Scientific Machine Learning for the No-Three-In-Line Problem: Energy-Based Continuous Dynamics and Neural ODEs

1 Motivation and Role in the Overall Framework

Previous work in this project has compared three main paradigms for the No-Three-In-Line problem:

- **Integer Linear Programming (ILP):** Provides provably optimal solutions but scales poorly with grid size.
- **PatternBoost Transformer:** Learns patterns from data and achieves near-optimal performance on moderate grids.
- **Reinforcement Learning (PPO):** Explores the combinatorial space via sequential decisions; effective on small grids but struggles as constraints tighten.

The SciML extension adds a fourth viewpoint:

> *Treat the discrete grid as a continuous field evolving under an ODE / Neural ODE, with a carefully designed energy functional whose minima correspond to good (or optimal) configurations.*

This gives:

- A physics-inspired, interpretable formulation.
- A continuous relaxation with polynomial-time ODE integration.
- A natural basis for hybrid methods (e.g., using SciML solutions as warm starts for ILP or RL).

2 Continuous Relaxation and State Representation

We relax the binary grid to continuous variables.

2.1 Grid and variables

Let n be the grid size. The discrete decision variables are:

$$x_{ij} \in \{0,1\}, \qquad i,j = 1,\dots,n.$$

We introduce a continuous relaxation:

$$x_{ij}(t) \in [0,1], \qquad i,j = 1,\dots,n,\ t \in [0,T],$$

and collect them into a vector:

$$\mathbf{x}(t) \in [0,1]^{n^2}.$$

Interpretation:

- $x_{ij}(t) \approx 1$: cell (i,j) contains a point.
- $x_{ij}(t) \approx 0$: cell (i,j) is empty.

Our goal is to define dynamics

$$\dot{\mathbf{x}}(t) = f_\theta(\mathbf{x}(t), t)$$

that drive $\mathbf{x}(t)$ toward low-energy configurations that are dense, nearly binary, and obey the "no three in a line" constraint.

3 Energy Functional Design

Let L denote the set of all triplets of collinear grid positions:

$$L = \Big\{ ((i_1,j_1),(i_2,j_2),(i_3,j_3)) \;\Big|\; \text{three distinct collinear cells} \Big\}.$$

The energy $E(\mathbf{x})$ combines three terms.

3.1 Collinearity penalty

We penalize any triple of cells that are simultaneously active:

$$E_{\text{collinear}}(\mathbf{x}) = \sum_{((i_1,j_1),(i_2,j_2),(i_3,j_3)) \in L} \alpha \, x_{i_1 j_1} x_{i_2 j_2} x_{i_3 j_3}, \tag{1}$$

where $\alpha > 0$ controls the strength of the penalty. If three cells along a line are close to 1, their product becomes large and the energy increases.

3.2 Point-count reward

We reward the presence of points by decreasing the energy when x_{ij} is large:

$$E_{\text{count}}(\mathbf{x}) = -\beta \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}, \tag{2}$$

with $\beta > 0$. Minimizing this term pushes the system toward grids with many active cells.

3.3 Binary regularization

To avoid trivial fractional configurations (e.g., $x_{ij} \approx 0.5$ everywhere), we use a double-well potential:

$$E_{\text{binary}}(\mathbf{x}) = \gamma \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}^2 (1 - x_{ij})^2, \tag{3}$$

with $\gamma > 0$. This term has minima at $x_{ij} = 0$ and $x_{ij} = 1$, encouraging near-binary states.

3.4 Total energy

The total energy is:

$$E(\mathbf{x}; \alpha, \beta, \gamma) = E_{\text{collinear}}(\mathbf{x}) + E_{\text{count}}(\mathbf{x}) + E_{\text{binary}}(\mathbf{x}). \tag{4}$$

The scalars α, β, γ can be:

- hand-tuned for a simple baseline, or
- learned via SciML as trainable parameters.

4 Dynamics: Gradient Flow and Neural ODE

4.1 Pure gradient-flow ODE

A natural choice is gradient flow in continuous time:

$$\dot{\mathbf{x}}(t) = -\nabla_{\mathbf{x}} E(\mathbf{x}(t)). \tag{5}$$

This is continuous-time gradient descent on the energy landscape. Fixed points correspond to local minima of E.

In SciML, we implement this as an ODE:

$$\frac{d}{dt}\mathbf{x}(t) = f(\mathbf{x}(t), t; \alpha, \beta, \gamma),$$

where f computes $-\nabla E$ using automatic differentiation.

