Graph Code Generation using Self-Play Reinforcement Learning

Overview

This guide implements a system that uses self-play reinforcement learning (inspired by the Absolute Zero paper) to generate Python matplotlib code that reproduces graphs from screenshots. The system evolves populations of code candidates through mutation, selection, and evaluation.

Key Components

1. Graph Code Generator

- **Purpose**: Generates and mutates matplotlib code
- **Methods**:
  - `generate_random_code()`: Creates initial random graph code
  - `mutate_code()`: Applies mutations to existing code
  - **Mutation types**: modify data points, change styles, add/remove curves

2. Similarity Evaluator

- **Purpose**: Measures how similar generated graphs are to target
- **Metrics**:
  - **SSIM (Structural Similarity)**: Compares overall image structure
  - **Visual Features**: Edges, contours, markers detection
  - **Data Patterns**: Curve shapes, marker positions
  - **Color Distribution**: For colored graphs

3. Self-Play Trainer

- **Purpose**: Evolves code population using evolutionary algorithms
- **Process**:
  - Initialize random population
  - Evaluate each candidate
  - Select best performers
  - Generate offspring through mutation
  - Repeat for multiple generations

Implementation Steps

Step 1: Prepare Your Target Image

python

```python
# Save your graph screenshot as 'target_graph.png'
target_image_path = 'target_graph.png'
```

Step 2: Initialize the Training System

python

```python
from graph_selfplay import SelfPlayTrainer

# Create trainer with your target image
trainer = SelfPlayTrainer(target_image_path)

# Configure parameters
trainer.population_size = 50  # Number of code candidates per generation
```

Step 3: Run the Training

python

```python
# Train for specified number of generations
trainer.train(generations=100)

# Get the best result
best_code = trainer.get_best_code()
print("Best generated code:")
print(best_code)

# Save the result
trainer.save_best_result("best_graph_code.py")
```

Advanced Similarity Metrics

The system uses multiple similarity metrics for robust evaluation:

1. Structural Similarity (SSIM)

- Measures pixel-level similarity
- Good for overall layout comparison
- Weight: 40% of final score

2. Visual Feature Similarity

- Compares edges, contours, shapes
- Detects curve patterns and markers
- Weight: 30% of final score

3. Data Pattern Similarity

- Analyzes curve density and distribution
- Counts data points and markers
- Weight: 30% of final score

Optimization Strategies

Population Evolution

1. **Elite Selection**: Keep top 25% performers
2. **Mutation Rate**: 80% mutation, 20% random generation
3. **Diversity Maintenance**: Prevent premature convergence

Mutation Types

1. **Data Modification**: Adjust x,y coordinates
2. **Style Changes**: Modify colors, markers, line styles
3. **Structure Changes**: Add/remove curves
4. **Parameter Tuning**: Adjust axis limits, labels

Evaluation Improvements

1. **Multi-metric Scoring**: Combine multiple similarity measures
2. **Feature Extraction**: Analyze specific graph characteristics
3. **Robust Comparison**: Handle different image sizes and formats

Expected Results for Your Graph

Based on your screenshot showing density vs electronic energy gap curves:

Target Characteristics

- **4 curves** with different colors (black, blue, green, red)
- **Exponential decay** patterns

- **Circular markers** on each curve
- **Specific axis ranges**: x: 1-16, y: 0-20
- **Scientific notation** and Greek symbols in labels

Generated Code Structure

python

```python
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(figsize=(8, 6))

# Multiple curves with different colors and decay patterns
ax.plot(x1, y1, 'ko-', markersize=6, linewidth=2)  # Black curve
ax.plot(x2, y2, 'bo-', markersize=6, linewidth=2)  # Blue curve
ax.plot(x3, y3, 'go-', markersize=6, linewidth=2)  # Green curve
ax.plot(x4, y4, 'ro-', markersize=6, linewidth=2)  # Red curve

ax.set_xlabel('Density (g cm⁻³)', fontsize=12)
ax.set_ylabel('Electronic energy gap (eV)', fontsize=12)
ax.set_xlim(1, 16)
ax.set_ylim(0, 20)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Performance Metrics

Training Progress Indicators

- **Generation Number**: Current evolution cycle
- **Best Similarity Score**: Highest achieved similarity (0-1 scale)
- **Population Diversity**: Variation in similarity scores
- **Convergence Rate**: Improvement over generations

Success Criteria

- **Similarity Score > 0.8**: Good visual match
- **Similarity Score > 0.9**: Excellent match
- **Feature Match**: Correct number of curves, colors, markers

Troubleshooting

Common Issues

1. **Low Similarity Scores**:
   - Increase population size
   - Run more generations
   - Adjust mutation parameters
2. **Slow Convergence**:
   - Increase mutation rate
   - Add more mutation types
   - Improve similarity metrics
3. **Premature Convergence**:
   - Maintain population diversity
   - Use tournament selection
   - Add randomness injection

Parameter Tuning

```python
# Adjust these parameters for better results
trainer.population_size = 100        # Larger population
trainer.mutation_rate = 0.9          # Higher mutation rate
trainer.elite_ratio = 0.2            # Keep top 20%
trainer.generations = 200            # More evolution cycles
```

Extensions and Improvements

1. Advanced Mutations

- **Smart data interpolation**: Generate realistic curve shapes
- **Style libraries**: Use predefined color/marker combinations
- **Template-based generation**: Start from graph templates

2. Better Evaluation

- **Perceptual metrics**: Human-like visual comparison
- **Mathematical analysis**: Compare curve equations

- **Domain-specific features**: Physics-aware evaluation

3. Interactive Training

- **Human feedback**: Manual ranking of candidates
- **Active learning**: Focus on difficult cases
- **Transfer learning**: Use knowledge from similar graphs

Usage Tips

1. **Start Simple**: Begin with basic graphs before complex ones
2. **Use Good Screenshots**: High contrast, clear images work best
3. **Monitor Progress**: Check intermediate results regularly
4. **Fine-tune Parameters**: Adjust based on your specific graph type
5. **Validate Results**: Always verify generated code produces correct output

Conclusion

This self-play approach provides a robust framework for automatically generating matplotlib code from graph images. The system learns through evolution, gradually improving code quality through intelligent mutations and comprehensive similarity evaluation. While it may take several generations to converge, the results can be highly accurate reproductions of complex scientific graphs.