ECE 651 Final Project Report: Maze Generation using Deep Learning

Amogu J. Uduka

December 15, 2024

1 Project Overview

This project implements a maze generator using deep learning techniques, specifically focusing on Generative Adversarial Networks (GAN) and Deep Reinforcement Learning (DRL). The primary objectives were to generate and analyze maze structures using these advanced AI techniques. The project consists of three main components:

1. Creation of a training dataset using traditional maze generation algorithms
2. Implementation of a GAN-based maze generator with two different architectures
3. Development of a DRL-based maze generator using Deep Q-Networks (DQN)

In addition, I implemented the optional DRL-based maze-solving component to complete the complete maze generation and navigation pipeline.

2 Training Dataset

The initial training dataset was created using the mazelib Python library, specifically using Prim's algorithm for maze generation. The maze output is shown in Figure 1. This approach was chosen for its ability to generate well-formed solvable mazes with the following specifications:

- Maze size (N): 8x8 grid structure
- Number of samples (M): 20 mazes
- Format: Binary matrices where 1 represents walls and 0 represents paths

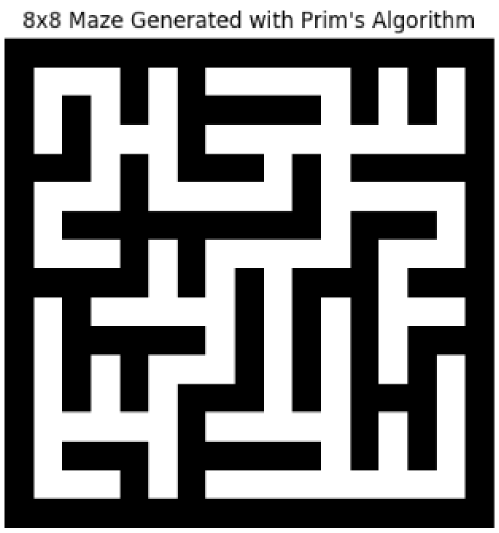

Figure 1: Example of a generated 8x8 maze using Prim's algorithm.

It is also worth pointing out that an 8 x 8 maze requires a 17 x 17 grid. This is illustrated as follows.

- In an 8x8 maze, you have 8 cells in each row and column
- Each cell represents a space where you can move in the maze
- Every cell needs to be separated by walls
- The maze also needs outer walls around its perimeter
- For each row/column: 8 cells (spaces where you can move) + 7 walls (between the cells) + 2 outer walls (on each end) = 17 total units

3 GAN-based Maze Generator

3.1 Model Selection and Primary Considerations

I implemented two GAN architectures:

1. Vanilla GAN: A simpler architecture using fully connected layers
2. Deep Convolutional GAN (DC-GAN): A more sophisticated architecture specifically designed for image generation

3.2 Design Configurations

3.2.1 Vanilla GAN Configuration

A Vanilla GAN is the simplest form of a Generative Adversarial Network, consisting of two neural networks: a generator that creates fake data from random noise and a discriminator that distinguishes real data from fake data. Both networks compete in a zero-sum game, with the generator trying to fool the discriminator and the discriminator learning to detect fakes. Vanilla GANs use fully connected layers and basic loss functions but often suffer from issues like mode collapse and unstable training.

- Generator: Three fully connected layers (100→512→1024→289)
- Discriminator: Three fully connected layers (289→512→256→1)
- Activation: Leaky ReLU (0.01) and Sigmoid
- Benefits: Simplicity, faster training
- Limitations: Less structured outputs, difficulty with spatial relationships

Listing 1: Vanilla GAN Generator Architecture

```python
class Generator(nn.Module):
    def __init__(self, z_dim, img_dim):
        super().__init__()
        self.gen = nn.Sequential(
            nn.Linear(z_dim, 512),
            nn.LeakyReLU(0.01),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.01),
            nn.Linear(1024, img_dim),
            nn.Sigmoid(),
        )
```

