Hyperbolic Restricted Boltzmann Machines for Hierarchical Learning

Kylie Hefner
M.S. Computational and Data Sciences, Chapman University

May 23, 2025

Abstract

Euclidean Restricted Boltzmann Machines (RBMs) struggle with strongly hierarchical data because the flat geometry forces wide, inefficient embeddings. Hyperbolic space, which expands exponentially with radius, offers a better solution. Building on Kobayashi's information–geometric theory of hyperbolic–valued Boltzmann Machines, I implement a Hyperbolic RBM (HRBM) and benchmark it against a Bernoulli RBM on a 1.2 k-node WordNet sub-hierarchy. The HRBM lowers reconstruction MAE by 78 −82 % and improves F_1 by 41 −53 % while training for only three epochs.

1 Background

1.1 Why Hyperbolic Geometry Fits Hierarchies

Consider a tree with depth d where each node has at most b children. The number of nodes grows exponentially with depth, about b^d. In contrast, euclidean volume in $\mathbb{R}^n$ only grows polynomially (r^n). To embed deep trees in Euclidean space, you need to push nodes farther apart (which increases distortion) or use more dimensions (which increases model size and complexity).

Hyperbolic space behaves differently. In the Poincaré ball model $\mathbb{B}^n$ (negative curvature $\kappa = -1$), the volume and surface area grow exponentially with radius, e^r. This matches the growth pattern of trees, so each level of a hierarchy can be represented at a fixed radial distance without crowding. As a result, hierarchical structures can be embedded much more compactly and accurately in low-dimensional hyperbolic space than in Euclidean space. For example, structures that require hundreds of Euclidean dimensions may fit cleanly into just two or three hyperbolic dimensions [2].

1.2 Boltzmann Machines

Boltzmann Machines are stochastic, generative neural networks that learn patterns by minimizing an energy function:

$$E(v,h) = -v^T W h - a^T v - b^T h$$

where v is the visible units vector , h is the hidden units vector, W is the weight matrix, a is the visible biases vector, and b is the hidden biases vector.

The probability of a configuration is given by the Boltzmann distribution:

$$P(v,h) = \frac{e^{-E(v,h)}}{Z}$$

where Z is a normalizing function, the sum of all possible state probabilities.

Think of $E(v,h)$ as a landscape of hills and valleys; learning shapes the landscape so that real samples fall into deep valleys and synthetic noise rolls away. A *Restricted* Boltzmann Machine (RBM) blocks all visible–visible and hidden–hidden edges, leaving a bipartite graph that is more computationally efficient to sample via Gibbs steps.

1.3 Hyperbolic Numbers

Standard RBMs operate in $\mathbb{R}$: parameters are real-valued scalars, and the energy function uses the standard dot product. Kobayashi [4] proposed extending this to *hyperbolic numbers*, written as $z = a + b\,\mathbf{j}$, where $\mathbf{j}^2 = +1$. These resemble complex numbers, but with a key difference: the squared norm becomes $|z|^2 = a^2 - b^2$ instead of $a^2 + b^2$.

This change leads to a new kind of inner product, called the Lorentzian inner product:

$$\langle u, v\rangle_{\mathbb{H}} = u_a v_a - u_b v_b,$$

which matches the geometry of hyperbolic space. It allows us to write an energy function with the same structure as the original RBM:

$$E_{\mathbb{H}}(v,h) = -\langle v, Wh\rangle_{\mathbb{H}} - \langle b, v\rangle_{\mathbb{H}} - \langle c, h\rangle_{\mathbb{H}}.$$

To generate valid probabilities, I applied the logistic function to the real part only and drop the unipotent component. This small change produces a working HRBM that captures hierarchical structure through negative curvature in the weight space.

2 Experimental Setup

2.1 Dataset

Using `nltk.corpus.wordnet` I extracted every noun synset under `mammal.n.01`. Each instance is a N-dimensional binary vector where $N = 1{,}170$; an entry is 1 if that synset (or any ancestor) is active.

