TOWARDS EXPRESSIVE AND ROBUST LEARNING WITH HYPERBOLIC GEOMETRY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tao Yu

August 2024

TOWARDS EXPRESSIVE AND ROBUST LEARNING WITH HYPERBOLIC GEOMETRY

Tao Yu, Ph.D.

Cornell University 2024

Machine learning models traditionally operate within the confines of Euclidean space, assuming the Euclidean nature of data. However, there is a growing interest in learning within non-Euclidean hyperbolic space, particularly in scenarios where data exhibits explicit or implicit hierarchies, such as in natural languages (with taxonomies and lexical entailment) or in tree-like and graphical data (as seen in biological and social networks). Embracing the geometry of the data not only leads to more expressive models but also offers deeper insights into the underlying mechanisms governing complex datasets.

An important foundation of machine learning lies in representing data as continuous values, a process known as embedding. Recent studies have demonstrated both theoretically and empirically that hyperbolic space can embed hierarchical data with lower dimensionality compared to Euclidean space. This insight has spurred the development of various hyperbolic networks, despite the challenge that hyperbolic space is not a vector space. To address this, we propose an end-to-end approach that adopts hyperbolic geometry from a manifold perspective. This approach includes an embedding framework that directly encodes data hierarchies, a method for hyperbolic-isometries-aware learning, and a demonstration of how our framework can enhance the performance of attention models, such as transformers, by capturing implicit hierarchies.

While hyperbolic geometry offers theoretical advantages, its practical im-

plementation faces challenges due to numerical errors stemming from floating-point computations, further exacerbated by the ill-conditioned hyperbolic metrics. This issue, often referred to as the "NaN" problem, arises when practitioners encounter Not-a-Number while running hyperbolic models. To address this, we introduce several robust and accurate representations using integer-based tilings and multi-component floating-point methods, which offer provably bounded numerical errors for the first time. Additionally, we present MCTensor, a PyTorch library that enables general-purpose and high-precision training of machine learning models. We demonstrate the effectiveness of our approach by applying multi-component floating-point to train large language models at low precision, mitigating the issue of reduced numerical accuracy and producing models of better performances.

In conclusion, our work aims to empower individuals and organizations to leverage the potential of hyperbolic geometry in machine learning, drawing a broad audience towards this promising and evolving research direction.

BIOGRAPHICAL SKETCH

Tao Yu was born in Lu'an, China and raised in Shaoxing, China for most of his adolescence, where he attended primary school, middle school and high school. In 2019, he obtained his Bachelor Degree in Mathematics and Applied Mathematics (Honors) from Zhiyuan college, Shanghai Jiao Tong University. He then began pursuing a Ph.D. degree in Computer Science at Cornell University, where he was advised by Professor Christopher De Sa. During his Ph.D., Tao was broadly interested in integrating data geometry into machine learning for expressive and efficient model developments. His research projects also extend to machine learning privacy and robustness, and vector symbolic architectures. During his doctoral curriculum, he completed internships at Apple Privacy team and Amazon Web Services AI Labs in California.

To my parents, whose support and belief in me have been unwavering, even during my most mischievous years.

To my girlfriend Jingyi, whose presence in my life is a gift I cherish every day.

ACKNOWLEDGEMENTS

Throughout my time at Cornell, I have had the privilege of collaborating with an outstanding group of researchers. I am deeply grateful to my advisor, Prof. Christopher De Sa, for his guidance, support, and encouragement throughout my doctoral studies. His intelligence, expertise, and insights have not only helped shape my research but also propel my academic growth, demonstrating how mathematical knowledge can be applied to solve practical problems. I would also like to thank Prof. Kilian Weinberger and Noah Stephens-Davidowitz for their valuable feedback and contributions as members of my dissertation committee. I am truly thankful for the internship experience at Prof. Kilian Weinberger's lab when I was an undergraduate, which exposed me to a wealth of research opportunities. I am indebted to Prof. John Hopcroft for his invaluable guidance into the field of machine learning and for encouraging me to pursue my Ph.D. overseas. Additionally, I am grateful to Prof. Vitaly Shmatikov for his mentorship in machine learning privacy research during the early years of my Ph.D.

My research achievements would not have been possible without the exceptional group of students and researchers at Cornell. I would like to extend my gratitude to Shengyuan Hu, Yichi Zhang, Toni Liu, Eugene Bagdasaryan, Chuan Guo, Tianyi Zhang, Wei-Lun Chao, and Prof. Zhiru Zhang for their collaboration and support. Being part of the Relax ML Lab led by Prof. Chris De Sa has been a privilege, and I am thankful for the opportunity to collaborate with other lab members, including Albert Tseng, Jianan Canal Li, Wentao Guo, and Tiancheng Yuan.

I am also thankful for the enriching experiences during my internships at Apple and AWS AI. I extend my gratitude to my mentors at Apple, Congzheng

Song, Mona Chitnis, Vojta Jina, Martin Pelikan, and Ulfar Erlingsson, as well as my mentors at AWS AI, Gaurav Gupta, Luke Huan, and Youngsuk Park, for their guidance and insightful discussions on various research challenges.

I am deeply appreciative of the support and contributions of these individuals, as well as anyone else who has contributed to my academic and personal development. Lastly, I am grateful to my family for their unwavering love and encouragement, and to my friends, Nathan Yan, Renjiu Hu, Junxiong Wang, Ke Wu and many others for their support and understanding throughout this challenging journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

Modern machine learning methods favor continuous data representations, as such representations can be easily used in differentiable deep models. Yet real-world data is often discrete; for example, natural language sentences are composed of characters and words, and graphs consist of nodes and edges [22, 104]. This discrete-continuous gap has resulted in a wide variety of embedding methods that map discrete data into continuous spaces. Popular methods such as word2vec [116] and GloVe [135] approach this problem by mapping into high-dimensional Euclidean spaces, which capture complex relations between data by using many dimensions.

However, not all data can be well represented in Euclidean Space [23, 147]. Hierarchical and graphical data, such as biological phylogenetic trees and social networks, are more naturally modeled with hyperbolic geometry [150, 147]. Intuitively, Euclidean geometry studies flat spaces with zero curvature (i.e., Euclidean space), while hyperbolic geometry focuses on spaces with negative curvature. It has been shown theoretically and empirically in [21, 150, 32, 23] that hyperbolic space is better- and naturally-suited for representing such data and capturing implicit hierarchies, outperforming Euclidean baselines.

Moreover, many well-established modern machine learning models are designed within the framework of Euclidean space. This underscores the necessity of exploring and developing machine learning models in hyperbolic space

to achieve more expressive and efficient learning from real data. However, this exploration comes with its own set of challenges. For example, since hyperbolic space is not a vector space, determining the principal approach for utilizing hyperbolic geometry to develop network operations (e.g., matrix-vector and matrix computations) is non-trivial. Additionally, hyperbolic operations can be computationally expensive and numerically unstable, posing challenges for the robust training of hyperbolic networks and models.

1.2 Hyperbolic Geometry Preliminaries

We detail some necessary preliminaries on hyperbolic geometry in this section. **Hyperbolic space** $\mathbb{H}_n$ is the unique n-dimensional simply connected Riemannian manifold [136] with constant negative curvature $-k, k > 0$ [7]. Typically, people work with hyperbolic space using *hyperbolic models*, which are mathematical representations within ambient Euclidean space in a way that can be visualized or understood more easily.

There exists multiple important models of hyperbolic space, most notably the Poincaré ball model, the Lorentz hyperboloid model, and the Poincaré upper-half space model [7]. These all model the same geometry in the sense that any two of them can be related by an isometry, a transformation that preserves geometrical properties of the space, including distances and gradient [25]. Generally, one can choose whichever model is best suited for a given task.

Poincaré ball model. The Poincaré ball model is the Riemannian manifold $(\mathcal{B}^n, g_p)$, where $\mathcal{B}^n = \{\boldsymbol{x} \in \mathbb{R}^n : \|\boldsymbol{x}\| < 1\}$ is the open unit ball. The metric and

distance on $\mathcal{B}^n$ are defined as

$$g_{\mathcal{B}}(\boldsymbol{x}) = \left(\frac{2}{1-\|\boldsymbol{x}\|^2}\right)^2 g_e, \quad d_{\mathcal{B}}(\boldsymbol{x},\boldsymbol{y}) = \operatorname{arcosh}\left(1 + 2\frac{\|\boldsymbol{x}-\boldsymbol{y}\|^2}{(1-\|\boldsymbol{x}\|^2)(1-\|\boldsymbol{y}\|^2)}\right),$$

where g_e is the standard Euclidean metric. The Poincaré ball model is also referred as the Poincaré disk model when $n = 2$. Due to its conformality (angles measured at a point in Euclidan space are the same as that in the actual hyperbolic space), convenient parameterization, and clear visualization results, the Poincaré ball model is widely used in many scenario. However, point distance within the Poincaré ball model changes rapidly when the points are close to the boundary (i.e. $\|x\| \approx 1$), and hence it becomes very poorly conditioned near the boundary.

Lorentz hyperboloid model. The Lorentz model is arguably the most natural model algebraically for hyperbolic space. It is defined in terms of a nonstandard scalar product called the *Lorentzian scalar product* $\langle\cdot,\cdot\rangle_L$, for example in $\mathbb{R}^3$

$$\langle \boldsymbol{x},\boldsymbol{y}\rangle_L = \boldsymbol{x}^T g_l \boldsymbol{y}, \quad \forall \boldsymbol{x},\boldsymbol{y} \in \mathbb{R}^3 \quad \text{where} \quad g_l = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The Lorentz model of 2-dimensional hyperbolic space is the upper-sheet of a hyperbola embedded in $\mathbb{R}^3$, formally defined as the Riemannian manifold $(\mathcal{L}^2, g_l)$, where $\mathcal{L}^2$ and associated distance function are given as

$$\mathcal{L}^2 = \{\boldsymbol{x} \in \mathbb{R}^3 : \langle \boldsymbol{x},\boldsymbol{x}\rangle_L = -1, x_0 > 0\}, \quad d_{\mathcal{L}}(\boldsymbol{x},\boldsymbol{y}) = \operatorname{arcosh}(-\langle \boldsymbol{x},\boldsymbol{y}\rangle_L).$$

This model generalizes easily to higher dimensional spaces by increasing the number of 1s on the diagonal of the matrix g_l. Points in the Lorentz model lie on the upper sheet of a two-sheeted n-dimensional hyperbola. Unlike the

Poincaré ball model, which is confined in the Euclidean ball, the Lorentz model is unbounded.

Poincaré upper-half space model. The Poincaré upper-half space model parameterizes the hyperbolic space using the upper-half of Euclidean space $\mathbb{R}^n$. It's defined as the Riemannian manifold $(\mathcal{U}^n, g_u)$ with $\mathcal{U}^n = \{\boldsymbol{x} \in \mathbb{R}^n : x_n > 0\}$. The metric and corresponding distance function are

$$g_u(\boldsymbol{x}) = \frac{g_e}{x_n^2}, \quad d_u(\boldsymbol{x}, \boldsymbol{y}) = \operatorname{arcosh}\left(1 + \frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2}{2x_n y_n}\right)$$

where g_e is the Euclidean metric. In the two dimensional case, the Poincaré upper-half space model can also be interpreted as a mapping on the complex plane (the "half-plane model").

1.2.1 Key Concepts in Hyperbolic Geometry

With the hyperbolic models defined, we are ready to introduce some key concepts for usages in later sections, which are common terms in Riemannian geometry, readers may refer [136, 7] for more details.

The **boundary** of $\mathbb{H}_n$ is the set of points which are infinitely far away from the origin. In the Poincaré ball model, the boundary $\partial\mathcal{B}^n$ is the sphere of radius 1 at the "edge" of the ball. In the Poincaré upper-half space model, the boundary $\partial\mathcal{U}^n$ consists of the 0-hyperplane ($x_n = 0$) and points at infinity. In the Lorentz hyperboloid model, the boundary $\partial\mathcal{L}^n$ is simply points at infinity.

An important concept in Riemannian geometry is the **tangent space**: it is the first order vector space approximation of a Riemannian manifold $\mathcal{M}$ locally

around a point $\boldsymbol{x}$, denoted as $T_{\boldsymbol{x}}\mathcal{M}$. $T_{\boldsymbol{x}}\mathcal{M}$ can also be interpreted as the set of tangent vectors of all smooth paths $\in \mathcal{M}$ through point $\boldsymbol{x}$.

The notion of locally flat tangent space allows us to define local distance metrics, $g_{\mathcal{M}}(\boldsymbol{x})$, everywhere on a Riemannian manifold. $g_{\mathcal{M}}(\boldsymbol{x})$ is termed **Riemannian metric** (as given in hyperbolic models), and it is a positive-definite quadratic form that assigns an "infinitesimal distance" to each tangent vector at a point on the manifold.

With the Riemannian metric $g_{\mathcal{M}}(\boldsymbol{x})$ defined, we are able to argue the Riemannian generalization of Euclidean straight lines, the **geodesics**. They are defined as smooth curves of locally minimal length connecting two points $\boldsymbol{x}$ and $\boldsymbol{y}$ on a Riemannian manifold $\mathcal{M}$. The length of a global geodesic can be understood as the local Riemannian metric $g_{\mathcal{M}}(\boldsymbol{x})$ integrated along the geodesic.

The tangent space $T_{\boldsymbol{x}}\mathcal{M}$ is connected to the Riemannian manifold $\mathcal{M}$ via the **exponential map** and the **logarithm map**. Specifically, the exponential map $\exp_{\boldsymbol{x}}(\boldsymbol{v})$, maps a vector $\boldsymbol{v} \in T_{\boldsymbol{x}}\mathcal{M}$ to another point in $\mathcal{M}$ by traveling along the geodesic from $\boldsymbol{x}$ in the direction of $\boldsymbol{v}$. The logarithm map $\log_{\boldsymbol{x}}(\boldsymbol{y})$ does the inverse thing: given $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{M}$, it finds the vector $\boldsymbol{v} \in T_{\boldsymbol{x}}\mathcal{M}$ such that $\exp_{\boldsymbol{x}}(\boldsymbol{v}) = \boldsymbol{y}$.

Many geometry objects and concepts in Euclidean space can be extended to their counterparts in Riemannian geometry or hyperbolic space in our case, such as spheres, balls, hyperplanes, equidistance curves and even laws like the law of sines. For example, a **hypercycle** is an equidistant curve that is on one side and at a given distance from a geodesic l, its axis. In the Poincaré ball, hypercycles of l are Euclidean circular arcs or chord of the boundary circle that intersect the boundary at l's ideal points at non-right angles.

In the Poincaré half space, if l is a Euclidean semicircle, then the hypercycles of l are again Euclidean circular arcs intersecting the boundary at l's ideal points at non-right angles. If l is a vertical ray, then hypercycles are Euclidean rays that intersect l's ideal point at a non-right angle.

An important rule we will use in Section 2.2 is the **hyperbolic law of sines**, like its Euclidean counterpart, it relates the angles of any hyperbolic triangle to its geodesic side lengths:

$$\frac{\sin A}{\sinh a} = \frac{\sin B}{\sinh b} = \frac{\sin C}{\sinh c} \tag{1.1}$$

where A, B, and C are the angles of a hyperbolic triangle, and a, b, and c are the geodesic edges of the sides opposite to these angles.

CHAPTER 2

EXPRESSIVE HIERARCHICAL DATA MODELING WITH HYPERBOLIC GEOMETRY

2.1 Background

Many kinds of real-world data are hierarchical or graph-like, such as the IBM Information Management System to describe the structure of documents, the large lexical database WordNet [57], various networks [37, 151, 79, 143] and natural language sentences [117, 118]. It is challenging but necessary to embed these structured data for the use of modern machine learning methods. Well-designed models and representations that take advantage of this data geometry could potentially ease the task and make learning more efficient (with less parameters).

Recent work [43, 122, 123, 26] proposed using *non-Euclidean hyperbolic spaces* to embed these structures and has achieved exciting results. Intuitively, Euclidean geometry studies flat spaces with zero curvature, while non-Euclidean geometry studies spaces with non-zero curvature. Hyperbolic space is a manifold with constant negative curvature and endowed with various geometric properties. In particular, hyperbolic space is characterized by a larger exponentially growing volume [198] compared to the polynomial growth observed in Euclidean space, thus it has more space for representations with a lower dimensionality. This exponential growth property resembles real data such as tree-, graph-style data and natural languages with latent hierarchies, where the interested hierarchies can grow exponentially. Indeed, [21] shows that any finite subset of an hyperbolic space looks like a finite tree according to the definition

in [65].

It has been shown both theoretically and empirically [21, 150, 32, 23] that hyperbolic space is naturally well suited for representing such data and capturing implicit hierarchies, outperforming Euclidean baselines. For example, [147] shows that hyperbolic space can embed trees without loss of information (arbitrarily low distortion), which cannot be achieved by Euclidean space of any dimension [31, 142]. [122, 123] proposed low-dimensional hyperbolic embeddings to capture (implicit) hierarchies on many tasks such as taxonomies, link prediction and inferring concept hierarchies, which achieves superior performances with light models that are unattainable in Euclidean space. Further more, there is an active line of research on developing ML models in hyperbolic space. Starting from hyperbolic neural networks (HNN) by [60], a variety of hyperbolic networks were proposed for different applications, including HNN++ [153], hyperbolic variational auto-encoders (HVAE, [111]), hyperbolic attention networks (HATN, [67]), hyperbolic graph convolutional networks (HGCN, [28]), hyperbolic graph neural networks (HGNN, [106]), and hyperbolic graph attention networks (HGAT [210]). The strong empirical results of HGCN and HGNN in particular on node classification, link prediction, and molecular-and-chemical-property prediction show the power of hyperbolic geometry for graph learning.

2.2 Partial Orders Embedding in Hyperbolic Shadow Cones

Overview

In this work, we focus on learning embeddings that capture partial orders on a set of data points X. In a partial order, certain pairs of points $u, v \in X$ possess entailment relations. That is, if $u \prec v$, u is an ancestor of v. Many hierarchical structures such as directed acyclic graphs (DAGs) can be expressed as partial orders, making partial orders a popular tool to represent such structures with [174, 60]. Furthermore, since not all pairs $u, v \in X$ need to be comparable (hence the "partial" nomer), partial orders are especially useful for graph prediction tasks, where multiple embeddings with different properties can exist for a single partial order over a set.

We propose a novel and physically intuitive partial order embedding framework, which we call the "shadow cones" framework. Shadow cones use a set of hyperbolic cones derived from shadows formed by light sources and opaque objects in hyperbolic space. Entailment relations between objects are modeled by subset relations between shadows, similar to how planets block each other out during solar eclipses. Shadow cones can be seen as a generalization of existing approaches that use hyperbolic cones to model partial orders ("entailment cones"), and are agnostic to choice of hyperbolic model. This results in better numerics and optimization properties, which allow shadow cones to empirically outperform existing entailment cone constructions on a wide variety of graph embedding tasks.

In the following sections, we present formal constructions of shadow cones

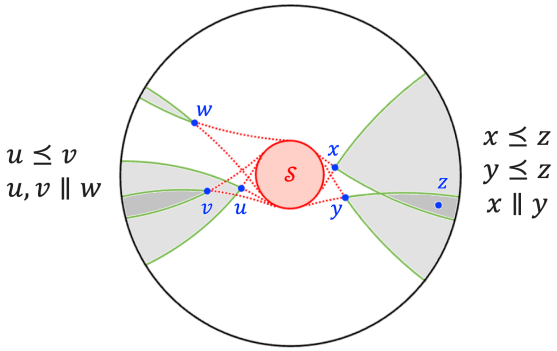


Figure 2.1: Example of two sets of shadow cone embeddings in the Poincaré ball, and the partial relations it encodes. Marked in black are the encoded partial relations, while in blue are the embeddings, $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ and $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$. Marked in red are the light source ($\boldsymbol{\mathcal{S}}$), and the dotted geodesics representing light rays. Shaded areas represent shadows. The symbol "||" denotes negative relations between unrelated, incomparable elements.

in the Poincaré ball and half-plane. Our main contributions are that we:

- Introduce the *shadow cones* framework to model partial order relations, and detail two self-contained formulations in two hyperbolic models, resulting in four embedding schemes.
- Define a differentiable energy function to measure disagreement between ground truth partial orders in data and those induced by shadow cones.
- Achieve state-of-the-art results on a wide range of graph embedding tasks, lending both empirical and theoretical support to the advantages of all four embedding schemes.

Related Work

We review several approaches to embedding partial orders and other hierarchical relations in Euclidean and hyperbolic space. Order embeddings were first

introduced in [174] to model partial orders in Euclidean space, by defining entailment relations as subset relations between axis-parallel cones at embedded points; that is, $\boldsymbol{u} \preceq \boldsymbol{v}$ iff the cone of $\boldsymbol{v} \subseteq$ the cone of $\boldsymbol{u}$. However, since all axis-parallel cones eventually intersect, order embeddings are incapable of model nonoverlapping categories, such as "canine" and "feline" in a taxonomy. Since volumes in Euclidean space only increase polynomially, sets of infinite cones are prone to heavy intersections. This results in provably limited space for disjoint regions that are necessary to model negative relations ($\boldsymbol{u} \parallel \boldsymbol{v}$). More recently, [207, 19] proposed box embeddings in Euclidean space using subset relations between axis-parallel boxes, to represent entailment relations. Yet these methods are subject to the drawbacks of Euclidean space, limiting their expressivity.

The "crowdedness" of Euclidean space places fundamental limits on its representation power for deep and wide hierarchical structures [147]. On the other hand, volumes in hyperbolic space grow exponentially, giving clear advantages for embedding hierarchies. Many works have explored the utility of hyperbolic embeddings for hierarchical data through likelihood scoring functions [122, 123, 198, 27]. For example, [122] used the following heuristic composed of norms and distances to rate the likelihood of **Is-A** $\prec$ relationships:

$$\text{score}(\textbf{Is-A}(\boldsymbol{u}, \boldsymbol{v})) = -(1 + \alpha(\|\boldsymbol{v}\| - \|\boldsymbol{u}\|))d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v}).$$

However, such approaches are limited, as they do not explicitly model partial orders and cannot easily recover learned hierarchies.

