Enhancing Snake Game Strategy with Deep Reinforcement Learning and Advanced Neural Networks

Amber Hasan, Mikiyas Midru, Luis Lobo Borobia
Executive Masters, The University of Texas at Dallas

***Abstract*— We are using Deep Reinforcement Learning with Q-Network and DQN Bellman Algorithm, as well as a recurrent neural network (RNN) with Long Short-Term Memory (LSTM).**

I. Introduction

In our class, we learned Supervised Learning and will learn Unsupervised Learning. We wanted to explore a third type of learning called Reinforcement Learning, by enhancing the popular Snake game.

Deep Reinforcement Learning (DRL) originates from pure Reinforcement Learning (RL), where problems are typically framed as Markov Decision Processes (MDPs). An MDP consists of a set of states S and actions A, with transitions between states governed by transition probabilities P, rewards R, and a discount factor γ. Deep Q-Networks (DQNs) enhance this by using neural networks to approximate the optimal action-value function, allowing RL to handle more complex and high-dimensional state spaces.

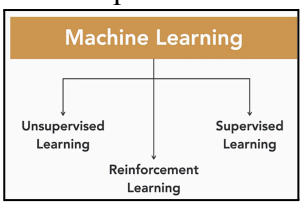

Fig. 1 We are exploring a third type of machine learning. .

II. Background Work

A. Creating the Basic Snake Game

Before adding Machine Learning, we first started with the basic snake game without any machine learning involved. A snake head can go three directions: straight, left, or right since a user can move with the left arrow, right arrow, or nothing. The snake can also eat the food or hit the borders or hit itself. We use QNet framework for a nicer UI.

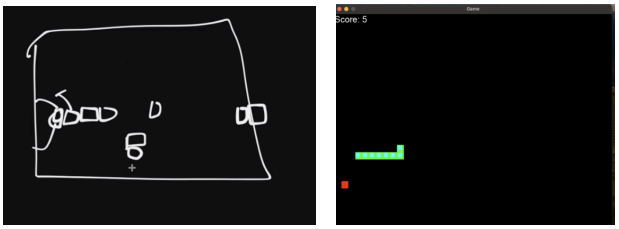

Fig. 2 Our brainstorming sketch of the game and our initial game.

B. Architecture

We have an agent responsible for controlling the game, loading the settings, and providing a report of its actions. We use a dispatcher to communicate specific instructions to the agent, like which activation function to use (e.g., ReLU), which optimizer to apply (e.g., Adam), the learning rate, the number of iterations, and other relevant parameters.

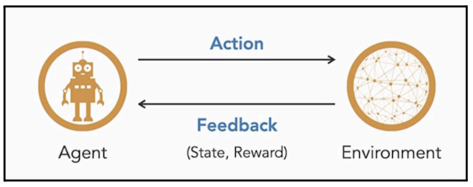

Fig. 3 The basic architecture we are aiming for.

C. Dispatcher UI with QT to control

We used the QT framework to create our dispatcher in training_setting.py so that we can control the max memory size and the speed of the snake moving.

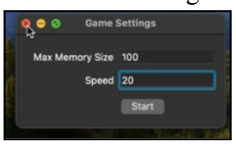

Fig. 4 The dispatcher is the initial component that will start the game.

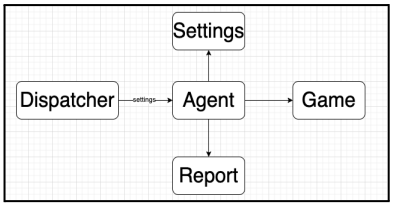

Fig. 5 Diagram showing how dispatcher interacts with the other components.

D. Agent

The agent is the entity that interacts with the environment to learn an optimal policy for achieving a specific goal. We have our state, long memory, short memory, action, and training loop.

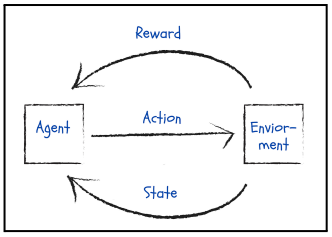

Fig. 6 Diagram showing how the general concept works.

E. Gymnasium Framework

Once we had the DQN game working, we put it into Gymnasium to have a better UI.

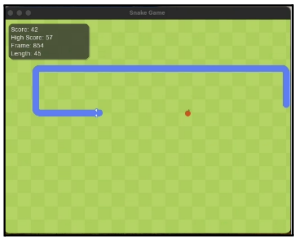

Fig. 7 The improved UI that we got for our snake using Gymnasium.

