2048 Game with Deep Q-Learning AI

By: Keenan Kalra

Project Overview

This project implements the classic 2048 puzzle game with a Deep Q-Network (DQN) AI agent that learns to play the game efficiently. The AI uses reinforcement learning to develop strategies for achieving high scores by making optimal moves based on the game state.

Features

- Classic 2048 game implementation with Pygame
- Deep Q-Learning agent that improves with training
- GPU acceleration for faster training
- Visualization of training progress
- Ability to play the game manually or watch the AI play
- Configurable training parameters
- Model saving/loading for continued training or evaluation

Installation

Prerequisites

- Python 3.10+
- PyTorch
- NumPy
- Pygame
- Matplotlib
- tqdm

Setup

How to Run

Play the Game Manually

Train the AI Agent

Watch the AI Play

Deep Q-Network (DQN) Explanation

What is DQN?

Deep Q-Network is a reinforcement learning algorithm that combines Q-Learning with deep neural networks. It enables an agent to learn optimal strategies in complex environments by approximating the Q-value function, which represents the expected future rewards for taking actions in different states.

Key Components of Our DQN Implementation:

1. Network Architecture

Our DQN uses a simple yet effective fully connected neural network:

- **Input layer:** 16 neurons (one for each cell in the 4x4 grid)
- **Hidden layers:** 2 layers with 256 neurons each, using ReLU activation
- **Output layer:** 4 neurons (representing up, down, left, right actions)

2. Experience Replay

The agent stores experiences `(state, action, reward, next_state, done)` in a replay buffer and randomly samples from this buffer during training. This breaks the correlation between consecutive training samples and improves learning stability.

3. Target Network

We use a separate target network that is periodically updated from the main network. This stabilizes training by providing consistent targets for Q-value updates.

4. Reward Function

Our reward function includes:

- Score increases from merging tiles
- Bonuses for creating higher value tiles
- Penalties for invalid moves

5. Exploration vs. Exploitation

The agent uses an epsilon-greedy policy:

- Initially explores randomly (high epsilon)
- Gradually shifts toward exploitation of learned values (decaying epsilon)
- Eventually settles on a minimal exploration rate (`epsilon_min`)

Learning Process

1. The agent observes the current board state
2. It selects an action (move direction) based on its policy
3. The game executes the action, providing a new state and reward
4. The agent stores this experience and learns by updating its Q-values

5. Over many episodes, the agent improves its strategy

Code Structure

Key Files:

- `agent.py`: Implements the `DQNAgent` class with neural network models, experience replay, and training logic
- `board.py`: Handles the 2048 board mechanics (moves, merging, game state)
- `train.py`: Manages the training process, including rewards, episode tracking, and model saving
- `play_ai.py`: Provides an interface to watch the trained AI play

Training Performance

The DQN agent typically shows significant improvement over time:

- **Early episodes (~100):** Random moves, rarely reaches tiles above 64
- **Mid training (~1000 episodes):** Develops basic strategies, regularly reaches 512 or 1024
- **Well-trained (~5000+ episodes):** Consistently achieves 2048 and beyond

Training visualizations are saved as `training_history.png`, showing score improvement and maximum tile values over time.

GPU Acceleration

The implementation supports GPU acceleration via PyTorch. When a CUDA-capable GPU is available, the training process automatically uses it for faster computation. Mixed-precision training is also enabled for compatible GPUs, further improving performance.

Tips for Better Results

Longer Training

- More episodes generally lead to better performance

Adjust Hyperparameters:

- Slower epsilon decay (e.g., `0.998`) for more thorough exploration
- Larger replay buffer for more diverse experiences
- Larger batch size for more stable updates

Advanced Reward Shaping:

- Consider modifying the reward function to encourage keeping large values in corners
- Add rewards for maintaining empty tiles
- Penalize board configurations that limit future moves

Acknowledgments

This project combines elements of reinforcement learning and game AI development. The 2048 game mechanics are based on the original game by Gabriele Cirulli.