Project Paper

Introduction

The project consists of a snake game implementation with a Deep Q Learning Agent on a Reinforcement Learning environment and is trained on neural networks. The repository consists of four files consisting of Snake Environment, implementation of DQNAgent, Test_the_Environment for checking if the environment is working and a training file to train the model.

Snake Environment

Firstly, the Agent's environment in reinforcement learning is the setting in which it interacts and lives. The agent can interact with the environment by taking certain actions, but those actions cannot change the environment's dynamics or laws.

At the start there are three function, encountering_with_food function takes care of situations where the snake has eaten food and hence incrementing the score. Another function called encountering_with_boundaries looks into situations where the snake has encountered a wall or the boundary of the grid as it makes sure that the game is over. Lastly, encountering_with_self function helps in situations where the snake has collided with it’s own body leading to a lose in the game.

The SnakeEnv class is a custom environment that follows the gym interface. It inherits from the gym.Env class to create a new environment for the Snake game. The environment class has built-in gym environment functions, which are essential for interacting with the agent.

First function is the init function where it initializes the action space and the observation space of the environment. The action space is defined using spaces.Discrete, which takes the number of possible actions (4 in this case). The observation space is defined using the spaces.Box method, taking the low and high values and the shape of the observation space.

The step function is responsible for specifying what happens at each step in the game. It takes an action as input and processes the actions accordingly. The function first appends the previous actions to the list, then updates the game display, and calculates the new position of the snake based on the given action. Using cv2, loaded image with different shapes and sizes with respect to snake and food, added a time delay as it was very fast to show the results. The reward variable is returned along with the updated state (observation) and other information in the step() method, which can be used by the learning algorithm to update the policy of the agent and reward equal to -10 will be the punishment.

If the snake encounters the food, the food position is updated, the score is increased, and the snake's length increases. The rewards are calculated based on the Euclidean distance between the snake and the food, and the difference between the total reward and the previous reward is returned.

If the snake encounters a boundary or itself, the game is considered done, and a message with the final score is displayed. The final score normally will show zero at the end since it considers each episode and not necessarily the last episode would be a successful one. The method returns an observation, the total reward, whether the game is done, and an empty info dictionary. A part where Euclidean distance is being calculated from the snake to food and gives a reward of 250 before scaling it down to give better learning outcomes.

The reset method is a built-in gym function that resets the game state when the snake hits a wall or itself. It initializes various variables such as the game state, snake position, food position, score, and rewards. It also sets up the initial observation array with the snake head, food position, snake length, and previous actions.

DQNAgent

A DQN agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. DQN is a variant of Q-learning. The agent's brain is the only distinction between Q-learning and DQN. In Q-learning, the Q-table serves as the agent's brain, but in DQN, the agent's brain is a deep neural network. Deep Q-Learning uses Experience Replay to learn in small batches to avoid skewing the dataset distribution of different states, actions, rewards, and next states.

The DQNAgent class is responsible for implementing the Deep Q-Network (DQN) agent. It has methods for building the neural network model, remembering experiences, selecting actions, and replaying experiences to learn from them.

The __init__ method initializes the agent's state size, action size, memory, and other parameters such as the discount rate (gamma), exploration rate (epsilon), learning rate, and the neural network model.

The _build_model method creates a Sequential neural network model for the DQN agent. It adds fully connected layers with 24 neurons and ReLU activation functions, followed by an output layer with a linear activation function to predict Q values. ReLU is considered optimal for binary operation scenarios where it’s either one way or the other. The model is compiled with mean squared error as the loss function and the Adam optimizer with the specified learning rate. The Adam optimizer is a stochastic gradient descent (SGD) modification that adaptively adjusts the learning rate.

The remember method is responsible for storing experiences in the agent's memory. It takes the state, action, reward, next_state, and done variables as inputs and appends them to the memory deque.

The act method chooses an action based on the current state. It implements an epsilon-greedy exploration strategy, meaning that with a probability of epsilon, a random action is chosen, and otherwise, the action with the highest predicted Q-value is selected.

The replay method is responsible for training the agent using experiences from its memory. It takes a batch size as input and samples a random mini batch of experiences from the memory. The method then iterates over the minibatch, calculating target Q values using the Bellman equation, and trains the model with the given states and Q values. The exploration rate is decayed if it is greater than the minimum value.

Test Environment

A random agent is being used by programming to test the Snake gaming environment. The environment is created by instantiating an object of the SnakeEnv class, which is imported from the "SnakeEnv" module. Then, using a for loop, the code executes 30 episodes of the game to check if the environment is working fine without errors and is ready to be trained.

By using the SnakeEnv object's reset() method, which returns the original observation, the environment is reset at the beginning of each episode. Once the episode is finished (done=True), which occurs when the snake either eats itself or encounters a wall, the while loop continues to run.

Using the env.action_space.sample() method, a random action is selected from the available action space at each iteration of the loop. After taking a step in the environment and returning the next observation, reward, done status, and further information, this action is then provided to the env.step() method.

