CSE 472 (Machine Learning Sessional) - Project Proposal

Problem Definition

Deep Reinforcement Learning to play Tetris

Reinforcement learning is a type of machine learning where an agent learns to make decisions in an environment by performing actions and receiving rewards or punishments. To apply reinforcement learning to playing Tetris, we can define the environment as the game, the actions as the moves the player can make (e.g. rotating or moving the falling block), the state as the current layout of the blocks on the playing field, and the rewards as points for clearing lines.

Dataset

We don't need a dataset for deep reinforcement learning to play a game automatically.Instead of using pre-existing data, the agent learns from its own interactions with the game environment. According to the results reported in the "Playing Atari with Deep Reinforcement Learning", the DQN algorithm was able to learn how to play Tetris and achieved a score of approximately 100,000 points.

Papers

Here are some research papers in the area of deep reinforcement learning applied to playing tetris and our work will be based on this papers:

1. Learn to Play Tetris with Deep Reinforcement Learning (2020)
2. Playing Tetris with deep reinforcement learning (2016)
3. The Game of Tetris in Machine Learning (2019)

Architecture

A general architecture could look like this:

1. **Input layer**: The input layer of the neural network would take the current game state as input and it could be an image of the current state of the game, or features that describe the state.
2. **Hidden layers**: The hidden layers of the neural network would consist of one or more dense or convolutional layers. These layers would apply transformations to the input to extract relevant features.
3. **Output layer**: The output layer of the neural network would produce a probability distribution over possible actions for the agent to take. The actions could be moving the block left or right, rotating it, or dropping it. The output layer can use a softmax activation function to ensure that the probabilities sum to 1.
4. **Replay Memory**: Replay memory refers to a technique where an agent stores and replays past experiences in order to improve its decision-making ability. Specifically, the agent stores the state of the environment, the action it took the next state it transitioned to and the resulting reward. The next state contributes in calculating **Q-value** by giving insight about how good the next state of an action is. It can help prevent the model from overfitting to the specific experiences it has encountered and can improve the model's ability to generalize to new situations.
5. **Reinforcement learning algorithm**: The reinforcement learning algorithm takes the output of the neural network and the reward signal as input and updates the network parameters to improve performance. **Deep Q-learning** can be suitable as a reinforcement learning algorithm.
6. **Reward function:** The reward function would determine the reward the agent receives for each action it takes. The reward function could be based on the number of lines cleared, the height of the blocks, and other factors that affect the score.

Deep Q-learning: Deep Q-Learning (DQL) is a reinforcement learning algorithm that combines deep neural networks with Q-Learning. It uses a neural network to approximate the Q-function, which maps states to action values, instead of a table-based approach.

Potential Features

Some performance metrics for Tetris can be:

1. **Score**: The most straightforward metric would be the score achieved by the agent over a series of games. We can track the average or maximum score.
2. **Blocks cleared:** Another metric could be the number of blocks cleared by the agent. This could give us insight into the efficiency and skill of the agent in playing the game.
3. **Game Duration:** We can also track the length of time the agent can play the game before losing. This could give us insight into the durability of the agent.

4. **Block placement**: Another metric could be the quality of block placement by the agent. This could be assessed by measuring the height of the blocks and the number of gaps or holes in the stack.
5. **Speed:** The speed at which the agent can clear lines could also be a useful performance metric. It can give us insight into the speed and agility of the agent.

6. **Holes:** Holes in Tetris are unoccupied spaces that are surrounded by blocks. They can arise due to poor placement of blocks and can have a significant impact on gameplay. In deep reinforcement learning of Tetris, it is important to consider holes as they can affect the optimal strategy for clearing lines and maximizing the score. Deep reinforcement learning approaches use techniques such as adding a penalty term for each hole or modifying the reward function to explicitly encourage hole-free play.

7. **Bumpiness:** Bumpiness in Tetris refers to the unevenness of the stack of blocks, with bumps and holes creating peaks and valleys. A stack with high bumpiness may require more complex block placements to clear lines, while a low bumpiness stack may allow for more straightforward line clear. Some deep reinforcement learning approaches use techniques such as adding a penalty term for bumpiness or modifying the reward function to explicitly encourage low bumpiness play.

These are the possible features to represent a state. We used **four** of them, namely holes, bumpiness, lines cleared, reward of a move. We selected these four feature from intuition as these largely affects reward or penalizes our next states.