[60] introduced the concept of entailment cones, which models entailment relations by subset relations between cones rooted at points. Formally, an entailment cone at $x \in X$, $\mathfrak{S}_{\boldsymbol{x}}$, is defined by mapping a convex cone S from the tangent space $T_{\boldsymbol{x}}\mathcal{M}$ to $\mathbb{H}$ using the exponential map $\mathfrak{S}_{\boldsymbol{x}} = \exp_x(S), S \subset T_x\mathcal{M} = T_x\mathbb{H}$. Differ-

ent choices of S give different sets of entailment cones for X. While entailment cones offer strong empirical performance, they also suffer from initialization issues that hinder optimization. For example, the specific construction presented in [60] leaves an ϵ-hole at the origin of the Poincaré ball where cones are undefined, and care must be taken to ensure that embeddings do not land in the ϵ-hole. To mitigate this issue, [60] initialize embeddings with pretrained embeddings from [122]. However, this approach constrains the representational capacity of entailment cones and complicates performance analysis, as it deviates from being a self-contained framework.

Ganea's entailment cones are a special case of our "penumbral shadow cones" (section 2.2.1) in the Poincaré ball, and our framework provides an intuitive explanation of why this ϵ-hole exists.[1] Additionally, our shadow cones framework allows for constructions that do not suffer from the ϵ-hole problem: we do this by instead defining penumbral cones in the Poincaré half-space and mapping the light source to infinity. Our experiments show that penumbral cones in the Poincaré half-space consistently outperform [60]'s strong baselines.

2.2.1 Formulations of Shadow Cones

The shadow cones framework defines entailment relations using shadows cast by a light source $\mathcal{S}$ from opaque objects representing embedded data points. The general idea is to geometrically represent the partial relation using subset relation between shadows: we say $\boldsymbol{u} \preceq \boldsymbol{v}$ when the "shadow cast" by $\boldsymbol{v}$ is a subset of the "shadow cast" by $\boldsymbol{u}$. Note that this embedding scheme automati-

[1] Specifically, it corresponds to the interior of the "light source" inside which shadows are not cast (Remark 1).

cally satisfies the transitivity of partial relations since subset relations are inherently transitive. However, explicitly characterizing the subset relations between shadowed regions is complicated. In this section, we introduce shadow cones which reduces region-region subset relations to point-region membership relations. Specifically, $\boldsymbol{v}$ is in the *shadow cone* of $\boldsymbol{u}$ iff the shadow of $\boldsymbol{v} \subseteq$ the shadow of $\boldsymbol{u}$. (Figure 2.1).

We categorize shadow cones into two types of cones, depending on the type of shadow used: umbral and penumbral. In umbral cones, the light source $\mathcal{S}$ is a point and data points are represented by objects with volume. Conversely, in penumbral cones, $\mathcal{S}$ has volume while data points are volumeless points. The exact shape of shadow cones depends on the shape and position of $\mathcal{S}$ and the objects associated with data points. In the following sections, we adopt $\mathcal{S}$ with shapes and positions that result in axially symmetric shadow cones. Given that hyperbolic space is endowed with continuous isometries capable of transforming non-axially symmetric placement of light sources into axially symmetric ones, our focus on symmetric cases greatly simplifies computations while at the same time do not lose any generality. We defer further discussions on isometries to Appendix A.1.2.

Umbral Cones

We provide constructions for umbral cones in the Poincaré ball and half-space models. In both constructions, embedded points are centers of associated hyperbolic balls of radius r. In both models, these hyperbolic balls correspond to Euclidean balls with shifted centers.

Definition 2.2.1. *Given a point light source $\mathcal{S}$ and points $x, y \in \mathbb{H}$, we say that y is*

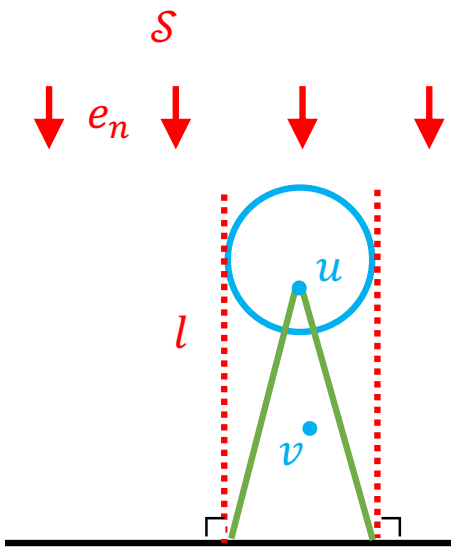


Figure 2.2: Umbral-half-space embeddings of partial relation $u \preceq v$. Marked in red are light source at infinity ($\mathcal{S}$), directions of light (e_n), and geodesic shadow boundaries (l). Blue is the object, and green the shadow cones.[2]

in the radius-r umbral cone *at x if for every point u in the (hyperbolic) ball of radius r centered at y, every geodesic between u and $\mathcal{S}$ passes through the (hyperbolic) ball of radius r centered at x.*

This definition formalizes the notion that the ball centered at y is entirely in the shadow of the ball centered at x. We provide several equivalent definitions of shadow cones in Appendix A.1.2 with a detailed analysis. Each resulting umbral cone is a subset of the umbral shadow, enclosed by the so-called equidistant curves (i.e., hypercycles) to the boundaries of the umbral shadow. We detail umbral cones and the boundaries in the Poincaré half-space and ball model separately below.

Formulation 1: Umbral-half-space We illustrate the shape of umbral cone when $\mathcal{S}$ is placed at $x_n = \infty$ in the Poincaré (upper) half-space model. Light

[2]Same notations and symbols apply to Figures 2.3-2.5.

travels along vertical ray geodesics in the direction $\boldsymbol{e}_n = (0, \ldots, 0, -1)$. The central axis of the cone induced by a point $\boldsymbol{u}$ is then $A^{\mathcal{U}}_{\boldsymbol{u}} = \{(u_1, \ldots, u_{n-1}, x_n) | 0 < x_n \leq u_n\}$. In the Poincaré half-space model, the hyperbolic ball of $\boldsymbol{u}$ is a Euclidean ball with center $\boldsymbol{c}_{\boldsymbol{u}} = (u_1, \ldots, u_{n-1}, u_n \cosh r)$ and radius $r_e = u_n \sinh r$. Thus, the shadow of $\boldsymbol{u}$'s ball is the region directly beneath its object, or equivalently, the region within Euclidean distance r_e from its central axis. Note that the umbral cone is a subset of this shadow, as *the entire ball* of $\boldsymbol{v}$ needs to be in this shadow for $\boldsymbol{v} \geq \boldsymbol{u}$.

This umbral cone is better characterized by studying its boundary. Note the set of light paths tangent to the boundary of $\boldsymbol{u}$'s ball is $\{(x_1, \ldots, x_{n-1}, t) | \sum_{i=1}^{n-1} (x_i - u_i)^2 = r_e^2, t > 0\}$. Let l be such a light path, then one boundary of the umbral cone is the curve equidistant to l, and passing through $\boldsymbol{u}$ (i.e., a hypercycle with axis l). Since l is a ray-geodesic, its equidistant curve is the Euclidean straight line through $\boldsymbol{u}$ and l's ideal point on the $x_n = 0$–hyperplane. Thus, the umbral cone of $\boldsymbol{u}$ is also a Euclidean cone with $\boldsymbol{u}$ as its apex and base on the $x_n = 0$–hyperplane with Euclidean radius r_e. The Poincaré half-space is conformal, so the half aperture of this cone is $\theta_{\boldsymbol{u}} = \arctan(r_e / u_n) = \arctan \sinh(r)$, which is fixed across different $\boldsymbol{u}$. This allows us to test if $\boldsymbol{u} \preceq \boldsymbol{v}$ by simply comparing the angle between $\boldsymbol{v} - \boldsymbol{u}$ and $\boldsymbol{e}_n$.

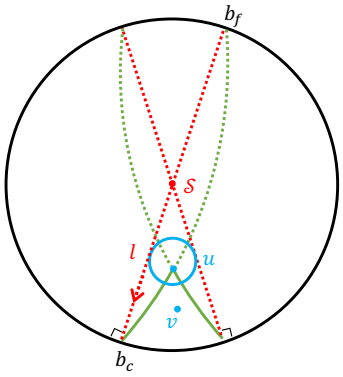


Figure 2.3: Umbral-Poincaré-ball embeddings of relation $u \preceq v$.

Formulation 2: Umbral-Poincaré-ball We now discuss the umbral cone when $\mathcal{S}$ is placed at the origin of the Poincaré ball model [3], which emits light along the

[3] Again, fixing the light source at the origin causes no loss of generality using isometries, for reasons discussed in the appendix.

radii. The umbral cone is symmetric with central axis $A^{\mathcal{B}}_{\boldsymbol{u}} = \{p\boldsymbol{u} | 1 \leq p < 1/\|\boldsymbol{u}\|\}$.

Again, we characterize the umbral cone by looking at its boundary. Let l be a light path tangent to the boundary of $\boldsymbol{u}$'s associated ball. Assume l intersects with the boundary at two ideal points $\boldsymbol{b}_f, \boldsymbol{b}_c$, where $\boldsymbol{b}_c$ is closer to $\boldsymbol{u}$ than $\boldsymbol{b}_f$. The corresponding boundary of the umbral cone is the curve equidistant to l, and passing through $\boldsymbol{u}$ (i.e., hypercycle passing through $\boldsymbol{u}$ with axis l). This is an arc passing through three points: $\boldsymbol{b}_f, \boldsymbol{b}_c, \boldsymbol{u}$. Thus, the related boundary of the umbral cone is the arc segment $\overline{\boldsymbol{u}\boldsymbol{b}_c}$ (see Figure 2.3).

Penumbral Cones

For penumbral cones, $\mathcal{S}$ is a convex region, while embedded points are points of no volume. We define the penumbral shadow of $\boldsymbol{u}$ as the region delineated by geodesic rays tangent to the light source's boundary $\partial\mathcal{S}$ and passing through $\boldsymbol{u}$. Consequently, $\boldsymbol{u} \preceq \boldsymbol{v}$ iff $\boldsymbol{v}$ is in the shadow of $\boldsymbol{u}$.

Definition 2.2.2. *Given a convex light source $\mathcal{S}$ and points $x, y \in \mathbb{H}$, we say that y is in the* penumbral cone *at x if the geodesic ray from y through x passes through $\mathcal{S}$.*

This definition formalizes the idea that x occludes some part of the light source as seen from y.

Formulation 3: Penumbral-Poincaré-ball We adopt a hyperbolic ball of radius r to be the light source. We parameterize the cone in the Poincaré ball model, and without loss of generality (up to an isometry), we place the center of $\mathcal{S}$ at the origin. The penumbral cone in this case is symmetric with central axis $A^{\mathcal{B}}_{\boldsymbol{u}}$.

Let l be an arc geodesic tangent to the light source's boundary $\partial \mathcal{S}$ at $\boldsymbol{w}$ and passes through $\boldsymbol{u}$. Then the penumbral cone induced by $\boldsymbol{u}$ is axially-symmetric, with one boundary being the segment of l starting from $\boldsymbol{u}$. The half aperture of the penumbral cone with apex $\boldsymbol{u}$ can be derived by applying the hyperbolic laws of sines (Equation 1.1 in Section 1.2.1) to the hyperbolic triangle $\triangle \boldsymbol{\mathcal{S}wu}$, with $\angle \boldsymbol{\mathcal{S}wu} = \pi/2$. The half aperture is then:

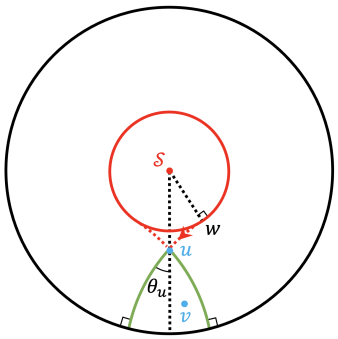


Figure 2.4: Penumbral-Poincaré-ball embeddings of relation $u \preceq v$.

$$\theta_{\boldsymbol{u}} = \arcsin\left(\frac{\sinh r}{\sinh d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{\mathcal{S}})}\right)$$

To test the partial order of any $\boldsymbol{u}, \boldsymbol{v}$, we need to compute the angle $\phi(\boldsymbol{v}, \boldsymbol{u})$ between the cone central axis and the geodesic connecting $\boldsymbol{u}, \boldsymbol{v}$ at $\boldsymbol{u}$, i.e., $\pi - \angle \boldsymbol{Ouv}$, which is given by [60] as

$$\phi(\boldsymbol{v}, \boldsymbol{u}) = \arccos\left(\frac{\langle \boldsymbol{u}, \boldsymbol{v} \rangle (1 + \|\boldsymbol{u}\|^2) - \|\boldsymbol{u}\|^2 (1 + \|\boldsymbol{v}\|^2)}{\|\boldsymbol{u}\| \, \|\boldsymbol{u} - \boldsymbol{v}\| \, \sqrt{1 + \|\boldsymbol{u}\|^2 \, \|\boldsymbol{v}\|^2 - 2\langle \boldsymbol{u}, \boldsymbol{v} \rangle}}\right),$$

One can test whether $\boldsymbol{u} \preceq \boldsymbol{v}$ simply by comparing $\phi(\boldsymbol{v}, \boldsymbol{u})$ with the half aperture $\theta_{\boldsymbol{u}}$. Note that larger radius r leads to wider aperture θ_u, which implies larger numbers of children. In Appendix A.1.3, we make r a learnable parameter, and observe a curious relation between the optimal radius and connectivity of the embedded dataset.

Formulation 4: Penumbral-half-space The penumbral shadow cone framework is very general and do not restrict the shape of the light source. Here we introduce one more formulation with the light source shaped as **horospheres** [83], which are hyperbolic analogs of hyper-planes in Euclidean spaces. In the

Poincaré half-space model, a horosphere is either a sphere tangent to the $x_n = 0$–hyperplane at an ideal point, or a Euclidean hyper-plane parallel to the $x_n = 0$–hyperplane. We use the latter as it induces symmetric shadows. In particular, we consider horospheres $\{(x_1, \ldots, x_{n-1}, e^h) | h > 0\}$, whose boundaries $\partial \mathcal{S}$ are parallel to and with a distance h from the $x_n = 0$–hyperplane.

Consider a half-circle geodesic l tangent to the horosphere light source, that passes through $\boldsymbol{u}$ and ends up at infinity. Then the boundary of the penumbral cone induced by $\boldsymbol{u}$ is the segment of l between $\boldsymbol{u}$ and infinity, that is, the $x_n = 0$–hyperplane. This is illustrated in Figure 2.5. The central axis of the cone is $A^{\boldsymbol{u}}_{\boldsymbol{u}}$. With some basic geometry, we derive the half aperture as $\theta_{\boldsymbol{u}} = \arcsin\left(u_n / e^h\right)$. Similarly, we compute the angle $\phi(\boldsymbol{v}, \boldsymbol{u})$ between the cone central axis and the geodesic connecting $\boldsymbol{u}, \boldsymbol{v}$ at $\boldsymbol{u}$, i.e., the angle between $\log_{\boldsymbol{u}}(\boldsymbol{v})$ and $\boldsymbol{e}_n$, where log is the logarithm map in the Poincaré half-space model [200], the formula of which is also provided in Appendix A.1.2.

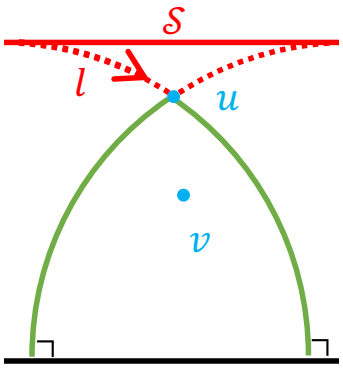


Figure 2.5: Penumbral-half-space embeddings of relation $u \preceq v$.

For completion, we also discuss the third and final category of celestial shadows - the antumbral shadows, apart from umbral and penumbral shadows. Antumbral shadows are formed when both the light source $\mathcal{S}$ and objects have volumes. The cones induced by antumbral shadows are in fact mathematically equivalent to those induced by penumbral shadows.

Antumbral Cones Antumbral shadows occur under two necessary conditions: 1. The radius r of the object must be smaller than the radius R of the

light source. 2. At least a portion of the object must be located outside the light source. In Figure 2.6, we illustrate antumbral cone in the half-space setting, where shadows are generally not axially symmetric.

Let l be geodesics tangent to the surface of the light source $\partial \mathcal{S}$ and the object $\partial \boldsymbol{u}$, such that $\boldsymbol{u}$ is between $\partial \mathcal{S}$ and the intersection $\boldsymbol{u}'$ of light paths. The antumbral shadow of $\boldsymbol{u}$ is then defined as the penumbral shadow of $\boldsymbol{u}'$. Note that by construction, for any object u with well-defined antumbal cone, it is always possible to find a surrogate point u', whose penumbral shadow is identical to the antumbral shadow of u.

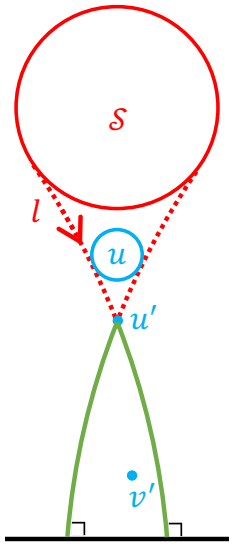


Figure 2.6: Antumbral cone in half-space

Therefore, to encode relation $\boldsymbol{u} \preceq \boldsymbol{v}$ using antumbral shadows, it is equivalent to require their surrogate points to satisfy $\boldsymbol{u}' \preceq \boldsymbol{v}'$ in the penumbral cone formulation. This establishes an equivalence between the entailment relations of antumbral and penumbral formulations.

Some Properties of Shadow Cones

We discuss some key properties of shadow cones in this section.

Theorem 2.2.1. *The shadow cone partial orders are transitive, i.e., if* $\boldsymbol{u} \preceq \boldsymbol{v}$ *and* $\boldsymbol{v} \preceq \boldsymbol{w}$, *then* $\boldsymbol{u} \preceq \boldsymbol{w}$.

This theorem follows immediately from the fact that the shadow cone relation is induced by the subset relation on shadows; a rigorous proof of this

theorem is provided in Appendix A.1.2. The shadow cone framework is thus well-suited framework to represent partial orders. In fact, shadow cones generalize entailment cones to a broad class of formulations, and the existing entailment cones in [60] is a special case of shadow cones, specifically, Formulation 3.

Theorem 2.2.2. *Entailment cones [60] are equivalent to Formulation 3, penumbral cones with a hyperbolic-ball-shaped light source at the origin.*

We prove this equivalence in Appendix A.1.2. As discussed in Section 1.2, the ϵ-hole problem of [60]'s entailment cones is a serious limitation of the model. Here, we give an intuitive explanation of this ϵ-hole problem around the origin in the shadow cones framework.

Remark 1 (Hole around the light source)**.** *The shadow is not defined when the light source $\mathcal{S}$ intersects with the point $\boldsymbol{u}$ (and its associated ball for umbral cones). We therefore require $d_{\mathbb{H}}(\boldsymbol{u}, \mathcal{S}) > r$, which results in a hole of hyperbolic radius r centered at $\mathcal{S}$.*

Umbral and penumbral cones also differ in many aspects that make them suitable for different purposes. For example,

Theorem 2.2.3. *Penumbral cones are geodesically-convex, while umbral cones are not due to the hypercycle boundary (Appendix A.1.1 and A.1.2).*

The half aperture of umbral cones are fixed for $\forall \boldsymbol{u} \in \mathbb{H}$, while that of penumbral cones become smaller as $\boldsymbol{u}$ approaches the 0-hyperplane. We note that the hyperbolic radius r and the level h in penumbral cones can be trainable parameters or even a function $r(\boldsymbol{u})$ and $h(\boldsymbol{u})$ in the space. However, in our main

Table 2.1: Properties of four shadow cone formulations

Cone	Model	Emb. Type	Convex?	Light Source	θ_u
Umbral	**Half-Space**	Ball	No	Point at ∞	Fixed
	Poincaré Ball	Ball	No	Point at Origin	Varying
Penumbral	**Poincaré Ball**	Point	Yes	Ball at Origin	Varying
	Half-Space	Point	Yes	Horosphere	Varying

experiments, we treat them as constants. A summary of these properties can be found in Table 2.1.

2.2.2 Optimization within Shadow Cones

In this section, we use our shadow cones to design an algorithm for embedding partial relational data. Given a dataset containing partial orders between pairs of elements, we seek to learn an embedding of all elements, such that the shadow cone framework can be used to evaluate the encoded partial order, and infer missing partial orders from the dataset.

A key component in learning with shadow cones is a differentiable energy function, or loss function, which quantifies both the confidence of a correctly classified pair $(\boldsymbol{u}, \boldsymbol{v})$, and the penalty associated with an incorrectly classified pair $(\boldsymbol{u}, \boldsymbol{v}')$. We propose to use the hyperbolic distance between $\boldsymbol{v}$ and the shadow cone of $\boldsymbol{u}$ as energy function.

The shortest path from $\boldsymbol{v}$ to the shadow cone induced by $\boldsymbol{u}$ can be classified into two distinct cases: (1) if $\boldsymbol{v}$ is at a higher "altitude" than $\boldsymbol{u}$, i.e. the apex of the cone, then the shortest path to the cone is the geodesic from $\boldsymbol{v}$ to the apex of the cone; (2) if $\boldsymbol{v}$ is at the same "altitude" as or lower than $\boldsymbol{u}$, then the shortest path

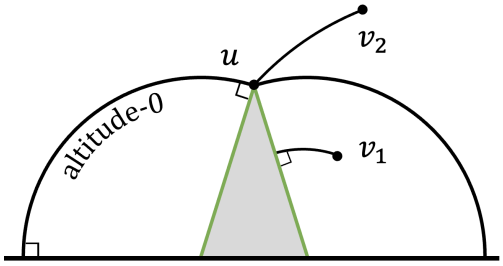


Figure 2.7: Shortest distances to u's cone in umbral-half-space, as computed differently for negative (v_1) and positive (v_2) altitude points respectively.