III. STUDY OF THE TECHNIQUE

So far, we've just had a brainless worm. Now we are going to be adding the brain in brain.py, which will hold the actual neural network. The agent will use the trainer (in brain.py) which will use the DQN (in brain.py)

A. Theoretical Study of the Algorithm

1. Bellman Equation and "SARSA-Max"

The Bellman Equation is a fundamental principle in reinforcement learning that defines the relationship between the value of a state-action pair and the values of subsequent state-action pairs. It provides a recursive method to calculate the expected cumulative reward for an agent's actions.

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

Fig. 8 The Bellman equation.

***Q**: The Q-value or quality value.*
***s**: The current state.*
***a**: The action taken from the current state.*
***Q(s, a)** represents the expected cumulative reward of taking action aaa in state sss and then following the optimal policy.*
***r**: The reward received after taking action*
***gamma**: discount factor between 0 and 1*

2. Deep Q-Network Algorithm and "SARSA-Max"

$$Q(s,a) = \overbrace{Q(s,a)}^{\text{Old value}} + \overbrace{\alpha}^{\text{learning rate}}(\overbrace{r}^{\text{reward}} + \overbrace{\gamma max Q(s'a')}^{\text{optimal future value}} - \overbrace{Q(s,a)}^{\text{Old value}})$$

Fig. 9 The Deep Q-Network Algorithm.

In our Snake game implementation, we use SARSA Max (Q-Learning) to train the agent. SARSA Max is a reinforcement learning algorithm that updates the Q-values based on the maximum expected future rewards. This method allows the agent to learn the optimal policy by considering the best possible action in future states. SARSA-MAX uses the greedy policy to select the second and following actions.

Steps for DQN:
1. Construct Q-table using equiprobable random policies (Neural Network Initiatlization)
2. Improve policy using Bellman equation
3. Update Q-table (Neural Network Training)

3. Off-Policy Temporal Difference

A policy is a strategy that defines the actions an agent takes in various states to maximize cumulative rewards. It acts as a blueprint for the agent's behavior in the environment.

The term "off-policy" refers to the way the algorithm learns about the target policy (the optimal policy) independently of the behavior policy (the policy used to generate the actions during learning). This is distinct from "on-policy" methods, where the learning process depends directly on the policy being followed.

Q-Learning is the most common off-policy Temporal Difference algorithm. The agent learns the value of the optimal policy, regardless of the actions taken by the current behavior policy. This allows the agent to learn from experiences not directly generated by the current policy.

Q-learning is different from the other TD methods because it uses a different policy to select a new action, unlike normal Sarsa which uses the same policy to select the next action before updating the Q-table.

4. Epsilon-Greedy Policy

We chose to use the Epsilon-Greedy policy, since it is an improvement on the Greedy policy. Epsilon-Greedy policy does not take only actions that lead to high rewards. It is used to *balance* exploration and exploitation during training, for the purpose of improving learning rather than just max reward. Here's how it fits into the process:

- **Exploration vs. Exploitation**: With probability ϵ, the agent selects a random action (exploration). With probability $1-\epsilon$, the agent selects the action with the highest Q-value (exploitation).
- **Adjusting Epsilon**: Over time, ϵ is typically decayed to shift from exploration to exploitation as the agent learns more about the environment.
- **Training vs. Testing**: We only use this policy during training for exploration vs exploitation. During testing, we do exploitation only. In the training, we have epsilon random. In the test, we did not put epsilon random since we don't want randomness when testing for real.
- **Epsilon Decay**: Over time, ϵ is decayed to reduce exploration as the agent learns more about the environment.

The optimal policy is the policy that maximizes the expected cumulative reward. During training, the agent uses the epsilon-greedy policy to learn this optimal policy. Once training is complete, the agent

can follow the learned optimal policy by always choosing the action with the highest Q-value (i.e., exploiting without exploration).

5. *Incremental Mean*

It helps us update the policy more often since we update after every episode.

6. *Long Short-Term Memory (LSTM)*

Regular neural networks are not good for predicting sequences because they do not model what happened in the past. Thus, we are using a recurrent neural network (RNN), where the same neural network is used at each time stamp. Each time step in an RNN two inputs (current input and previous hidden state) and two outputs (current output and next hidden state). A one-to-one recurrent neural network is the same as a regular neural network. We also have hidden state creation. The value of hidden state for a given time step is used to compute the output for that time step.

