NOTEBOOK GUIDE

Notebook Guide

Overview

This project trains neural networks to act as Gene Regulatory Networks (GRNs) that optimize cell fate pattern formation. Cells evolve through stochastic dynamics governed by a regulatory function `f(s̄)`, and the goal is to discover patterns that maximize an information-theoretic utility function: **U = S_pattern - S_reproducibility**.

The project successfully implements evolutionary training (ES) to discover lateral inhibition patterns, and investigates gradient-based training via backpropagation, however not yet to the desired stability and effeciency.

Notebook 01: The Utility Function

Purpose:
Derives, implements, and validates the utility function that serves as the optimization objective for the entire project.

Key Features:

- **Shannon Entropy Calculations**: Pattern entropy (S_pat) and reproducibility entropy (S_rep)
- **Hard Utility Function**: Uses `jnp.unique` to count discrete patterns (non-differentiable)
- **Soft Utility Function**: Differentiable approximation using sigmoid soft thresholding and KDE
- **Utility Landscape Visualization**: Explores how utility varies with fate ratios and pattern diversity

Key Insight:
The optimal patterns achieve high pattern entropy (diverse cell fates) while maintaining low reproducibility entropy (consistent patterns across replicates).

Mathematical Methods:

- Shannon entropy: $H = -\Sigma\ p\ \log_2(p)$
- Binary pattern encoding via base-2 representation
- Kernel Density Estimation (KDE) for differentiable pattern probability distributions
    - See full derivation: Differentiable Reproducibility Entropy via Kernel Density Estimation

Notebook 02: The Dynamics

Purpose:
Implements the stochastic differential equation (SDE) that governs cell state evolution over time.

Key Features:

- **Euler-Maruyama Integration**: Numerical method for SDEs with Gaussian noise
- **Neighbor Averaging**: Cells sense the average state of immediate neighbors (diffusive communication)
- **State Clipping**: Maintains states within [0,1] bounds
- **Parallel Replicates**: Uses `jax.vmap` to run multiple independent simulations in parallel
- **Trajectory Visualization**: Space-time plots showing cell state evolution

Mathematical Framework:

- Stochastic differential equation: `ds/dt = f(s̄) + η`
- Euler-Maruyama: `s(t+Δt) = s(t) + f(s̄) * Δt + σ * √Δt * ε`, where `ε ~ N(0,1)`
- Fixed boundary conditions (edge cells have only one neighbor)

Key Technical Features:

- JAX JIT compilation for performance
- Vectorization with `vmap` for batch operations
- Functional programming style (pure functions, no side effects)

Notebook 03: The Regulatory Network

Purpose:
Sets up the neural network architecture that serves as the regulatory function `f(s̄)`.

Key Features:

- **Flax Neural Network**: Small feedforward network (default: 3 hidden layers × 8 neurons)
- **Architecture**: Input (scalar s̄) → Hidden Layers (tanh activation) → Output (scalar ds/dt)
- **Parameter Management**: Initialization, flattening/unflattening for evolutionary algorithms
- **Visualization Tools**: Plot learned regulatory functions

Technical Implementation:

- Flax `nn.Module` class for network definition
- Parameter initialization with configurable random keys

- Conversion between Flax pytree structure and flat 1D arrays (needed for ES)
- Regulatory function closure that captures trained parameters

---

Notebook 04: Training The Network

Purpose:
Implements evolutionary strategy (ES) training to discover optimal regulatory functions.

Key Features:

- **(μ+λ)-Evolution Strategy**: Population-based, gradient-free optimization
- **Fitness Evaluation**: Uses hard utility function to score parameter sets
- **Gaussian Mutations**: Add noise to parameters to explore search space
- **Elitism**: Keep top-performing individuals across generations
- **JIT-Compiled Population Evaluation**: Massive speedup (15-40×) via `jax.vmap` + `@jit`

Training Configuration:

- Population size: 20 individuals
- Generations: 200
- Mutation standard deviation: 0.1
- Number of elite parents: 5
- Replicates per evaluation: 100
- Simulation time: T=20.0 (2000 steps)

Results:
Successfully trains networks to discover **lateral inhibition**: cells suppress neighbors when their average state is high, leading to alternating on-off patterns (e.g., [0,1,0,1,0,1,0]).

Why ES Works:

- Evaluates final outcomes directly (no gradient tracing required)
- Robust to noise and discrete operations
- Industry standard for differentiable simulation problems

Notebook 05: Backpropagation

Purpose:
Attempts gradient-based training using Adam optimizer and soft differentiable utility.

Key Features:

- **Soft Thresholding**: Sigmoid instead of hard binary threshold (enables gradient flow)
- **Soft Utility via KDE**: Differentiable pattern probability estimation
- **Straight-Through Estimator (STE)**: Custom gradient for threshold operation
- **Adam Optimizer**: Adaptive learning rate optimization

Result:
Training fails - gradients vanish to zero, loss remains constant, no learning occurs.

