The snake moves to the y direction and more precise upwards if the y-velocity is equal to −1 and moving downwards will be equal to 1 too.

When it comes to storing memory, A queue of length MAX\_MEMORY is called an experience replay buffer is used to hold (state, action, reward, next\_state,) done at each stage of the snake. The oldest memory in the replay buffer will be removed once it is filled. Given that this algorithm employs both long-term and short-term training techniques, the experience replay buffer is an essential component of the training process. Short-term training is carried out using just very recent memory; that is, all training steps are carried out using the most recent memory. After the snake dies, long-term training utilizing Memory Replay takes place. From this buffer, we select a random length BATCH\_SIZE chunk as a memory, and we train our Deep Q network on this.

**Implementation**

We implemented a Deep Q-network that consisted of a simple neural network. Simple means that it had no hidden layers. The algorithm generally followed Q-learning, except the table of Q values was converted to a single-layer neural network with N outputs. Each of the N outputs corresponds to one of the stwo valid actions in the game. A simple neural network uses N outputs because that is equivalent to the amount of memory needed to store the N Q values in a table. Bellman’s equation is used to update the action-state pair.

**Q(St, At) = (1 − α)Q(St, At) + α(R + γQ(St+1, a))**

We initialize the replay memory buffer for random actions with a uniform random probability distribution equal to the total number of actions in the game. The actions were executed by the game but never learned because a random action isn't based on the learning network. If the action is random, the learning network does not correlate with the random number that was chosen. The memory buffer's primary purpose is to memorize future actions to be taken by the game while the network is being updated. If the network can remember k past actions that were taken, then the future memory will be the previous experience multiplied by k. Below you can see the current training values from our algorithms.

The code structure for our project mainly includes,

**Agent Class**: It is very important to the project becoming the bone of contention for all the decision-making aspects of the AI player. It consists of methods to Retrieve the game state (get\_state), Retrieve action on what the AI will perform(get\_action), and Train the neural network based on the short-term and long-term memory (train\_short\_memory and train\_long\_memory).

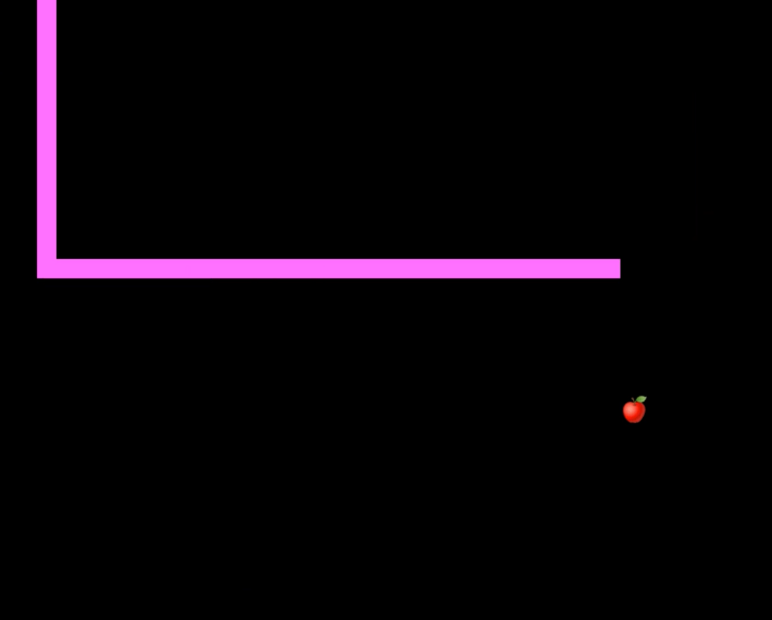
**Game Engine Class**: It engages in performing the main editing for the snake game. The methodology incorporated in this section includes resetting the game, placing foods, playing the game, checking the collisions, updating the user interface, and moving the snake. Incrementing the gameplay experience by adding real apples and obstacles.

**Neural Network and Trainer**: It is a remote control for the neural network in which the AI agent is utilized for making decisions. The Trainer class contains the process of training the neural network in updating the weights of the neural network via feedback from the actual gameplay.

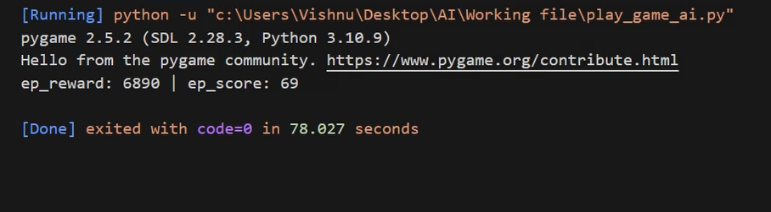
Libraries like Numpy, Pygame, and PyTorch were also used in the implementation stage.

**Results**

The below figure represents the AI game running.



The project has come off with producing a snake Game AI that can play the game.



From the above Snapshot, we can see there are two columns, namely the ‘ep\_reward’ and the ‘ep\_score’ column. The reward helps the agent reach the Target position during the training phase, to present the agent’s behavior. Whereas the ‘ep\_score’ is a pure score of the snake agent from the start till the game ends.

The snake’s agent is trained from scratch without using any GPU to speed up the result, we have manually trained the agent over a period of time.

**References**

[1] Engineer, Python. “Python Snake Game with Pygame - Create Your ... - Youtube.” YouTube, 8 Dec. 2020, <https://www.youtube.com/watch?v=–nsd2ZeYvs>.

[2] Google Snake. <https://www.googlesnake.com/>

[3] Sebastianelli, Alessandro & Tipaldi, Massimo & Ullo, Silvia & Glielmo, Luigi. (2021). A Deep Q-learning-based approach was applied to the Snake game. 10.1109/MED51440.2021.9480232.

[4]<https://cs230.stanford.edu/projects_fall_2021/reports/103085287.pdf>

[5] https://youtu.be/L8ypSXwyBds?feature=shared