Emergent Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis

Brent Hartshorn: June 26, 2025

brenthartshorn@proton.me

Abstract:

Traditional neural networks face significant hurdles when integrating non-differentiable, dynamic systems, often requiring complex approximations for gradient-based learning. This paper presents a novel computational paradigm that bypasses these limitations by leveraging the deterministic complexity of the Burning Ship fractal to directly initialize a Conway's Game of Life (GOL) grid. The subsequent spatiotemporal evolution of this GOL process is then analyzed using a 3D Fast Fourier Transform (FFT) to extract key spectral energy bands. These bands are then fed into a small, feedforward neural network classifier, which learns to interpret the spectral patterns and produce the system's output. Optimization is achieved through a hybrid approach: a gradient-free mutation and selection process applied to the fractal's parameters, coupled with traditional gradient-based training for the classifier. This approach demonstrates a unique form of emergent computation, where the system learns to identify fractal regions that deterministically yield GOL dynamics with specific spectral characteristics that a separate classifier can interpret, offering a compelling alternative for dynamic pattern recognition and bio-inspired computing.

1. Introduction

Artificial Neural Networks (ANNs) have revolutionized numerous fields by excelling at learning complex mappings from data. Their success largely stems from the ability to efficiently train parameters via backpropagation, a gradient-based optimization algorithm that demands differentiable components throughout the network. However, this reliance creates a bottleneck when attempting to integrate inherently non-differentiable dynamical systems, such as discrete, rule-based cellular automata, directly into the computational flow of a single, end-to-end differentiable ANN. Such integration, particularly with systems exhibiting complex emergent behaviors like cellular automata, holds immense potential for modeling intricate natural phenomena and developing novel forms of intelligent computation.

Previous attempts to bridge this gap, as explored in earlier work, involved embedding a Conway's Game of Life (GOL) simulation within a neural network's forward pass. While conceptually rich, the non-differentiable nature of GOL proved a formidable challenge for effective gradient propagation *through the GOL simulation itself*, leading to prohibitively long training times and unstable convergence for an end-to-end differentiable system. This paper introduces a hybrid optimization strategy that circumvents these direct backpropagation limitations. It proposes leveraging the inherent properties of fractal geometry to initiate and control emergent GOL dynamics, whose spatiotemporal characteristics are interpreted through advanced spectral analysis and then *classified by a separate, small neural network*.

2. The Fractal-GOL-FFT Architecture

Our proposed computational model consists of three core integrated components, designed to process external inputs and produce a classified output (e.g., solving XOR) through an iterative optimization process:

2.1. Burning Ship Fractal Initialization of GOL

At the heart of the system is the Burning Ship fractal, a complex fractal similar to the Mandelbrot set but characterized by its unique "burning ship" appearance due to the absolute value operations in its iteration formula. Instead of a neural network generating initial GOL states, a specific region of this fractal directly determines the initial 32x32 binary grid for the GOL simulation. Parameters defining this region—namely the fractal's center_x, center_y coordinates, zoom level, and max_iterations for its computation—become the primary "tunable variables" of our system. The fractal generation process outputs a grid of iteration counts, which are then binarized (e.g., values above a threshold become 'alive', others 'dead') to seed the GOL. This leverages the fractal's inherent intricate patterns to provide rich and varied starting conditions for the cellular automaton. Crucially, external inputs (e.g., the two bits of an XOR problem) are encoded into specific rows of this initial GOL grid, ensuring that the GOL's evolution is directly modulated by the problem's input.

2.2. Spatiotemporal GOL Evolution

Once initialized, the GOL grid evolves for 64 discrete steps. Conway's Game of Life is a zero-player game, meaning its evolution is determined solely by its initial state and a simple set of rules governing cell birth, survival, and death based on the number of living neighbors in a 3x3 grid. Critically, each intermediate state of the GOL grid throughout its 64-step evolution is recorded. This sequence of 32x32 grids over 64 time steps forms a 64x32x32 three-dimensional dataset, capturing the complete spatiotemporal dynamics of the GOL process.

2.3. 3D Fast Fourier Transform and Spectral Classification

The raw 3D dataset representing the GOL's full evolution is then subjected to a 3D Fast Fourier Transform (FFT). This transformation moves the data from the spatial-temporal domain into a three-dimensional frequency domain (spatial frequencies in X and Y, and temporal frequency in T). The resulting magnitude spectrum, after being shifted to center its zero-frequency component, provides a comprehensive representation of the patterns and periodicities present across both space and time within the GOL's dynamics. From this rich 3D spectrum, a fixed number of distinct spectral energy bands (e.g., 16 bands) are extracted. These normalized band energies then serve as the input features for a small, feedforward neural network classifier. This classifier, a standard differentiable component, learns the mapping from these spectral features to the desired output (e.g., 0 or 1 for the XOR problem). This modular design isolates the non-differentiable GOL dynamics from the gradient-based learning, allowing the classifier to efficiently learn patterns in the spectral domain.