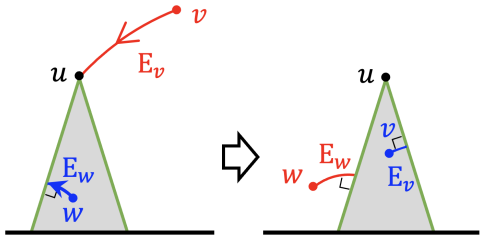


Figure 2.8: A toy optimization example: $\boldsymbol{u} \preceq \boldsymbol{v}, \boldsymbol{u} \parallel \boldsymbol{w}$. (Left) $\boldsymbol{v}$ is a child of $\boldsymbol{u}$, but is wrongly initialized outside of $\boldsymbol{u}$'s cone. $\boldsymbol{w}$ is incomparable with $\boldsymbol{u}$ but is initialized inside the cone. (Right) The energy gradients pulls $\boldsymbol{v}$ inside the cone, and pushes $\boldsymbol{w}$ outside. Red denotes positive energy, and blue negative.

to the cone is the shortest geodesic from $\boldsymbol{v}$ to the cone's boundary. Figure 2.7 illustrates the two distances for the umbral-half-space formulation.

To easily check which distance measure to use, we introduce a relative "altitude" function, $H(\boldsymbol{v}, \boldsymbol{u})$. If $H(\boldsymbol{v}, \boldsymbol{u}) > 0$, then $\boldsymbol{u}$ is at a higher altitude than the apex of $\boldsymbol{u}$'s cone, corresponding to case 1. Otherwise, we are in case 2. we present this relative altitude function along with the shortest distances to the cone, using the umbral-half-space formulation. We leave the discussion of other cases, such as penumbral cones, and detailed derivations to Appendix A.1.2.

Lemma 2.2.4 (Umbral-half-space)**.** *Define temperature* $t = (\sqrt{\sum_{i=1}^{n-1}(u_i - v_i)^2} - u_n \sinh \sqrt{k}r)/v_n$*, then the relative altitude function of* $\boldsymbol{v}$ *with respect to* $\boldsymbol{u}$ *is* $H(\boldsymbol{v}, \boldsymbol{u}) = v_n^2(1 + t^2) - u_n^2 \cosh^2 \sqrt{k}r$.

Theorem 2.2.5 (Shortest Distance to Umbral Cones)**.** *For umbral cones with temperature* t *and relative altitude function* $H(\boldsymbol{v}, \boldsymbol{u})$*, the signed-distance-to-boundary function is* $\frac{1}{\sqrt{k}} \operatorname{arsinh}(t) + r$*. Thus, the shortest distance from* $\boldsymbol{v}$ *to the umbral cone induced*

by $\boldsymbol{u}$ *is*

$$d(\boldsymbol{v}, Cone(\boldsymbol{u})) = \begin{cases} d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v}) & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) > 0, \\ \frac{1}{\sqrt{k}} \operatorname{arsinh}(t) + r & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) \leq 0. \end{cases}$$

We note that the sign of distance-to-boundary serves as another way to test partial order of $\boldsymbol{v}, \boldsymbol{u}$.

To learn embeddings in a geometrically meaningful manner, we introduce "shadow loss", which is designed to draw child nodes $\boldsymbol{v}$ closer to the cones of their patent nodes $\boldsymbol{u}$ while simultaneously pushing away randomly sampled, negative child nodes $\boldsymbol{v}'$:

$$\mathcal{L}_{\gamma_1,\gamma_2} = -\sum_{(\boldsymbol{u},\boldsymbol{v})\in P} \log \frac{\exp(-\max(E(\boldsymbol{u},\boldsymbol{v}),\gamma_2))}{\sum_{(\boldsymbol{u}',\boldsymbol{v}')\in N} \exp(\max(\gamma_1 - E(\boldsymbol{u}',\boldsymbol{v}'),0))}, \tag{2.1}$$

where P is the edge set of positive relations, N that of negative relations, and $E(\boldsymbol{u},\boldsymbol{v}) = d(\boldsymbol{v},\text{Cone}(\boldsymbol{u}))$ is the two-case distance defined previously. In addition, this loss allows us to choose how far to push negative samples away from the cone (distance $\gamma_1 > 0$), and how deep to pull positive samples into the cone (distance $\gamma_2 > 0$). Intuitively, positive samples shrink the embedding while negative ones dilate. In Figure 2.8, we show how the distance-energies are used to optimize a toy embedding for umbral-half-space.

Compared to the max-margin energy used in previous works [174, 60], namely $\mathcal{L} = \sum_{(\boldsymbol{u},\boldsymbol{v})\in P} E(\boldsymbol{u},\boldsymbol{v}) + \sum_{(\boldsymbol{u}',\boldsymbol{v}')\in N} \max(0, \gamma - E(\boldsymbol{u}',\boldsymbol{v}'))$, our distance-based shadow loss offers several advantages: (1) The learning dynamics are refined by considering the distance or depth of both wrongly and correctly classified points relative to the cone. (2) We adopt the contrastive-style loss proposed by [122], which we demonstrate to be effective in Section 2.2.3.

Our distance-based measure offers a more nuanced understanding of hier-

archical relationships compared to the angle-based energy used in [60]:$E(\boldsymbol{u}, \boldsymbol{v}) = \max(0, \phi(\boldsymbol{v}, \boldsymbol{u}) - \theta_{\boldsymbol{u}})$, where $\theta_{\boldsymbol{u}}$ is the half aperture and $\phi(\boldsymbol{v}, \boldsymbol{u})$ is the angle between the geodesic $\boldsymbol{uv}$ and cone's central axis at $\boldsymbol{u}$. This angle-based energy lacks the ability to capture the depth or confidence of the hierarchy. For example, all points $\boldsymbol{v}$ on the the cone's axis associated with $\boldsymbol{u}$ have $\phi(\boldsymbol{v}, \boldsymbol{u}) = 0$, yet some may require further optimization to better reflect hierarchical depth. Additionally, the angle-based energy, when used with max-margin energy loss in [60], leads to vanishing gradients for misclassified negative samples in the shadow cone [60]. Our approach effectively avoids this issue.

2.2.3 Experiments

This section showcases shadow cones' ability to represent and infer hierarchical relations on four datasets (detailed statistics in Appendix A.1): MCG [181, 185, 105], Hearst patterns [75, 101], WordNet Noun [36], and its Mammal sub-graph. We consider only the **Is-A** type relations in these datasets.

Transitive reduction and transitive closure. MCG and Hearst are pruned until they are acyclic, as detailed in Appendix A.1.3. We then compute their transitive reduction and closure [3]. Transitive reduction reduces the graph to a minimal set of relations from which all other relations could be inferred. Consistent with [60], we refer to this minimal set as the "basic" edges. On the other hand, transitive closure encompasses all pairs of points connected via transitivity. Transitive reduction and closure respectively offer the most succinct and exhaustive representations of DAGs.

Table 2.2: F1 score (%) on mammal sub-graph with best numbers **bolded**

Non-basic-edge Percentage	Dimension = 2					Dimension = 5				
	0%	10%	25%	50%	90%	0%	10%	25%	50%	90%
GBC-box	23.4	25.0	23.7	43.1	48.2	35.8	60.1	66.8	83.8	**97.6**
VBC-box	20.1	26.1	31.0	33.3	34.7	30.9	43.1	58.6	74.9	69.3
Entailment Cone	54.4	61.0	71.0	66.5	73.1	56.3	81.0	**84.1**	83.6	82.9
Umbral-half-space	**57.7**	73.7	**77.4**	**80.3**	**79.0**	**69.4**	81.1	83.7	**88.5**	91.8
Umbral-Poincaré-ball	44.6	58.9	60.5	65.3	63.6	62.4	67.4	81.4	81.9	92.2
Penumbral-half-space	52.8	**74.1**	70.9	72.3	76.0	67.8	**82.0**	83.5	87.6	89.9
Penumbral-Poincaré-ball	44.6	60.8	62.7	68.4	67.9	60.8	69.5	78.2	84.4	92.6

Table 2.3: F1 score (%) on WordNet noun, MCG, and Hearst with best numbers **bolded**

Dataset		Noun				MCG				Hearst			
Non-basic-edge Percentage		0%	10%	25%	50%	0%	10%	25%	50%	0%	1%	2%	5%
Entailment Cone	d=5	29.2	78.1	84.6	92.1	25.3	56.1	52.1	60.2	22.6	45.2	54.6	55.7
	d=10	32.1	82.9	91.0	95.2	25.5	58.9	55.5	63.8	23.7	46.6	54.9	58.2
Umbral-half-space	d=5	45.2	87.8	94.2	96.4	36.8	80.9	85.0	89.1	**32.8**	63.4	77.1	80.7
	d=10	**52.2**	**89.4**	**95.7**	**97.0**	**40.1**	**81.9**	**87.5**	**91.3**	32.6	**65.1**	**81.2**	**86.9**
Penumbral-half-space	d=5	44.6	82.6	86.2	88.3	35.0	78.6	81.1	85.3	26.8	62.8	72.3	78.8
	d=10	51.7	84.1	88.3	89.8	37.6	81.9	85.3	89.2	28.4	54.4	68.1	79.3

Training and testing. Non-basic edges can be inferred from the basic ones by transitivity, but the reverse is not true. Therefore, it is critical to include all basic edges in the training sets. Consistent with [60], we create training sets with varying levels of difficulty by progressively adding 1% to 90% of the non-basic edges. The remaining 10% of non-basic edges are evenly divided between the validation and test sets. Our main testing regime is to first train an embedding using a collection of basic and non-basic edges, and then use it to predict unseen non-basic edges. In Appendix A.1.3 we discuss other downstream tasks such as distance-based classification.

Initialization. The initial embeddings have been observed to be important for training [60]. Following the convention established by [122], we initialize our

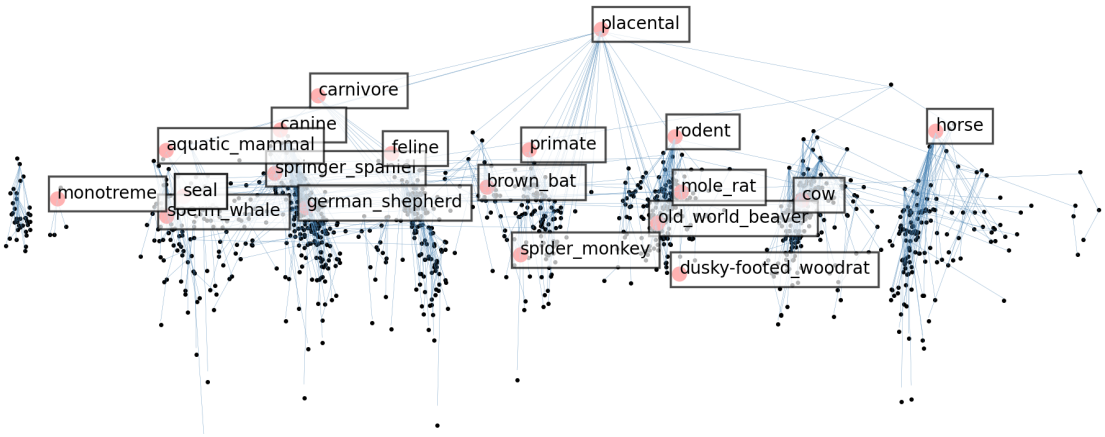


Figure 2.9: Mammal sub-graph embeddings in the 2D Umbral-Half-Space formulation. Black points represent the embedded nodes, while blue lines the basic edges between them.

embeddings as a uniform distribution around origin in $[-\epsilon, \epsilon]$ in each dimension. Note the origin of the Poincaré half-space model is $(0, \ldots, 0, 1)$. In the Poincaré ball model, since embedded points and light sources can not overlap, we project the initialized embeddings away until they are at least of distance r (radius of hyperbolic ball) from the origin. We note that [60] adopted a pre-trained 100-epoch embedding from [122] as initialization.

We benchmark against the entailment cones proposed in [60] by following their original implementation with max-margin angle-based energy loss, which to our knowledge is state-of-the-art model using cone embeddings in hyperbolic space. For completion, we also compare with the latest box embeddings in Euclidean space: GBC-box and VBC-box [207, 19]. Table 2.2 summarizes all four shadow cone's performance on Mammal, which nearly outperform all previous methods, Euclidean and Hyperbolic, in terms of generalization.

Discussion on the impact of ϵ-hole. We note that the Poincaré half-space formulations of shadow cones consistently outperform their Poincaré-ball counter-

parts. We attribute this difference to the initial embeddings prior to optimization. In the Poincaré-ball model, due to the hole around the light source at the origin, the initial embeddings are randomly projected away onto the boundary of the hole, which can destroy the hierarchical relationship between two objects by potentially pushing them to opposite directions. However, in contrast, shadow cones in the Poincaré half-space model do not suffer from this issue as the light source is placed far away from the hyperbolic origin.

On large datasets (WordNet Noun, MCG, and Hearst), we compare shadow cones in the Poincaré half-space against the baseline, as shown in 2.3. In all experiments, umbral-half-space cones consistently outperform the baselines and penumbral-half-space cones for all non-basic-edge percentages. This is likely because penumbral-half-space has a height limit while umbral-half-space does not. Finally, we visualize one of our trained embeddings: Umbral-half-space on Mammals, in 2.9. The points represent taxonomic names, with blue edges indicating basic relations. It's noteworthy that the learnt embedding naturally organizes points into clusters, roughly corresponding to families. The depth of points within these clusters may be interpreted as taxonomic ranks, such as the Canidae family to the German Shepherd species. Our code is available on github [4].

Geodesic Convexity of Penumbral Cones. In our experiments thus far, penumbral cones have not demonstrated performance on par with umbral cones in the half-space model, potentially due to their height limit and variable aperture. However, we would like to highlight a theoretical advantage unique to penumbral cones - geodesic convexity. Suppose $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are both within the

[4] `https://github.com/ydtydr/ShadowCones`

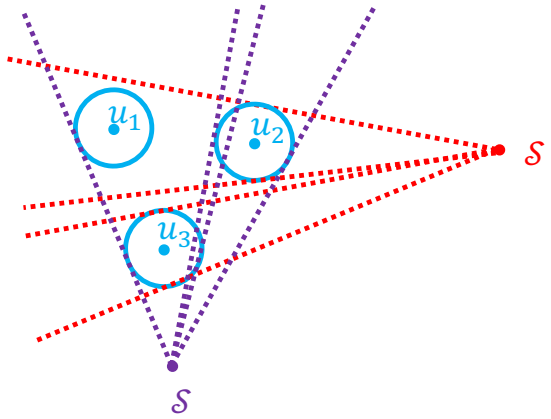


Figure 2.10: Multi-relation embedding in Euclidean Space. Marked in purple and red are the two different light sources, and the respective shadow boundaries they cast.

penumbral cone of $\boldsymbol{u}$, then the entire geodesic segment $\overline{v_1 v_2}$ also resides within the penumbral cone. Such convexity lends itself to more meaningful geometric operations, such as interpolation. We thus conjecture that penumbral embeddings are better suited for word2vec or GloVe-style semantic analysis [164]. Semantic analysis in Euclidean spaces necessitates the comparison of vectors in Euclidean space, which is straightforward. However, comparing geodesics in general Riemannian manifolds requires careful approaches using methods such as exponential maps and parallel transport.

Multi-relation Embeddings. The shadow cones framework also allows one to capture multiple relation types within one embedding by using multiple light sources, with the shadows cast by each light source capturing one type of relation. It is possible to enrich the same embedding simultaneously with various types of relations, such as entailment and causality. This can be done while relaxing the classification accuracy for each relation types. Our framework readily facilitates this, as we can utilize multiple light sources, each casting shadows of

a distinct color that captures a different type of relation. An example is depicted in Figure 2.10, which is set in Euclidean space for simplicity. This approach may enable more comprehensive representation of complex relational datasets.

2.3 HyLa: Random Hyperbolic Laplacian Features

Overview

In Section 2.1 and Section 2.2, we see hyperbolic space can support high-fidelity embeddings of tree- and graph-structured data, upon which various hyperbolic networks have been developed. Presently, most well-known and -established deep neural networks are built in Euclidean space. The standard approach is to pass the input to a Euclidean network and hope the model can learn the features and embeddings. But this flat-space approach can encode the wrong prior in tasks for which we know the underlying data has a different geometric structure, such as the hyperbolic-space structure implicit in tree-like graphs.

Existing hyperbolic networks adopt hyperbolic geometry at every layer of the model. Since hyperbolic space is not a vector space, operations such as addition and multiplication are not well-defined; neither are matrix-vector multiplication and convolution, which are key components of a deep model that uses hyperbolic geometry at every layer. A common solution is to treat hyperbolic space as a *gyro-vector space* by equipping it with a non-commutative, non-associative addition and multiplication, allowing hyperbolic points to be processed as features in a neural network forward. However, this complicates the use of hyperbolic geometry in neural networks because the imposition of an extra structure on hyperbolic space beyond its manifold properties—making the approach somehow non-geometric. A second problem with using hyperbolic points as intermediate features is that these points can stray far from the origin (just as Euclidean DNNs require high dynamic range [89]), especially for deeper networks. This can cause significant numerical issues when the

space is represented with ordinary floating-point numbers: the representation error is unbounded and grows exponentially with the distance from the origin. Much careful hyperparameter tuning is required to avoid this "NaN problem" [147, 198, 200]. These issues call for a simpler and more principled way of using hyperbolic geometry in DNNs.

Here we propose such a simple approach for learning with hyperbolic space. The insight is to (1) encode the hyperbolic geometric priors *only at the input* via an embedding into hyperbolic space, which is then (2) mapped once into Euclidean space by a random feature mapping $\phi : \mathbb{H}_n \to \mathbb{R}^d$ that (3) respects the geometry of hyperbolic space in that its induced kernel $k(\boldsymbol{x}, \boldsymbol{y}) = \mathbb{E}[\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{y}) \rangle]$ is *isometry-invariant*, i.e. $k(\boldsymbol{x}, \boldsymbol{y})$ depends only on the hyperbolic distance between $\boldsymbol{x}$ and $\boldsymbol{y}$, followed by (4) passing these Euclidean features through some downstream Euclidean network. This approach both avoids the numerical issues common in previous approaches (since hyperbolic space is only used once early in the network, numerical errors will not compound) and eschews the need for augmenting hyperbolic space with any additional non-geometric structure (since we base the mapping only on geometric distances in hyperbolic space). Our contributions are as follows:

- In Section 2.3.1 we propose a random feature extraction called HyLa which can be sampled to be an unbiased estimator of any isometry-invariant kernel on hyperbolic space. This generalizes the classic method of random Fourier features proposed for Euclidean space by [140].
- In Section 2.3.2 we show how to adopt HyLa in an end-to-end graph learning architecture that simultaneously learns the embedding of the initial objects and the Euclidean graph learning model.

- In Section 2.3.3, we evaluate our approach empirically. Our HyLa-networks demonstrate better performance, scalability and computation speed than existing hyperbolic networks: HyLa-networks consistently outperform HGCN, even on a tree dataset, with 12.3% improvement while being 4.4× faster. Meanwhile, we argue that our method is an important hyperbolic baseline to compare against due to its simple implementation and compatibility with any graph learning model.

Related Work

To encode geometric priors into neural networks, many versions of *hyperbolic neural networks* have been proposed. But while (matrix-) addition and multiplication are essential to develop a DNN, hyperbolic space is not a vector space with well-defined addition and multiplication. To handle this issue, several approaches were proposed in the literature.

Gyrovector space. Many hyperbolic networks, including HNN [60], HNN++ [153], HVAE [111], HGAT [210], and GIL [214], adopt the framework of *gyrovector space* as an algebraic formalism for hyperbolic geometry, by equipping hyperbolic space with non-associative addition and multiplication: Möbius addition $\oplus$ and Möbius scalar multiplication $\otimes$, which is defined for $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{B}^n$ and a scalar $r \in \mathbb{R}$

$$\boldsymbol{x} \oplus \boldsymbol{y} := \frac{(1 + 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle + ||\boldsymbol{y}||^2)\boldsymbol{x} + (1 - ||\boldsymbol{x}||^2)\boldsymbol{y}}{1 + 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle + ||\boldsymbol{x}||^2||\boldsymbol{y}||^2}, \quad r \otimes \boldsymbol{x} := \tanh(r \tanh^{-1}(||\boldsymbol{x}||))\frac{\boldsymbol{x}}{||\boldsymbol{x}||}.$$

However, Möbius addition and multiplication are complicated with a high computation cost; high level operations such as Möbius matrix-vector multiplication

are even more complicated and numerically unstable [200, 201], due to the use of ill-conditioned functions like $\tanh^{-1}$. Also problematic is the way hyperbolic space is treated as a gyrovector space rather than a manifold, meaning this approach can not be generalized to other manifolds that lack this structure.

Push-forward & Pull-backward. Since many operations are well-defined in Euclidean space but not in hyperbolic space, a natural idea is to map $\mathbb{H}_n$ to $\mathbb{R}^d$ via some mappings, apply well-defined operations in $\mathbb{R}^d$, then map the results back to hyperbolic space. Many works, including HATN [67], HGCN [28], HGNN [106], HGAT [210], Lorentzian GCN [211], and GIL [214], adopt this method:

- pull the hyperbolic points to Euclidean space with a "pull-backward" mapping;
- apply operations such as multiplication and convolution in Euclidean space; and then
- push the resulting Euclidean points to hyperbolic space with a "push-forward" mapping.

Since hyperbolic space and Euclidean space are different spaces, no isomorphic maps exist between them. A common choice of the mappings [28, 106] is the exponential map $\exp_{\boldsymbol{x}}(\cdot)$ and logarithm map $\log_{\boldsymbol{x}}(\cdot)$ (Section 2.1), where $\boldsymbol{x}$ is usually chosen to be the origin $\boldsymbol{O}$.

This approach is more straightforward and natural in the sense that hyperbolic space is only treated as a manifold object with no more structures added, so it can be generalized to general manifolds (although it does privilege the origin). However, $\exp_{\boldsymbol{o}}(\cdot)$ and $\log_{\boldsymbol{o}}(\cdot)$ are still complicated and numerically unstable.