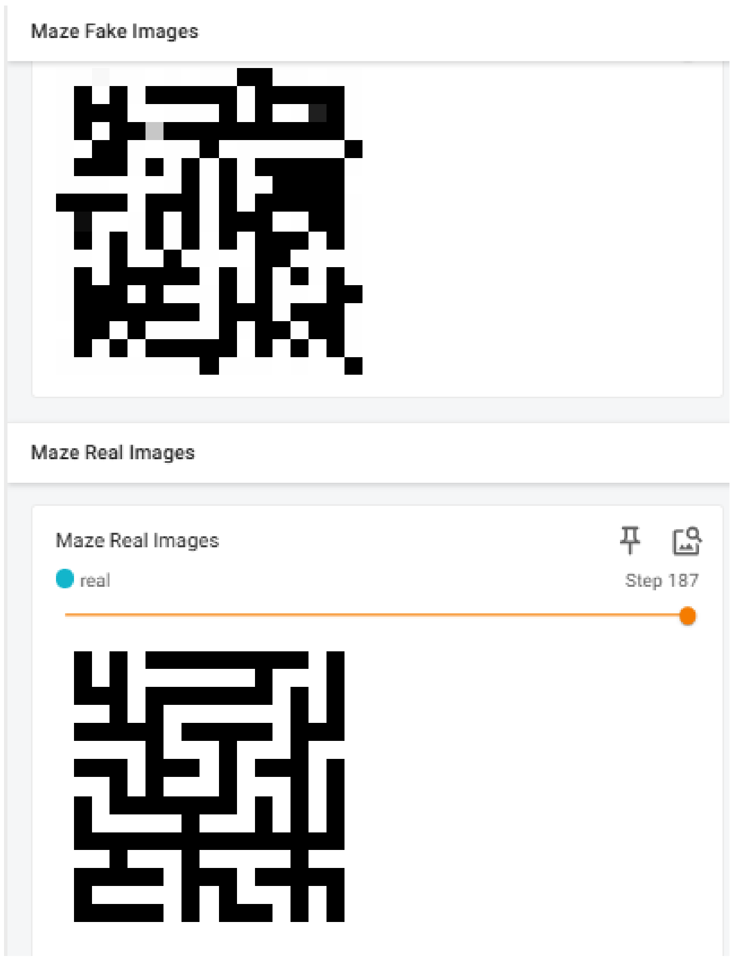

Figure 2: Maze Output by Vanilla GAN (obtained using TensorBoard)

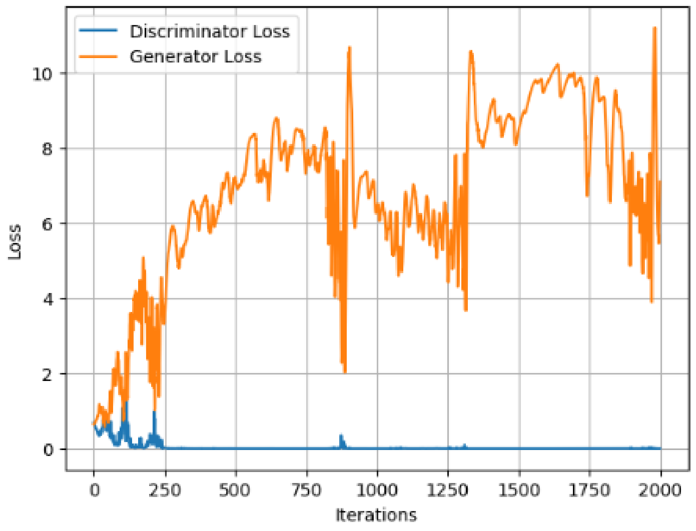

Figure 3: Training losses for Generator and Discriminator

3.2.2 DC-GAN Configuration

A Deep Convolutional GAN (DC-GAN) is an improved GAN architecture that replaces fully connected layers with convolutional and transposed convolutional layers. It is designed for image generation, offering better spatial awareness, more realistic outputs, and more stable training compared to Vanilla GANs. DC-GAN uses techniques like batch normalization and Leaky ReLU, making it a foundational GAN variant for image-related tasks.

- Generator: Transposed convolutions with batch normalization
- Discriminator: Convolutional layers with batch normalization
- Benefits: Better handling of spatial relationships, more stable training
- Limitations: More complex architecture, longer training time