3. Optimization Through Parameter Mutation

This architecture employs a hybrid optimization strategy. The outer loop uses a gradient-free mutation and selection process to optimize the non-differentiable Burning Ship fractal parameters. The inner optimization involves standard gradient-based learning to train the small neural network classifier, which interprets the spectral outputs generated by the GOL/FFT process.

3.1. Iterative Mutation and Evaluation

The process begins with a randomly initialized set of Burning Ship parameters. In each subsequent iteration, a new candidate set of parameters is generated by applying a small, random mutation (perturbation) to the current best-performing parameters. This new set of fractal parameters is then used to:

- Generate a GOL initial grid for each XOR input (e.g., [0,0], [0,1], [1,0], [1,1]), incorporating the input into the grid as described in prior work.
- Run the GOL simulation for 64 steps for each input, recording its entire spatiotemporal evolution.
- Perform the 3D FFT on each history and extract the fixed number of energy bands.

These generated spectral band features (e.g., 16 features for each XOR input) are then used as the training data for the small neural network classifier. The classifier is trained for a fixed number of epochs using a standard loss function (e.g., Mean Squared Error) and an optimizer (e.g., Adam) to learn the mapping from these spectral features to the corresponding XOR outputs. The 'loss' for the outer, gradient-free optimization loop (which evaluates the Burning Ship parameters) is then derived from the performance of this *trained classifier* on the XOR problem, such as its average prediction error across all XOR inputs.

3.2. Selection and Convergence

If the average classification error (loss) of the neural network, when trained on data generated by the mutated Burning Ship parameters, is lower than the best error achieved so far, these new parameters replace the current best. This acts as a selection mechanism within the outer, gradient-free optimization loop, continuously refining the fractal parameters to enable better separability of the XOR inputs in the spectral domain. This hybrid method gracefully handles the non-differentiability of the GOL and FFT steps because the gradient-based learning is confined to the classifier, which operates on numerical inputs. Convergence is observed when the system consistently achieves a low classification error for the XOR problem, indicating that the Burning Ship parameters are effectively generating spectrally distinct GOL dynamics for each XOR input that the classifier can reliably map to the correct output.

Conclusion:

The exploration presented here pushes the boundaries of conventional computational models, evolving a novel paradigm that leverages the inherent complexity and deterministic chaos of fractal geometry. Here, the intricate patterns of the Burning Ship fractal directly initialize the GOL grid, with the fractal's parameters (center coordinates, zoom, and iteration depth) serving as the system's primary tunable variables. The GOL simulation then unfolds, and its entire spatiotemporal evolution (a 3D volume of time x grid_x x grid_y) is subjected to a 3D Fast Fourier Transform (FFT) to comprehensively capture its emergent dynamics. Crucially, the spectral energy bands extracted from this 3D FFT are then fed into a separate, small neural network classifier. This classifier learns to interpret the complex spectral signatures, performing tasks such as solving the XOR problem. The 'training' process for this combined system involves a nested optimization: an outer, gradient-free loop that mutates the Burning Ship fractal's parameters, and an inner loop where the classifier is trained via traditional gradient descent on the GOL-generated spectral features.

This approach offers several compelling advantages and insights:

- **Bypassing Non-Differentiability:** It gracefully sidesteps the formidable challenge of backpropagating gradients through discrete, non-linear, and non-differentiable systems like GOL. The non-differentiable components (fractal generation, GOL, FFT) are placed outside the direct gradient path of the classifier, allowing their parameters to be optimized through an outer, gradient-free search. The deterministic nature of the fractal and GOL ensures consistent outputs for given parameters, making this evaluation feasible.
- **Harnessing Inherent Complexity:** Instead of learning to generate complex patterns from scratch (as a neural network might), this model learns to *select* regions within a pre-defined, naturally complex mathematical space (the Burning Ship fractal) that inherently lead to desired emergent GOL dynamics.
- **Spatiotemporal Feature Extraction:** The use of 3D FFT on the entire GOL process represents a powerful method for extracting holistic spatiotemporal features, capturing not just static patterns but their dynamic evolution over time.
- **Hybrid Optimization:** The system employs a novel hybrid optimization strategy. While the inner classifier leverages efficient gradient-based learning, the outer loop for the fractal parameters utilizes a gradient-free iterative mutation and selection process. This dual-loop approach effectively navigates complex, non-linear parameter spaces where purely gradient-based methods would fail due to the non-differentiable nature of the core GOL dynamics.

