DQN City Planning Agent with PyTorch

This project implements a Deep Q-Network (DQN) agent using PyTorch to solve an urban planning optimization problem. The primary goal is to train an AI agent to strategically place residential houses on a grid, maximizing the total number of validly connected houses while adhering to various environmental constraints and optimizing spatial utilization.

Features

- **Deep Q-Network (DQN)**: Utilizes the DQN algorithm for reinforcement learning, incorporating experience replay and a target network for stable training.
- **Convolutional Neural Network (CNN)**: Employs a CNN as the Q-network to effectively process the grid-based state representation and learn complex spatial features.
- **Multi-channel State Representation**: The environment state is represented as an 8-channel input to the CNN, providing detailed information about building types, distances to public buildings, and road connectivity.
- **Advanced Reward Shaping**: Custom reward functions guide the agent towards efficient house placements by incentivizing:
    - Block alignment for dense packing.
    - Placements that allow for future block extensions.
    - Clustering of houses.
    - Proximity to the Town Center.
- **Modular Environment Design**: The environment is structured to allow for relatively straightforward extensions of building types and rules.

How to Run

Prerequisites

Make sure you have the following Python libraries installed:

- numpy
- matplotlib
- torch
- collections (standard library)
- copy (standard library)
- time (standard library)
- math (standard library)

You can install the main dependencies using pip:
pip install numpy matplotlib torch

Execution

To train the DQN agent and visualize the results, simply run the Python script:

python dqn_city_planning.py

The script will print training progress to the console and display plots of episode rewards and the final best grid layout upon completion.

Code Overview

The project is primarily composed of two main classes: EnhancedCityPlanningEnv (the environment) and DQNAgent (the learning agent), which uses a QNetwork (the CNN).

1. The Environment (EnhancedCityPlanningEnv)

- **Grid Management**: Represents the city as a 2D NumPy array, tracking EMPTY, HOUSE, ROAD, TOWN_CENTER, and TRADING_POST cells.
- **Building Placement Logic**: Handles the placement of houses, ensuring no overlaps and that placements are on existing roads.
- **Connectivity**: Uses Breadth-First Search (BFS) to verify that all placed houses remain connected to the Kontor and Town Center via roads.
- **State Generation (_get_reduced_state)**: Transforms the grid into an 8-channel NumPy array suitable for the CNN. Channels include one-hot encodings for building types, normalized distance to the Town Center, and road connectivity to both the Kontor and Town Center.
- **Reward Calculation (_calculate_reward_shaping)**: Provides dense rewards to guide the agent, promoting efficient and strategic house placements.

2. The Q-Network (QNetwork)

- A PyTorch nn.Module that defines the CNN architecture.
- Takes the 8-channel grid state as input.
- Consists of three convolutional layers (with Batch Normalization and ReLU activation) to extract spatial features, followed by two fully connected layers to output Q-values for each possible house placement action.

3. The DQN Agent (DQNAgent)

- **Policy and Target Networks**: Maintains two instances of QNetwork for stable learning. The policy_net learns, while the target_net provides fixed targets for Q-value updates.
- **Experience Replay**: Stores past (state, action, reward, next_state) transitions in a buffer, from which random batches are sampled for training. This helps decorrelate experiences and improve learning stability.
- **Epsilon-Greedy Policy**: Implements an epsilon-greedy strategy for action selection, balancing exploration (random actions) and exploitation (choosing actions with the highest predicted Q-value). Epsilon decays over training episodes.
- **Learning**: Uses the Adam optimizer and Mean Squared Error (MSE) loss to update the policy_net based on the Bellman equation.

4. Training Loop (train_dqn_agent)

- Orchestrates the entire training process over a specified number of episodes.
- Manages episode resets, action selection, environment steps, experience storage, and agent learning.
- Tracks and prints training progress, including average rewards, current house count, epsilon value, and the best overall solution found.

Extending the Project

The modular design allows for significant extensions, but new rules and interactions require careful implementation.

Adding New Public Buildings (e.g., a "Market")

1. **Define Constants**: Add new integer constants for the building type (e.g., MARKET = 5) and its _DIMS.
2. **Environment Reset**: Implement logic in EnhancedCityPlanningEnv.reset() to randomly place the new building, ensuring it's on roads and connected to the Kontor. Store its coordinates and center.
3. **State Representation**: **Crucially**, add new input channels to the CNN's state in _get_reduced_state(). This would typically involve:
    - A one-hot encoded channel for the new building's locations.
    - A normalized Euclidean distance channel to the new building's center.
    - A binary road connectivity channel to the new building.
    - Update the input_channels parameter in DQNAgent initialization.
4. **Connectivity Checks**: Extend _check_all_buildings_connectivity() to include connectivity requirements for the new building if applicable.
5. **Reward Shaping**: Add new terms to _calculate_reward_shaping() to incentivize placements relative to the new building's influence.

Houses Requiring Multiple Public Building Needs ("Stages")

If houses need to fulfill conditions from multiple public buildings (e.g., 50% tile coverage from both TC and a Market):

1. **Influence Logic**: Modify is_within_influence() or create a new helper function to combine influence checks from all relevant public buildings.
2. **Reward Shaping**: Introduce a significant bonus in _calculate_reward_shaping() when a house successfully meets these combined requirements, effectively "upgrading" its stage.
3. **State Representation**: Consider adding a specific channel to _get_reduced_state() that indicates cells meeting these combined influence criteria, providing direct signals to the CNN.

Increasing Grid Dimensions

Scaling up the grid (e.g., from 20x20 to 40x40) dramatically increases the problem's complexity:

- **Computational Cost**: The action space (possible house placements) grows quadratically, leading to a larger QNetwork output layer and increased computation per step.
- **Reward Balancing**: Rewards might become sparser. Consider **increasing the magnitude of positive rewards** and ensure **reward shaping remains robust** to provide guiding signals.
- **Epsilon (Exploration-Exploitation)**:
    - **Slower Decay Rate**: Use a much slower epsilon_decay_rate (e.g., 0.9998 or 0.9999) to allow for more extensive exploration over more episodes.
    - **Higher min_epsilon**: Keep min_epsilon slightly higher (e.g., 0.1 to 0.2) to ensure continuous exploration and avoid local optima.
- **DQN Hyperparameters**:
    - **replay_capacity**: **Significantly increase** the replay buffer capacity (e.g., to 100,000 or more).
    - **batch_size**: Consider a **larger batch_size** (e.g., 64, 128, or 256) for more stable gradient updates.
    - **target_update_freq**: You might need to **increase this value** (e.g., to 500 or 1000) for target network stability.
    - **Network Architecture**: Potentially add **more convolutional layers** or increase filter counts in QNetwork to handle the larger input and learn more complex features.

Scaling up will require significantly longer training times and extensive hyperparameter tuning.