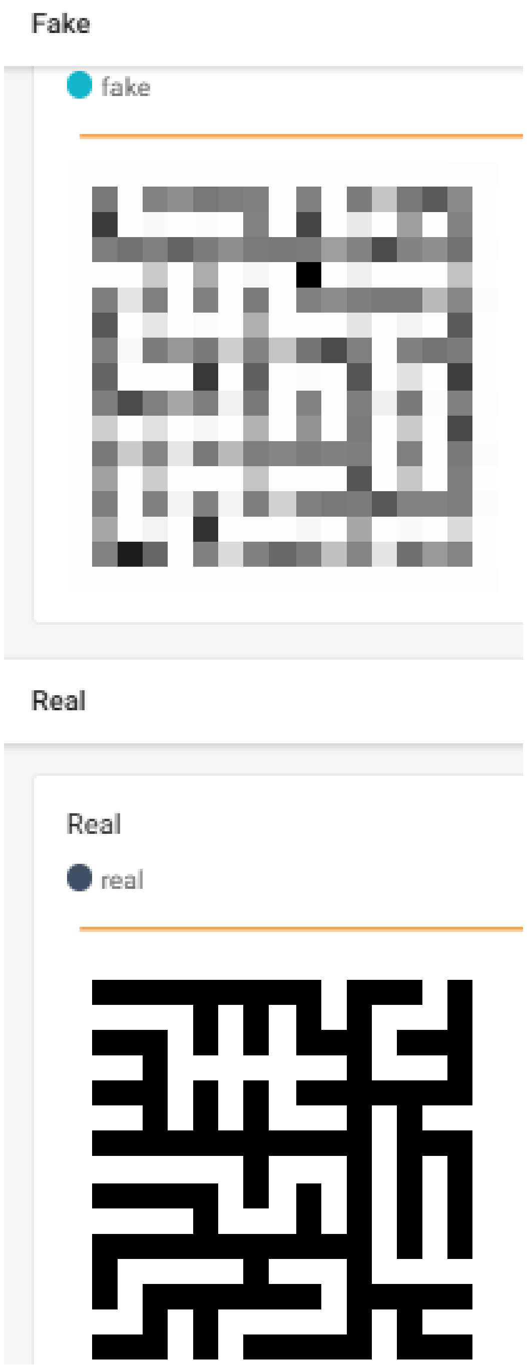

Figure 4: Maze Output by DC-GAN (Obtained using TensorBoard)

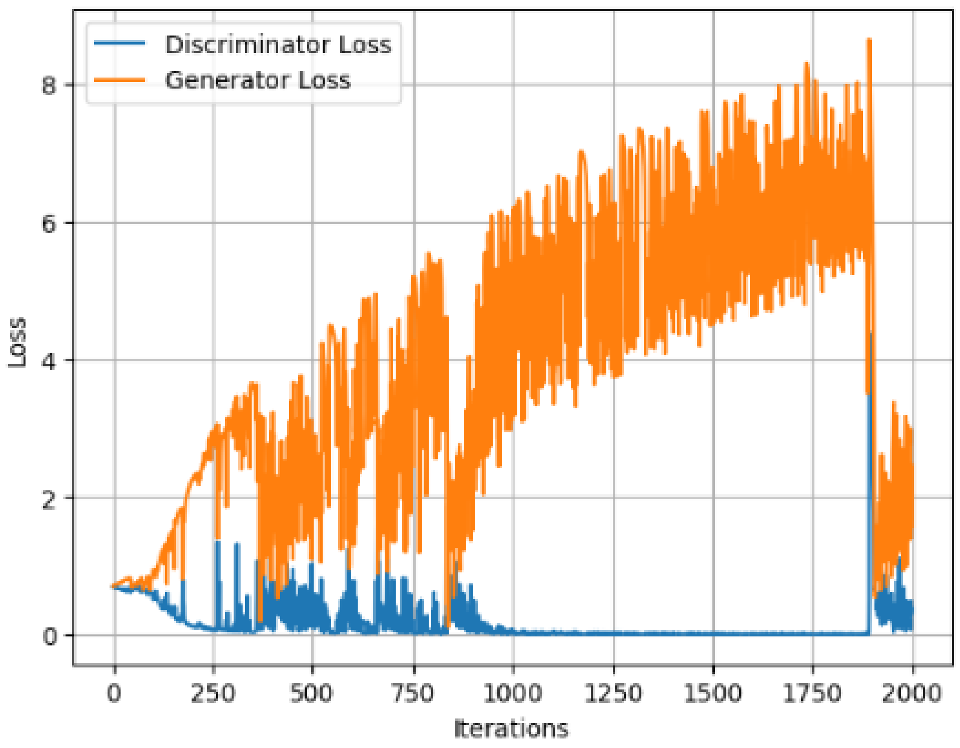

Figure 5: Training losses for Generator and Discriminator (DC-GAN)

4 DRL-based Maze Generator

4.1 Algorithm Selection

Going forward, a deep Q-Network (DQN) approach was used for the generation of mazes, with the following components:

- State space: This includes the current maze configuration, agent position, and key points.
- Action space: These are the possible actions (wall modifications, movement, point placement) the agent takes. They include; Add Treasure, Edit North, Go Up, Go Right, Edit West, Go Left, Add Start, Go Down, Add End, Edit South, and Edit East.
- Reward structure: Based on the existence and connectivity of the path. These conditions are good to avoid the cluster of closed boxes.
  - +10 for increasing connectivity
  - -10 for increasing connectivity