2.2 Model Configurations

Euclidean RBM. Implemented with `scikit-learn`'s `BernoulliRBM` (50 hidden units, learning rate 0.1, 50 epochs).

Hyperbolic RBM. Differences:

1. **Hyperbolic parameters**: each weight is $a + b\,\mathbf{j}$.
2. **Lorentzian energy**: dot product $v^\top Wh \;\to\; \langle v, Wh\rangle_{\mathbb{H}}$.
3. **Activation**: logistic on real part only.
4. **Learning**: CD-1 with Riemannian update on the real part; unipotent part clipped to $[-1, 1]$.

Hyperparameters are the same as the baseline except epochs = 3.

2.3 Evaluation Metrics

I report MAE and precision/recall/F_1 for two roots (`pug.n.01`, `dog.n.01`) and visualise reconstructions.

3 Implementation

I used Python to implement a baseline RBM using `scikit-learn`'s `BernoulliRBM`. I then created a `HyperbolicNumber` class with hyperbolic arithmetic functions and a `HyperbolicRBM` class covering weight initialization, forward/backward passes, Gibb's sampling, and a training function `fit()`.

3.1 Model structure: `HyperbolicRBM`

Listing 1: Model initialization and parameter setup

```
class HyperbolicRBM:
    def __init__(self, n_visible, n_hidden, learning_rate=0.1):
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.learning_rate = learning_rate

        # Initialize weights and biases as hyperbolic numbers
        self.W = np.array([
            [HyperbolicNumber(np.random.normal(0, 0.01), np.random.normal(0, 0.01))
            for j in range(n_visible)]
            for i in range(n_hidden)
        ])

        # Initialize hidden and visible biases
        self.h_bias = np.array([HyperbolicNumber(0.0, 0.0)
            for i in range(n_visible)])
        self.v_bias = np.array([HyperbolicNumber(0.0, 0.0)
            for j in range(n_hidden)])
```

Each parameter is a hyperbolic number, which carries a real and unipotent component. The weights are stored as a 2D matrix of these numbers. Small random values are used for both the real and imaginary parts.

Listing 2: Forward pass to compute hidden activations

```
def _compute_hidden_prob(self, v):
    h_probs = [] # list to store hidden probabilities
    for i in range(self.n_hidden):
        total = HyperbolicNumber(0.0, 0.0) # initialize hidden units
        for j in range(self.n_visible):
            # multiply hyperbolic weight with binary visible input and add to total
            total += v[j] * self.W[i][j]
        total += self.h_bias[i] # add hidden bias for unit i
        h_probs.append(hyperbolic_sigmoid(total)) # apply hyperbolic sigmoid
    return h_probs
```

Here, I compute the hidden activation probabilities using hyperbolic numbers in each step. The hyperbolic sigmoid function was defined earlier in the code. The computation of visible probabilities is analogous.

Listing 3: Gibbs Sampling

```
def gibbs_sample(self, v0):
    h_probs = self._compute_hidden_prob(v0)
    h_states = np.array([1 if prob.a > np.random.rand() else 0 for prob in h_probs])

    v_probs = self._compute_visible_prob(h_states)
    v_states = np.array([1 if p > np.random.rand() else 0 for p in v_probs])

    return v_states
```

Gibbs sampling is manually coded to perform one step starting from a visible vector v0.

Listing 4: Contrastive divergence within fit() function

```
for i in range(self.n_hidden):
    for j in range(self.n_visible):
        # positive association: hidden probability * visible unit activation
        pos = h_prob[i].a * sample[j]
        # negative association: hidden probability * reconstructed visible
        neg = h_probs_neg[i].a * v_reconstructed[j]
        # compute update using difference scaled by the learning rate
        delta = self.learning_rate * (pos - neg)
        # Update only the real part of the weight (for stability and simplicity)
        self.W[i][j].a += delta
```

This fit() function performs the learning algorithm iterating over the samples. First the hidden unit probabilities and hidden states are computed, then the visible units and states are reconstructed using Gibbs sampling. The contrastive divergence loop shown above is run. Finally, the hidden and visible biases are updated.

