GNN based Tic Tac Toe Project

Lekh Sisodiya

August 21, 2024

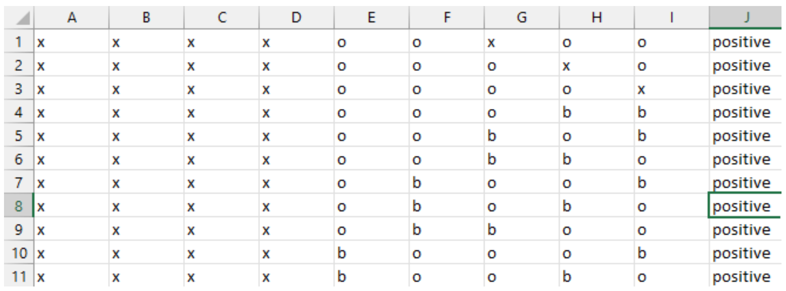
1 Project Overview

The Tic Tac Toe GNN project aims to develop an AI that can play Tic Tac Toe using Graph Neural Networks (GNNs). The model is trained to predict optimal moves based on the current game state. The GNN architecture includes convolutional and fully connected layers and is evaluated through simulated games.

2 Data Preparation

2.1 Dataset

The dataset contains sequences of Tic Tac Toe moves and their outcomes (win, loss, draw).



	A	B	C	D	E	F	G	H	I	J
1	x	x	x	x	o	o	x	o	o	positive
2	x	x	x	x	o	o	o	x	o	positive
3	x	x	x	x	o	o	o	o	x	positive
4	x	x	x	x	o	o	o	b	b	positive
5	x	x	x	x	o	o	b	o	b	positive
6	x	x	x	x	o	o	b	b	o	positive
7	x	x	x	x	o	b	o	o	b	positive
8	x	x	x	x	o	b	o	b	o	positive
9	x	x	x	x	o	b	b	o	o	positive
10	x	x	x	x	b	o	o	o	b	positive
11	x	x	x	x	b	o	o	b	o	positive

Figure 1: Dataset

2.2 Preprocessing

Data augmentation techniques are applied, ensuring moves are represented as integers.

```
import numpy as np

def augment_data(data):
    augmented_data = []
    for game in data:
        augmented_data.append(np.flip(game, axis=0))  # Flip rows
        augmented_data.append(np.flip(game, axis=1))  # Flip columns
        augmented_data.append(np.rot90(game, k=1))    # Rotate 90 degrees
    return np.array(augmented_data)
```

3 Model Architecture

3.1 Layers

- **GCNConv Layers**: Three layers with output dimensions 16, 32, and 64, respectively.
- **Conv1d Layer**: Processes the graph data.
- **Fully Connected Layers**: Two layers for final classification.

```
import torch
import torch.nn as nn
import torch_geometric.nn as pyg_nn

class TicTacToeGNN(nn.Module):
    def __init__(self):
        super(TicTacToeGNN, self).__init__()
        self.conv1 = pyg_nn.GCNConv(in_channels=9, out_channels=16)
        self.conv2 = pyg_nn.GCNConv(in_channels=16, out_channels=32)
        self.conv3 = pyg_nn.GCNConv(in_channels=32, out_channels=64)
        self.conv1d = nn.Conv1d(in_channels=64, out_channels=64, kernel_size=3)
        self.fc1 = nn.Linear(64, 32)
        self.fc2 = nn.Linear(32, 9)
        self.dropout = nn.Dropout(p=0.01)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        x = torch.relu(x)
        x = self.conv3(x, edge_index)
        x = torch.relu(x)
        x = x.unsqueeze(0)  # Add batch dimension for Conv1d
        x = self.conv1d(x)
        x = x.mean(dim=2)  # Pooling over sequence
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

4 Training and Evaluation

4.1 Training Metrics

Here are the training and validation metrics from the final epochs:

- **Epoch 48/50**: Training Loss: 0.3268, Validation Loss: 0.2620, Training Accuracy: 0.8641, Validation Accuracy: 0.8802
- **Epoch 49/50**: Training Loss: 0.2736, Validation Loss: 0.1789, Training Accuracy: 0.8967, Validation Accuracy: 0.9427
- **Epoch 50/50**: Training Loss: 0.3204, Validation Loss: 0.2313, Training Accuracy: 0.8614, Validation Accuracy: 0.9115