4.2 Network Structure and Training Setup

Listing 2: DQN Network Architecture

```
class DeepQNetwork(nn.Module):
    def __init__(self, input_dim, n_actions, lr=0.001):
        super(DeepQNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, n_actions)
```

5 DRL for Maze Solving

5.1 Network Configuration (4 x 4 Maze)

The DQN implemented for maze solving uses:

- Input size: 5 (current position, start, end, treasure coordinates)
- Hidden layers: 2 layers with 128 neurons each
- Output size: 11 (possible actions)
- Activation functions: ReLU for hidden layers, linear for output

5.2 Reward Function Design

The reward function incorporates the following:

1. Positive rewards for finding valid paths
2. Distance-based rewards for path optimization
3. Penalties for invalid moves or disconnected paths
4. Additional rewards for finding optimal paths through treasure points

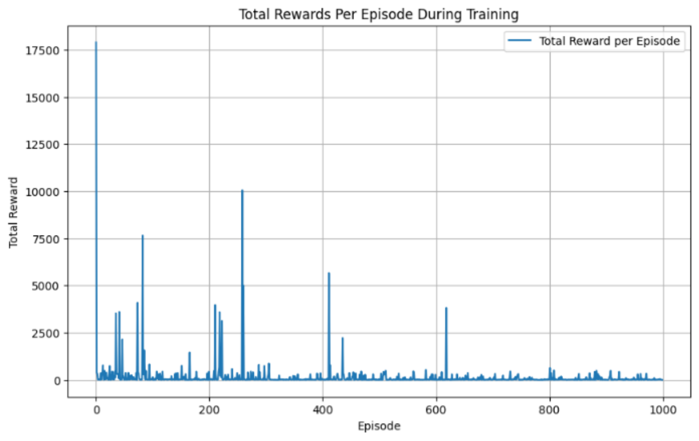

Figure 6: DRL Training Rewards over Episodes (4 x 4)

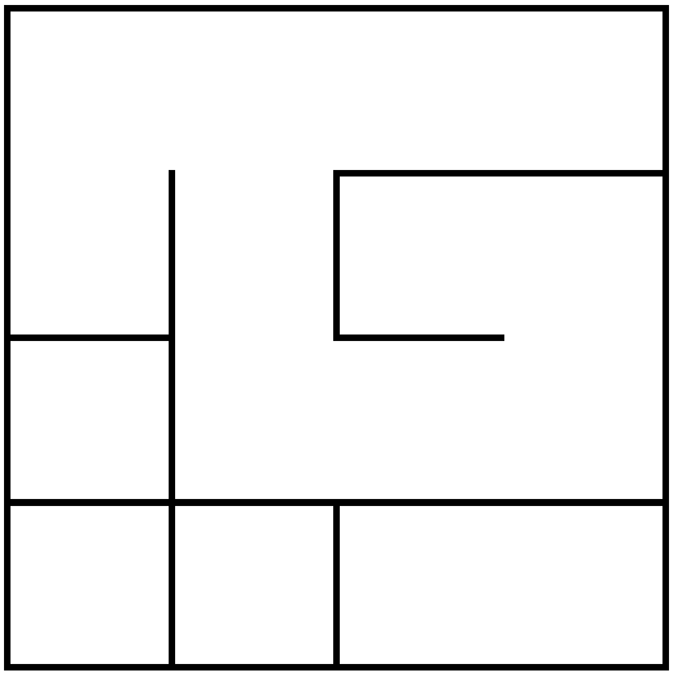

Figure 7: DRL Maze (4 x 4)

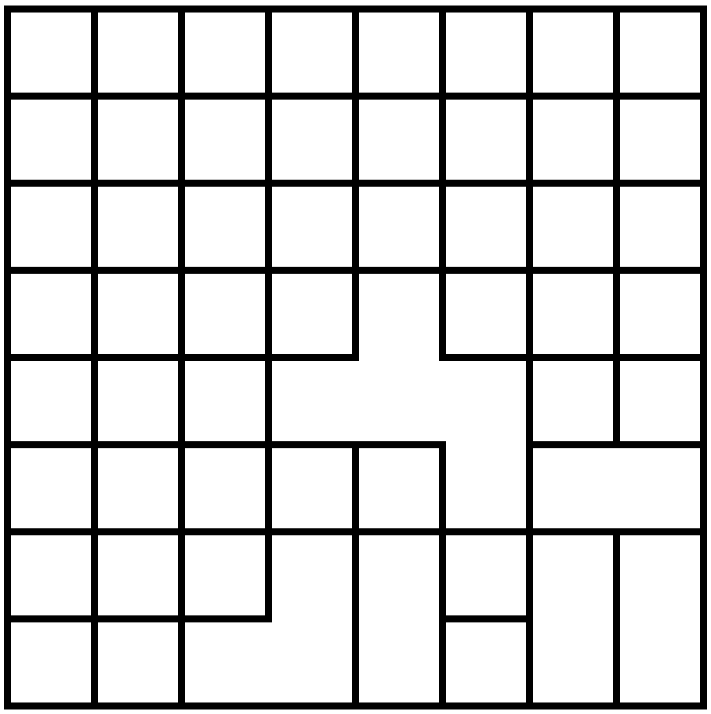
Figure 8: DRL Maze (8 x 8)