In a basic RNN, we have a vanishing gradient (output is based on immediate prior time step inputs) since memory of previous layers decays over time steps. LSTM uses multiple gates to decide if previous hidden state needs to be passed or ignored which ensures long term memory of certain features.

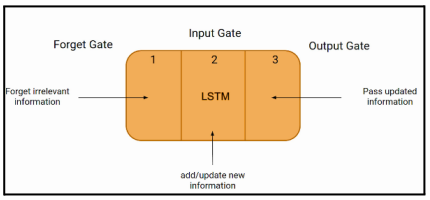

Fig. 10 There are three gates: Forget Gate, Input Gate, Output Gate

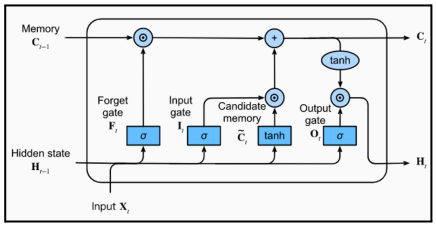

Fig. 11 We either pass the candidate hidden state or the previous hidden state at each gate.

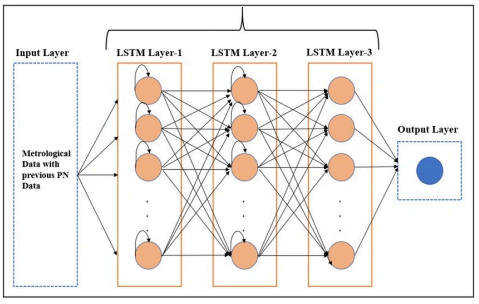

Fig. 12 Overview of the RNN using LSTM

B. Conceptual Study of the Algorithm

Category	States
Possible Moves	Up, Down, Left, Right
Presence of Collisions	Up, Left, Right
Directions of the Food	Up, Down, Left, Right

Fig. 13 Our 11 features which will be inputs.

With our 11 features in a network, we have it like this with our Q-Learning where each row is an example of an instance and each column is a feature. In this project, we utilize Q-Learning to develop a reinforcement learning model for a Snake game where action values are stored as Q-values. Since we'd have a massive table with a lot of states, we are doing Q-network (not simple Q-table).

- Each row in the provided table represents a distinct state of the game, characterized by various features such as the potential moves the snake can make (up, down, left, right), the presence of collisions in those directions, and the directions in which food is located relative to the snake's head.
- The columns labeled 'Move Up', 'Move Down', 'Move Left', and 'Move Right' contain Q-values that estimate the expected rewards for the snake taking those actions in the given state.
- Binary indicators under 'Collision Up', 'Collision Left', and 'Collision Right' denote whether a collision would occur if the snake moved in the specified direction.
- Similarly, the columns 'Food Up', 'Food Down', 'Food Left', and 'Food Right' indicate the presence of food in those directions.

By updating these Q-values based on the rewards received from taking actions, the model learns to optimize the snake's movements to maximize its score. Each state can be seen as a snapshot of the game at a specific moment, and the table helps in deciding the best action for the snake to take in each state to achieve optimal performance. This approach enables the model to refine its strategy over multiple games, ultimately learning to avoid collisions and move towards the food efficiently.

State	Move Up	Move Down	Move Left	Move Right	Collision Up	Collision Left	Collision Right	Food Up	Food Down	Food Left	Food Right
S1	0.5	0.2	0.1	0.7	0	1	0	1	0	1	0
S2	0.1	0.7	0.2	0.3	1	0	1	0	1	0	1
S3	0.3	0.1	0.4	0.2	0	1	0	1	0	1	0
S4	0.4	0.3	0.5	0.6	1	0	1	0	1	0	1

Fig. 14 The table presents the Q-values and various state features for a Q-Learning Snake game. Each cell under the columns "Move Up", "Move Down", "Move Left", and "Move Right" contains a Q-value.

Alternatively, we can organize it in this way so it's easier to understand:

	State 1	State 2	State 3	State 4	State 5	State 6
Forward	0.5	0.2	0.1	0.7	0.6	0.3
Backward	0.1	0.7	0.2	0.3	0.4	0.5
Right	0.3	0.1	0.4	0.2	0.3	0.6
Left	0.4	0.3	0.5	0.6	0.2	0.1

Fig. 15a Q-Table.