```
import torch.optim as optim
from torch_geometric.data import DataLoader

def train(model, data_loader, epochs=50):
    criterion = nn.CrossEntropyLoss()
```

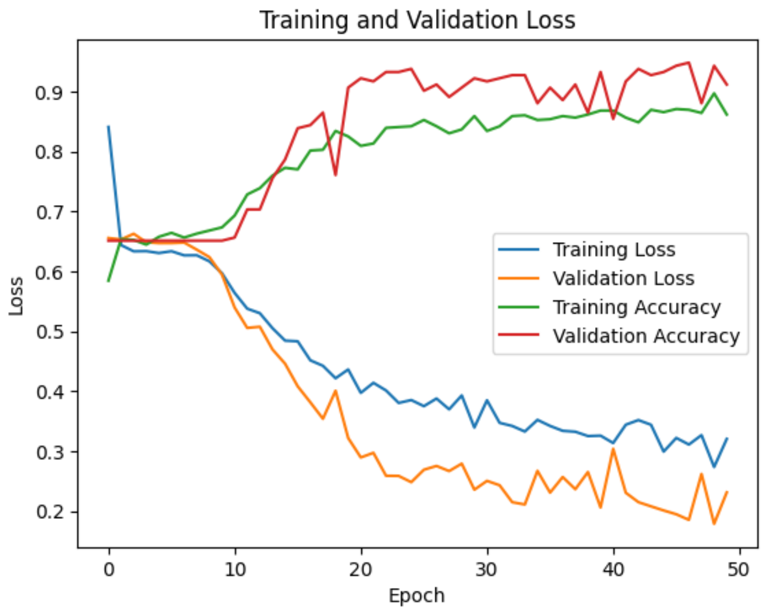


Figure 2: Training and Accuracy

```
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    for epoch in range(epochs):
        model.train()
        for data in data_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, data.y)
            loss.backward()
            optimizer.step()
        print(f'Epoch {epoch + 1}, Loss: {loss.item()}')

def evaluate(model, data_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for data in data_loader:
            output = model(data)
            pred = output.argmax(dim=1)
            correct += (pred == data.y).sum().item()
    accuracy = correct / len(data_loader.dataset)
    print(f'Accuracy: {accuracy * 100:.2f}%')
```

5 Game Implementation

5.1 Game Logic

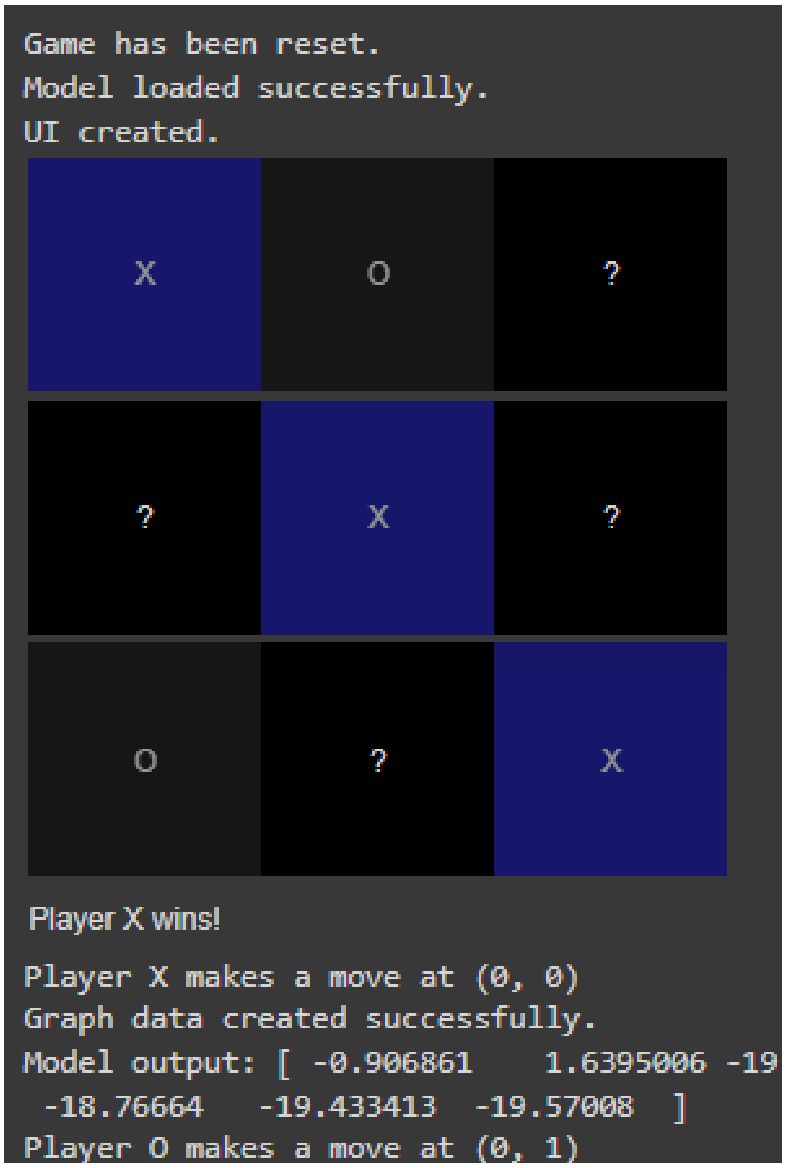
Handles player and AI moves and updates the game state.

