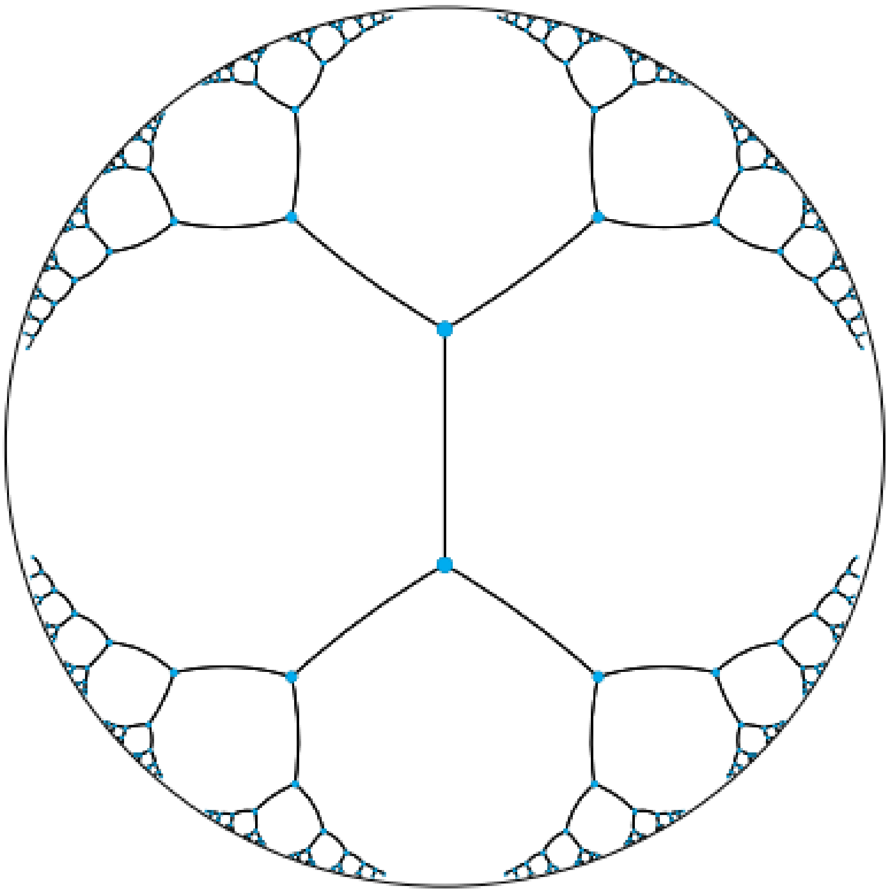
Hyperbolic embeddings

Hyperbolic space is more suitable to embed data that has a tree-like structure.

This is because the space grows exponentially with distance to the origin, which better captures the branching nature of trees. The deeper I am in the tree, the more nodes it takes me to get to another node on the same hierarchy I started. This behaviour is not easily emulated by distances in euclidean space, so very high dimensions are needed to embed the topology faithfully.



A tree in hyperbolic space. Distances between all points are equal (because the space grows with distance from the origin) In euclidean space, this would not be possible in two dimensions.

Proteins have a tree-like structure.

The distances between proteins are the result of evolution, which is inarguably tree-like, even more than the latent structure of words (which most research focuses on).
A common assumption in protein LMs is that the model learns biophysical laws from the data. Henrik once mentioned that this is just one way to think about it, the other would be that the models learn “evolvability”.
If the models really learn the evolutionary space, hyperbolic space might be more fitting.

Embedding Text in Hyperbolic Spaces

https://www.aclweb.org/anthology/W18-1708.pdf

The authors used Skip-Thougths, a sentence embedding model that works by encoding the sentence (GRU) and then predicting the preceding and the following sentence from the hidden state. This hidden state shall then be the sentence vector.

Their framework allows the use of normal, non-hyperbolic encoders. This is done by learning a constrained reparametrization function that maps the latent vector to the Poincare ball, by computing a norm magnitude p and a direction vector v. The new latent is $\theta = p * v$. This enforces the embedding to be withhin the ball, meaning the norm of the embedding needs to be smaller than 1, as the ball has unit radius.