Both push-forward and pull-backward mappings are used at every hyperbolic layer, which incurs a high computational cost in both the model forward and backward loop. As a result, this prevents hyperbolic networks from scaling to large graphs. Moreover, the Push-forward & Pull-backward mappings act more like nonlinearities instead of producing meaningful features.

Kernel Methods and Horocycle Features. [33] proposed hyperbolic kernel SVM for nonlinear classification without resorting to ill-fitting tools developed for Euclidean space. Their approach differs from ours in that they map hyperbolic points to another (higher-dimensional) hyperbolic feature space, rather than an Euclidean feature space. They also constructed feature mappings only for the Minkowski inner product kernel: it's unknown how to construct feature mappings of their type for general kernels. Another work by [55] develops several valid positive definite kernels in hyperbolic spaces and investigates their usages; they do not provide any sampling-based features to approximate these kernels. [178] constructed hyperbolic neuron models using a push-forward mapping along with the *hyperbolic Poisson kernel*

$$P_n(\boldsymbol{x}, \boldsymbol{\omega}) = (\frac{1 - ||\boldsymbol{x}||^2}{||\boldsymbol{x} - \boldsymbol{\omega}||^2})^{n-1}, \quad \boldsymbol{x} \in \mathcal{B}^n, \ \ \boldsymbol{\omega} \in \partial\mathcal{B}^n$$

as the backbone of an even more complicated feature function. [155] theoretically proposes a continuous version of shallow fully-connected networks on non-compact symmetric space (including hyperbolic space) using Helgason-Fourier transform, where some network functions coincidentally share some similarities to features proposed in this paper.

Background on Laplacian

Laplace operator (Euclidean). The Laplace operator Δ on Euclidean space $\mathbb{R}^n$ is defined as the divergence of the gradient of a scalar function f, i.e.,

$$\Delta f = \nabla \cdot \nabla f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}.$$

It serves as an important characterization of the space, which has a physical interpretation for non-equilibrium diffusion as the extent to which a point represents a source or sink of chemical concentration.

The eigenfunctions of Δ are the solutions of the *Helmholtz equation* $-\Delta f = \lambda f, \lambda \in \mathbb{R}$, and can form an *orthonormal basis* for the Hilbert space $L^2(\Omega)$ when $\Omega \in \mathbb{R}^n$ is compact [62], i.e., a linear combination of them can represent any function/model that is L^2-integrable. A notable parameterization for these eigenfunctions are the *plane waves*, given by $f(\boldsymbol{x}) = \exp(i\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle)$, where $\boldsymbol{\omega} \in \mathbb{R}^n$ and $\langle \cdot, \cdot \rangle$ is the Euclidean inner product. A standard result given in [76] states that any eigenfunction of Δ can be written as a linear combination of these plane waves (Theorem A.2.1). The famous result of [140] used these eigenfunctions to construct feature maps, called *random Fourier features*, for arbitrary shift-invariant kernels in Euclidean space.

Theorem 2.3.1 (Bochner's theorem, [146])**.** *For any shift-invariant continuous kernel $k(\boldsymbol{x}, \boldsymbol{y}) = k(\boldsymbol{x} - \boldsymbol{y})$ on $\mathbb{R}^n$, let $p(\boldsymbol{\omega})$ be its Fourier transform and $\xi_\omega(\boldsymbol{x}) = \exp(i\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle)$. Then k is positive definite if and only if $p \geq 0$, in which case if we draw $\boldsymbol{w}$ proportionally to p,*

$$k(\boldsymbol{x} - \boldsymbol{y}) = \int_{\mathbb{R}^n} p(\boldsymbol{\omega}) \exp(i\langle \boldsymbol{\omega}, \boldsymbol{x} - \boldsymbol{y} \rangle)\, d\boldsymbol{\omega} = k(\boldsymbol{0}) \cdot \mathbb{E}_{\omega \sim p}\left[\xi_\omega(\boldsymbol{x}) \xi_\omega(\boldsymbol{y})^*\right].$$

Since both the probability distribution $p(\boldsymbol{\omega})$ and the kernel $k(\boldsymbol{x} - \boldsymbol{y})$ are real, the integral is unchanged when we replace the exponential with a co-

sine. [140] leveraged this to produce real-valued features by setting $z_{\omega,b}(\boldsymbol{x}) = \sqrt{2}\cos(\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle + b)$, arriving at a real-valued mapping that satisfies the condition $\mathbb{E}_{\boldsymbol{w} \sim p, b \sim \text{Unif}[0,2\pi]}\left[z_{\omega,b}(\boldsymbol{x}) z_{\omega,b}(\boldsymbol{y})\right] = k(\boldsymbol{x} - \boldsymbol{y})$. [140] approximated functions such as Gaussian, Laplacian and Cauchy kernels with this technique.

Laplace-Beltrami operator (Hyperbolic). The Laplace-Beltrami operator $\mathcal{L}$ is the generalization of the Laplace operator to Riemannian manifolds, defined as the divergence of the gradient for any twice-differentiable real-valued function f, i.e., $\mathcal{L}f = \nabla \cdot \nabla f$. In the n-dimensional Poincaré disk model $\mathcal{B}^n$, the Laplace-Beltrami operator takes the form [2]

$$\mathcal{L} = \frac{1}{4}(1 - \|\boldsymbol{x}\|^2)^2 \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2} + \frac{n-2}{2}(1 - \|\boldsymbol{x}\|^2) \sum_{i=1}^{n} x_i \frac{\partial}{\partial x_i}.$$

Just as in the Euclidean case, the eigenfunctions of $\mathcal{L}$ in hyperbolic space can be derived by solving the Helmholtz equation. We might hope to find analogs of the "plane waves" in hyperbolic space that are eigenfunctions of $\mathcal{L}$. One way to approach this is via a geometric interpretation of plane waves.

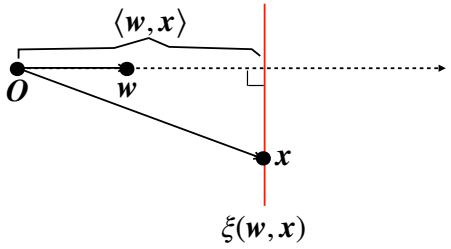


Figure 2.11: Euclidean hyperplane.

In the Euclidean case, for unit vector $\boldsymbol{\omega}$ and scalar λ, $f(\boldsymbol{x}) = \exp(i\lambda\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle)$ is called a "plane wave" because it is constant on each hyperplane perpendicular to $\boldsymbol{\omega}$. We can interpret $\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle$ as the signed distance from the origin $\boldsymbol{O}$ to the hyperplane $\xi(\boldsymbol{\omega}, \boldsymbol{x})$ which contains $\boldsymbol{x}$ and is perpendicular to $\boldsymbol{w}$ (Figure 2.11).

In the Poincaré ball model of hyperbolic space, the geometric analog of the hyperplane is the *horocycle*. For any $\boldsymbol{z} \in \mathcal{B}^n$ and unit vector $\boldsymbol{\omega}$ (i.e., $\boldsymbol{\omega} \in \partial\mathcal{B}^n$), the horocycle $\xi(\boldsymbol{\omega}, \boldsymbol{z})$ is the Euclidean circle that passes through $\boldsymbol{\omega}, \boldsymbol{z}$ and is tangent

to the boundary $\partial\mathcal{B}^n$ at $\boldsymbol{\omega}$, as indicated in Figure 2.12. We let $\langle \boldsymbol{\omega}, z\rangle_H$ denote the signed hyperbolic distance from the origin $\boldsymbol{O}$ to the the horocycle $\xi(\boldsymbol{\omega}, z)$. In the Poincaré ball model, this takes the form

$$\langle \boldsymbol{\omega}, z\rangle = \log \frac{1 - \|z\|^2}{\|z - \boldsymbol{\omega}\|^2} = \log P(z, \boldsymbol{\omega}), \quad z \in \mathcal{B}^n, \quad \boldsymbol{\omega} \in \partial\mathcal{B}^n,$$

If we define the "hyperbolic plane waves" $\exp(\mu\langle \boldsymbol{\omega}, z\rangle_H)$, where $\mu \in \mathbb{C}$. Unsurprisingly, they are indeed eigenfunctions of the hyperbolic Laplacian [2].

$$\mathcal{L}\exp(\mu\langle \boldsymbol{\omega}, z\rangle_H) = \mu(\mu - n + 1)e(\mu\langle \boldsymbol{\omega}, z\rangle_H).$$

Since we are interested in finding real eigenfunctions (via the same exp-to-cosine trick used in [140]), we restrict our attention to μ that yield a real eigenvalue. This happens when $\mu = \frac{n-1}{2} + i\lambda$ for real λ, in which case the eigenvalue is $\mu(\mu - n + 1) = -\lambda^2 - (n-1)^2/4$. Just as the Euclidean plane waves $\exp(i\langle \boldsymbol{\omega}, \boldsymbol{x}\rangle)$ span the eigenspaces of the Euclidean Laplacian, the same result holds for these "hyperbolic plane waves" (Theorem A.2.2).

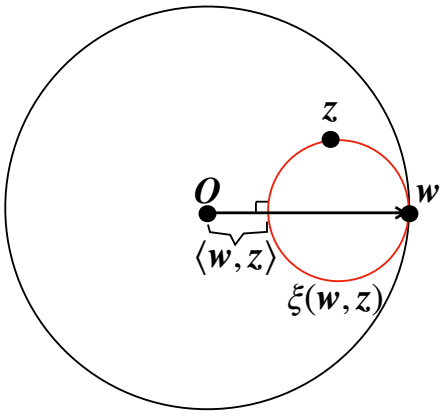


Figure 2.12: Hyperbolic horocycle.

2.3.1 HyLa: Euclidean Features from Hyperbolic Embeddings

In this section, we present HyLa, a feature mapping that can approximate an isometry-invariant kernel over hyperbolic space $\mathbb{H}_n$ in the same way that the random Fourier features of [140] approximate any shift-invariant kernel over $\mathbb{R}^n$. In place of the Euclidean plain waves, which are the eigenfunctions of the

Euclidean Laplacian, here we derive our feature extraction using the hyperbolic plain waves, which are eigenfunctions of the hyperbolic Laplacian. Since the hyperbolic plane wave $\exp((\frac{n-1}{2} - i\lambda)\langle\langle \boldsymbol{\omega}, \boldsymbol{x}\rangle_H)$ is an eigenfunction of the real operator $\mathcal{L}$ with real eigenvalue, so will this function multiplied by any phase $\exp(-ib)$, as will its real part. Call the result of this

$$\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(z) = \exp\left(\frac{n-1}{2}\langle \boldsymbol{\omega}, z\rangle_H\right)\cos\left(\lambda\langle \boldsymbol{\omega}, z\rangle_H + b\right). \tag{2.2}$$

This parameterization, which we call **HyLa** (for **Hy**perbolic **La**placian features), yields real-valued eigenfunctions of the Laplace-Beltrami operator with eigenvalue $-\lambda^2 - (n-1)^2/4$.

HyLa eigenfunctons have the nice property that they are bounded in almost every direction, as $\langle \boldsymbol{\omega}, z\rangle_H$ approaches 0 as z approaches any point on the boundary of $\mathcal{B}^n$ except $\boldsymbol{\omega}$. Note that HyLa eigenfunctions are invariant to isometries of the space: any isometric transformation of HyLa yields another HyLa eigenfunction with the same λ but a transformed $\boldsymbol{\omega}$ (depending on how the isometry acts on the boundary $\partial\mathcal{B}^n$). It is easy to compute, parameterized by continuous instead of discrete parameters, and are analogous to random Fourier features. Moreover, HyLa can be extended to eigenfunctions on other manifolds (e.g. symmetric spaces) since we only use manifold properties of $\mathbb{H}_n$.

We show that for any $\lambda \in \mathbb{R}$, under uniform sampling of $\boldsymbol{\omega}$ and b, the product $\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{x})\,\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{y})$ is an unbiased estimate of an isometry-invariant kernel $k(\boldsymbol{x}, \boldsymbol{y})$.

Theorem 2.3.2. *Let $\boldsymbol{\omega}$ be sampled uniformly (under the Euclidean metric) from the boundary $\partial\mathcal{B}^n$ and let b be sampled uniformly from $[0, 2\pi]$. Then $\mathbb{E}\left[\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{x})\,\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{y})\right] = k_\lambda(\boldsymbol{x}, \boldsymbol{y})$ for any $\lambda \in \mathbb{R}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{H}_n$, where the function*

k_λ is an isometry-invariant kernel given by

$$k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \cdot {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right),$$

where ${}_2F_1$ is the hypergeometric function, defined via analytic continuation by the power series

$${}_2F_1\left(a, b; c; z\right) = \sum_{n=0}^{\infty} \frac{(a)_n (b)_n}{(c)_n} \frac{z^n}{n!} = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \qquad (|z| < 1).$$

Note that despite the presence of an i in the formula, this kernel is clearly real because the hypergeometric function satisfies the properties ${}_2F_1(a, b; c; z) = {}_2F_1(b, a; c; z)$ and ${}_2F_1(a, b; c; z)^* = {}_2F_1(a^*, b^*; c^*; z^*)$. In practice, as with random Fourier features, instead of choosing one single λ, we select them at random from some distribution $\rho(\lambda)$. The resulting kernel will be an isometry-invariant kernel that depends on the distribution of λ, as follows:

$$k(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\int_{-\infty}^{\infty} {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right) \cdot \rho(\lambda)\, d\lambda. \quad (2.3)$$

This formula gives a way to derive the isometry-invariant kernel from a distribution $\rho(\lambda)$; if we are interested in finding a feature map for some particular kernel, we can invert this mapping to get a distribution for λ which will produce the desired kernel.

Theorem 2.3.3. *Suppose that $k(\boldsymbol{x}, \boldsymbol{y}) = k(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))$ is an isometry-invariant positive semidefinite kernel. Assume the existence of an associated density $\rho(\lambda)$ with the kernel, then*

$$\rho(\lambda) = \lambda \tanh\left(\frac{\pi\lambda}{2}\right) \int_{\mathcal{B}^n} \int_{\partial\mathcal{B}^n} k(d_{\mathbb{H}}(\boldsymbol{z}, \boldsymbol{O})) \exp\left(\left(\frac{n-1}{2} - i\lambda\right)\langle \boldsymbol{\omega}, \boldsymbol{z}\rangle_H\right) d\boldsymbol{\omega}\, d\boldsymbol{z}.$$

i.e., $\rho(\lambda)$ is the spherical transform of the kernel, and if we draw λ proportional to ρ, $\boldsymbol{\omega}$ uniformly on $\partial\mathcal{B}^n$, and b uniformly on $[0, 2\pi]$, then

$$k(0) \cdot \mathbb{E}\left[\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{x})\, \mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{y})\right] = k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = k(\boldsymbol{x}, \boldsymbol{y}).$$

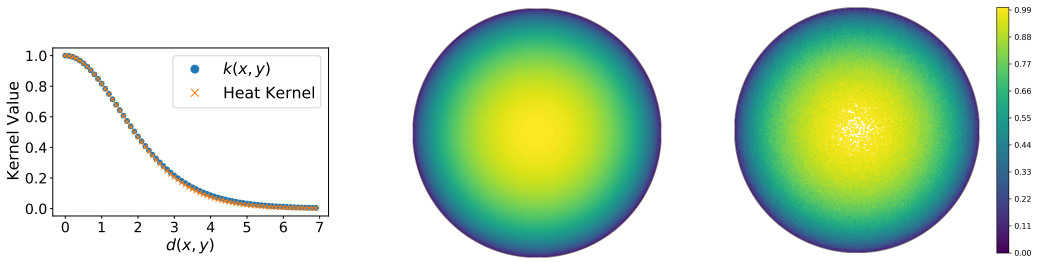


Figure 2.13: Visualization of the kernel $k(\boldsymbol{x}, \boldsymbol{y})$ when $\rho(\lambda) = \mathcal{N}(0, 0.5^2)$. (left) Distributions of $k(\boldsymbol{x}, \boldsymbol{y})$ and the heat kernel (at temperature $t = 6$) in 3D hyperbolic space; (middle) $k(\boldsymbol{x}, \boldsymbol{O})$ in 2D Poincaré disk model; (right) $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{y}) \rangle$ for HyLa features based on $D = 1000$ samples.

Although Theorem 2.3.3 lets us find a HyLa distribution for any isometric kernel, for simplicity in this paper, because the closed-forms of many kernels in $\mathbb{H}_n$ are not available, rather than arriving at a distribution via this inverse, we will instead focus on the case where ρ is a Gaussian. This corresponds closely to a heat kernel [64], as illustrated in Figure 2.13 (left).

We will use HyLa eigenfunctions to produce Euclidean features from hyperbolic embeddings, using the same random-features approach as [140]. Concretely, to map from $\mathbb{H}_n$ to $\mathbb{R}^D$, we draw D independent samples $\lambda_1, \ldots, \lambda_D$ from ρ, D independent samples $\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_D$ uniform from $\partial \mathcal{B}^n$, and D independent samples $b_1, \ldots, b_D$ uniform from $[0, 2\pi]$, and then output a feature map ϕ the kth coordinate of which is $\phi_k(\boldsymbol{x}) = \frac{1}{\sqrt{D}} \mathrm{HyLa}_{\lambda_k, b_k, \boldsymbol{\omega}_k}(\boldsymbol{x})$. It is easy to see that this will yield feature vectors with $\mathbb{E}\left[\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{y}) \rangle\right] = k(\boldsymbol{x}, \boldsymbol{y})$ as given in Eq. 2.3.

We visualize the kernel $k(\boldsymbol{x}, \boldsymbol{O})$ for $\rho(\lambda) = \mathcal{N}(0, 0.25)$ in the 2-dimensional Poincarè disk in Figure 2.13, evaluating the integral in Eq. 2.3 using Gauss–Hermite quadrature. In Figure 2.13 (right), we sample random HyLa features with $D = 1000$ and plot $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{O}) \rangle$. Visibly, the HyLa features approximate the kernel well. A discussion of the estimation error is provided in Section A.2.2.

Connection to Euclidean Activation. There is a close connection between the HyLa eigenfunction and the Euclidean activations used in Euclidean fully connected networks. Given a data point $\boldsymbol{x} \in \mathbb{R}^n$, a weight $\boldsymbol{\omega} \in \mathbb{R}^n$, a bias $b \in \mathbb{R}$ and nonlinearity σ, a Euclidean DNN activation can be written as

$$\sigma(\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle + b) = \sigma(\|\boldsymbol{\omega}\| \langle \frac{\boldsymbol{\omega}}{\|\boldsymbol{\omega}\|}, \boldsymbol{x} \rangle + b),$$

In hyperbolic space, for $\boldsymbol{z} \in \mathcal{B}^n$, $\boldsymbol{\omega} \in \partial\mathcal{B}^n$, $\lambda \in \mathbb{R}$, $b \in \mathbb{R}$ and $\sigma = \cos$, we can reformulate the HyLa eigenfunction as

$$\mathrm{HyLa}_{\lambda,b,\boldsymbol{\omega}}(\boldsymbol{z}) = \sigma\left(\lambda \langle \boldsymbol{\omega}, \boldsymbol{z} \rangle_H + b\right) \exp\left(\frac{n-1}{2} \langle \boldsymbol{\omega}, \boldsymbol{z} \rangle_H\right),$$

HyLa generalizes Euclidean activations to hyperbolic space, with an extra factor $\exp\left(\frac{n-1}{2} \langle \boldsymbol{\omega}, \boldsymbol{z} \rangle_H\right)$ from the curvature of $\mathbb{H}_n$.

From a functional perspective, any $f \in L^2(\mathbb{H}_n)$ can be expanded as an infinite linear combination (integral form) of HyLa (Theorem 4.3 in [155]). This statement holds whenever the non-linearity σ is a tempered distribution on $\mathbb{R}$, i.e., the topological dual of the Schwartz test functions, including ReLU and cos. Though the features will not approximate a kernel on hyperbolic space if $\sigma \neq \cos$, this suggests variants of HyLa with other nonlinearities may be interesting to study.

2.3.2 HyLa for Graph Learning

In this section, we show how to use HyLa to encode geometric priors for end-to-end graph learning.

Background on Graph Learning. A graph is defined formally as $\mathcal{G} = (\mathcal{V}, \mathbf{A})$, where $\mathcal{V}$ represents the vertex set consisting of n nodes and $\mathbf{A} \in \mathbb{R}^{n \times n}$ represents the symmetric adjacency matrix. Besides the graph structure, each node in the graph has a corresponding d-dimensional feature vector: we let $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote the entire feature matrix for all nodes. A fraction of nodes are associated with a label indicating one (or multiple) categories it belongs to. The *node classification* task is to predict the labels of nodes without labels or even of nodes newly added to the graph.

An important class of Euclidean graph learning model is the graph convolutional neural network (GCN) [92, 47]. The GCN is widely used in graph tasks including semi-supervised learning for node classification, supervised learning for graph-level classification, and unsupervised learning for graph embedding. Many complex graph networks and GCN variants have been developed, such as the graph attention networks (GAT [172]), FastGCN [30], GraphSage [72], and others [173, 190]. An interesting work to understand GCN is *simplifying GCN* (SGC [183]): a linear model derived by removing the non-linearities in a K-layer GCN as:

$$f(\mathbf{A}, \mathbf{X}) = \text{softmax}(\mathbf{S}^K \mathbf{X} \mathbf{W}) \ ,$$

where $\mathbf{S}$ is the "normalized" adjacency matrix with added self-loops and $\mathbf{W}$ is the trainable weight. This is essentially a multi-class logistic regression on the pre-processed features $\mathbf{S}^K\mathbf{X}$. Note that the pre-processed features $\mathbf{S}^K\mathbf{X}$ can be computed before training, which enables large graph learning and greatly saves memory and computation cost.

End-to-End Learning with HyLa. We propose a feature-extracted architecture via the following recipe: embed the data objects (graph nodes or features, de-

tailed below) into some space (e.g., Euclidean or hyperbolic), map the embedding to Euclidean features $\overline{\mathbf{X}}$ via the kernel transformation (e.g., RFF or HyLa), and finally apply an Euclidean graph learning model $f(\mathbf{A}, \overline{\mathbf{X}})$. This recipe only manipulates the input of the graph learning model and hence, this architecture can be used with *any* graph learning model. The graph model and the hyperbolic embedding are learned simultaneously with backpropagation. In theory, the embedding space can be any desired space, just use the same Laplacian recipe to construct features. Below we only show the pipeline for hyperbolic space, while we also include Euclidean space (with random Fourier features) as a baseline in our experiments.