	State 1	State 2	State 3	State 4	State 5	State 6
Collision Up	0	1	0	1	0	1
Collision Down	0	0	1	0	1	0
Collision Left	1	0	1	0	1	0
Collision Right	0	1	0	1	0	1

Fig. 15b Collision Table

	State 1	State 2	State 3	State 4	State 5	State 6
Food Up	1	0	1	0	1	0
Food Down	0	1	0	1	0	1
Food Left	1	0	1	0	1	0
Food Right	0	1	0	1	0	1

Fig. 15c Food Table.

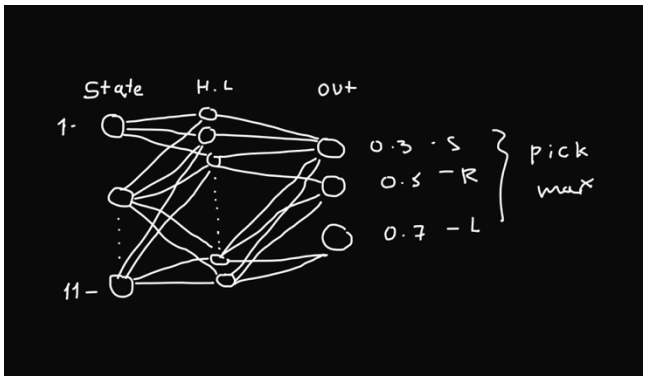

Fig. 16 A rough drawing of how the neural network works. There are 11 inputs we have for possible moves, presence of collisions, and directions of the food. There are 3 output options (move Straight, Right, or Left), where we choose the output based on highest Q-Learning value.

Optimal policies in reinforcement learning refer to the strategy or set of actions that an agent should follow to maximize its expected cumulative reward over time.

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Fig. 17 An optimal policy is derived from taking the max Q-value

IV. Preliminary Results

1. *Exploitation vs. Exploration Issue*

We had initial problems with these results having more exploitation over exploration:

- "Lazy Suicidal Snake": Once the snake was able to get the food a few times, the snake kept going in the same direction repeatedly in the same line and direction over and over until it died, regardless of where the food was.
- "Lazy Guardian Snake": Once the snake gets the food on the perimeter, it keeps surrounding the perimeter without dying. It's stuck in a loop and wouldn't die.
- "Indie 500": Doing laps over and over in a circle

The solution was to have epsilon decay. It was initially 0 epsilon decay so the snake would not explore other options. We had to increase it to a value greater than 0.