Our Approach

1. **Architecture**: We used the basic approach of deep reinforcement learning to form an agent. A fully connected layer was used to train the agent.
2. **Deep Reinforcement Learning**: Q learning was used to generate the best action. Q value had contribution from current reward and discounted reward from future state.
3. **Replay Memory**: We used replay memory to save past performances.
4. **Environment**: A built in tetris game was used as environment. It comes with custom speed which helped us render while training and understand which features could be used.
5. **Framework**: We used PyTorch framework to construct the model, replay memory and other stuffs.

Loss Function:

There were multiple steps involved in calculation loss. These are:

1. **Target and Policy Network**: Before training, values were predicted from the network for the state batches. These were considered as target values. After training, values were predicted for the states.
2. **Loss**: We had target and predicted values from the previous state. Mean square error(**MSE**) loss function were used to generate loss. These function was already implemented in **PyTorch** framework.
3. **Backward**: Loss values from previous step were used to generate gradients and update corresponding weights.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE = mean squared error
n = number of data points
Y_i = observed values
$\hat{Y}_i$ = predicted values

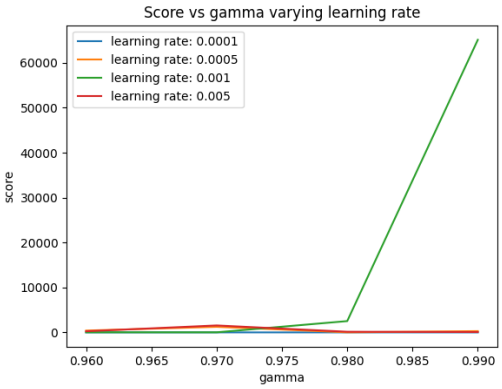
Performance Metrics

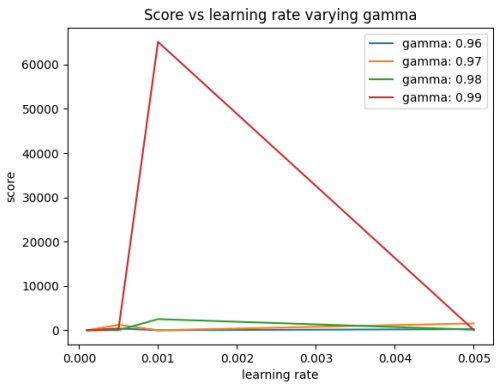
General scoring convention was used to measure performance of our agent. The score was calculated as follows:

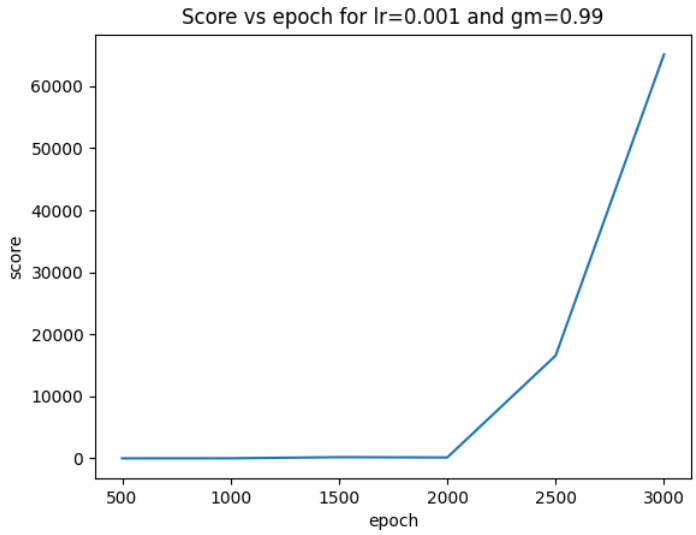
- Score 1 for successfully falling a piece on the board
- Score x*x * y for clearing a line in tetris where x is the number of lines cleared till now and y is the width of the line cleared.

We tried to observe how agents behave with respect to some parameters namely gamma, episode number, learning rate. Scores wrt. to these parameters are plotted below.

We observed best agent performance for learning rate = 0.001, discount gamma = 0.99 and episode number = 3000.







Challenges

1. Finding suitable environment was a challenge. We tried **OpenAI Gym Tetris** environment. Setting up the environment was challenging. Moreover, game speed, rendering was not customable. So we decided to not use this environment and find a suitable one.
2. Adjusting the parameters was not easy. The agent did not perform well for low episode numbers. Experimenting on the parameters with high episode count was an issue on computational complexity.
3. Visualization of the agent was our goal from the beginning. Thus we could not use virtual environments like colab as they don't support redering features.