Tic Tac Toe AI Engine Development Framework

Table of Contents

Overview

This project provides a framework for developing and testing various AI engines for playing Tic Tac Toe (Piskvorky). It includes visualization tools, training capabilities, and comparison functionalities to help you develop and evaluate different AI strategies.

User Workflow

1. Getting Started:
   - Clone the repository
   - Review EngineLinear.py as a reference implementation
   - Create your own engine file (e.g., MyEngine.py)
2. Development Process:
   - Implement required engine methods
   - Register your engine in main.py's SETTING dictionary
   - Train your engine using main.py
   - Save promising versions to temp_engines/
3. Testing & Evaluation:
   - Test against AI using human_vs_ai.py
   - Compare with other engines using main_komparace.py
   - Run tournaments using main_tournament.py
   - Iterate and improve based on results

Application Components

Main Applications

- `main.py` - Main training application with visualization
  - Controls: Play/Stop and Save buttons

  - Configuration via SETTING dictionary
  - Real-time visualization of training games
- `main_komparace.py` - Engine comparison tool
  - One-on-one engine comparisons
  - Detailed statistics and visualization
  - Configurable number of games
- `main_tournament.py` - Tournament system
  - Multiple engine comparison
  - Round-robin tournament structure
  - Comprehensive results analysis
- `human_vs_ai.py` - Interactive testing
  - Play against trained engines
  - Mouse-based interface
  - Real-time board visualization

Configuration Settings

```
SETTING = {
    "RUN": True,         # Controls if training is running or paused
    "SAVE": False,       # When True, saves current engine parameters
    "Engine": EngineAI, # Your engine class to train
    "Generations": 7,    # Number of generations to train
    "PaMutation": 0.1    # Mutation probability
}
```

Creating Your Own Engine

Basic Requirements

Each engine must: 1. Be a class that inherits basic game evaluation functionality 2. Implement required methods for parameter handling and board evaluation 3. Be able to save/load its parameters for future use

Template Structure

Use `EngineLinear.py` as a reference. Your engine must implement these methods:

```
class YourEngine:
    def __init__(self):
        self.v_engine = "0.0.1"  # Version tracking
        self.board = None
        self.initialize_parameters()

    def initialize_parameters(self):
        # Initialize your engine's parameters
        # Example:
```

```
        self.parameters = {
            'param1': np.random.randn(size),
            'param2': np.zeros(size)
        }

    def mutate(self, mutation_rate=0.1, mutation_scale=0.1):
        # Implement parameter mutation for training
        pass

    def evaluation(self):
        # Implement board evaluation
        # Should return a numerical value representing board state
        # Positive values favor player 1, negative values favor player -1
        pass

    def evaluate_board(self, board):
        self.board = board.copy()
        return self.evaluation()

    def get_parameters(self):
        # Return copy of parameters
        return self.parameters.copy()

    def set_parameters(self, parameters):
        # Set parameters from saved state
        self.parameters = parameters.copy()

    def load_params(self, file=""):
        # Load parameters from file
        return self.set_parameters(np.load(file))
```

Board Representation

- The game board is a 5x5 numpy array
- Values: 0 (empty), 1 (player 1), -1 (player 2)
- Winning condition: 4 in a row (horizontal, vertical, or diagonal)

Using the Framework

Training Engines

Use `main.py` to train your engine:

```
# In main.py, set your engine class
SETTING = {
    "Engine": YourEngine,  # Your engine class
    "Generations": 7,      # Number of generations
```

```
    "PaMutation": 0.1      # Mutation probability
}
```

The training process: 1. Creates multiple instances of your engine 2. Makes them play against each other 3. Selects winners based on game outcomes 4. Applies mutations to create new generations 5. Saves successful engines to `temp_engines/`

Testing Against Other Engines

Use `main_komparace.py` to compare two engines:

```
comparison = EngineComparison(
    engine1_class=YourEngine,
    engine1_datafile="./temp_engines/your_engine.npz",
    engine2_class=OtherEngine,
    engine2_datafile="./temp_engines/other_engine.npz",
    num_games=20,
    display_game=True
)
comparison.run_comparison()
```

