Differentiable Reproducibility Entropy via Kernel Density Estimation

1. The Problem

We have a system of cells that evolve into binary patterns (states of 0 or 1). We want to train a neural network to maximize the **reproducibility** of these patterns.

The standard metric is **Reproducibility Entropy**:

$$H = -\sum_{k \in \text{Unique Patterns}} p_k \log(p_k)$$

However, this metric is **non-differentiable** for two reasons:

1. **Thresholding:** The step function mapping $s < 0.5 \to 0$ has zero gradient.
2. **Counting:** The `unique` function used to calculate probabilities (p_k) is discrete and breaks the computation graph.

To fix this, we replace the discrete operations with differentiable approximations using **Soft Discretization** and **Kernel Density Estimation (KDE)**.

---

2. The Solution Pipeline

Step A: Soft Discretization (The Sigmoid)

First, we smooth the "cliff" of the threshold function. Instead of a hard step, we use a sigmoid function with a temperature parameter T.

$$\tilde{s} = \sigma(T \cdot (s - 0.5)) = \frac{1}{1 + e^{-T(s-0.5)}}$$

- **Low T:** Smooth, linear-like transition.
- **High T:** Approaches a hard binary step function.

Step B: Soft Matching (The Gaussian Kernel)

In the discrete world, two patterns x and y are either identical (distance 0) or not. In the continuous world, we measure similarity using a **Gaussian Kernel**.

We define the squared Euclidean distance between two patterns x_i and x_j as $d_{ij}^2 = ||x_i - x_j||^2$.

The similarity (kernel value) is:

$$K(x_i, x_j) = \exp\left(-\frac{d_{ij}^2}{2\sigma^2}\right)$$

Key Properties:

1. **If $x_i \approx x_j$:** Distance is ≈ 0, so $K \approx 1$.
2. **If $x_i \neq x_j$:** Distance is large, so $K \approx 0$.
3. **No Normalization:** We deliberately do **not** include the standard normalization factor ($\frac{1}{\sqrt{2\pi\sigma^2}}$). We want the "self-similarity" to be exactly 1.0, effectively counting "1 match".

Step C: Soft Counting

To estimate how frequent a specific pattern x_i is within a batch of size N, we simply sum the similarities it has with every other pattern in the batch.

$$C_i = \sum_{j=1}^{N} K(x_i, x_j)$$

- **Interpretation:** C_i represents the "Soft Count" of pattern i. If there are 5 identical patterns, $C_i \approx 5$.

Step D: Estimating Probabilities

We convert the soft count into an estimated probability $\hat{p}_i$ by normalizing over the batch size N.

$$\hat{p}_i = \frac{C_i}{N}$$

---

3. The Entropy Calculation & "Double Counting"

We calculate the entropy using the **Resubstitution Estimator**, which averages the log-probability over the samples (rows) rather than summing over unique types.

$$H_{batch} = -\frac{1}{N} \sum_{i=1}^{N} \log(\hat{p}_i)$$

Why does this work? (Addressing Double Counting)

It might seem incorrect to sum over N rows if many rows are identical. However, this summation naturally performs the weighting required by Shannon Entropy.

The Proof:
Let there be a unique pattern type U that appears M times in the batch.

1. For any sample i that is an instance of U, the soft count is $C_i \approx M$.

2. The estimated probability is $\hat{p}_i \approx \frac{M}{N}$.
3. This specific value, $\log(\frac{M}{N})$, appears in our summation exactly M times (once for each row).

We can rewrite the sum by grouping the unique patterns:

$$\begin{aligned} H_{batch} &= -\frac{1}{N} \sum_{i=1}^{N} \log(\hat{p}_i) \\ &= -\frac{1}{N} \left[\sum_{U \in \text{Unique}} M_U \cdot \log\left(\frac{M_U}{N}\right) \right] \\ &= -\sum_{U \in \text{Unique}} \left(\frac{M_U}{N}\right) \cdot \log\left(\frac{M_U}{N}\right) \end{aligned}$$

Since $\frac{M_U}{N}$ is exactly the probability $P(U)$, we recover the standard Shannon Entropy formula:

$$H = -\sum_{U \in \text{Unique}} P(U) \log P(U)$$

Thus, iterating over the rows and "double counting" correctly applies the probability weight $P(U)$ to the term.

---

4. JAX Implementation

Here is the implementation combining these concepts.

```
import jax
import jax.numpy as jnp

def differential_entropy_loss(final_states, temperature=10.0, sigma=0.5):
    """
    Calculates the differentiable entropy of the pattern distribution.

    Args:
        final_states: (Batch, Cells) array in range [0, 1]
        temperature:  Steepness of the sigmoid (Soft Discretization)
        sigma:        Bandwidth of the kernel (Soft Matching)

    Returns:
        Scalar entropy value.
        0.0       => Perfect reproducibility (All patterns identical)
        ln(Batch) => Zero reproducibility (All patterns unique)
    """
    batch_size = final_states.shape[0]

    # 1. Soft Discretization
    # Push values towards 0 or 1
```

```
    soft_patterns = jax.nn.sigmoid(temperature * (final_states - 0.5))

    # 2. Pairwise Distance Matrix
    # Shape: (Batch, Batch, Cells)
    diff = soft_patterns[:, None, :] - soft_patterns[None, :, :]

    # Squared Euclidean Distance
    # Shape: (Batch, Batch)
    dists_sq = jnp.sum(diff**2, axis=-1)

    # 3. Soft Counting (Unnormalized Gaussian Kernel)
    # K(x, y) = exp(-d^2 / 2*sigma^2)
    # We use log-space for stability: log(K) = -d^2 / 2*sigma^2
    log_kernels = -dists_sq / (2 * sigma**2)

    # 4. Log-Probability Estimation
    # log(p_i) = log( sum(exp(log_kernels)) ) - log(N)
    # This represents log(Count / N)
    log_probs = jax.scipy.special.logsumexp(log_kernels, axis=1) -
jnp.log(batch_size)

    # 5. Entropy Calculation (Resubstitution Estimator)
    # H = - Mean( log(p_i) )
    entropy = -jnp.mean(log_probs)

    # Optional: Normalize by system size if needed
    # return entropy / final_states.shape[1]
    return entropy
```