Poincare Ball

次元 $\mathbb{D}^n = \{ x \in \mathbb{R}^n : \|x\| < 1 \}$

双曲線距離

$$d(u,v) = \operatorname{arcosh}\left(1 + \frac{2\|u-v\|^2}{(1-\|u\|^2)(1-\|v\|^2)} \right)$$

ボールの中心に近いほど「上位」，遠いほど「下位」の意味を持つ．

埋め込みの学習方法

⇒ "近接ペア (u,v) は近くに，その他のノード (u,v') は離す" のを目的としている!!

☆ 損失関数

$$L = \sum_{(u,v)\in D} \log \frac{e^{-d(u,v)}}{\sum_{v'\in N(u)} e^{-d(u,v')}}$$

D：観測されたペアの集合
$N(u)$：negative sampleの集合
$d(u,v)$：双曲距離

$N(u) = \{ v \mid (u,v) \notin D \} \cup \{u\}$ ← 入くわからない　$d(u,u) = 0$　$e^{-0} = 1$

1つの positive sample に対して，10コの negative sample が必要。
↳ 既に実装している。どこをinputにするかに決めた方が良さそう．

最適化方法（optimization）

↳ Riemannian SGD（RSGD）が採用されている．〜 通常のユークリッド勾配降下法（GD：Gradient Descent）では
（Stochastic Gradient Descent）　Poincaré Ball の外に出てしまうから．

☆ Haskellではは RSGD が実装されていなさそうなので自作ミ

$$\theta_{t+1} = R_{\theta t}(-\eta_t \nabla_R L(\theta_t)) \quad \longleftrightarrow \text{同じ意味(?)} \quad \theta_{t+1} = \mathrm{proj}\left(\theta_t - \eta_t \cdot \frac{(1-\|\theta_t\|^2)^2}{4} \cdot \nabla_E L(\theta_t) \right)$$

$\frac{(1-\|\theta_t\|^2)^2}{4} \cdot \nabla_E L(\theta_t) = \nabla_R L(\theta_t)$

θ_t：tステップ目の埋め込みベクトル（Embedding）
η_t：学習率（learning rate）
$\nabla_R L(\theta_t)$：リーマン勾配（Riemannian gradient）
$R_{\theta t}$：再びマップ（retraction）〜 接空間上のベクトルを，元のリーマン多様体上に戻す

$R_{\theta t}$ 接空間のステップ $(-\eta_t \nabla_R L(\theta_t))$ をそのまま θ_t に加算して proj する関数

① ユークリッド勾配 $\nabla_E L(\theta_t)$ を計算 ⟶ ☆

② リーマン勾配に変換

$$\underbrace{\nabla_R L(\theta_t)}_{\text{リーマン勾配}} = \left(\frac{(1-\|\theta_t\|^2)^2}{4} \right) \underbrace{\nabla_E L(\theta_t)}_{\text{ユークリッド勾配}}$$

RSDGは1サンプル（バッチ）ごとに更新だから $\|\theta\|$ が定義できるミ

③ $R_{\theta t}$ にわたす
→ $\theta_{t+1} = \mathrm{proj}\left(\theta_t - \eta_t \nabla_R L(\theta_t) \right)$

$$\mathrm{proj}(\theta) = \begin{cases} \dfrac{\theta}{\|\theta\| - \varepsilon} & \text{if } \|\theta\| > 1 \to \text{はみ出たときに戻す} \\ \theta & \text{otherwise} \to \text{そのまま} \end{cases}$$

$\varepsilon = 1e\text{-}5$（引数で指定できるようになっていた）

☆ $\nabla_E = \frac{\partial L(\theta)}{\partial d(\theta, x)} \frac{\partial d(\theta, x)}{\partial \theta}$ を求める。

手計算もできなくはなさそうだが、できれば $\nabla_E L(\theta_t)$ の計算はやってもらえるようにしたい....

☆'' optimizer に RSGD を追加する？できる？
runStep の実装を1から書きたくない.....

```
data GD = GD deriving (Show)

-- | Stateless gradient descent step
gd :: LearningRate -> Gradients -> [Tensor] -> [Tensor]
gd lr (Gradients gradients) parameters = zipWith step parameters gradients
  where
    step p dp = p - (lr * dp)

-- | Gradient descent step with a dummy state variable
gd' :: LearningRate -> Gradients -> [Tensor] -> GD -> ([Tensor], GD)
gd' lr gradients depParameters dummy = (gd lr gradients depParameters, dummy)

instance Optimizer GD where
  step = gd'

sgd :: LearningRate -> [Parameter] -> [Tensor] -> [Tensor]
sgd lr parameters = zipWith step depParameters
  where
    step p dp = p - (lr * dp)
    depParameters = map toDependent parameters
```

↳ あまり分かっていない!!

全データを使って損失関数の勾配を計算して
パラメータを更新. 1エポックごとに1回更新

1サンプル(またはいくつか)ずつで勾配を計算して
パラメータを更新。1エポックに何回も更新される

Torch.Optim をまねて、RSGD用に手元に作ってimportして使う？？ いくつか追加関数が必要かも
[RGD]

拡張したい内容
↳ ・∇_E から ∇_R への変換 (係数を計算してかける)
・proj 関数

☆'' その他に変更が必要な点

→ training rate について

・burn_in という引数があり、burn_in = 1.0 C のとき、learning rate = $\frac{\alpha}{C}$ になる.
・burn_in_alpha = α 0.01

これは RSGD には関わらないが
training の実装で必要になりそう.

最初のCエポックは learning_rate をさらに低くして始める.