This research contributes to a broader understanding of how emergent properties can be harnessed for computational tasks through a hybrid learning architecture, potentially informing novel approaches in areas like dynamic system control, complex pattern recognition, and the synthesis of artificial life. By elegantly combining gradient-free optimization of non-differentiable, generative dynamics with a standard, differentiable classifier, this paradigm offers a powerful blueprint for integrating diverse computational elements to achieve emergent intelligence.

SOURCE CODE

```
import random, math, sys, torch, time; import numpy as np; import torch.nn as nn; import torch.optim as optim
# --- Constants ---
GOL_GRID_SIZE = 32; GOL_SIM_STEPS = 64; NUM_FFT_BANDS_FOR_CLASSIFIER = 16; OPTIMIZATION_ITERATIONS = 100; MIN_LOSS=0.01
MUTATION_RATE = 0.05; CLASSIFIER_RATE = 5; device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# --- Conway's Game of Life Implementation ---
def gol_step_np(grid):
    new_grid = np.copy(grid); rows, cols = grid.shape
    for r in range(rows):
        for c in range(cols):
            live_neighbors = 0
            for i in range(-1, 2):
                for j in range(-1, 2):
                    if i == 0 and j == 0:
                        continue
                    nr, nc = (r + i) % rows, (c + j) % cols
                    live_neighbors += grid[nr, nc]
            if grid[r, c] == 1:
                if live_neighbors < 2 or live_neighbors > 3: new_grid[r, c] = 0
            else:
                if live_neighbors == 3: new_grid[r, c] = 1
    return new_grid

# --- Burning Ship Fractal Generation (still in NumPy/Python) ---
def generate_burning_ship_grid_np(grid_size, center_x, center_y, zoom, max_iter=100):
    grid = np.zeros((grid_size, grid_size), dtype=float)
    scale_x = 3.0 / zoom / grid_size
    scale_y = 3.0 / zoom / grid_size
    for row in range(grid_size):
        for col in range(grid_size):
            cx = center_x + (col - grid_size / 2) * scale_x
            cy = center_y + (row - grid_size / 2) * scale_y
            x, y = 0.0, 0.0
            for i in range(max_iter):
                x_new = x*x - y*y + cx
                y_new = 2 * abs(x * y) + cy
                x = abs(x_new); y = abs(y_new)
                if x*x + y*y > 4.0: break
            grid[row, col] = i
    max_val = np.max(grid)
    if max_val > 0: grid = grid / max_val
    return grid

# --- GOL Simulation and 3D FFT (with simple XOR input encoding) ---
def run_gol_and_3d_fft_torch(burning_ship_params, xor_input): # Added xor_input
    center_x, center_y, zoom, fractal_max_iter = burning_ship_params
    # 1. Initialize GOL grid from Burning Ship fractal (NumPy)
    fractal_raw_grid_np = generate_burning_ship_grid_np(GOL_GRID_SIZE, center_x, center_y, zoom, fractal_max_iter)
    current_gol_grid_np = (fractal_raw_grid_np > 0.5).astype(int)
    # 2. Encode XOR input into the GOL grid (overwriting or adding to fractal init)
    # Using rows 0, 1 for xor_input[0] and rows 2, 3 for xor_input[1].
    current_gol_grid_np[0:4, :] = 0 # Reset specific rows to ensure clean encoding
    # Encode xor_input[0]
    if xor_input[0] == 0:
        current_gol_grid_np[0, :] = 1 # Fill row 0
    elif xor_input[0] == 1:
        current_gol_grid_np[0, :] = 1 # Fill row 0
        current_gol_grid_np[1, :] = 1 # Fill row 1
    # Encode xor_input[1]
    if xor_input[1] == 0:
        current_gol_grid_np[2, :] = 1 # Fill row 2
    elif xor_input[1] == 1:
        current_gol_grid_np[2, :] = 1 # Fill row 2
        current_gol_grid_np[3, :] = 1 # Fill row 3
    # Now the grid has a combined pattern of Burning Ship and XOR input.
    # 3. Store entire GOL process for 3D FFT (as PyTorch tensor)
    gol_process_history = torch.zeros((GOL_SIM_STEPS, GOL_GRID_SIZE, GOL_GRID_SIZE), dtype=torch.float32, device=device)
    for step in range(GOL_SIM_STEPS):
        # Transfer current_gol_grid_np to torch tensor for storage, then back to numpy for next step
        gol_process_history[step, :, :] = torch.from_numpy(current_gol_grid_np).to(device).float()
        current_gol_grid_np = gol_step_np(current_gol_grid_np)
    # 4. Perform 3D FFT on the entire GOL process (PyTorch)