This mean that the whole hyperbolic nature comes from engineering the training objective, as the parameters of the mapping function are learned.

Prediction & Loss in Skip-Thoughts

Standard loss function of Skip-Thoughts: Reconstruction loss on the following and preceding sentences, conditioned on the hidden state of the current sentence (word-by-word, with teacher forcing). E.g. Crossentropy, just not on its own tokens but on the ones of the next sentence.

They reparametrize the model, dropping the GRU decoder, and make the reconstruction depend on the inner product of the hidden state(averaged) and the embedding of the current token.
The rationale for this is

v_w : word embedding of w $f_\theta(s_i$: sentence embedding

$$c_t = \frac{1}{2K} \Sigma_{k=1}^{K} v'_{w_{t-k}} + v'_{w_{t+k}}$$

c_t is the average word embedding of a local context window of size K.

$$P(w_t|w_{\neq t}, f_\theta(s_i)) \propto \exp(v_{w_t}^T f_\theta(s_i) + v_{w_t}^T c_t)$$

$P(w_t|w_{\neq t}, f_\theta(s_i))$ is what i want to maximize (minimize the negative log - this is the loss).

Crossentropy Loss = LogSoftmax + Negative log likelihood loss. Means, they talk about crossentropy here.

Reparametrization for hyperbolic embeddings

Probability is expressed as inner product between the embedding of w_t and the sentence embedding plus the inner product of the embedding of w_t and the average surrounding embedding (unsure why exp). Thus, the loss is two inner products.
Because it is an inner product, it is equivalent to an expression in distances. With hyperbolic distance d, the expression becomes:

$$P(w_t|w_{\neq t}, f_\theta(s_i)) \propto \exp(-\lambda_1 d(v_{w_t}^T f_\theta(s_i)) - \lambda_2 d(v_{w_t}^T c_t))$$

Really neat trick, just make the distances inside the exp negative. We still dont get negative probabilities thanks to exp(), but probability goes up when distance goes to 0.

λ_1, λ_2 are learned coefficients to control the influence of either term. The reparametrization to make $v_{w_t}^T, f_\theta(s_i)$ and c_t meet the Poincaré ball constraint is:

$$\phi_{dir}(\mathbf{x}) = W_1^T \mathbf{x}$$
$$\phi_{norm}(\mathbf{x}) = W_2^T \mathbf{x}$$

$$\bar{\mathbf{v}} = \phi_{dir}(\mathbf{e}(s)), \mathbf{v} = \frac{\bar{\mathbf{v}}}{||\bar{\mathbf{v}}||}$$

$$\bar{p} = \phi_{norm}(\mathbf{e}(s)), p = \sigma(\bar{p})$$

$$\theta = p\mathbf{v}$$

With $\mathbf{x}$ being the euclidean version of $v_{w_t}^T, f_\theta(s_i)$ and c_t. This means, i only transition to hyperbolic space when I make the prediction. Encoding is fully euclidean. ### Application to the language modeling task

I cannot reuse exactly their prediction setup, as I do not have neighboring sequences, always just one protein at a time. This leaves me with two options: - Averaged hidden state of the full sequence as $f_\theta(s_i)$. Maybe this gives too much information away. - Just have one inner product. In the tied embedding weights case, i get my prediction from taking the inner product of the hidden state at each position with the embeddings.

When the weight of the embedding layer with the final decoder is shared, the situation in an LSTM with hidden state h is:

$$P(w_t|w_1,..,w_{t-1}) \propto \exp(v_{w_t}^T h_t)$$

Hyperbolic equivalent should be, with the same reparametrization as above

$$P(w_t|w_1,..,w_{t-1}) \propto \exp(d(v_{w_t}^T h_t))$$

Note: I did not consider the bias here.

```
class PoincareReparametrize(nn.Module):
    def __init__(self, dim):
        self.phi_dir = nn.Linear(dim,dim)
        self.phi_norm = nn.Linear(dim,dim)
    def forward(x):
        v_bar  = self.phi_dir(x)
        p_bar = self.phi_norm(x)
        v = v_bar / torch.norm(v_bar)
        p = nn.functional.sigmoid(p_bar)

        return p*v
```