2. *Early Stage: A couple minutes into the game*

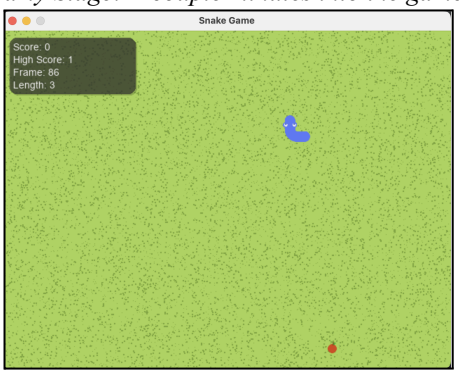

Fig. 18a *Early Stage*: Worm is learning and is small and going random places

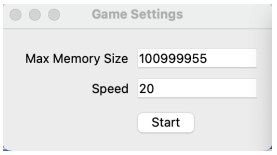

Fig. 18b *Early Stage*: Settings that we used to start the game

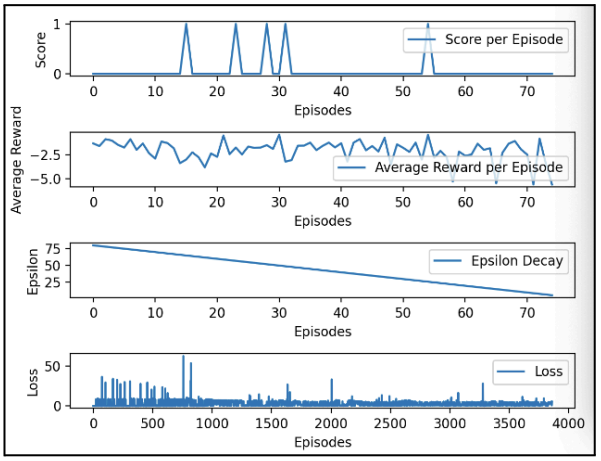

Fig. 18c *Early Stage*: Results that we have so far

3. *Intermediate Stage: Five minutes into the game*

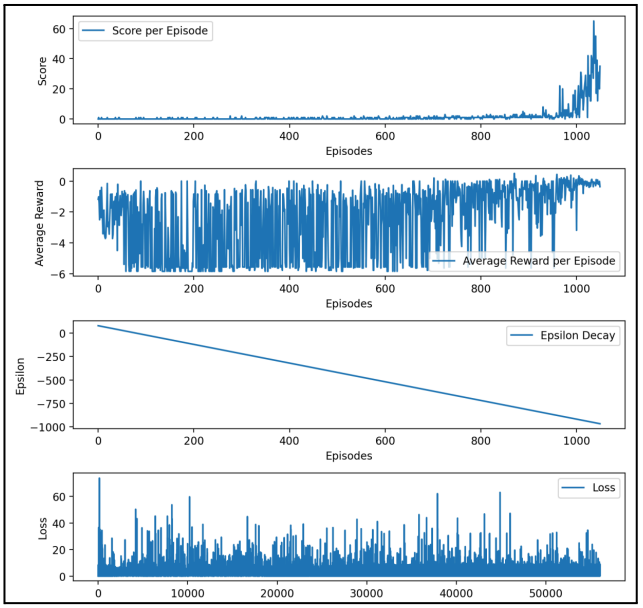

Fig. 19 *Intermediate stage* after 1100 iterations of the game (5 min)

V. Results and Analysis

1. Score Per Episode

A. Observation

The scores start very low and remain consistently low for a significant number of episodes. Around the 800th episode, the scores begin to increase rapidly and continue to rise.

B. Interpretation

Initially, the agent struggles to achieve higher scores, which is expected as it is learning the environment and policy. The rapid increase in scores after a substantial number of episodes indicates that the agent has started to learn effective strategies to maximize its reward.

2. Average Reward per Episode

A. Observation

The average reward per episode fluctuates significantly with many negative rewards early on. Over time, the fluctuations continue but the rewards gradually become less negative and more stable around the 800th episode.

B. Interpretation

Initial high variability and negative average rewards suggest the agent is frequently making suboptimal decisions, such as colliding with itself or the environment. As training progresses, the agent learns to avoid negative outcomes more effectively, leading to more stable and higher average rewards.

3. Epsilon Decay

A. Observation

Epsilon decays linearly over the episodes, starting from a high value and decreasing towards zero.

B. Interpretation

A linear decay of epsilon indicates a typical epsilon-greedy strategy, where the agent starts with a high exploration rate (high epsilon) to explore the environment. Over time, epsilon decreases, leading to more exploitation of the learned policy. This gradual shift from exploration to exploitation allows the agent to first gather diverse experiences and then focus on refining its strategy based on what it has learned.

4. Loss

A. Observation

The loss fluctuates significantly throughout the training process. There are periods of high loss values interspersed with lower loss values.

B. Interpretation

The fluctuations in loss indicate the learning process where the model continuously updates its parameters. High loss values can occur when the agent encounters new situations or when the Q-values are adjusted significantly. The overall pattern of loss suggests that the model is learning, although the high variability indicates that the learning process could benefit from further tuning.

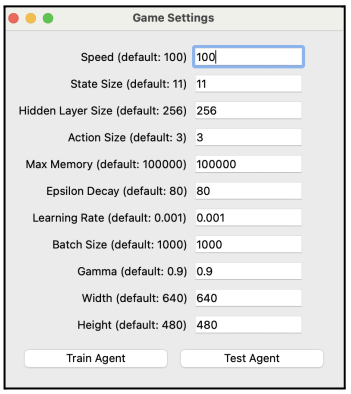

Fig. 20a *Late stage:* Updated UI of the Dispatcher Game Settings

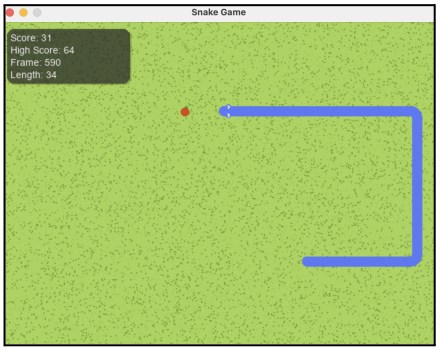

Fig. 20b *Late stage* after 1800 iterations of the game (1-2 hours)