Why Backpropagation Fails (brief summary):

1. **Long simulations**: 2000 sequential steps cause vanishing gradients (similar to deep RNN problem)
2. **Stochastic noise**: Gradient signal destroyed by accumulated randomness
3. **Multiple saturation points**: Tanh activations, state clipping, soft threshold sigmoid all contribute
4. **Weak sensitivity**: Large KDE bandwidth makes utility insensitive to small pattern changes

Documentation: See `FixTheBackprop.md` for detailed analysis and proposed solutions.

Key Takeaway:
This demonstrates the fundamental challenge of differentiable simulation through long stochastic trajectories - an active research area in differentiable physics, neural ODEs, and model-based RL.

Notebook 06: Evolve And Select Fixed Boundary

Purpose:
Extension of Notebook 04 with fixed boundary conditions (edge cells pinned to state 0.0).

Key Features:

- **Modified Dynamics**: `simulate_fixed_boundary()` enforces edge states = 0.0 at every timestep
- **Same ES Training**: Identical evolutionary algorithm as Notebook 04
- **Boundary Constraints**: Tests whether fixed boundaries affect pattern discovery

Technical Differences:

- Uses `get_neighbor_average_fixed_boundary()` for edge handling
- Edge cells forced to state 0.0 before neighbor averaging
- All other components identical to standard training

Purpose of Fixed Boundaries:
Explores whether boundary constraints help or hinder the discovery of alternating patterns, providing insight into the role of boundary conditions in pattern formation.

Supporting Code Structure

`src/utility_function.py`

Contains all entropy and utility calculations (both hard and soft versions).

Key Functions:

- `compute_entropy()` : Shannon entropy implementation
- `compute_pattern_entropy()` : Pooled cell fate distribution
- `compute_hard_utility()` : Non-differentiable version (for ES)
- `compute_soft_utility()` : Differentiable version via KDE (for backprop)
- `apply_soft_threshold()` : Sigmoid approximation of binary threshold

`src/dynamics.py`

Implements the stochastic dynamics simulation.

Key Functions:

- `euler_step()` : Single timestep of Euler-Maruyama integration
- `simulate()` : Full trajectory simulation
- `apply_threshold()` : Convert continuous states to binary patterns (with STE gradient)
- `run_multiple_replicates()` : Parallel execution via `jax.vmap`
- Fixed boundary variants: `*_fixed_boundary()` versions of above

`src/neural_network.py`

Neural network architecture and evolutionary algorithm components.

Key Functions:

- `RegulatoryNetwork` : Flax module defining network architecture
- `init_params()` : Initialize network weights
- `flatten_params()` / `unflatten_params()` : Convert between pytree and 1D array
- `compute_fitness()` : Evaluate utility for parameter set
- `evaluate_population()` : JIT-compiled batch fitness evaluation
- `mutate_population()` : Gaussian mutation with elitism

Additional Documentation

`FixTheBackprop.md`

Comprehensive analysis of why gradient-based training fails, including:

- Detailed investigation of vanishing gradients
- Gradient flow analysis through different components
- Proposed solutions (reduce simulation length, lower noise, adjust hyperparameters)
- Comparison between ES and backpropagation approaches

`Differentiable Reproducibility Entropy via Kernel Density Estimation.md` (Obsidian Vault)

Mathematical derivation of the soft utility function:

- Problem statement (non-differentiability of discrete operations)
- Solution pipeline (soft discretization + KDE)
- Proof that resubstitution estimator recovers Shannon entropy
- JAX implementation details

Located at: `/Users/johannes/Library/CloudStorage/OneDrive-Persönlich/Johannes Vault/PHD/Basel/MyCellFateNNRepoDoc/`

Key Technologies Used

- **JAX**: Auto-differentiation, JIT compilation, GPU-ready computation
- **Flax**: Neural network library built on JAX
- **NumPy**: Array operations and numerical computing
- **Matplotlib/Seaborn**: Visualization
- **Jupyter**: Interactive development and exploration

Expected Outcome

The trained neural network converges to a **lateral inhibition function** approximately:

```
f(s̄) ≈ −a·tanh(b·(s̄ − 0.5))
```

This function causes cells to:

- Decrease when surrounded by high-state neighbors
- Increase when surrounded by low-state neighbors
- Result: Self-organized alternating patterns with maximal utility

Project Status

✅ **Completed**:

- Utility function implementation (hard + soft)
- Dynamics simulation (Euler-Maruyama with noise)
- Neural network architecture
- Evolutionary training (successfully discovers lateral inhibition)
- Fixed boundary variant

⚠️ **Investigated but unsuccessful**:

- Gradient-based training (backpropagation fails due to vanishing gradients)

Running the Code

See `README.md` for setup instructions:

```
# Setup environment
python -m venv venv
source venv/bin/activate
pip install jax jaxlib flax numpy matplotlib seaborn jupyter

# Run notebooks
jupyter notebook notebooks/
```

Run notebooks in order (01 → 06) to follow the development progression.