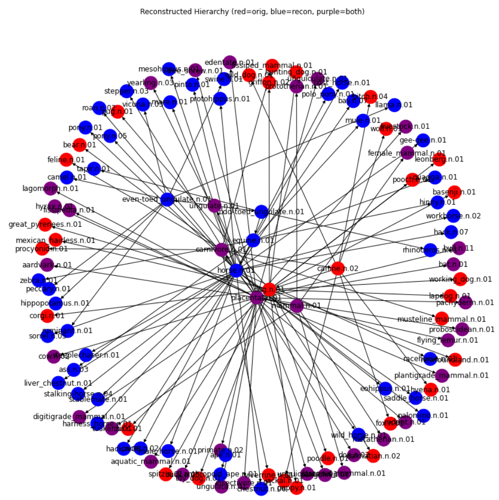
4 Results

The HRBM lowers reconstruction MAE by $78-82\,\%$ and improves F_1 by $41-53\,\%$ while training for only three epochs.

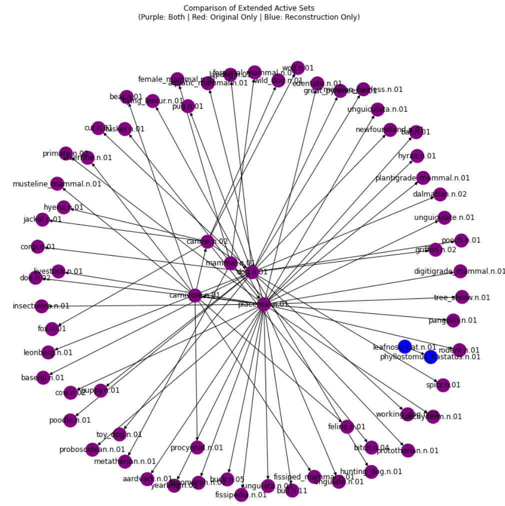
Table 1: Reconstruction on `dog.n.01`. HRBM wins despite 17× fewer epochs.

Model	Epochs	MAE	Precision	Recall	F_1
Euclidean RBM	50	0.0051	0.429	0.600	0.500
Hyperbolic RBM	3	0.0009	0.833	1.000	0.909

Figures 1–2 compare two-layer deep reconstructions.



(a) Euclidean graph



(b) HRBM graph

```
Chosen Sample Node:
pug.n.01

Original active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'mammal.n.01', 'placental.n.0
1', 'pug.n.01']

Reconstructed active synsets:
['ape.n.01', 'equine.n.01', 'even-toed_ungulate.n.01', 'horse.n.01', 'mamma
l.n.01', 'odd-toed_ungulate.n.01', 'placental.n.01']
```