    fft_result_3d = torch.fft.fftn(gol_process_history)
    fft_shifted_3d = torch.fft.fftshift(fft_result_3d)
    magnitude_spectrum_3d = torch.abs(fft_shifted_3d)
    return magnitude_spectrum_3d

# --- Energy Band Calculation from 3D FFT (unchanged) ---
def calculate_3d_fft_energy_bands_torch(magnitude_spectrum_3d, num_bands):
    power_spectrum = magnitude_spectrum_3d**2
    flattened_power = power_spectrum.flatten()
    band_energies = torch.zeros(num_bands, dtype=torch.float32, device=device)
    segment_size = len(flattened_power) // num_bands
    for i in range(num_bands):
        start_idx = i * segment_size
        end_idx = start_idx + segment_size
        if i == num_bands - 1: end_idx = len(flattened_power)
        band_energies[i] = torch.sum(flattened_power[start_idx:end_idx])
    total_energy_bands = torch.sum(band_energies)
    if total_energy_bands > sys.float_info.min: band_energies = band_energies / total_energy_bands
    else: band_energies = torch.full((num_bands,), 1.0 / num_bands, device=device)
    return band_energies
```

```python
# --- Small Neural Network Classifier (PyTorch) ---
class ClassifierNN(nn.Module):
    def __init__(self, num_input_bands, num_hidden_neurons, num_output_neurons):
        super(ClassifierNN, self).__init__()
        self.fc1 = nn.Linear(num_input_bands, num_hidden_neurons)
        self.sigmoid = nn.Sigmoid()
        self.fc2 = nn.Linear(num_hidden_neurons, num_output_neurons)
    def forward(self, x):
        x = self.fc1(x); x = self.sigmoid(x); x = self.fc2(x); x = self.sigmoid(x)
        return x

# --- Optimization Loop (Modified to pass XOR input to GOL function) ---
def solve_xor_with_burning_ship_and_classifier_torch():
    # XOR input/output pairs
    xor_data = [
        ([0, 0], 0), ([0, 1], 1), ([1, 0], 1), ([1, 1], 0)
    ]
    classifier_nn = ClassifierNN(
        num_input_bands=NUM_FFT_BANDS_FOR_CLASSIFIER,
        num_hidden_neurons=8,
        num_output_neurons=1
    ).to(device)
    optimizer = optim.Adam(classifier_nn.parameters(), lr=0.1)
    criterion = nn.MSELoss()
    classifier_nn_epochs_per_iter = CLASSIFIER_RATE
    best_bs_params = [
        random.uniform(-1.0, 0.0), random.uniform(0.0, 1.0), random.uniform(1.0, 5.0), random.randint(50, 150)
    ]
    best_overall_loss = float('inf')
    print(f"Starting optimization for XOR with Burning Ship and PyTorch Classifier. Iterations: {OPTIMIZATION_ITERATIONS}")
    print(f"Initial Best Burning Ship Parameters: {best_bs_params}")
    start_time = time.time()
    for iteration in range(OPTIMIZATION_ITERATIONS):
        current_bs_params = list(best_bs_params)
        for i in range(len(current_bs_params)):
            if i < 3:
                current_bs_params[i] += random.uniform(-MUTATION_RATE, MUTATION_RATE) * abs(current_bs_params[i])
                if i == 2: current_bs_params[i] = max(0.1, current_bs_params[i])
            else:
                current_bs_params[i] += random.randint(
                    -max(1, int(MUTATION_RATE * current_bs_params[i])), max(1, int(MUTATION_RATE * current_bs_params[i])))
                current_bs_params[i] = max(20, min(200, current_bs_params[i]))
        # --- Evaluate current Burning Ship parameters ---
        current_band_data_list = []; targets_list = []
        for xor_input_val, target in xor_data: # Looping through XOR inputs
            # Pass the current xor_input_val to the GOL function!
            magnitude_spectrum = run_gol_and_3d_fft_torch(current_bs_params, xor_input_val)
            bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
            current_band_data_list.append(bands)
            targets_list.append(float(target))
        bands_batch = torch.stack(current_band_data_list).to(device)
        targets_batch = torch.tensor(targets_list, dtype=torch.float32).unsqueeze(1).to(device)
        classifier_nn.train()
        current_avg_nn_loss_val = 0.0
        for epoch in range(classifier_nn_epochs_per_iter):
            optimizer.zero_grad()
            predictions = classifier_nn(bands_batch)
            loss = criterion(predictions, targets_batch)
            loss.backward()
            optimizer.step()
            current_avg_nn_loss_val += loss.item()