4.2 Physics-informed Neural ODE

To increase expressive power, we can add a learned correction:

$$\dot{\mathbf{x}}(t) = -\nabla_{\mathbf{x}} E(\mathbf{x}(t)) + g_\theta(\mathbf{x}(t), t), \tag{6}$$

where g_θ is a neural network (e.g., MLP or convolutional network over the grid), parameterized by θ.

This yields a *physics-informed Neural ODE*, where:

- $-\nabla E$ encodes known structure and constraints.
- g_θ learns to improve convergence and escape poor local minima.

5 Training Objectives

We describe two training regimes, one fully unsupervised and one teacher-guided using ILP solutions.

5.1 Unsupervised energy-minimization training

In unsupervised mode, we simply ask the system to learn parameters that produce low-energy configurations from random initial states.

Let $\mathbf{x}^{(b)}(t)$ denote the solution trajectory for the b-th initial condition, and let $\mathbf{x}^{(b)}(T)$ be the final state at terminal time T. The unsupervised loss is:

$$\mathcal{L}_{\text{energy}}(\theta, \alpha, \beta, \gamma) = \frac{1}{B} \sum_{b=1}^{B} E(\mathbf{x}^{(b)}(T); \alpha, \beta, \gamma), \tag{7}$$

where B is the batch size. Parameters are updated by backpropagating through the ODE using adjoint sensitivity methods.

5.2 Teacher-guided training with ILP or PatternBoost

When optimal (or near-optimal) discrete solutions $\hat{\mathbf{y}}^{(k)} \in \{0,1\}^{n^2}$ are available from ILP or PatternBoost for small grids, we can add a teacher loss.

For the k-th training instance, with final state $\mathbf{x}^{(k)}(T)$, define:

$$\mathcal{L}_{\text{teacher}} = \frac{1}{K} \sum_{k=1}^{K} \left\| \sigma(\mathbf{x}^{(k)}(T)) - \hat{\mathbf{y}}^{(k)} \right\|_2^2, \tag{8}$$

where $\sigma(\cdot)$ is a smooth squashing function (e.g., elementwise sigmoid), and K is the number of teacher-labeled instances in a batch.

The total loss becomes:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{energy}} + \lambda_{\text{teacher}} \mathcal{L}_{\text{teacher}}, \tag{9}$$

with a hyperparameter $\lambda_{\text{teacher}} \geq 0$ controlling the strength of teacher guidance.

6 From Continuous to Discrete Solutions

After solving the ODE from $t = 0$ to $t = T$, we obtain $\mathbf{x}(T)$. To obtain a discrete grid:

1. **Thresholding:**
$$\hat{x}_{ij} = \begin{cases} 1, & x_{ij}(T) \geq \tau, \\ 0, & x_{ij}(T) < \tau, \end{cases}$$
with threshold $\tau \in (0,1)$ (e.g., $\tau = 0.5$).

2. **Local repair (optional):** For any line that contains three or more active points, we greedily deactivate points to satisfy the constraint, prioritizing removals that minimize the decrease in total point count.

This yields a feasible configuration that can be evaluated against the ILP optimum.

7 Training-Time and Complexity Analysis

Let:

- n: grid size.

- $d = n^2$: number of continuous variables.
- $|L|$: number of collinear triples; asymptotically $|L| = O(n^3)$.
- N_t: number of time steps (RHS evaluations) used by the ODE solver per trajectory.
- B: batch size (number of trajectories per iteration).
- N_{iter}: number of training iterations.

7.1 Cost per ODE solve

Each evaluation of the vector field requires computing the gradient $\nabla_{\mathbf{x}} E(\mathbf{x})$.

- Computing $E_{\text{collinear}}(\mathbf{x})$ is $O(|L|)$ since we sum over all collinear triples.
- With automatic differentiation, computing $\nabla_{\mathbf{x}} E(\mathbf{x})$ is also $O(|L|)$ up to a constant factor.

The ODE solver requires N_t evaluations, so:

$$\text{Cost per ODE solve} \approx O(N_t\,|L|) = O(N_t\,n^3).$$

7.2 Cost per training iteration and total cost

With batch size B, a single training iteration requires B ODE solves:

$$\text{Cost per iteration} \approx O(B\,N_t\,n^3).$$

Over N_{iter} iterations:

$$\text{Total training cost} \approx O(N_{\text{iter}}\,B\,N_t\,n^3).$$

7.3 Qualitative comparison

- **ILP:** Worst-case exponential in problem size; empirically hits practical limits around $n = 19$ for exact solutions.
- **PatternBoost / PPO:** Training cost grows with number of episodes, model size, and horizon length; the combinatorial action space becomes challenging at large n.
- **SciML (this work):** Training dominated by ODE solves with cost $O(n^3)$ per trajectory, plus backpropagation. This offers a different scaling profile and potentially better behavior than ILP for moderate-to-large grids.