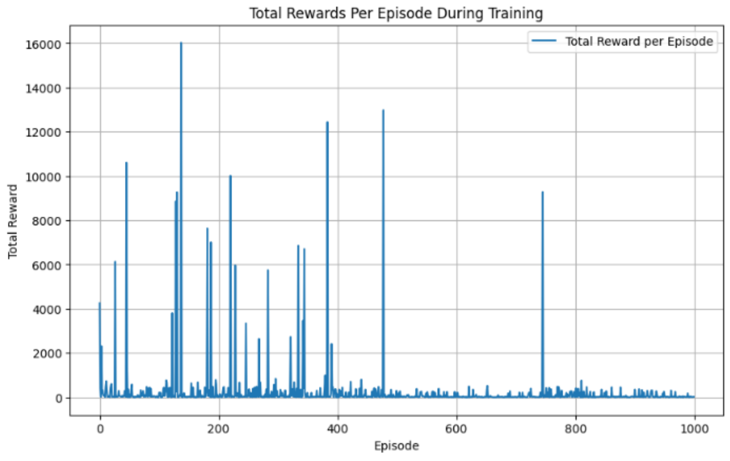

Figure 9: DRL Training Rewards over Episodes (8 x 8)

6 Comparison of Reinforcement Learning Performance in Different Maze Sizes

6.1 Initial Learning Phase

The 4×4 maze demonstrates superior early-stage learning characteristics, achieving peak rewards of approximately 16,000 units during the initial training episodes. In contrast, the 8×8 maze exhibits a delayed learning curve, which can be attributed to its more complex state space dimensionality.

6.2 Peak Performance Analysis

- Peak reward metrics:
  - 4×4 maze: ≈16,000 units
  - 8×8 maze: ≈15,000 units
- This differential is consistent with theoretical expectations, as shorter optimal trajectories in the 4×4 environment inherently accumulate fewer negative step penalties.

6.3 Stability Characteristics

Both environments exhibit significant reward variance throughout the training period. The 8×8 maze demonstrates persistent performance spikes, notably around episode 800, while the 4×4 configuration displays enhanced stability post-400 episodes.

6.4 Long-term Behavioral Patterns

$$\text{Performance Characteristics} = \begin{cases} 8 \times 8 \text{ maze:} & \text{Continued exploration} \\ 4 \times 4 \text{ maze:} & \text{Convergent behavior} \end{cases} \tag{1}$$

7 Summary: Observations and Discussion

7.1 Visualization Comparison

- GAN-generated mazes showed more structural consistency
- DRL-generated mazes exhibited more varied patterns
- Training dataset mazes provided balanced reference points

7.2 Evaluation Metrics

Key metrics used for evaluation:

- Path complexity and solvability
- Structural diversity
- Reward-based evaluation for DRL performance

7.3 Analysis and Future Work

Key insights from the project:

1. DC-GAN proved more effective than vanilla GAN for structured maze generation
2. DRL approach offered more control over maze properties
3. Combined approach demonstrates potential for automated game level design

Future work directions:

- Conditional generation of mazes with specific difficulty levels
- Integration of additional maze features and constraints
- Scalability to larger maze dimensions

8 References

1. Mazelib: A Python library for creating and solving mazes.
   https://github.com/john-science/mazelib

2. PyTorch-GAN: PyTorch implementations of Generative Adversarial Networks.
   https://github.com/eriklindernoren/PyTorch-GAN

3. CleanRL: Clean implementations of RL algorithms. https://github.com/vwxyzjn/cleanrl

4. Maze generator with Reinforcement Learning.
   https://https://github.com/AmT42/Maze_generator_with_ReinforcementLearning

5. Radford, A., Metz, L., Chintala, S. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv preprint arXiv:1511.06434.
   https://arxiv.org/abs/1511.06434