        current_avg_nn_loss_val /= classifier_nn_epochs_per_iter
        if current_avg_nn_loss_val < best_overall_loss:
            best_overall_loss = current_avg_nn_loss_val; best_bs_params = current_bs_params
            elapsed_time = time.time() - start_time
            print(f"Iteration {iteration}, New Best BS Params, Avg NN Loss: {best_overall_loss:.6f}")
            print(f"Params: {[f'{p:.4f}' for p in best_bs_params]}, Time: {elapsed_time:.2f}s")

        if iteration % (OPTIMIZATION_ITERATIONS // 100) == 0:
            classifier_nn.eval()
            total_test_loss = correct_predictions = 0
            print(f"\n--- Iteration {iteration} Progress Check ---")
            test_bands_batch_list = []; test_targets_batch_list = []
            for test_input, test_target in xor_data:
                test_spectrum = run_gol_and_3d_fft_torch(best_bs_params, test_input)
                test_bands = calculate_3d_fft_energy_bands_torch(test_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
                test_bands_batch_list.append(test_bands); test_targets_batch_list.append(float(test_target))
            test_bands_batch = torch.stack(test_bands_batch_list).to(device)
            test_targets_batch = torch.tensor(test_targets_batch_list, dtype=torch.float32).unsqueeze(1).to(device)
            with torch.no_grad(): test_predictions = classifier_nn(test_bands_batch)
            for j in range(len(xor_data)):
                test_prediction_val = test_predictions[j].item(); test_target = test_targets_batch[j].item()
                test_loss = (test_target - test_prediction_val)**2; total_test_loss += test_loss
                predicted_xor = round(test_prediction_val)
                if predicted_xor == test_target: correct_predictions += 1
                print(f"Test Input: {xor_data[j][0]}, Target: {int(test_target)}")
                print(f"  Pred Val: {test_prediction_val:.4f}, Pred XOR: {predicted_xor}, Loss: {test_loss:.4f}")

            avg_test_loss_current_iter_report = total_test_loss / len(xor_data)
            print(f"  Average Test Loss (current best BS params): {avg_test_loss_current_iter_report:.4f}")
            print(f"Correct: {correct_predictions}/{len(xor_data)}")
            print("---------------------------------------\n")
            if correct_predictions == len(xor_data) and avg_test_loss_current_iter_report < MIN_LOSS:
                print(f"\nConverged at Iteration {iteration}!"); break
    end_time = time.time()
    print(f"\n--- Optimization Finished ---"); print(f"Total time elapsed: {end_time - start_time:.2f} seconds")
    print(f"Best Burning Ship Parameters Found: {best_bs_params}")
    print(f"Lowest Average Classifier Loss Achieved: {best_overall_loss:.6f}")
    print("\n--- Final Evaluation with Best Parameters ---")
    classifier_nn.eval()
    for xor_input_val, xor_target in xor_data: # Pass the xor_input_val to the GOL function for final eval
        magnitude_spectrum = run_gol_and_3d_fft_torch(best_bs_params, xor_input_val)
        predicted_bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
        with torch.no_grad(): final_prediction_val = classifier_nn(predicted_bands.unsqueeze(0)).item()
        final_predicted_xor = round(final_prediction_val)
        print(f"Input: {xor_input_val}, Target: {xor_target}")
        print(f"Final Pred Val: {final_prediction_val:.4f}, Predicted XOR: {final_predicted_xor}")

if __name__ == '__main__':# Run the optimization
    solve_xor_with_burning_ship_and_classifier_torch()
```

OUTPUT

```
Converged at Iteration 27!

--- Optimization Finished ---
Total time elapsed: 63.79 seconds
Best Burning Ship Parameters Found: [-0.0977173116643562, 0.8158841281631091, 3.39380398526714, 120]
Lowest Average Classifier Loss Achieved: 0.009079

--- Final Evaluation with Best Parameters ---
Input: [0, 0], Target: 0, Final Pred Val: 0.0455, Predicted XOR: 0
Input: [0, 1], Target: 1, Final Pred Val: 0.8596, Predicted XOR: 1
Input: [1, 0], Target: 1, Final Pred Val: 0.9572, Predicted XOR: 1
Input: [1, 1], Target: 0, Final Pred Val: 0.0893, Predicted XOR: 0
```

References:

[1] Iterating a Fractal-like Self Awareness Naturally https://ai.vixra.org/abs/2505.0195