(c) Euclidean synsets

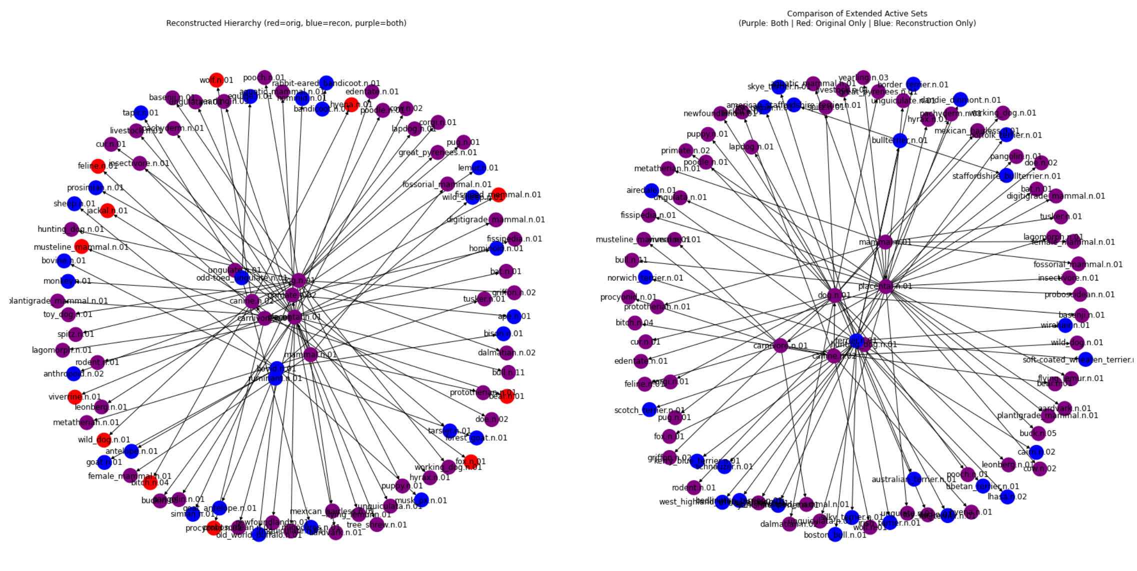
```
Chosen Sample Node:
pug.n.01

Original active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'mammal.n.01', 'placental.n.0
1', 'pug.n.01']

Reconstructed active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'mammal.n.01', 'phyllostomus_
hastatus.n.01', 'placental.n.01']
```

(d) HRBM synsets

Figure 1: Reconstruction for `pug.n.01`.



(a) Euclidean graph

(b) HRBM graph

```
Chosen Sample Node:
dog.n.01

Original active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'mammal.n.01', 'placental.n.0
1']

Reconstructed active synsets:
['bovid.n.01', 'dog.n.01', 'mammal.n.01', 'odd-toed_ungulate.n.01', 'placen
tal.n.01', 'primate.n.02', 'rabbit-eared_bandicoot.n.01']
```

(c) Euclidean synsets

```
Chosen Sample Node:
dog.n.01

Original active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'mammal.n.01', 'placental.n.0
1']

Reconstructed active synsets:
['canine.n.02', 'carnivore.n.01', 'dog.n.01', 'hunting_dog.n.01', 'mammal.
n.01', 'placental.n.01']
```

(d) HRBM synsets

Figure 2: Reconstruction for `dog.n.01`.

5 Future Work

- **GPU Acceleration**: The current HRBM implementation takes a long time to train. Rewriting the model using PyTorch would enable GPU acceleration, reduce training time, and allow scalable experimentation.
- **Curvature Control**: Hyperbolic geometry assumes constant negative curvature ($\kappa = -1$), but real-world data may benefit from learning or tuning curvature dynamically. Future models could treat curvature as a hyperparameter or optimize it jointly with weights.
- **Deeper Architectures**: This work focuses on a single-layer RBM. Extending to hyperbolic Deep Belief Networks (DBNs) could capture more complex hierarchies.
- **Benchmark Expansion**: This report evaluates only a subset of WordNet. A larger evaluation—e.g., across full WordNet, biomedical ontologies, or hierarchical image taxonomies—would provide stronger evidence of HRBM effectiveness.

6 Conclusion

This project shows that using hyperbolic-valued weights in a Restricted Boltzmann Machine can significantly improve its ability to model hierarchical data. The Hyperbolic RBM (HRBM) achieved better reconstruction and classification metrics than a standard Euclidean RBM, even with fewer training epochs. While the implementation is currently slow, the results demonstrate that hyperbolic geometry is a promising direction for energy-based models applied to structured data. All code is available at `https://github.com/kyliehefner/hyperbolic-rbm`.

References

[1] G. E. Hinton, S. Osindero, and Y. W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, 2006.

[2] M. Nickel and D. Kiela, "Poincaré Embeddings for Learning Hierarchical Representations," *Advances in Neural Information Processing Systems*, 2017.

[3] B. P. Chamberlain, J. Clough, and M. De Bie, "Deep Learning in Hyperbolic Space," *International Conference on Machine Learning*, 2019.

[4] M. Kobayashi, "Information Geometry of Hyperbolic-Valued Boltzmann Machines," *Entropy*, 2020.