Directly Embed Graph Nodes. We embed each node into a low dimensional hyperbolic space $\mathcal{B}^{d_0}$ as hyperbolic embedding $\mathbf{Z} \in \mathbb{R}^{n \times d_0}$ for all nodes in the graph, which can be either a pretrained fixed embedding or as parameters learnt together with the subsequent graph learning model during training time. To compute with the HyLa eigenfunctions, first sample d_1 points uniformly from the boundary $\partial\mathcal{B}^{d_0}$ to get $\mathbf{\Omega} \in \mathbb{R}^{d_1 \times d_0}$, then sample d_1 eigenvalues and biases separately from $\mathcal{N}(0, s^2)$ and Uniform($[0, 2\pi]$) to get $\mathbf{\Lambda}, \mathbf{B} \in \mathbb{R}^{d_1}$, where s is a scale constant. The resulting feature matrix ($\overline{\mathbf{X}} \in \mathbb{R}^{n \times d_1}$) computation follows; please refer to Algorithm 1 for a detailed

Algorithm 1 End-to-End HyLa

input: n objects, Poincaré disk $\mathcal{B}^{d_0}$, HyLa feature dimension d_1, adjacency matrix $\mathbf{A}$, node feature matrix $\mathbf{X}$, graph neural network f
initialize $\mathbf{Z} \in \mathbb{R}^{n \times d_0}$ {hyperbolic embeddings}
sample boundary pts matrix $\mathbf{\Omega} \in \mathbb{R}^{d_1 \times d_0}$, eigenvalues $\mathbf{\Lambda} \in \mathbb{R}^{n \times d_1}$ and biases $\mathbf{B} \in \mathbb{R}^{n \times d_1}$
compute $\mathbf{P} = \langle \mathbf{\Omega}, \mathbf{Z} \rangle_H \in \mathbb{R}^{n \times d_1}$ {Horocycle distance}
compute $\overline{\mathbf{X}} = \exp\left(\frac{n-1}{2}\mathbf{P}\right)\cos(\mathbf{\Lambda} \cdot \mathbf{P} + \mathbf{B})$ {HyLa}
if embedding features: $\overline{\mathbf{X}} = \mathbf{X}\overline{\mathbf{X}} \in \mathbb{R}^{n \times d_1}$
return $\mathbf{Y} = f(\mathbf{A}, \overline{\mathbf{X}})$ {e.g., SGC}

breakdown. The mapped features $\overline{\mathbf{X}}$ are then fed into the chosen graph learning model $f(\mathbf{A}, \overline{\mathbf{X}})$ for prediction.

Embed Features. One can also embed the given features into hyperbolic space so as to derive the node features implicitly. Specifically, when a node feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ is given, we initialize a hyperbolic embedding for each of the d dimensions to derive hyperbolic embeddings $\mathbf{Z} \in \mathbb{R}^{d \times d_0}$. The Euclidean features $\overline{\mathbf{X}}$ can be computed in the same manner following Algorithm 1. However, to get the new node feature matrix, an extra aggregation step $\overline{\mathbf{X}} = \mathbf{X}\overline{\mathbf{X}} \in \mathbb{R}^{n \times d_1}$ is required before fedding into a graph learning model.

Embedding graph nodes is better-suited for tasks where no feature matrix is available or meaningful features are hard to derive. However, the size of the embedding $\mathbf{Z}$ will be proportional to the graph size, hence it may not scale to very large graphs due to memory and computation constraints. Furthermore, this method can only be used in a transductive setting, where nodes in the test set are seen during training. In comparison, embedding features can be used even for large graphs since the dimension d of the original feature matrix is usually fixed and much lower than the number of nodes n. Note that as the hyperbolic embeddings are built for each feature dimension, they can be used in both transductive and inductive settings, as long as the test data shares the same set of features as the training data. One limitation is that its performance depends on the existence of a feature matrix $\mathbf{X}$ that contains sufficient information for learning.

Any graph learning model can be used in the proposed feature-extracted architecture. In our experiments, we focus primarily on the simple linear graph

network SGC, which takes the form $f(\mathbf{A}, \overline{\mathbf{X}}) = \text{softmax}(\mathbf{A}^K \overline{\mathbf{X}} \mathbf{W})$ with a trainable weight matrix $\mathbf{W}$. Note that just as the vanilla SGC case, $\mathbf{A}^K$ or $\mathbf{A}^K \mathbf{X}$ can be pre-computed in the same way before training and inference. For the purpose of end-to-end learning, we jointly learn the embedding parameter $\mathbf{Z}$ and weight $\mathbf{W}$ in SGC during the training time. It's also possible to adopt a two-step approach, i.e., first pretrain a hyperbolic embedding following [122, 123, 147, 156, 28], then fix the embedding and train the graph learning model only. We defer this discussion to Appendix A.2.3.

2.3.3 Experiments

Node Classification

Task and Datasets. The goal of this task is to classify each node into a correct category. We use transductive datasets: Cora, Citeseer and Pubmed [151], which are standard citation networks benchmarks, following the standard splits adopted in [92]. We also include datasets adopted in HGCN [28] for comparison: disease propagation tree and Airport. The former contains tree networks simulating the SIR disease spreading model [9], while the latter contains airline routes between airports from OpenFlights. To measure scalability, we supplement our experiment by predicting community structure on a large inductive dataset Reddit following [183]. More experimental details are provided in Appendix A.2.4.

Experiment Setup. Since all datasets contain node features, we choose to embed features most of the time, since it applies to both small and large graphs,

and transductive and inductive tasks. The only exception is the Airport dataset, which contains only 4 dimensional features—here, we use HyLa/RFF after embedding the graph nodes to produce better features $\overline{\mathbf{X}}$. We then use SGC model as softmax($\mathbf{A}^K\overline{\mathbf{X}}\mathbf{W}$), where both $\mathbf{W}$ and $\mathbf{Z}$ are jointly learned. Further training details are provided in Appendix A.2.4, our code is available at github[5]

Baselines. On Disease, Airport, Pubmed, Citeseer and Cora dataset, we compare our HyLa/RFF-SGC model against both Euclidean models (GCN, SGC and GAT) and Hypebolic models (HGCN, LGCN and HNN) using their publicly released version in Table 2.4, where all hyperbolic models adopt a *16-dimensional* hyperbolic space for consistency and a fair comparison. For the largest Reddit dataset, a 50-dimensional hyperbolic space is used. We also compare against the reported performance of supervised and unsupervised variants of GraphSAGE and FastGCN in Table 2.4. Note that GCN-based models (e.g., HGCN, LGCN) could not be trained on Reddit because its adjacency matrix is too large to fit in memory, unless a sampling way is used for training. Worthy to mention, despite of the fact that standard Euclidean GCN literature [92, 183] train the model for *100* epochs on the node classification task, most hyperbolic (graph) networks including HGCN, LGCN, GIL[6] report results of training for (5-)thousand epochs with early stopping. For a fair comparison, we train all models for a maximum of *100* epochs with early stopping, except from HGCN[7], whose results were taken from the original paper trained for 5,000 epochs.

[5] `https://github.com/ydtydr/HyLa.git`

[6] We cannot replicate results of GIL from their public code.

[7] HGCN requires pretraining embeddings from a link prediction task to achieve reported results on node classification task for Pubmed and Cora.

Table 2.4: Test accuracy/Micro F1 Score (%) averaged over 10 runs on node classification task. Performance of some baselines are taken from their original papers. **OOM:** Out of memory.

Dataset	Disease	Airport	Pubmed	Citeseer	Cora
Hyperbolicity δ	0	1.0	3.5	5.0	11
GCN	69.7 ± 0.4	81.4 ± 0.6	78.1 ± 0.2	70.5 ± 0.8	81.3 ± 0.3
SGC	69.5 ± 0.2	80.6 ± 0.1	78.9 ± 0.0	71.9 ± 0.1	81.0 ± 0.0
GAT	70.4 ± 0.4	81.5 ± 0.3	79.0 ± 0.3	72.5 ± 0.7	**83.0** ± 0.7
HGCN	74.5 ± 0.9	90.6 ± 0.2	**80.3** ± 0.3	64.0 ± 0.6	79.9 ± 0.2
LGCN	81.3 ± 4.0	57.6 ± 0.7	77.8 ± 0.7	65.9 ± 0.8	78.0 ± 0.6
HNN	41.0 ± 1.8	80.5 ± 0.5	69.8 ± 0.4	52.0 ± 1.0	54.6 ± 0.4
RFF-SGC	83.4 ± 1.9	94.8 ± 0.8	77.6 ± 0.1	71.3 ± 0.6	81.6 ± 0.4
HyLa-SGC	**86.8** ± 2.1	**95.2** ± 0.5	**80.3** ± 0.9	**72.6** ± 1.0	82.5 ± 0.5

Model	Test F1
GCN	**OOM**
HGCN	**OOM**
LGCN	**OOM**
SAGE-mean	95.0
SAGE-GCN	93.0
FastGCN	93.7
SGC	94.9
RFF-SGC	93.9
HyLa-SGC	94.5

Analysis. Our feature-extracted architecture is particularly strong and expressive to encode geometric priors for graph learning from Table 2.4. Together with SGC, HyLa-SGC outperforms state-of-the-art hyperbolic models on nearly all datasets. In particular, HyLa-SGC beats not only Euclidean SGC/GCN, but also an attention model GAT, except on the Cora dataset with a comparable performance. For the tree network disease with lowest hyperbolicity (more hyperbolic), the improvements of HyLa-SGC over HGCN is about 12.3%! The results suggest that the more hyperbolic the graph is, the more improvements will be gained with HyLa. On Reddit, HyLa-SGC outperforms sampling-based GCN variants, SAGE-GCN and FastGCN by more than 1%. However, the performance is close to SGC, which may indicate that the extra weights and nonlinearities are unnecessary for this particular dataset. Notably, RFF-SGC, embedding into Euclidean space and using RFF, sometimes can be better than GCN/SGC, while HyLa-SGC is consistently bettter than RFF-SGC.

Visualization. In Figure 2.14, we visualize the learned node Euclidean features using HyLa on Cora, Airport and Disease datasets with t-SNE [170] and PCA projection. This shows that HyLa achieves great label class separation (in-

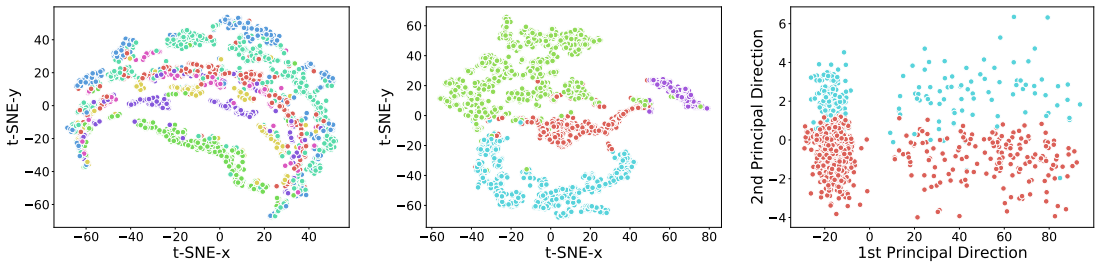


Figure 2.14: Visualization of node HyLa features on Cora, Airport and Disease datasets, where nodes of different classes are indicated by different colors. (left) t-SNE on Cora; (middle) t-SNE on Airport; (right) PCA on Disease.

dicated by different colors).

Text Classification

Task and Datsets. We further evaluate HyLa on **transductive** and **inductive** text classification task to assign documents labels. We conducted experiments on 4 standard benchmarks including R52 and R8 of Reuters 21578 dataset, Ohsumed and Movie Review (MR) follows the same data split as [193, 183]. Detailed dataset statistics are provided in Table A.3 in Appendix A.2.4.

Experiment Setup. In the transductive case, previous work [193] and [183] apply GCN and SGC by creating a corpus-level graph where both documents and words are treated as nodes in the graph. For weights and connections in the graph, word-word edge weights are calculated as pointwise mutual information (PMI) and word-document edge weights as normalized TF-IDF scores. The weights of document-document edges are unknown and left as 0. We follows the same data processing setup for the transductive setting, and embed the whole graph since only the adjacent matrix is available. In the inductive setting, we take the sub-matrix of the large matrix in the transductive setting, includ-

Table 2.5: Test accuracy (%) averaged over 10 runs on transductive and inductive text classification task except from the LR mode. **Bold** numbers: best in both transductive and inductive setting; Underlined numbers: best in inductive setting.

Setting	Methods	R8	R52	Ohsumed	MR
Transductive	TextGCN	97.1 ± 0.1	93.5 ± 0.2	68.4 ± 0.6	**76.7** ± 0.2
	TextSGC	97.2 ± 0.1	94.0 ± 0.2	**68.5** ± 0.3	75.9 ± 0.3
	RFF-SGC	96.5 ± 0.3	94.0 ± 0.5	67.2 ± 0.4	73.1 ± 0.4
	HyLa-SGC	96.9 ± 0.4	**94.1** ± 0.3	67.3 ± 0.5	76.2 ± 0.3
Inductive	TextGCN	95.8 ± 0.3	88.2 ± 0.7	57.7 ± 0.4	74.8 ± 0.3
	LR	93.3	85.6	56.6	73.0
	HyLa-LR	**97.4** ± 0.2	93.5 ± 0.2	64.9 ± 0.3	75.5 ± 0.3
	RFF-LR	97.0 ± 0.4	92.2 ± 0.2	61.6 ± 0.3	**76.0** ± 0.3

ing only the document-word edges as the node representation feature matrix $\mathbf{X}$, then follow the procedure in Section 2.3.2 to embed features and apply HyLa/RFF to get $\overline{\mathbf{X}}$. Since the adjacency matrix of documents is unknown, we replace SGC with a logistic regression (LR) formalized as $\mathbf{Y} = \text{softmax}(\overline{\mathbf{X}}\mathbf{W})$. We train all models for a maximum of 200 epochs and compare it against TextSGC and TextGCN in Table 2.5.

Performance Analysis. In the transductive setting, HyLa-based models can match the performance of TextGCN and TextSGC. The corpus-level graph may contain sufficient information to learn the task, and hence HyLa-SGC does not seem to outperform baselines, but still has a comparable performance. HyLa shows extraordinary performance in the inductive setting, where less information is used compared to the (transductive) node level case, i.e. a submatrix of corpus-level graph. With a linear regression model, it can already outperform inductive TextGCN, sometimes even better than the performance of a transductive TextGCN model, which indicates that there is indeed redundant informa-

tion in the corpus-level graph. From the results on the inductive text classification task, we argue that HyLa (with features embedding) is particularly useful in the following three ways. First, it can solve the OOM problem of classic GCN model in large graphs, and requires less memory during training, since there are limited number of lower level features (also $\mathbf{X}$ is usually sparse), and there is no need to build a corpus-level graph anymore. Second, it is naturally inductive as HyLa is built at feature level (for each word in this task), it generalizes to any unseen new nodes (documents) that uses the same set of words. Third, the model is simple: HyLa follows by a linear regression model, which computes faster than classical GCN models.

Efficiency. Following [183], we measure the training time of HyLa-based models on the Pubmed dataset, and we compare against both Euclidean models (SGC, GCN, GAT) and Hyperbolic models (HGCN, HNN). In Figure 2.15, we plot the timing performance of various models, taking into account the pre-computation time of the models into training time. We measure the training time on a NVIDIA GeForce RTX 2080 Ti GPU and show the specific timing statistics in Appendix A.2.4. HyLa-based models achieve the best performance while incurring a minor computational slowdown, which is 4.4× faster than HGCN.

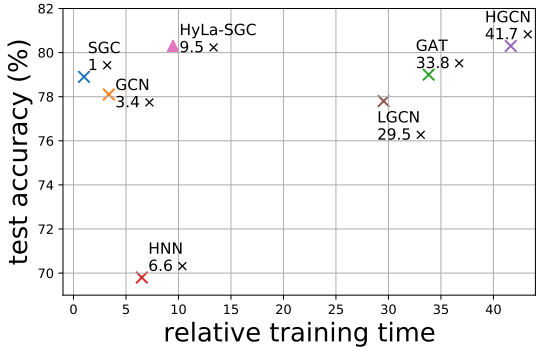


Figure 2.15: Performance over training time on Pubmed. HyLa-SGC achieves best performance with minor computation slowdown.

2.4 Coneheads: Hierarchy Aware Attention

Overview

Beyond the usage of hyperbolic embeddings for graph learning in section 2.3, we further extend the potential of hierarchy-aware hyperbolic embedding to improve the parameter efficiency of attention models, such as graph networks and transformers, with the shadow cones framework in section 2.2.

In recent years, attention networks have achieved highly competitive performance in a variety of settings, often outperforming highly-engineered deep neural networks [24, 54, 171]. The majority of these networks use dot product attention, which defines the similarity between two points $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$ by their inner product $\boldsymbol{u}^\top \boldsymbol{v}$ [171]. Although dot product attention empirically performs well, it also suffers from drawbacks that limit its ability to scale to and capture complex relationships in large datasets [180, 161]. The most well known of these issues is the quadratic time and memory cost of computing pairwise attention. While many works on attention mechanisms have focused on reducing the computational cost of dot product attention, few have considered the properties of the dot product operator itself [180, 44].

Many real world datasets exhibit complex structural patterns and relationships which may not be well captured by an inner product [168, 110]. For example, NLP tasks often contain hierarchies over tokens, and images may contain clusters over pixels [192, 74]. Motivated by this, we propose a new framework based on hyperbolic entailment cones to compute attention between sets of points [60, 203]. Our attention mechanism, which we dub "cone attention",

utilizes partial orderings defined by hyperbolic cones to better model hierarchical relationships between data points. More specifically, we associate two points by the depth of their lowest common ancestor (LCA) in the cone partial ordering, which is analogous to finding their LCA in a latent tree and captures how divergent two points are.

Cone attention effectively relies on two components: hyperbolic embeddings and entailment cones. Hyperbolic embeddings, which use the underlying geometric properties of hyperbolic space, give low-distortion embeddings of hierarchies that are not possible with Euclidean embeddings [147]. Entailment cones, which rely on geometric cones to define partial orders between points, allow us to calculate explicit relationships between points, such as their LCA [60, 203]. To the best of our knowledge, we are the first to define a hierarchy-aware attention operator with hyperbolic entailment cones.

Functionally, cone attention is a drop-in replacement for dot product attention. We test cone attention in both "classical" attention networks and transformers, and empirically show that cone attention consistently improves end task performance across a variety of NLP, vision, and graph prediction tasks. Furthermore, we are able to match dot product attention with significantly fewer embedding dimensions, resulting in smaller models. To summarize, our contributions are:

- We propose cone attention, a hierarchy aware attention operator that uses the lowest common ancestor of points in the partial ordering defined by hyperbolic entailment cones.
- We evaluate cone attention on NLP, vision, and graph prediction tasks, and show that it consistently outperforms dot product attention and other

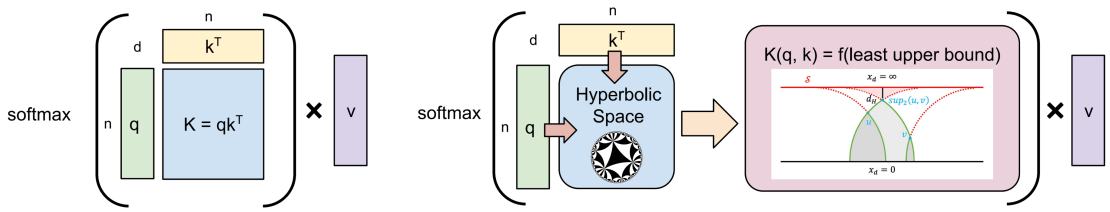


Figure 2.16: Overview of cone attention vs. dot product attention. In dot product attention (left), similarity scores are calculated with $K = qk^\top$. In cone attention (right), q and k are first projected onto hyperbolic space. Then, pairwise similarity is calculated from the lowest common ancestor of points in the partial ordering defined by entailment cones. Cone attention allows us to explicitly encode notions of hierarchy in attention, and empirically gives better performance than dot product attention.

baselines. For example, we achieve +1 BLEU and +1% ImageNet Top-1 Accuracy on the `transformer_iwslt_de_en` and DeiT-Ti models, respectively.

- We test cone attention at low embedding dimensions and show that we can significantly reduce model size while maintaining performance relative to dot product attention. With cone attention, we can use 21% fewer parameters for the IWSLT'14 De-En NMT task.

2.4.1 Background

In this section, we provide background on attention mechanisms and describe our choice of entailment cones to encode partial orderings.

Attention

The attention operator has gained significant popularity as a way to model interactions between sets of tokens [171]. At its core, the attention operator A

performs a "lookup" between a single query $q_i \in \mathbb{R}^d$ and a set of keys $k \in \mathbb{R}^{n \times d}$, and aggregates values $v \in \mathbb{R}^{n \times d}$ associated with the keys to "read out" a single value for q_i. Mathematically, this can be represented as

$$A(q_i, k) = C \sum_j \Big(K(q_i, k_j) v_j \Big),$$

where $C \sum_j K(q_i, k_j) = 1$. In "traditional" dot product attention, which has generally superseded "older" attention methods such as Additive Attention and Multiplicative Attention [13, 109],

$$K(q_i, k_j) = \exp\left(\frac{q_i k_j}{\sqrt{d}}\right) \qquad C = \frac{1}{\sum_j K(q_i, k_j)} \qquad A(q_i, k) = \text{softmax}\left(\frac{q_i k^\top}{\sqrt{d}}\right) v,$$

similarity is scored with a combination of cosine similarity and embedding magnitudes.