This analysis can be directly reported in the ICML paper to highlight the trade-offs between methods.

8 Experimental Plan for SciML

For each grid size $n \in \{8, 10, 12, 14, 16, 19\}$, we propose to evaluate:

- **SciML-GF:** Pure gradient-flow dynamics ($g_\theta \equiv 0$).
- **SciML-NODE:** Physics-informed Neural ODE with non-zero g_θ.

For each method and each n:

1. Run R trajectories from different random initial conditions (e.g., $R = 100$).
2. Integrate the ODE to terminal time T, apply thresholding and optional repair.
3. Measure:
   - Feasible success rate (fraction of runs with zero collinear triples).
   - Average number of points in feasible solutions.
   - Best-of-R number of points.
   - Average training time per iteration and total training time.
   - Inference time per trajectory (single ODE solve).

These metrics can be compared to ILP (optimal where available), PatternBoost, and PPO.

9 Pseudocode

9.1 Precomputing collinear triples

Algorithm 1 Compute collinear triples L for an $n \times n$ grid

```
function ComputeCollinearTriples($n$)
    $L \leftarrow \emptyset$
    coords $\leftarrow \{(i, j) \mid i = 1..n, j = 1..n\}$
    for each triple of distinct points $(p_1, p_2, p_3)$ from coords do
        if Collinear($p_1, p_2, p_3$) then
            Append $(p_1, p_2, p_3)$ to $L$
        end if
    end for
    return $L$
end function
```

Two points (x_1, y_1), (x_2, y_2), (x_3, y_3) are collinear if the triangle area is zero:

$$x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) = 0.$$

9.2 Energy and vector field

9.3 Training loop (high-level)

Algorithm 2 Energy and ODE vector field

```
function ENERGY($\mathbf{x}, L, \alpha, \beta, \gamma$)
    $E_{\text{col}} \leftarrow 0$
    for each triple $(p_1, p_2, p_3) \in L$ do
        $i_1 \leftarrow$ INDEXOF($p_1$)
        $i_2 \leftarrow$ INDEXOF($p_2$)
        $i_3 \leftarrow$ INDEXOF($p_3$)
        $E_{\text{col}} \leftarrow E_{\text{col}} + \alpha\, x_{i_1} x_{i_2} x_{i_3}$
    end for
    $E_{\text{count}} \leftarrow -\beta \sum_k x_k$
    $E_{\text{binary}} \leftarrow \gamma \sum_k x_k^2 (1 - x_k)^2$
    return $E_{\text{col}} + E_{\text{count}} + E_{\text{binary}}$
end function
function VECTORFIELD($\mathbf{x}, t, \theta, \alpha, \beta, \gamma, L$)
    $g \leftarrow \nabla_{\mathbf{x}}$ENERGY($\mathbf{x}, L, \alpha, \beta, \gamma$)
    physics $\leftarrow -g$
    if $\theta$ is used then
        correction $\leftarrow g_\theta(\mathbf{x}, t)$
    else
        correction $\leftarrow 0$
    end if
    return physics + correction
end function
```

Algorithm 3 SciML Training Loop

```
Initialize parameters $\theta, \alpha, \beta, \gamma$
$L \leftarrow$ COMPUTECOLLINEARTRIPLES($n$)
for iter = 1 to $N_{\text{iter}}$ do
    $\mathcal{L} \leftarrow 0$
    for $b = 1$ to $B$ do
        Sample random initial state $\mathbf{x}_0^{(b)}$
        Solve ODE: $\mathbf{x}^{(b)}(T) \leftarrow$ SOLVEODE(VECTORFIELD, $\mathbf{x}_0^{(b)}, [0, T]$)
        $\mathcal{L} \leftarrow \mathcal{L} +$ ENERGY($\mathbf{x}^{(b)}(T), L, \alpha, \beta, \gamma$)
        if teacher data is available then
            Sample teacher solution $\hat{\mathbf{y}}^{(b)}$
            $\mathcal{L} \leftarrow \mathcal{L} + \lambda_{\text{teacher}} \|\sigma(\mathbf{x}^{(b)}(T)) - \hat{\mathbf{y}}^{(b)}\|_2^2$
        end if
    end for
    $\mathcal{L} \leftarrow \mathcal{L}/B$
    Compute gradients of $\mathcal{L}$ w.r.t. $\theta, \alpha, \beta, \gamma$
    Update parameters with chosen optimizer (e.g., Adam)
end for
```

10 Julia SciML Skeleton (Simulation Only)

Below is a compact, self-contained Julia example that:

- builds an $n \times n$ grid,
- precomputes collinear triples,
- defines an energy functional,
- implements gradient-flow dynamics via finite-difference gradient (for clarity), and
- solves the ODE from a random initial condition and thresholds the final state.