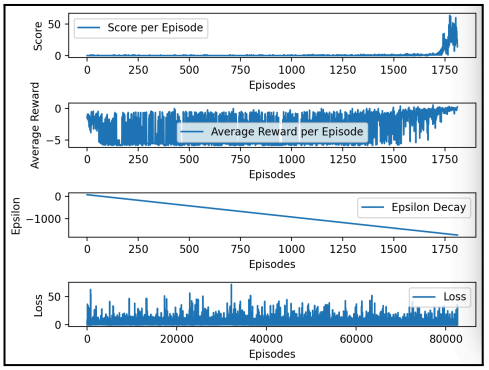

Fig. 20c *Late stage* after 1800 iterations of game (1-2 hours)

The results demonstrate the typical learning curve of a DQN model in a reinforcement learning task:

1. *Early Stage*

Low and inconsistent scores with highly negative rewards. This indicates the exploration phase where the agent is trying out different actions to understand the environment.

2. *Mid Stage*

Gradual improvement in scores and average rewards. The agent starts learning from its experiences and improves its strategy.

3. *Late Stage*

Significant improvement in scores with more stable and higher average rewards. The agent shifts towards exploiting the learned policy, leading to better performance.

4. *Loss:*

Continuous updates and fluctuations in loss values, showing the model's learning dynamics.

VI. Conclusions

We successfully integrated Deep Reinforcement Learning with Neural Network Regression and the DQN Bellman Algorithm into the Snake game, enabling strategic decision-making. By implementing the epsilon-greedy policy, we balanced exploration and exploitation, while the neural network approximated optimal action-value functions in high-dimensional state spaces. The model successfully learns to navigate the environment and improve its performance over time, as evidenced by the increasing scores and stabilizing rewards.

VII. Future Work

Currently, our system uses the Bellman equation to decide between three possible actions—indicating danger in the front, left, or right. This limited output range (taking the max of three) restricts the system's ability to handle complex scenarios effectively, particularly when we get to the higher scores like in the 70s and cannot go higher.

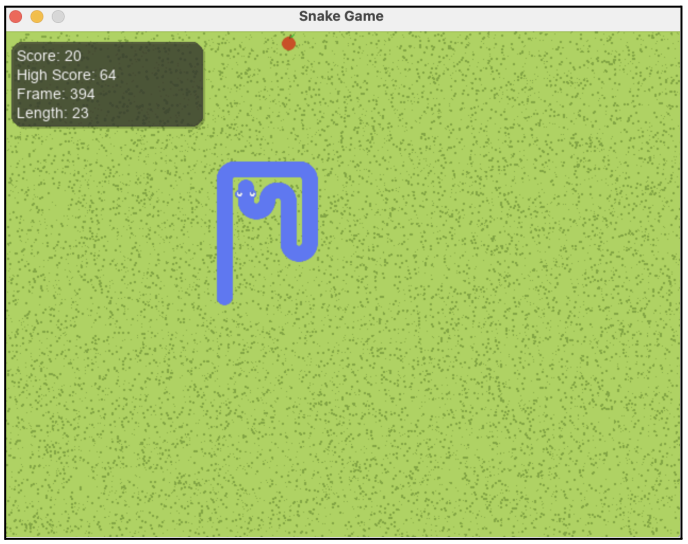

Fig. 21 This issue can be fixed with Monte Carlo Tree Search algorithm

To address this, we propose using Monte Carlo Tree Search Algorithm (MCTS), a technique similar to those used in strategic games like Go and Chess. MCTS will allow the system to explore all possible moves each cycle and select the best move based on a score evaluation. By constructing a graph of possible moves and their outcomes, MCTS will enable the system to make more informed decisions and avoid getting trapped. This approach will help improve the overall performance and adaptability of the system.