Existing works have proposed replacing dot product attention to various degrees. A large body of works focus on efficiently computing dot product attention, such as with Random Fourier Features and low-rank methods [132, 180]. These methods generally perform worse than dot product attention, as they are approximations [212]. Some recent works, such as EVA attention, parameterize these approximations and effectively get a larger class of dot-product-esque attention methods [212]. Beyond this, [167] replace K with compositions of classical kernels. Others extend dot product attention, such as by controlling the "width" of attention with a learned Gaussian distribution or by using an exponential moving average on inputs, although extensions do not usually depend on K [69, 110]. Closer to our work, [67] introduced hyperbolic distance attention, which defines $K(q_i, k_i) = \exp(-\beta d_{\mathbb{H}}(q_i, k_i) - c)$ where $d_{\mathbb{H}}$ is the hyperbolic distance, $\beta \in \mathbb{R}^+$, and $c \in \mathbb{R}$. K can be interpreted as an analog of the distance from q_i to k_i on a latent tree. Finally, in an orthogonal direction, [160] ignore token-token interactions and synthesize attention maps directly from random

alignment matrices. Whether token-token interactions are actually needed is outside the scope of this work, and we compare cone attention accordingly.

Entailment Cones

Entailment cones in hyperbolic space were first introduced by [60] to embed partial orders. The general concept of Ganea's entailment cones is to capture partial orders between points with membership relations between points and geodesically convex cones rooted at said points. That is, if $u \in$ the cone of v, then $v \prec u$. Ganea's cones (Figure 2.17 right) are defined on the Poincaré ball by a radial angle function $\psi(r)$, with an ϵ-ball around the origin where cones are undefined [8, 60]. This makes learning complicated models with Ganea's cones difficult, as optimization on the Poincaré ball is nontrivial and the ϵ-ball negatively impacts embedding initializations [203, 60].

In this work, we instead use the shadow cone construction introduced in section 2.3 and operate on the Poincaré half-space, which makes computing our desired attention function numerically simpler. Shadow cones are defined by shadows cast by points and a single light source $\mathcal{S}$, and consist of the penumbral and umbral settings (Figure 2.17 left quadrant). In the penumbral setting, $\mathcal{S}$ is a ball of fixed radius and points are points. The shadow and cone of u are both the region enclosed by geodesics through u tangent to $\mathcal{S}$. In the umbral setting, $\mathcal{S}$ is instead a point, and points are centers of balls of fixed radius. Here, the shadow of u is the region enclosed by geodesics tangent to the ball around u that intersect at $\mathcal{S}$. However, to preserve transitivity, the cone of u is a subset of the shadow of u (see Figure 2.17). The shadow cone formulation can also be achieved with subset relations between shadows instead of membership

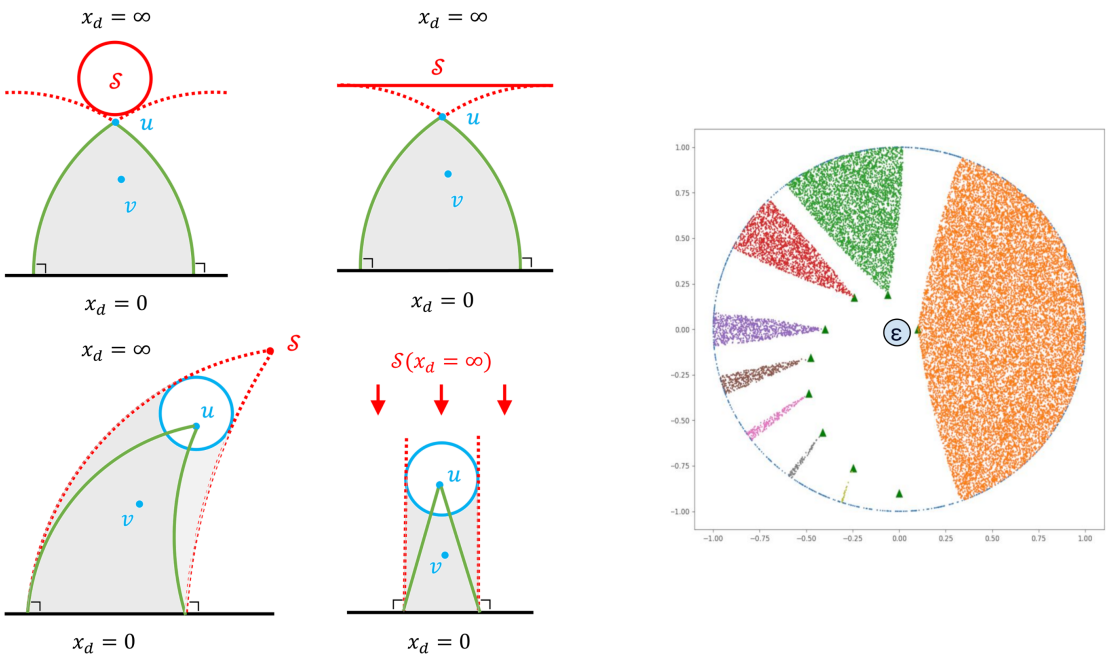


Figure 2.17: (Left Quadrant) Clockwise from top left: finite setting penumbral cone, infinite setting penumbral cone, infinite setting umbral cone, and finite setting umbral cone. All figures are in H^2. Shadows are represented by shaded regions, and cones are enclosed in green. In all figures, $u \prec v$ since v is in the cone of u. (Right) Ganea's entailment cones, as defined on the Poincaré ball. Note the ϵ-ball where cones are not defined, which makes optimization nontrivial. Figure from [60].

relations between points and cones, which may be conceptually clearer.

Infinite-setting Shadow Cones. For penumbral cones, when $\mathcal{S}$ is a ball with center at $x_d = \infty$, $\mathcal{S}$'s boundary is a horosphere of user-defined height h. Here, all shadows are defined by intersections of Euclidean semicircles of Euclidean radius h. Notably, the infinite setting penumbral cone construction is similar to Ganea's cones under an isometry from the Poincaré half-space to the Poincaré ball where $\mathcal{S}$ maps to the ϵ-ball [203]. For umbral cones, when $\mathcal{S}$ is $x_d = \infty$, shadows are regions bounded by Euclidean lines perpendicular to the x-axis.

Unlike Ganea's cones and penumbral cones, umbral cones are not geodesi-

cally convex [203]. That is, the shortest path between two points in an umbral cone may not necessarily lie in the cone. In a tree, this corresponds to the shortest path between two nodes not being in the subtree of their LCA, which is not possible. Empirically, while still better than dot product attention, umbral attention usually performs worse than penumbral attention.

Lowest Common Ancestor (LCA) and Least Upper Bound

The LCA of two nodes u, v in a directed acyclic graph is the lowest (deepest in hierarchy) node that is an ancestor of both u and v. Although the two terms are similar, the LCA is *not* the same as the least upper bound of a partial ordering. The least upper bound of two points x, y in a partial ordering, denoted $\sup(x, y)$, is the point p such that $p \preceq x, y$ and $\forall q$ where $q \preceq x, y$, $q \preceq p$. The key difference is that all other upper bounds must precede the least upper bound, while not all ancestors of u and v must also be ancestors of LCA(u, v). Furthermore, p may not actually exist.

2.4.2 Hierarchy Aware Attention

Here, we describe cone attention using the shadow cone construction, and discuss projection functions onto $\mathbb{H}^d$ that allow us to use cone attention within attention networks. All definitions use the infinite-setting shadow cones, and proofs and derivations are in the appendix. For clarity, we refer to Ganea's entailment cones as "Ganea's cones" and the general set of cones that captures entailment relations (e.g. Ganea's cones and shadow cones) as "entailment cones" [60, 203]. Our methods are agnostic entailment cone choice, and can also be

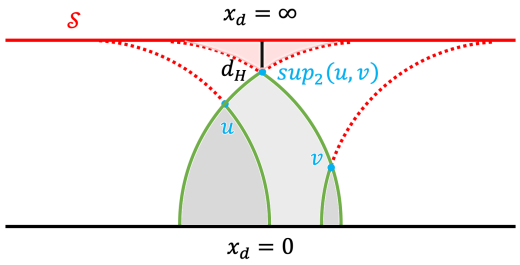


Figure 2.18: In this example using penumbral cones in H^d, $\sup_2(u, v)$ is the lowest common ancestor of u and v. The red region is the set of points P s.t. $p \in P \preceq u, v$. Of these, $\sup_2(u, v)$ is the lowest point whose cone (light gray) contains both u and v. Here, $\mathcal{S}$ is the root of all hierarchies, and points closer to $x_d = 0$ are closer to the "leaves" of hierarchies. If $d = 2$, then $\sup_2(u, v)$ is also the least upper bound of u and v in the partial ordering defined by entailment cones.

used with Ganea's cones.

Cone Attention

We wish to associate $u, v \in \mathsf{H}^d$ by their LCA in some latent tree T, which is analogous to finding their LCA, denoted $\sup_2(u, v)$, in the partial ordering defined by entailment cones. Formally,

$$\sup_2(u, v) = r\left(\arg\max_{C: u, v \in C} d_H(\mathcal{S}, r(C))\right)$$

where $r(C)$ denotes the root of a cone C. This corresponds to finding the cone that is farthest away from $\mathcal{S}$, which is the root of all hierarchies in the shadow cones construction. When $d = 2$, $\sup_2(u, v)$ also has the nice property of being the least upper bound $\sup(u, v)$ in the partial ordering defined by shadow cones. Then, we define the similarity between u and v as

$$K(u, v) = f(d_{\mathbb{H}}(\sup_2(u, v), \mathcal{S}))$$

where f is a user-defined monotonically increasing function. If $K(u, v) > K(u, w)$, then $d_{\mathbb{H}}(\sup_2(u, v), \mathcal{S}) > d_{\mathbb{H}}(\sup_2(u, w), \mathcal{S})$, which implies that u and v have a more

recent "ancestor" than u and w. Thus, K gives a higher similarity score to points who have a more recent "ancestor" in T. In the infinite-setting shadow cone construction, $\sup_2(u, v)$ is root of the minimum height (literal lowest) cone that contains both u and v, or $\sup_2(u, v) = r\left(\arg\min_{C:u,v\in C} r(C)_d\right)$.

Using this, we provide definitions for $K(u, v)$ in the infinite-setting shadow cone construction. Both the umbral and penumbral definitions correspond to the Euclidean height of $\sup_2(u, v)$. In the penumbral setting, when $\sup_2(u, v)$ does not exist, we return the Euclidean height of lowest light source where $\sup_2(u, v)$ exists. x_d denotes the last dimension of x, and $x_{:-1}$ denotes the first $d-1$ dimensions of x. $\gamma \in \mathbb{R}^+$ corresponds to the softmax "temperature" in attention. In the penumbral setting, $r \in \mathbb{R}^+$ is the user-defined height of the horosphere light source. In the umbral setting, $r \in \mathbb{R}^+$ is the user-defined radius of the ball centered at each point.

Definition 2.4.1. ***Penumbral Attention:***

$$K(u, v) = \exp\left(-\gamma \max\left(u_d, v_d, \sqrt{r^2 - \left(\frac{\sqrt{r^2 - u_d^2} + \sqrt{r^2 - v_d^2} - \|u_{:-1} - v_{:-1}\|}{2}\right)^2}\right)\right)$$

when there exists a cone that contains u and v, and

$$K(u, v) = \exp\left(-\gamma \sqrt{\left(\frac{\|u_{:-1} - v_{:-1}\|^2 + u_d^2 - v_d^2}{2\|u_{:-1} - v_{:-1}\|}\right)^2 + v_d^2}\right)$$

otherwise. There exists a cone that contains u and v when

$$\left(\|u_{:-1} - v_{:-1}\| - \sqrt{r^2 - u_d^2}\right)^2 + v_d^2 < r^2$$

Definition 2.4.2. ***Umbral Attention:***

$$K(u, v) = \exp\left(-\gamma \max\left(u_d, v_d, \frac{\|u_{:-1} - v_{:-1}\|}{2\sinh(r)} + \frac{u_d + v_d}{2}\right)\right)$$

These definitions possess a rather interesting property – when $u_d = v_d$ and K is normalized across a set of vs, cone attention reduces to the Laplacian kernel $K(u_{:-1}, v_{:-1}) = \exp(-\gamma\|u_{:-1} - v_{:-1}\|)$. Since Euclidean space is isomorphic to a horosphere, and u and v are on the same horosphere if $u_d = v_d$, cone attention can also be seen as an extension of the Laplacian kernel [124].

Cone attention and dot product attention both take $O(n^2 d)$ time to compute pairwise attention between two sets of n d-dimensional tokens $q, k \in \mathbb{R}^{n \times d}$ [180]. However, cone attention takes more operations than dot product attention, which computes $qk^\top$. In transformers, our PyTorch cone attention implementations with `torch.compile` were empirically 10-20% slower than dot product attention with `torch.bmm` (cuBLAS)[129] (see section A.3.3). `torch.compile` is not optimal, and a raw CUDA implementation of cone attention would likely be faster and narrow the speed gap between the two methods [129].

Mapping Functions

Here, we discuss mappings from Euclidean space to the Poincaré half-space. These mappings allow us to use cone attention within larger models. The canonical map in manifold learning is the exponential map $\text{Exp}_x(v)$, which maps a vector v from the tangent space of a manifold $\mathcal{M}$ at x onto $\mathcal{M}$ by following the geodesic corresponding to v [134, 8]. In the Poincaré half-space [200],

$$\text{Exp}_x(v)_{:-1} = x_{:-1} + \frac{x_d}{\|v\|/\tanh(\|v\|) - v_d} v_{:-1} \qquad \text{Exp}_x(v)_d = \frac{x_d}{\cosh(\|v\|) - v_d \sinh(\|v\|)/\|v\|}$$

While $\text{Exp}_x(v)$ is geometrically well motivated, it suffers from numerical instabilities when $\|v\|$ is very large or small. These instabilities make using the exponential map in complicated models such as transformers rather difficult.

Using `fp64` reduces the risk of numerical over/underflows, but `fp64` significantly reduces performance on GPUs, which is highly undesirable [35].

Instead, we use maps of the form $(x_{:-1}, x_d) \to (x_{:-1} f(x_d), f(x_d))$ that preserve the exponential volume of hyperbolic space while offering better numerics for large-scale optimization. Our use of alternatives to $\text{Exp}_x(v)$ follows [67]'s psuedopolar map onto the Hyperboloid. Geometrically, since n-dimensional Euclidean space is isomorphic to a horosphere in $\mathbb{H}^{n+1}$, these maps corresond to selecting a horosphere with $f(x_d)$ and then projecting $x_{:-1}$ onto that horosphere [124]. To achieve exponential space as $x_d \to -\infty$, we use functions f of the form $\exp(\cdot)$.

For the infinite-setting umbral construction, since there is no restriction on where points can go in the Poincaré half-space, we map $x \in \mathbb{R}^d$ to H^d with $\psi : \mathbb{R}^d \to \mathsf{H}^d$:

$$\psi(x)_{:-1} = x_{:-1} \exp(x_d) \qquad \psi(x)_d = \exp(x_d)$$

For the infinite-setting penumbral construction, since the mapped point cannot enter the light source at height h, we map $x \in \mathbb{R}^d$ to area below the light source with $\xi : \mathbb{R}^d \to \mathsf{H}^d$

$$\xi(x)_{:-1} = x_{:-1} \frac{h}{1 + \exp(-x)} \qquad \xi(x)_d = \frac{h}{1 + \exp(-x)}$$

While ξ is structurally the sigmoid operation, note that $\text{sigmoid}(x) = \exp(-\text{softplus}(-x))$. Since $\text{softplus}(x) \approx x$ for large values of x, ξ preserves the exponential volume properties we seek.

2.4.3 Experiments

Here, we present an empirical evaluation of cone attention in various attention networks. For each model we test, our experimental procedure consists of changing K in attention and training a new model from scratch. Unless otherwise noted in the appendix, we use the code and training scripts that the authors of each original model released. We assume released hyperparameters are tuned for dot product attention, as these models were state-of-the-art (SOTA) when new.

Baselines. Our main baseline is dot product attention, as it is the most commonly used form of attention in modern attention networks. Additionally, we compare cone attention against [67]'s hyperbolic distance attention and the Laplacian kernel. To the best of our knowledge, few other works have studied direct replacements of the dot product for attention.

[67]'s original hyperbolic distance attention formulation not only computed the similarity matrix K in the Hyperboloid, but also aggregated values v in hyperbolic space by taking the Einstein midpoint with respect to weights α in the Klein model (see appendix for definitions) [134]. However, the Einstein midpoint, defined as

$$m(\alpha, v) = \sum_i \left(\frac{\alpha_i \gamma(v_i)}{\sum_j \alpha_j \gamma(v_j)} \right)$$

where $\gamma(v) = 1/\sqrt{1 - \|v\|^2}$, does not actually depend on how α is computed. That is, α could be computed with Euclidean dot product attention or cone attention and $m(\alpha, v)$ would still be valid. The focus of our work is on computing similarity scores, so we do not use hyperbolic aggregation in our experiments. We test hyperbolic distance attention using both [67]'s original pseudopolar projection

onto the Hyperboloid model and with our ψ map onto the Poincaré half-space.

Models. We use the following models to test cone attention and the various baselines. These models span graph prediction, NLP, and vision tasks, and range in size from a few thousand to almost 250 million parameters. While we would have liked to test larger models, our compute infrastructure limited us from feasibly training billion-parameter models from scratch.

Graph Attention Networks. Graph attention networks (GATs) were first introduced by [172] for graph prediction tasks. GATs use self-attention layers to compute node-level attention maps over neighboring nodes. The original GAT used a concatenation-based attention mechanism and achieved SOTA performance on multiple transductive and inductive graph prediction tasks [172]. We test GATs on the transductive Cora and inductive multi-graph PPI datasets [112, 71].

Neural Machine Translation (NMT) Transformers. Transformers were first applied to NMT in [171]'s seminal transformer paper. We use the fairseq `transformer_iwslt_de_en` architecture to train a German to English translation model on the IWSLT'14 De-En dataset [126, 56]. This architecture contains 39.5 million parameters and achieves near-SOTA performance on the IWSLT'14 De-En task for vanilla transformers [126]. As this model is the fastest transformer to train out of the tested models, we use it for ablations.

Vision Transformers. Vision transformers (ViT) use transformer-like architectures to perform image classification [54]. In a ViT, image patches are used as a tokens in a transformer encoder to classify the image. We use the *Data Efficient* Vision Transformer (DeiT) model proposed by FAIR, which uses a student-

teacher setup to improve the data efficiency of ViTs [165]. DeiTs and ViTs share the same architecture, and the only differences are how they are trained and the distillation token. We train DeiT-Ti models with 5 million parameters on the ImageNet-1K dataset for 300 epochs[165, 49]. We also train cone and dot product attention for 500 epochs, as we observed that training for more iterations improves performance.

Adaptive Input Representations for Transformers. Adaptive input representations were introduced by Baevski and Auli for transformers in language modeling tasks [12]. In adaptive inputs, rarer tokens use lower dimensional embeddings, which serves as a form of regularization. We use the fairseq `transformer_lm_wiki103` architecture (246.9M parameters) and train models on the WikiText-103 language modeling dataset with a block size of 512 tokens [126, 113]. We also test the same architecture without adaptive inputs, which has 520M parameters. This version converges in significantly fewer iterations, allowing us to train such a large model.

Diffusion Transformers. Diffusion transformers (DiT) are diffusion models that replace the U-Net backbone with a transformer [131]. DiTs operate on latent space patches, so we expect there to be less hierarchical information vs. taking image patches. We use DiTs to test cone attention when the data is less hierarchical. We train DiT-B/4 models with 130M parameters on ImageNet-1K [131, 49].

Table 2.6: Performance of various attention methods across models and tasks. * indicates the model failed to converge or ran into NaN errors. ↑ indicates higher is better, and ↓ indicates lower is better. Cone attention methods (†) generally outperform dot product attention and other baselines. Model default refers to a model's default attention method, which is dot product attention except for GATs.

Method	NMT IWLST (BLEU ↑)	DeiT-Ti Imagenet Top-1 / 5 (Acc. ↑) 300 Epochs	DeiT-Ti Imagenet Top-1 / 5 (Acc. ↑) 500 Epochs	GAT Cora / PPI (Acc. ↑)
Model Default	34.56	72.05 / 91.17	73.65 / 91.99	0.831 / 0.977
Dot Product	34.56	72.05 / 91.17	73.65 / 91.99	0.834 / 0.985
Penumbral†	**35.56**	72.67 / 91.12	74.34 / 92.38	0.835 / **0.990**
Umbral†	35.07	**73.14 / 91.82**	**74.46 / 92.54**	**0.836** / 0.989
$d_{\mathbb{H}}$ H^n ξ	32.54	49.19* / 74.68*	–	0.834 / 0.987
$d_{\mathbb{H}}$ Hyperboloid	33.80	0.10* / 0.45*	–	0.13* / 0.989
Laplacian Kernel	34.68	71.25 / 90.84	–	0.823 / 0.986

Method	Adaptive Inputs WikiText-103 0 / 480 Context Window (Ppl. ↓)	Without Adaptive Inputs	DiT-B/4 @ 400K Steps (FID-50K ↓)
Dot Product	20.86 / 19.22	26.62 / 24.73	68.9
Penumbral	**20.72 / 19.01**	**26.44 / 24.31**	67.7
Umbral	21.16 / 19.59	27.82 / 26.73	**67.6**

Results

Table 2.6 summarizes the performance of cone attention across various models and tasks. Both penumbral and umbral attention significantly outperform baselines on the NMT IWSLT and DeiT-Ti Imagenet tasks. For DeiT-Ti, umbral attention achieves 73.14% top-1 and 91.82% top-5 accuracy at 300 epochs and 74.46% top-1 and 92.54% top-5 accuracy at 500 epochs, which matches a distilled DeiT-Ti with more parameters (74.5% / 91.9%) [165]. On the GAT tasks, cone attention again outperforms baselines and achieves Graph Convolutional Network-level performance on the PPI dataset. Interestingly, almost all baselines, including dot product attention, outperform the concatenation-based attention in the original GAT paper. The two GAT tasks also reveal some interesting differences between penumbral and umbral attention. The Cora citation

dataset is more tree-like with an average of 2 edges per node and chronological dependencies between nodes (a paper cannot cite a newer paper), while the PPI dataset is closer to a clustered graph with 14.3 edges per node [172, 71, 112]. As noted in [203], umbral cones appear to be better for strict hierarchies, while penumbral cones capture more complicated relationships such as those in the PPI dataset.