The observation details the game's current situation, including the snake and food's whereabouts. A scalar value known as the reward represents how well the agent performed in the previous time step. The award is written to the console for each time step in this code.

Train

In the code to train a model based on DQNAgent and the snake environment, Trainer class is used with two functions init and train. In init function, attributes like environment, agent, batch size and scores are initialized.

In train function, environment resets by calling env.reset function after every iteration of the for loop. After resetting, reshaping of the state is done to be compatible with the neural network. Another loop starts where we get the actions of the current state, reshapes the next state for compatibility, checks if the iteration is done and adds the score to the scores list and then plots the performance plot episodes vs scores graph.

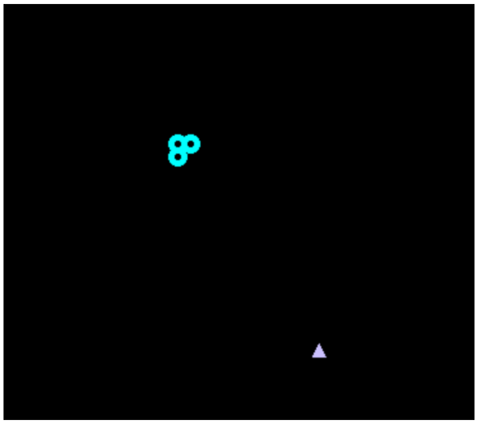
Challenges faced

There was confusion with visualization of the game interface as there are lot of options to explore and I chose pygame library at the start of the project but was facing errors. Then I switched to opencv for visualization which helped a lot with easily accessible functions.

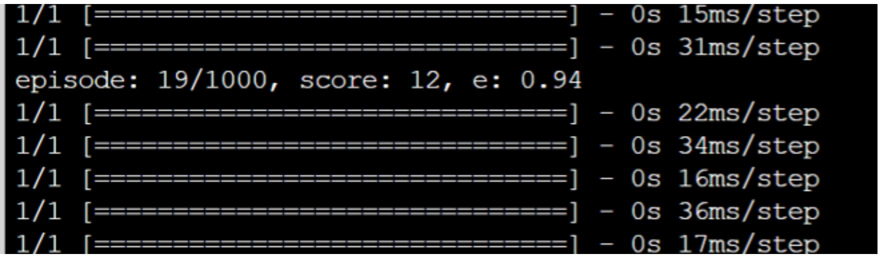
Another confusion was with Atari, firstly I was using this for my gameplay but was facing errors in learning. Then I used Adam optimizer which was very convenient with lesser computation time and required fewer parameter for tuning the model.

Test Results

As shown in the below image, this is what the game window looks like when the model is getting trained. The snake starts with length 3 as set by default with no score or reward.

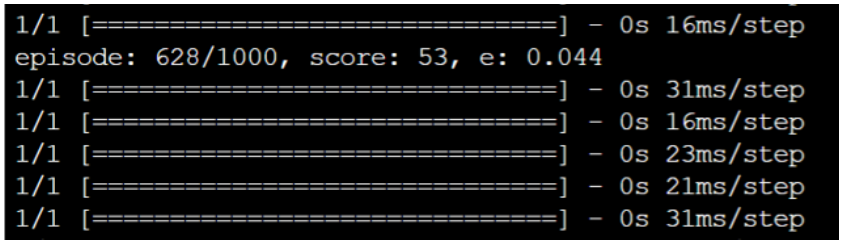


Firstly, the model was trained with 1000 episodes limit but without the Euclidean distance being calculated to the food and the reward was just set up by taking the length of the snake. Then the start of the training was with a bit lower score with decaying epsilon as shown below,



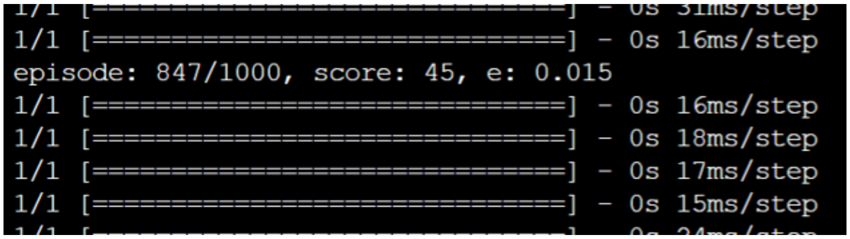
```
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 31ms/step
episode: 19/1000, score: 12, e: 0.94
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 17ms/step
```

After almost 600 episodes, it gives a score which is recorded to be the highest among all the episodes with a score of 53 but the episode almost reached the saturation point of 0.01.

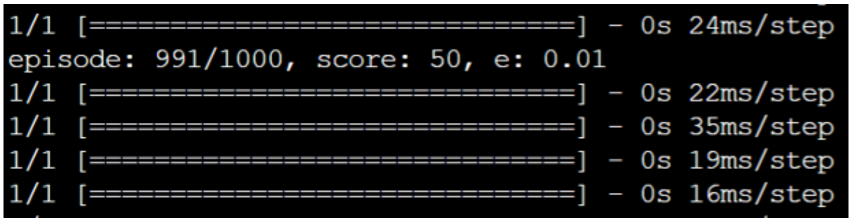


```
1/1 [==============================] - 0s 16ms/step
episode: 628/1000, score: 53, e: 0.044
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 31ms/step
```

But as we go further, there was not much improvement from the above score without Euclidean method for distance calculation. Around episode 847 and 991 the scores were lower than the highest score with epsilon at the saturation point.