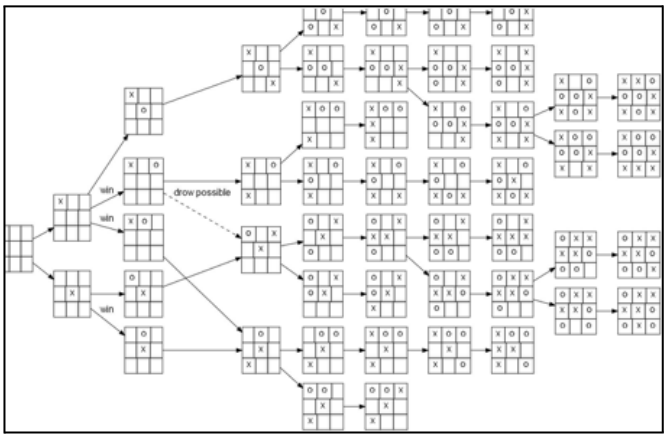

Fig. 22a *Monte Carlo Tree Search: Tic Tac Toe*

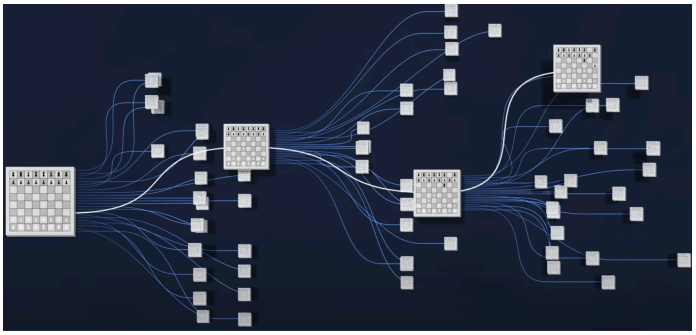

Fig. 22b *Monte Carlo Tree Search: Chess*

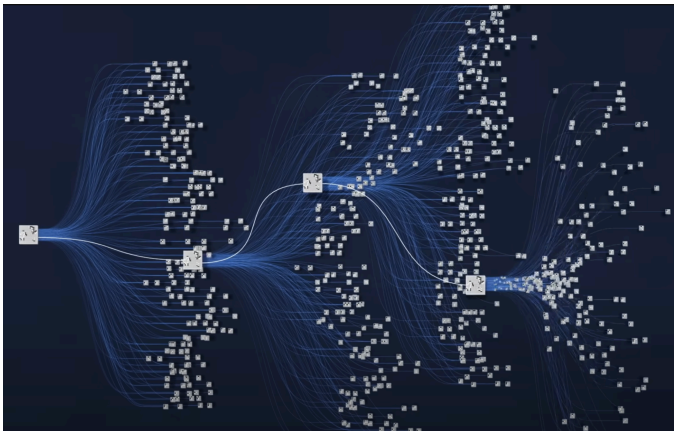

Fig. 22c *Monte Carlo Tree Search: Go*

In Go, the number of possible moves in each spot is 200, which and the number of configurations on the board is greater than the number of atoms in the universe stage

Some ideas to expand this project into future work would be to dive into robotics or autonomous driving, which could validate our findings further. To enhance performance, it would be good to Investigate alternative neural network architectures, such as CNNs and transformers. We could also optimize hyperparameters and exploring advanced techniques like Double DQN or Proximal Policy Optimization (PPO). Lastly, incorporating transfer learning and multi-agent scenarios could provide deeper insights into more complex interactions and strategies.

Acknowledgment

We would like to express our sincere gratitude to Dr. Anurag Nagar for teaching us the foundational concepts of supervised and unsupervised machine learning, which allowed us to extend to this third type for a well-rounded education. We also appreciate the valuable libraries provided by PyTorch, and the ease of importing the Snake game through Pygame. We are thankful for the Bellman DQN algorithm, which facilitated the training process, and the contributions of Keras. Additionally, we acknowledge the role of Long Short-Term Memory (LSTM) networks in enabling us to retain essential information.

References

[1] https://www.pythonguis.com/tutorials/pyqt-basic-widgets/
[2] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
[3] https://pythonprogramming.net/introduction-deep-learning-python-tensorflow-keras/
[4] https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/
[5] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
[6] https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/
[7] https://markus-x-buchholz.medium.com/deep-reinforcement-learning-introduction-deep-q-network-dqn-algorithm-fb74bf4d6862
[8] https://opengameart.org/content/snake-game-assets
[9] https://www.youtube.com/watch?v=NFo9v_yKQXA&list=PLzvYlJMoZ02Dxtwe-MmH4nOB5jYlMGBjr&pp=iAQB
[10] https://www.linkedin.com/learning/machine-learning-with-python-foundations/what-is-reinforcement-learning?autoplay=true&resume=false
[11] https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
[12] https://www.youtube.com/watch?v=WXuK6gekU1Y