The adaptive inputs method regularizes models by reducing d for rare words. We expect rarer words to be closer to the leaves of a word hierarchy, so the adaptive inputs method also acts as a hierarchical prior on the data [192]. We use adaptive inputs to test cone attention when combined with other hierarchical priors. Penumbral attention outperforms dot product attention with or without adaptive inputs, but the gap between the two methods is larger without adaptive inputs. On the other hand, umbral attention performs worse than dot product attention and penumbral attention on the WikiText103 task, with or without adaptive inputs. Since umbral cones are not geodesically convex, which means that the shortest path between two points in a cone may not necessarily lie entirely in that cone, we suspect that convexity is important for the WikiText-103 language modeling task. In the DiT model, patches are taken at the latent space-level, which we suspect gives less hierarchical information than when patches are taken from an input image [131]. Here, cone attention still outperforms dot product attention, but to a lesser degree. This mirrors our expectation that the DiT model does not benefit as much from hierarchical attention, and verifies that in such cases, using cone attention does not hurt performance. Interestingly, umbral cones slightly outperform penumbral cones in the DiT-B/4 task, which may be a result of patches being over the latent space, and not the input image.

Table 2.7: Comparison of various mappings from Euclidean space to hyperbolic models for the NMT IWSLT task (BLEU scores, higher is better). The exponential map generally performs worse than ψ and the pseudopolar map. While ψ also performs worse than the psuedopolar map, table 2.6 indicates that the pseudopolar map is less numerically stable than ψ.

Method	$\text{Exp}_O(v) \to \mathsf{H}^d$	$\xi \to \mathsf{H}^d$	$\psi \to \mathsf{H}^d$	Pseudopolar $\to$ Hyperboloid
Penumbral	35.12	**35.56**	-	-
Umbral	34.63	-	**35.07**	-
$d_{\mathbb{H}}$	30.59	-	32.54	**33.80**

Effect of Mapping Functions

Table 2.7 compares how ψ and ξ compare to the exponential map and psueodpolar map (section 2.4.2) on the NMT IWSLT task. Here, we use the exponential map at the origin of H^d, $O = (0, 0, ..., 1)$. To compare $\text{Exp}_O(v)$ to ψ, we first take $\text{Exp}_O(v)$ and then project the point onto the boundary of L if $\text{Exp}_O(v)$ is in L. In the infinite penumbral setting, this corresponds to taking $\text{Exp}_O(v)_d = \min(\text{Exp}_O(v)_d, h)$. ψ and ξ generally perform much better than taking the exponential map at the origin O, which suggests that ψ and ξ have better optimization properties. For $d_{\mathbb{H}}$ attention, Gulcehre et al.'s pseudopolar map slightly outperforms ξ. However, table 2.6 indicates that outside of this specific task, using the psuedopolar map and Hyperboloid is less numerically stable.

Attention Efficiency and Model Size

Figure 2.19 shows the performance of cone attention vs. dot product attention at low *token* (q, k) embedding dimensions (d from before) on the NMT IWSLT task. Both umbral and penumbral attention are able to achieve significantly better performance than dot product attention in this regime, matching dot product

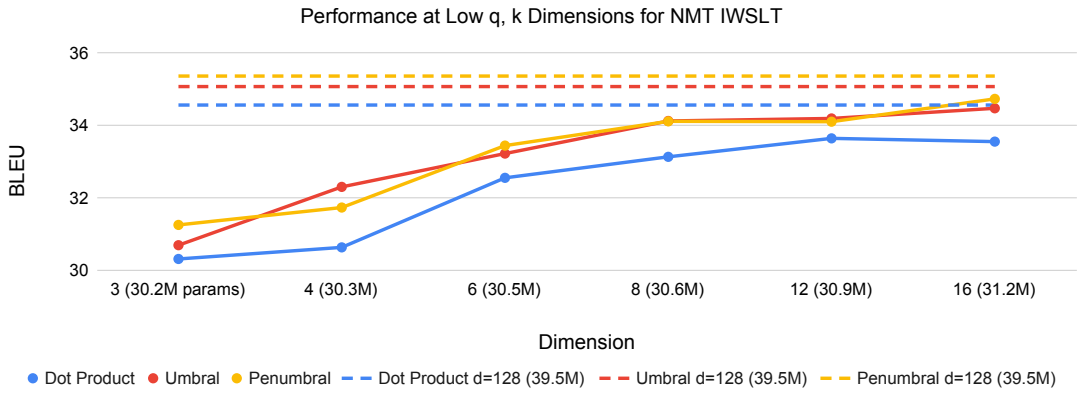


Figure 2.19: Performance of cone and dot product attention at low dimensions on NMT IWSLT. The base architecture uses 128 dimensions, and cone attention is able to match dot product attention with only 16 dimensions. For this model, this allows us to use 21% fewer parameters to reach performance parity, indicating that cone attention more efficiently captures hierarchical information.

Table 2.8: Performance of DeiT-Ti at 64 (default) and 16 dimensions, 300 epochs.

Method	d = 64	d = 16
Dot Product	72.05 / 91.17	71.29 / 90.54
Penumbral	72.67 / 91.12	**72.25 / 91.20**
Umbral	**73.14 / 91.82**	71.67 / 90.71

attention at $d = 128$ with only $d = 16$. For this model, using 16 dimensions reduces the number of parameters from 39.5M to 31.2M. Table 2.8 shows the performance of DeiT-Ti at $d = 16$. Here, penumbral attention at $d = 16$ is able to outperform dot product attention at $d = 64$, again indicating that cone attention is able to more efficiently capture hierarchies.

Sensitivity to Initialization

Hyperbolic embeddings are known to be sensitive to initialization [60, 203]. To test cone attention's sensitivity to initialization, we trained 5 seeds for the IWSLT De2En task for cone attention and dot product attention. Dot product had an

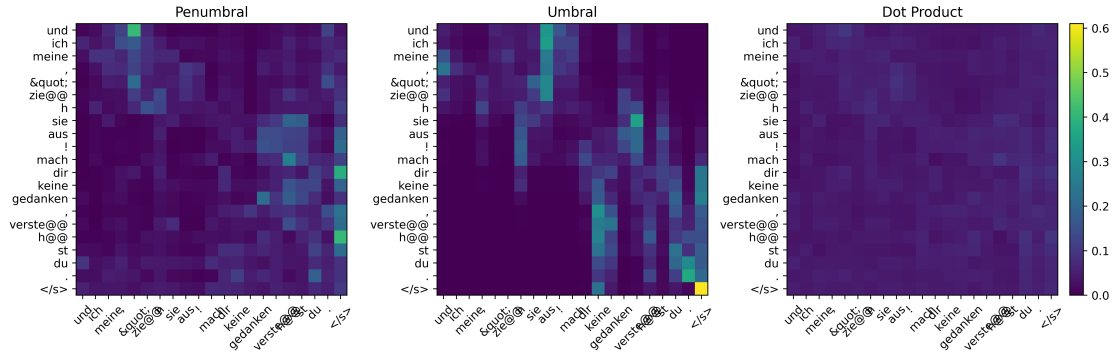


Figure 2.20: Attention heatmaps from an attention head in a trained IWSLT De2En translation model for the tokenized validation sequence *"Und ich meine, "Zieh sie aus! Mach dir keine gedanken, verstehst du.'*, which translates to *"I'm like, "Get it off! Don't worry about it, you know."* The cone attention heatmaps (left and center) have a clear distinction between the two parts of the sequence separated by "!", whereas the dot product heatmap (right) does not have a clear separation.

average BLEU score of 34.59 and standard deviation 0.12, penumbral achieved 35.41 ± 0.14, and umbral achieved 35.03 ± 0.30. There was one outlier in the umbral attention trials, and with the outlier removed umbral achieved 35.16 ± 0.09. The cone attention methods appear to have slightly higher variance than dot product attention, but not significantly so.

Attention Heatmaps

A natural question arises about what cone attention methods actually learn. Figure 2.20 shows heatmaps from an attention head in a trained IWSLT De2En translation model. The heatmaps for penumbral and umbral attention show clearer separation than the dot product attention heatmap. Furthermore, the separation in the two cone attention methods happens at the exclamation mark, a natural segmentation of the sequence into two parts. Intuitively, cone attention can be seen as an attention method that satisfies certain "logical con-

straints," such as "if $z \prec y$ and $y \prec x$, then $z \prec x$," which leads to relations between attention scores. For example. if $K(x, y)$ and $K(y, z)$ are both high, then $K(x, z)$ should also be relatively high in cone attention. In dot product attention, this is not guaranteed. If $x = (1, 1, 0, 0), y = (10, 10, 10, 10)$, and $z = (0, 0, 1, 1)$, then $\langle x, y \rangle = \langle y, z \rangle = 20$, but $\langle x, z \rangle = 0$. We suspect this is a reason why cone attention methods show better separation than dot product attention, which can aid performance.

CHAPTER 3

ROBUST AND EFFICIENT NUMERICAL COMPUTATIONS

3.1 Background

In the second part of my research, I focused on the practical model deployment. Particularly the impact of numerical precision on model performance and efficiency, within the realm of (low-precision) floating-point arithmetic – a cornerstone of virtually all modern computing systems.

The advantages of hyperbolic space sketched in Section 2 do not come for free. It's surprising that the numerical error of floating-point representations could drastically affect the performance of hyperbolic models, even with float32, which typically performs well for Euclidean models. The large volumes that are so beneficial from a embedding-capacity perspective create unavoidable numerical problems. Informally called "the NaN problem" [199], representing hyperbolic space with ordinary floating-point numbers can cause significant errors and lead to severe numerical instability that compounds throughout training, causing NaNs to pop out everywhere in the computations. Common models of hyperbolic space, such as the Poincaré ball model and the Lorentz hyperboloid model introduced in Section 1.2, all suffer from significant numerical errors. These errors originate in the ordinary floating-point representation, and are further amplified in subsequent computations by the very ill-conditioned Riemannian metrics involved in their construction.

To address this when embedding a graph, one technical solution exploited by [150] is to carefully scale down all the edge lengths by a factor before embed-

ding, then recover the original distances afterwards by dividing by the factor. However, this scaling increases the distortion of the embedding, and the distortion gets worse as the scale factor increases. [147] show that this sort of tradeoff is unavoidable when embedding in hyperbolic space: one can either choose to increase the number of bits used for the floating-point numbers or increase the dimension of the space, and this is independent of the underlying models. However, increasing the dimension is not a panacea, as the 'NaN' problem still exists in high dimensional space (albeit to a lesser degree).

Another straightforward solution to the 'NaN' problem would be to compute with high precision floating-point numbers, such as BigFloats (specially designed with a large quantity of bits), as adopted in the combinatorial construction of hyperbolic embedings in [147], where floating-point numbers with thousands of bits are used with the support of Julia [17]. However, there are two evident problems with representing hyperbolic space using BigFloats. Firstly, BigFloats are not supported on ML accelerator hardware like GPUs: computations using BigFloats on CPUs will be slow and hence not realistic for many meaningful tasks. Secondly, BigFloats are currently not supported in most deep learning frameworks even on CPUs, so one would need to develop the algorithms from scratch so as to learn over hyperbolic space, which is tedious and fraught with the danger of numerical mistakes.

The impact of numerical precision further extends beyond hyperbolic models. Recent success of large models using transformers backend has gathered the attention of community for generative language modeling (GPT-4 [125], LaMDA [163], LLaMa [166]), image generation (e.g., Dall-E [16]), speech generation (such as Meta voicebox, OpenAI jukebox [102, 53]), and multimodality

(e.g. gemini [162]) motivating to further scale such models to larger size and context lengths. However, scaling models is prohibited by the hardware memory and also incur immense compute cost in the distributed training, such as ~1M GPU-hrs for LLaMA-65B [166], thus asking the question whether large model training could be made efficient while maintaining the accuracy?

Previous works have attempted to reduce the memory consumption and run models more efficiently by reducing precision of the parameter's representation, at training time [208, 96, 97, 133] and post-training inference time [40, 141, 34]. The former one is directly relevant to our work using *low-precision storages* at training time, but it suffers from issues such as numerical inaccuracies and narrow representation range. Researchers developed algorithms such as loss-scaling and mixed-precision [115, 154] to overcome these issues. Existing algorithms still face challenges in terms of memory efficiency as they require the presence of high-precision clones and computations in optimizations. One critical limitation of all the aforementioned methods is that such methods keep the "standard format" for floating-points during computations and lose information with a reduced precision.

3.2 Numerically Accurate Hyperbolic Embeddings Using Tiling-Based Models

Overview

While methods proposed in Section 3.1 can greatly improve the accuracy of an embedding empirically, they come with a computational cost, and the floating-point error is still unbounded everywhere. As points move far away from the origin, the error caused by using floating-point numbers to represent them will be unbounded. Even if we try to compensate for this effect by using BigFloats, no matter how many bits we use, there will always be numerical issues for points sufficiently far away from the origin. No amount of BigFloat precision is sufficient to accurately represent points *everywhere* in hyperbolic space.

To address this problem, it is desirable to have a way of representing points in hyperbolic space that: (1) can represent any point in the space with small fixed bounded error; (2) supports standard geometric computations, such as hyperbolic distances, with small numerical error; and (3) avoids potentially expensive BigFloat arithmetic.

One solution is to avoid floating-point arithmetic and do as much computation as possible with integer arithmetic, which introduces no error. To gain intuition, imagine solving the same problem in the more familiar setting of the Euclidean plane. A simple way to construct a constant-error representation is by using the integer-lattice square tiling (or tessellation) [38] of the Euclidean plane. With this, we can represent any point in the plane by (1) storing the coordinates of the square where the point is located as integers and (2) storing the

coordinates of the point within that square as floating point numbers. In this way, the worst-case representation error (3.2.1) will only be proportional to the machine epsilon of the floating-point format—but not the distance of the point from the origin.

We propose to do the same thing in the hyperbolic space: we call this a *tiling-based model*. Given some tiling of hyperbolic space, we can represent a point in hyperbolic space as a pair of (1) the tile it is on and (2) its position within the tile represented with floating point coordinates. In this paper, we show how we can do this, and we make the following contributions:

- We identify tiling-based models for both the hyperbolic plane and for higher-dimensional hyperbolic space in various dimensions. We prove that the representation error (3.2.1) is bounded by a fixed value, further, the error of computing distances and gradients are independent of how far the points are from the origin.
- We show how to compute efficiently over tiling-based models, and we offer algorithms to compress and learn embeddings for real-world datasets.

3.2.1 A Tiling-Based Model for Hyperbolic Plane

We first show explicitly that the standard models of hyperbolic space exhibit unbounded numerical error as the hyperbolic distance from the origin increases.

Definition 3.2.1. *[Representation error] We are concerned with representing points in hyperbolic space $\mathbb{H}^n$ using floating-points* fl*. Define the* representation error *of a particular point $x \in \mathbb{H}^n$ as $\delta_{\mathrm{fl}}(x) = d_{\mathbb{H}^n}(x, \mathrm{fl}(x))$, and the* worst case representation

error *of floating-points representation as a function of the distance-to-origin d, which is the maximum representation error of any point with a distance-to-origin at most d,*

$$\delta_{\text{fl}}^{d} = \max_{x \in \mathbb{H}^n,\, d_{\mathbb{H}^n}(x,O) \leq d} \delta_{\text{fl}}(x).$$

Theorem 3.2.1. *The worst-case representation error (Definition 3.2.1) in the Lorentz model using floating-point arithmetic (with machine epsilon ϵ_m) is $\delta_l^d = \operatorname{arcosh}(1 + \epsilon_m(2\cosh^2(d) - 1))$, where d is the hyperbolic distance to origin. This becomes $\delta_l^d = 2d + \log(\epsilon_m) + o(\epsilon_m^{-1}\exp(-2d))$ if $d = O(-\log \epsilon_m)$.*

Herein, we will describe a tiling-based model that avoids this problem. Our model is constructed on top of the Lorentz model for the two-dimensional hyperbolic plane $\mathbb{H}^2$.

In hyperbolic geometry, a uniform tiling [38, 45, 158] is an edge-to-edge filling of the hyperbolic plane which has regular congruent polygons as faces and is vertex-transitive (there is an isometry mapping any vertex onto any other) [137]. Any tiling is associated with a discrete group G of orientation-preserving isometries of $\mathbb{H}^2$ that preserve the tiling [204, 94]; discrete subgroups of isometries of $\mathbb{H}^2$ (like G) are called *Fuchsian groups* [90, 15, 176]. Importantly, not only does the tiling determine G, but G also determines the shape of the tiling. One way to see this is to consider the images of a single point in $\mathbb{H}^2$ under the group action G (called an *orbit* of the action). Then the Voronoi diagram associated with the orbit (which partitions each point in $\mathbb{H}^2$ into tiles based on which point in the orbit it is closest to) will be a regular tiling of $\mathbb{H}^2$. This equivalence between tilings and groups means that we can reason about tilings by reasoning about Fuchsian groups.

In the 2-dimensional Lorentz model, isometries can be represented as ma-

trices operating on $\mathbb{R}^3$ that preserve the Lorentzian scalar product. That is, a matrix $A \in \mathbb{R}^{3\times3}$ is an isometry if $A^T g_l A = g_l$. If we have some discrete group of isometries G, and we choose the tile which contains the origin to be the *fundamental domain* [176, 175] F, then we can start to define a tiling-based model on top of the Lorentz model of the hyperbolic plane.

***L*-tiling model.** Our first insight is to represent points in the hyperbolic plane as a pair consisting of an element of the group and an element of the fundamental domain. The point represented by this pair is the result of the group element applied to the fundamental domain element. For example, the ordered pair $(\boldsymbol{g}, \boldsymbol{x}) \in G \times F$ would represent the point $\boldsymbol{gx}$. The L-tiling model of the hyperbolic plane is defined as the Riemannian manifold $(\mathcal{T}_l^n, g_{lt})$, where $g_{lt} = g_l$ and

$$\mathcal{T}_l^n = \{(\boldsymbol{g}, \boldsymbol{x}) \in G \times F : \langle \boldsymbol{x}, \boldsymbol{x} \rangle_L = -1\}, \quad d_{lt}((\boldsymbol{g}_x, \boldsymbol{x}), (\boldsymbol{g}_y, \boldsymbol{y})) = \operatorname{arcosh}\left(-\boldsymbol{x}^T \boldsymbol{g}_x^T g_{lt} \boldsymbol{g}_y \boldsymbol{y}\right).$$

Of course, this is useless unless we have a group G that we can store and compute with easily. Our second insight is to construct a Fuchsian group that can be represented with *integers* so that group operations can be computed exactly and efficiently. The naive way to do this is to try the subgroup of orientation-preserving isometries in $\mathbb{R}^{3\times3}$ that have all-integer coordinates: unfortunately, this group (called the modular group) results in a tiling with unbounded fundamental domain, which makes it impossible to bound the representation error, so it is not suitable for our purpose. Instead, we constructed a special Fuchsian group to get a particularly useful L-tiling model of hyperbolic plane.

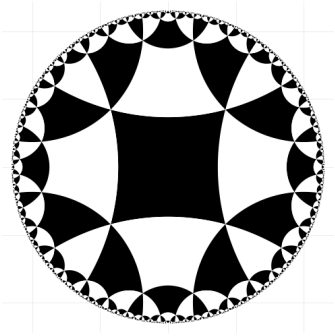


Figure 3.1: The regular quadrilateral tiling of hyperbolic space produced by the group G on the Poincaré disk.

Algorithm 2 Map Lorentz model to L-tiling model

Require: $\boldsymbol{x} \in \mathcal{L}^2$
  initialize $R \leftarrow I$
  while $x \notin F$ **do**
    if $x_2 \leq -|x_3|$ **then** $S \leftarrow g_a^{-1}$
    else if $x_2 \geq |x_3|$ **then** $S \leftarrow g_b^{-1}$
    else if $x_3 < -|x_2||$ **then** $S \leftarrow g_b$
    else if $x_3 > |x_2|$ **then** $S \leftarrow g_a$
    $(R, \boldsymbol{x}) \leftarrow (R \cdot S, L \cdot S^{-1} \cdot L^{-1} \cdot \boldsymbol{x})$
    $x_1 = \sqrt{x_2^2 + x_3^2 + 1}$ ▷ renormalize $\boldsymbol{x}$
  end while
  Output $(R, \boldsymbol{x})$

Definition 3.2.2. *Let g_a and $g_b \in \mathbb{Z}^{3\times3}$ and $L \in \mathbb{R}^{3\times3}$ be defined as*

$$g_a = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 0 & -1 \\ 3 & 2 & 0 \end{bmatrix}, \; g_b = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 0 & -1 \\ -3 & 2 & 0 \end{bmatrix}, \text{ and } L = \begin{bmatrix} \sqrt{3} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Define G to be the fuchsian group generated by $L \cdot g_a \cdot L^{-1}$ and $L \cdot g_b \cdot L^{-1}$. It is straightforward to verify that $(L \cdot g_a \cdot L^{-1})^T g_l (L \cdot g_a \cdot L^{-1}) = (L \cdot g_b \cdot L^{-1})^T g_l (L \cdot g_b \cdot L^{-1}) = g_l$. Note that $g_a^6 = g_b^6 = (g_a g_b)^3 = I$, and so this group has presentation

$$G = L \cdot \langle g_a, g_b | g_a^6 = g_b^6 = (g_a g_b)^3 = 1 \rangle \cdot L^{-1}.$$

Importantly, *any* element of G can be represented in the form $g = LZL^{-1}$ where $Z \in \mathbb{Z}^{3\times3}$ is an all-integers matrix. For this reason, we can store elements of G and take group products and inverses using *only integer arithmetic*. This property makes G of particular interest for use with an L-tiling model. But before we can construct an L-tiling model for this group, we need to choose an appropriate fundamental domain.

Theorem 3.2.2. $F = \{(x_1, x_2, x_3) \in \mathcal{L}^2 | \max(2x_2^2 - x_3^2, 2x_3^2 - x_2^2) < 1\}$ *is a fundamental*

domain of G. Any point in $\mathcal{L}^2$ can be mapped by G to one unique point in F or to a point on its boundary.