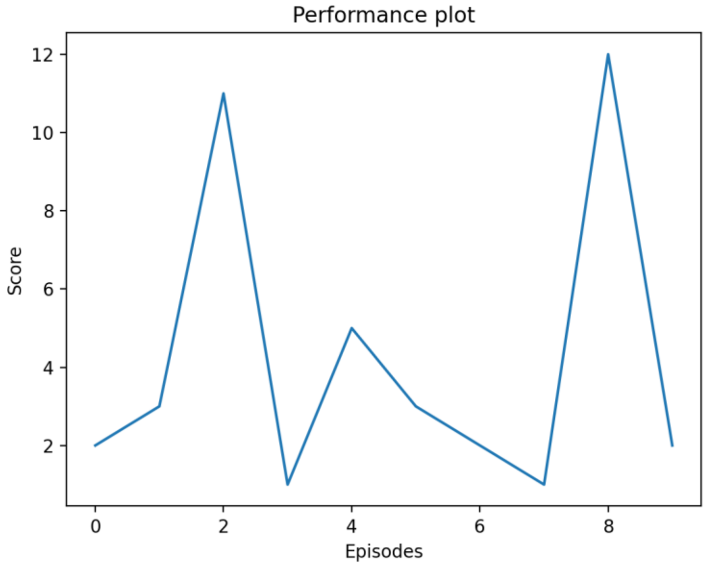


```
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 16ms/step
episode: 847/1000, score: 45, e: 0.015
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 24ms/step
```

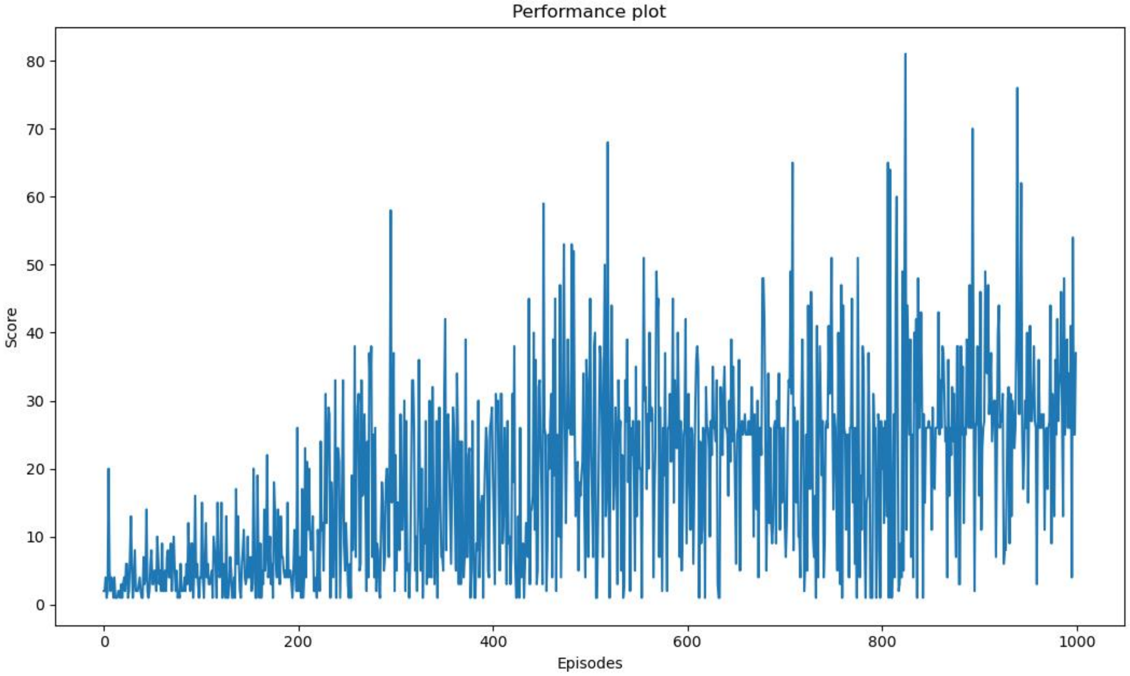


```
1/1 [==============================] - 0s 24ms/step
episode: 991/1000, score: 50, e: 0.01
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 16ms/step
```

After this I tried plots for performance and trained small number of episodes to check the accuracy of plots. Below plot was done on 10 episodes with the highest score of 12 at episode 8,



After this I trained the reinforcement learning model with the Euclidean distance being calculated to the nearest food from the snake which helped in improving the scores a lot. The performance plot is shown below,



From the above illustration, we can see that the highest score is around 80 around episode 800 which is a huge improvement from the previous trials. If we look at the trend, it's an increasing trend with higher scores towards the end of 1000 episodes.