```
class TicTacToeGame:
    def __init__(self):
        self.board = np.zeros((3, 3), dtype=int)  # 0: empty, 1: X, 2: O

    def make_move(self, x, y, player):
        if self.board[x, y] == 0:
            self.board[x, y] = player
            return True
        return False

    def check_winner(self):
        # Check rows, columns, and diagonals
        for player in [1, 2]:
            if any(np.all(self.board[i, :] == player) for i in range(3)) or \
               any(np.all(self.board[:, i] == player) for i in range(3)) or \
               np.all(np.diag(self.board) == player) or \
               np.all(np.diag(np.fliplr(self.board)) == player):
                return player
        return 0  # No winner yet

    def is_draw(self):
        return np.all(self.board != 0) and self.check_winner() == 0
```

5.2 User Interface

Updates the UI based on game events and AI decisions.

```
def update_ui(game_state):
    # Example function to update the UI
    print("Current Board State:")
    print(game_state)
    if game.check_winner() == 1:
        print("Player X wins!")
```



```
Game has been reset.
Model loaded successfully.
UI created.
```

Player X wins!

```
Player X makes a move at (0, 0)
Graph data created successfully.
Model output: [ -0.906861    1.6395006 -19
 -18.76664   -19.433413  -19.57008   ]
Player O makes a move at (0, 1)
```

Figure 3: Game UI

```
elif game.check_winner() == 2:
    print("Player O wins!")
elif game.is_draw():
    print("The game is a draw!")
```

6 Future Optimizations

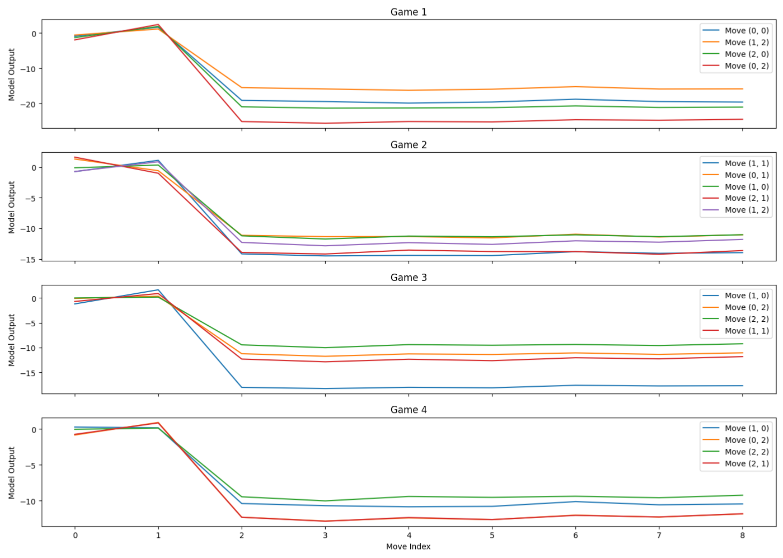


Figure 4: Future Optimisations Observation

- **Bias Mitigation**: Analyze predictions to identify and address biases. Include a wider variety of game situations in training data.
- **Data Enhancement**: Apply diverse augmentation techniques. Ensure balanced representation of game states.
- **Loss Function Adjustment**: Experiment with different loss functions that better reflect the objective of predicting optimal moves.
- **Hyperparameter Tuning**: Experiment with learning rates, dropout rates, and batch sizes to optimize performance.

- **Feature Engineering**: Re-evaluate feature extraction and representation for better game state capture.
- **Model Complexity**: Adjust the model architecture to balance complexity and performance.

7 Conclusion

The Tic Tac Toe GNN project highlights significant advancements in applying Graph Neural Networks (GNNs) to game AI. Key learnings include the effective use of GNNs for capturing game state dependencies and improving decision-making. The integration of graph-based approaches has demonstrated the capability of GNNs to model complex relationships within game states, leading to enhanced gameplay strategies. The project has successfully showcased the potential of GNNs in game AI, providing a solid foundation for future enhancements. Further optimizations will focus on refining the model to improve its accuracy and strategic depth, ensuring more robust and intelligent gameplay.