Tournament Mode

Use `main_tournament.py` to compare multiple engines:

```
engine_configs = [
    (YourEngine, "./temp_engines/your_engine1.npz"),
    (YourEngine, "./temp_engines/your_engine2.npz"),
    (OtherEngine, "./temp_engines/other_engine.npz")
]

tournament = TournamentManager(
    engines_config=engine_configs,
    games_per_match=5,
    display_game=True
)
tournament.run_tournament()
```

Playing Against Your Engine

Use `human_vs_ai.py` to test your engine interactively:

```
game = HumanVsAI(
    engine_class=YourEngine,
    engine_datafile="./temp_engines/your_engine.npz",
    ai_starts=False  # True for AI to start
)
game.run_game()
```

Engine Development Tips

Evaluation Function

The evaluation function should: - Return higher values for positions favorable to player 1 - Return lower values for positions favorable to player -1 - Consider factors like: - Piece positions - Potential winning lines - Blocking opponent's moves - Board control

Parameter Management

- Keep parameters in a dictionary for easy saving/loading
- Use numpy arrays for efficient computation
- Consider using multiple parameter sets for different game stages

Mutation Strategy

- Balance exploration vs exploitation
- Consider adaptive mutation rates
- Test different mutation scales

Example Engines

Linear Engine (Provided Example)

- Uses linear regression for board evaluation
- Simple but effective starting point
- Good reference for parameter handling

Possible Approaches

1. Convolutional Neural Networks
    - Pattern recognition
    - Spatial feature detection
2. Deep Neural Networks
    - Complex pattern evaluation
    - Multiple evaluation criteria
3. Heuristic-based
    - Hand-crafted features
    - Strategic patterns

Testing and Evaluation

Key Metrics

- Win rate
- Draw rate
- Average game length
- Position evaluation accuracy

Visualization

The framework provides visualization for: - Game progress - Training progress - Tournament results - Comparative analysis

Best Practices

1. Version your engines
2. Save successful parameters
3. Test against multiple opponents
4. Use visualization tools for debugging
5. Implement gradual improvements
6. Document your engine's strategy

Project Structure (For Developers)

Core Components

- `Player.py`: Implements game logic and player behavior
- `GameBoard.py`: Handles game board visualization
- `Button.py`: UI component for controls
- `Manager.py`: Manages training and game execution
- `SaveHandler.py`: Handles parameter saving
- `StartStopHandler.py`: Controls training flow

Visualization Components

- `colors.py`: Color definitions for UI
- `my_dataclasses.py`: Data structures for game state

Engine Interface

Required methods for custom engines: - `initialize_parameters()` - `mutate()` - `evaluation()` - `evaluate_board()` - `get_parameters()` - `set_parameters()` - `load_params()`

Data Management

- `temp_engines/`: Directory structure for saved engines
  - Naming convention: engineType_version.npz
  - Example: LinEngine1.npz, ConvEngine1.npz

Training Flow

1. Manager initializes games
2. Players use engines for moves
3. Winners selected based on game outcomes
4. Parameters mutated for next generation

5. Successful engines saved to temp_engines/

Common Issues and Solutions

1. Overfitting to Specific Opponents

- Problem: Engine performs well against training opponents but poorly against others
- Solutions:
  - Test against various strategies
  - Implement regularization
  - Increase training population diversity

2. Poor Generalization

- Problem: Engine fails to adapt to new situations
- Solutions:
  - Increase training diversity
  - Add randomization to evaluation
  - Improve feature extraction

3. Slow Evaluation

- Problem: Engine takes too long to make decisions
- Solutions:
  - Optimize calculations
  - Use vectorized operations
  - Simplify evaluation criteria

4. Unstable Training

- Problem: Training results are inconsistent
- Solutions:
  - Adjust mutation parameters
  - Implement early stopping
  - Save successful intermediate results

Getting Help

- Review the example implementations in EngineLinear.py
- Use visualization tools to understand engine behavior
- Test against basic engines to validate improvements
- Analyze tournament results for performance insights
- Check the code comments for additional implementation details

Remember that developing a successful engine is an iterative process. Start with simple implementations and gradually add complexity based on performance analysis.