You can replace the finite-difference gradient with automatic differentiation (e.g., Zygote) and wrap this in DiffEqFlux for full training.

```julia
using OrdinaryDiffEq
using Random

# -------------------------------
# 1. Grid & index helpers
# -------------------------------
const n = 6 # example grid size
const N = n * n # total variables

# Map (i, j) -> linear index in 1..N
linear_index(i, j) = (i - 1) * n + j

# -------------------------------
# 2. Precompute collinear triples
# -------------------------------
function compute_collinear_triples(n::Int)
    triples = Tuple{Tuple{Int,Int},Tuple{Int,Int},Tuple{Int,Int}}[]
    coords = [(i, j) for i in 1:n for j in 1:n]

    # Helper: check collinearity of three lattice points
    function collinear(p1, p2, p3)
        (x1, y1) = p1
        (x2, y2) = p2
        (x3, y3) = p3
        # Area of triangle == 0 -> collinear
        return (x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2)) == 0
    end

    # Enumerate all distinct 3-combinations of points
    L = length(coords)
    for a in 1:(L-2), b in (a+1):(L-1), c in (b+1):L
        p1 = coords[a]
        p2 = coords[b]
        p3 = coords[c]
        if collinear(p1, p2, p3)
            push!(triples, (p1, p2, p3))
        end
    end
    return triples
end

const L_triples = compute_collinear_triples(n)

# -------------------------------
# 3. Energy function
# -------------------------------
function energy(x::Vector{Float64};
               ::Float64 = 5.0,
               ::Float64 = 1.0,
               ::Float64 = 2.0)

    # Collinearity term
    E_col = 0.0
    @inbounds for (p1, p2, p3) in L_triples
        i1 = linear_index(p1[1], p1[2])
        i2 = linear_index(p2[1], p2[2])
        i3 = linear_index(p3[1], p3[2])
        E_col +=  * x[i1] * x[i2] * x[i3]
    end

    # Count term (reward more points)
    E_count = - * sum(x)

    # Binary regularization (double well at 0 and 1)
    E_bin = 0.0
    @inbounds for k in 1:length(x)
        E_bin += x[k]^2 * (1.0 - x[k])^2
    end
    E_bin *=

    return E_col + E_count + E_bin
end

# -------------------------------
# 4. Approximate gradient via finite differences
# (for demonstration; replace with AD in real experiments)
# -------------------------------
function grad_energy_fd(x::Vector{Float64};
                      h::Float64 = 1e-5,
                      ::Float64 = 5.0,
                      ::Float64 = 1.0,
                      ::Float64 = 2.0)
    g = similar(x)
    E0 = energy(x; =, =, =)
    for k in 1:length(x)
        x[k] += h
        Ek = energy(x; =, =, =)
        x[k] -= h
        g[k] = (Ek - E0) / h
    end
    return g
```

```
end

# -------------------------------
# 5. ODE vector field: pure gradient flow
# -------------------------------
function f!(dx, x, p, t)
    , ,  = p
    g = grad_energy_fd(x; =, =, =)
    @. dx = -g # gradient descent direction
end

# -------------------------------
# 6. Solve the ODE from a random initial condition
# -------------------------------
Random.seed!(1234)
x0 = rand(N) # random in (0,1)

tspan = (0.0, 5.0) # time horizon
p = (5.0, 1.0, 2.0) # (, , )
prob = ODEProblem(f!, x0, tspan, p)
sol = solve(prob, Tsit5(); reltol=1e-4, abstol=1e-6)

# Extract final state and threshold
xT = Array(sol(end))
 = 0.5
x_binary = [val >=  ? 1 : 0 for val in xT]
grid = reshape(x_binary, (n, n))

println("Final discrete grid (1 = point, 0 = empty):")
for i in 1:n
    println(grid[i, :])
end
```

This skeleton can be extended in several directions:

- Replace `grad_energy_fd` with an automatic differentiation-based gradient.
- Wrap the ODE in a DiffEqFlux training loop to learn α, β, γ or parameters of a neural correction term g_θ.
- Scale up n and integrate this module into your ILP / PatternBoost / PPO comparison.