Figure 3.1 illustrates the tiling generated by group G and F centered at the origin in the Poincaré disk model. Now that we have a group and a fundamental domain, we can start computing with our new L-tiling model. The first step is to build a relationship between standard hyperbolic models and the L-tiling model, i.e., convert points into the L-tiling model from other models: to this end, we offer a "normalization" procedure (Algorithm 2), which transforms the Lorentz model to the L-tiling model. The convergence and complexity of this algorithm are characterized in Theorem 3.2.3.

Theorem 3.2.3. *For any point in the Lorentz model, Algorithm 2 converges and stops within $1 + 7d$ steps, where $d = d(\boldsymbol{x}, \boldsymbol{O})$ denotes the distance from $\boldsymbol{x}$ to the origin.*

Representing points. For a point $(g, \boldsymbol{x})$ in the L-tiling model, where $g \in G$, $\boldsymbol{x} \in F$, we represent this point with $(g, \mathrm{fl}(\boldsymbol{x}))$. Here g is exact because it is represented by the related integer matrix, while fl denotes float arithmetic with error bounded by some machine epsilon ϵ_m. This floating point arithmetic introduces some representation error, which we can bound as follows:

$$d_{lt}((g, \boldsymbol{x}), (g, \mathrm{fl}(\boldsymbol{x}))) = \operatorname{arcosh}(-\boldsymbol{x}^T g^T g_{lt} g \mathrm{fl}(\boldsymbol{x})) = \operatorname{arcosh}(-\boldsymbol{x}^T g_{lt} \mathrm{fl}(\boldsymbol{x}))$$

Since $\boldsymbol{x} \in F$, which is bounded as shown in Theorem 3.2.2, this approximation error can also be bounded (Theorem 3.2.4). In comparison, for the Lorentz model, the worst case error (Theorem 3.2.1) is unbounded.

Theorem 3.2.4. *The representation error (Definition 3.2.1) in L-tiling model is bounded as $\delta_{lt}^d \leq \sqrt{5\epsilon_m} + 15\epsilon_m/4 + o(\epsilon_m)$, where ϵ_m is the machine error.*

By convention, for $(g, \boldsymbol{x})$ in the L-tiling model, where $g \in G$, $\boldsymbol{x} \in F$, firstly we will usually denote g using its related integer matrix $\hat{g} = L^{-1}gL$; Secondly for the point $x \in F$, even though x is part of the Lorentz model and lies in 3-dimensional space, in fact only two coordinates suffice to determine its position. For simplicity, we define a biejective function $h(x_2, x_3) = (\sqrt{1 + x_2^2 + x_3^2}, x_2, x_3)$ which maps $\mathbb{R}^2$ to the hyperboloid model (this is sometimes called the *Gans model* [61]). In this way, we can represent $(g, \boldsymbol{x}) \in \mathcal{T}_{lt}^2$ as $(\hat{g}, h^{-1}(\boldsymbol{x}))$. We can then store the integer matrix and floating-point coordinates $h^{-1}(\boldsymbol{x}) \in \mathbb{R}^2$. In future sections, we assume we will use this integer matrix and two-coordinate representation rather than $(g, \boldsymbol{x})$ unless otherwise specified.

3.2.2 Learning in the L-tiling Model

In this section, we provide an efficient and precise way to compute distances and gradients accordingly in the L-tiling model, with which we can construct learning algorithms to train and derive embeddings. We also present error bounds for these computations, which avoid the "Nan" problem.

Distance and Gradient. For two points $(U, u), (V, v)$ in the L-tiling model, the formula to compute distance is

$$d((U, u), (V, v)) = \operatorname{arcosh}(h(u)^T L^{-T} Q L^{-1} h(v))$$

where $Q = -U^T L^T g_{lt} L V$ can be computed exactly with integer arithmetic. A potential difficulty here is that the entries in Q can be very large (possibly even larger than can be represented in floating-point). To solve this, observe that Q_{11} has the largest absolute value in the matrix (B.1.2). So we define and compute

$\hat{Q} = Q/Q_{11}$, which is guaranteed to not overflow the floating-point format, since all the entries of $\hat{Q}$ are in $[-1, 1]$. Let $d_c = h(u)^T L^{-T} \hat{Q} L^{-1} h(v)$, this reduces our distance to

$$d((U,u),(V,v)) = \operatorname{arcosh}(Q_{11} \cdot d_c) = \log(Q_{11}) + \log\left(d_c + \sqrt{d_c^2 - Q_{11}^{-2}}\right)$$

Note that (assuming that we can compute $\log(Q_{11})$ without overflow) this expression can be computed in floating-point without any overflow, since all the numbers involved are well within range. The corresponding formula for the gradient can also be derived as

$$\nabla_u d((U,u),(V,v)) = \frac{\nabla h(u)^T L^{-T} \hat{Q} L^{-1} h(v)}{\sqrt{d_c^2 - Q_{11}^{-2}}},$$

where

$$\nabla h(u) = \left[\frac{u}{\sqrt{1 + \|u\|^2}},\ I\right]$$

Again, this avoids any possibility of overflow. We provide the error of computing distance (Theorem B.1.3) and gradient (Theorem B.1.4) in L-tiling model together with that in Lorentz model in Appendix. By computing with integer arithmetic, the error will be independent of how far the points are from the origin, which guarantees that it avoids the "NaN" problem. Since we

Algorithm 3 RSGD in the L-tiling model

Require: Objective function f, fuchsian group G with fundamental domain F, exponential map $\exp_{\beta_t}(v) = \cosh(\|v\|_L)\beta_t + \sinh(\|v\|_L)\frac{v}{\|v\|_L}$, where $\|v\|_L = \sqrt{\langle v, v\rangle_L}$.

Require: $(\beta_t, U_t) \in F \times G$, Epochs T, and learning rate η

for $t = 0$ to $T - 1$ **do**

$l_t \Leftarrow g_{\beta_t}^{-1} \nabla_{\beta_t} f(L U_t L^{-1} \beta_t)$ ▷ Riemannian

grad $f \Leftarrow l_t + \langle \beta_t, l_t \rangle_L \beta_t$ ▷ Projection

$\beta_{t+1} \Leftarrow \exp_{\beta_t}(-\eta \operatorname{grad} f)$ ▷ Update

if $\beta_{t+1} \notin F$ **then**

$W \Leftarrow \arg\min_{W \in G} d(L W^{-1} L^{-1} \beta_{t+1}, O)$

$U_{t+1} \Leftarrow U_t \cdot W$ ▷ Normalize if $\beta_{t+1} \notin F$

$\beta_{t+1} \Leftarrow L W^{-1} L^{-1} \beta_{t+1}$

else

$U_{t+1} \Leftarrow U_t$

end if

end for

Output (β_{t+1}, U_{t+1})

can compute distances and derivatives, we can use all the standard gradient-based optimization algorithms. In Algorithm 3 we present the most powerful one, RSGD, adapted for use with the L-tiling model.

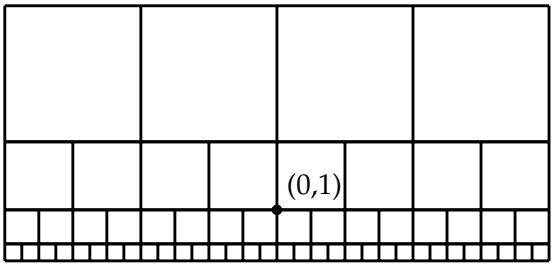
3.2.3 Extension to Higher Dimensional Space

Extending the L-tiling model to higher dimension seems simple: just find a co-compact (to ensure a bounded fundamental domain) discrete subgroup of the higher-dimensional space's isometry group. Such a group would induce a *honeycomb*, a higher-dimensional analog of a regular tiling of the hyperbolic plane. Unfortunately, a classic result by [41] says this is impossible in general: there are no such regular honeycombs in six or more dimensions.

In order to derive a high dimensional tiling-based model which may be necessary for complicated datasets, we consider two possibilities.

- Take the Cartesian product of multiple copies of the L-tiling model in the hyperbolic plane. The use of multiple copies of models in the hyperbolic plane was previously proposed in [66].
- Construct honeycombs and tilings from a set of isometries that is not a group.

Practically we can embed data into products of $\mathbb{H}^2$s as we do in section 3.2.4, however, the first possibility (tilings over $\mathbb{H}^2 \times \mathbb{H}^2 \times \cdots \mathbb{H}^2$) is something fundamentally different from tiling a single high dimensional hyperbolic space (tilings over $\mathbb{H}^n$), which we aims to do in this section. Fortunately for the second possibility, in half-space model, we find that horizontal translation and homotheties are hyperbolic isometries, which can produce the (infinite) square tiling



Datasets	Nodes	Edges
Bio-yeast[144]	1458	1948
WordNet[57]	74374	75834
↳ Nouns	82115	769130
↳ Verbs	13542	35079
↳ Mammals	1181	6541
Gr-QC[104]	4158	13422

Figure 3.2: (Left) The infinite square tiling of hyperbolic space on the half-plane model; (Right) Dataset statistics.

illustrated in Figure 3.2 (Left) [6, 25]. It consists of the image of the unit square S, with vertical and horizontal sides and whose lower left corner is at $(0, ..., 0, 1)$, under the maps

$$p \to 2^j(p + k), (j, k) \in \mathbb{Z} \times (\mathbb{Z}^{n-1} \times \{0\}).$$

Here each square is isometric to every other square, and the unit square S takes on the role of the fundamental domain in Theorem 3.2.2. With these maps, we can define a tiling-based model on top of the half-space model as follows.

***H*-tiling model.** The H-tiling model of the hyperbolic space is defined as the Riemannian manifold $(\mathcal{T}_h^n, g_{ht})$, where

$$\mathcal{T}_h^n = \{(j, \boldsymbol{k}, \boldsymbol{x}) \in \mathbb{Z} \times (\mathbb{Z}^{n-1} \times \{0\}) \times S\}, \qquad g_{ht}(j, \boldsymbol{k}, \boldsymbol{x}) = \frac{g_e}{(2^j x_n)^2}$$

The associated distance function on $\mathcal{T}_h^n$ is then given as

$$d((j_1, \boldsymbol{k}_1, \boldsymbol{x}), (j_2, \boldsymbol{k}_2, \boldsymbol{y})) = \operatorname{arcosh}\left(1 + \frac{\|2^{j_1} z_1 - 2^{j_2} z_2 + 2^{j_1} k_1 - 2^{j_2} k_2)\|^2}{2^{j_1 + j_2 + 1} z_{1n} z_{2n}}\right).$$

Similarly, we derive the representation error for this model, which is bounded by a constant depending on the machine epsilon as shown in Theorem 3.2.5.

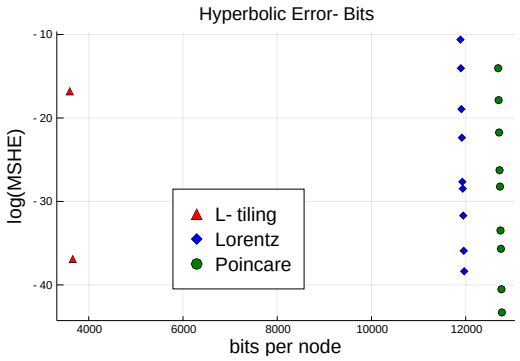
Theorem 3.2.5. *The representation error (Definition 3.2.1) in H-tiling model is bounded as* $\delta_{ht}^{d} = \sqrt{(n+3)\epsilon_m}/2 + (n+3)\epsilon_m/4 + o(\epsilon_m)$*, where* ϵ_m *is the machine error.*

We can compute distances and gradients in a numerically accurate way, and run RSGD algorithm on this model for optimization, just as we could in the *L*-tiling model. For lack of space, we defer that discussion and more learning details of sections 3.2.2 and 3.2.3 to B.1.1. Also note that we are not tied to the half-space model here: while the half-space model gives a convenient way to describe the set of transformations we are using, we could use the same transformations with any underlying model we choose by adding an appropriate conversion.

3.2.4 Experiments

Compressing embeddings. We consider storing 2-dimensional embeddings using the *L*-tiling model for compression: storage using few bits. While storing the integer matrices exactly is convenient for computation, it does tend to take up a lot of extra memory (especially when BigInts are needed to store the integer values in the matrix). This motivates us to look for alternative storage methods. To store the matrix g, we prorpose and evaluate the following methods:

- Matrix: store all 9 integers in the matrix g as Int or BigInt.
- Entries: store just g_{21}, g_{31} as Int or BigInt, which we can show is sufficient to reconstruct the whole matrix (Theorem B.1.1 in Appendix B.1.3).
- Order: store the generator order with respect to g_a, g_b as a string.
- VBW: store the generator order with respect to g_a, g_b using a variable bit-width encoding. We use binary code `10` to represent g_a^1 and g_b^1, `001` to



Models	size (MB)	bzip (MB)
Poincaré(16512B)	372	119
Poincaré(12688B)	287	81
Lorentz(11898B)	396	171
Matrix (VLQ)	600 (286)	260 (251)
Entries (VLQ)	132 (63)	57 (55)
Order	111	8.52
VBW	33.1	6.07
f32-FD	6.2	1.96
f16-FD	4.25	1.07

Figure 3.3: (Left) Hyperbolic error for WordNet Nouns; (Right) Compression statistics for WordNet under the same MSHE, first block contains the size of original poincare embedding, second block contains the size of compressed baseline models, third block contains the size of matrix part in the *L*-tiling model (size of compressed integers using VLQ is also reported), the last block contains size of float points (f32 or f16) in the fundamental domain (FD) of *L*-tiling model.

represent g_a^2 and g_b^2, `010` to represent g_a^3 and g_b^3, `011` to represent g_a^4 and g_b^4, `11` to represent g_a^5 and g_b^5, and `000` to represent the end of the string. This encoding disambiguates the generators by taking advantage of the fact that powers of g_a and g_b must alternate to appear.

The generator order and corresponding VBW encoding of a given matrix can be derived using Algorithm 2 as shown in Lemma B.1.1. Additionally, for Int or BigInt, we can use variable length quantity (VLQ) to compress [148]. To test our compression methods, we use combinatorial construction [147] to derive 2-dimensional Poincaré disk embeddings for WordNet (Tree-like) and Bio-yeast datasets (Figure 3.2 Right Table), then we transform embeddings and compress them. We calculate the mean squared hyperbolic error (MSHE) with respect to the original embedding to show the error of compression.

For Bio-yeast, we evaluate different compressions using MSHE and mean average precision (MAP). As shown in Table 3.1, representation and compres-

Table 3.1: Compression error of Bio-yeast

Models	MSHE	MAP
Poincaré(8128B)	0.00	0.873
Poincaré(6360B)	4.84e-17	0.873
Poincaré(1832B)	1.01e+03	0.310
L-tiling-f64(1832B)	9.76e-17	0.873
L-tiling-f32(1768B)	5.12e-08	0.873
L-tiling-f16(1736B)	4.30e-05	0.873
L-tiling-f0(1704B)	5.90e-01	0.873

Table 3.2: Embedding performances of Mammals

Models	MAP	MR
Poincaré	0.805±0.011	2.22±0.10
Lorentz	0.855±0.013	1.89±0.13
L-tiling-SGD	0.892±0.031	2.14±0.70
L-tiling-RSGD	**0.930**±0.005	**1.49**±0.09
H-tiling-RSGD	0.923±0.016	1.56±0.20

sion in the *L*-tiling model (with different floating number for points in the fundamental domain) does not hurt MAP performance, while the compression of the Poincaré embedding to the same size hurts MAP severely. For WordNet, we plot the scatter of the relationship between log(MSHE) and bits to store per node in Figure 3.3 (Left). Under the same MSHE, the *L*-tiling model requires approximately 2/3 less bits per node compared to that of Lorentz and Poincaré models. We measure the size of different models under the same MSHE in Figure 3.3 (Right). The *L*-tiling model can represent the hyperbolic embedding with only (6.07+1.07) MB, which is 2% of the original 372 MB, while it will cost at least 81 MB for any reasonably accurate baseline model.

Learning embeddings. As we have shown, our tiling-based models represent hyperbolic space accurately, and so they can be used for learning embeddings with generic objective functions. However, since we analyzed hyperbolic distance and gradient computation error in this paper, we evaluate our learning methods empirically on objective functions that depend on distances. As proposed in [122], to consider the ability to embed data that exhibits a clear latent hierarchical structure, we conduct reconstruction experiments on the transitive closure of the Gr-QC, WordNet Nouns, Verbs and Mammals hierarchy as sum-

marized in Table 3.2. We firstly embed the data and then reconstruct it from the embedding to evaluate the representation capacity of the embedding. Let $\mathcal{D} = \{(u, v)\}$ be the set of observed relations between objects. We aim to learn embeddings of $\mathcal{D}$ such that related objects are close in the embedding space. To do this, we minimize the loss [122]

$$\mathcal{L}(\Theta) = \sum_{(u,v)\in\mathcal{D}} \log \frac{e^{-d(\boldsymbol{u},\boldsymbol{v})}}{\sum_{\boldsymbol{v}'\in\mathcal{N}(u)} e^{-d(\boldsymbol{u},\boldsymbol{v}')}}, \tag{3.1}$$

where $\mathcal{N}(u) = \{v \mid (u, v) \notin \mathcal{D}\} \cup \{u\}$ is the set of negative examples for u (including u). We randomly sample $|\mathcal{N}(u)| = 50$ negative examples per positive example during training.

We consider the L-tiling models trained with RSGD and SGD, H-tiling models trained with RSGD and the Cartesian product of multiple copies of 2-dimensional L-tiling models (proposed in [66]). The Poincaré ball model [122] and Lorentz model [123] were included as baselines. All models were trained in float64 for 1000 epochs with the same hyper-parameters. To evaluate the quality of the embeddings, we make use of the standard graph embedding metrics in [20, 121]. For an observed relationship (u, v), we rank the distance $d(\boldsymbol{u}, \boldsymbol{v})$ among the set $\{d(u, v')|(u, v') \in \mathcal{D})\}$, then we evaluate the ranking on all objects in the dataset and record the mean rank (MR) as well as the mean average precision (MAP) of the ranking.

We start by evaluating all 2-dimensional embeddings on the Mammals dataset. As shown in Table 3.2, all tiling-based models outperform baseline models: the performances of L-tiling model and H-tiling model with RSGD are nearly the same. In particular, the L-tiling model achieves a 8.8% MAP improvement on Mammals compared to Lorentz model.

Embedding experiments on other three large datasets are presented in Ta-

Dimension	Models	WordNet Nouns		WordNet Verbs		Gr-QC	
		MAP	MR	MAP	MR	MAP	MR
2	Poincaré	0.124±0.001	68.75±0.26	0.537±0.005	4.74±0.17	0.561± 0.004	67.91±1.14
	Lorentz	0.382±0.004	17.80±0.55	**0.750**±0.004	2.11±0.06	0.563±0.003	68.40±1.20
	H-tiling-rsgd	0.390±0.002	17.18±0.52	0.747±0.003	2.10±0.05	0.560±0.004	66.17±1.05
	L-tiling-sgd	0.341±0.001	20.27±0.39	0.696±0.003	2.33±0.07	**0.574**±0.005	**63.04**±1.97
	L-tiling-rsgd	**0.413**±0.007	**15.26**±0.57	0.746±0.004	**2.07**±0.03	0.564± 0.002	63.88±1.47
4	2×Lorentz	0.460±0.001	10.12±0.03	**0.873**±0.001	1.31±0.01	**0.718**±0.003	11.59±0.32
	2×L-tiling-rsgd	**0.464**±0.002	**9.99**±0.09	0.871±0.004	1.33±0.01	0.716±0.005	**10.88**±0.42
5	Poincaré	0.848±0.001	4.16±0.04	0.948±0.001	1.19±0.01	0.714±0.000	34.60±0.52
	Lorentz	0.865±0.005	**3.70**±0.12	0.947±0.001	**1.16**±0.00	**0.715**±0.003	33.51± 1.04
	H-tiling-rsgd	**0.869**±0.001	**3.70**±0.06	**0.949**±0.001	**1.16**±0.01	0.714±0.002	**33.46**±0.66
10	Poincaré	0.876±0.001	3.47±0.02	0.953±0.002	1.16±0.01	0.729±0.000	29.51±0.21
	Lorentz	0.865±0.004	3.36±0.04	0.948±0.001	1.15±0.00	0.724±0.001	29.34±0.23
	H-tiling-rsgd	**0.888**±0.004	**3.22**±0.02	0.954±0.002	1.15±0.00	0.729±0.001	27.75±0.39
	5×Lorentz	0.672±0.000	4.42±0.00	0.958±0.003	1.07±0.01	0.944±0.007	3.06±0.03
	5×L-tiling-rsgd	0.674±0.000	4.41±0.00	**0.961**±0.002	**1.06**±0.00	**0.953**±0.002	**3.03**±0.01

Table 3.3: Embedding experiments on different datasets. Results are averaged over 5 runs and reported in mean+std style.

ble 3.3. These results show that tiling-based models generally perform better than baseline models in various dimensions. We found three observations particularly interesting here. First, the group-based tiling model (L-tiling) performs better than the non-group tiling model (H-tiling) in two dimensions. Second, tiling-based models perform particularly better than baseline models for the largest WordNet Nouns dataset, which further validates that numerical issue happens for standard models when the embeddings are far from the origin and affects the embedding performances. Third, the Cartesian product of multiple copies of 2-dimensional L-tiling models performs even better than high dimensional models when the datasets are not too large and complex such as WordNet Verbs and Gr-QC, especially for the dense graph Gr-QC.

More experiment details are provided in B.1.1. We release our compression code[1] in Julia and learning code[2] in PyTorch publicly for reproducibility.

[1] https://github.com/ydtydr/HyperbolicTiling_Compression
[2] https://github.com/ydtydr/HyperbolicTiling_Learning

3.3 Representing Hyperbolic Space Accurately using Multi-Component Floats (MCF)

Overview

The *tiling-based* hyperbolic model in Section 3.2 allows for a guaranteed bound on the numerical error of learning in hyperbolic space. However, their approach is limited somewhat in that its guarantees apply to loss functions that depend only the hyperbolic distance between points. A bigger issue with this type of approach is computational: the big-integer matrix multiplication involved in the model is not supported on GPUs and is not yet implemented in many deep learning frameworks such as PyTorch [128] and TensorFlow [1]. This effectively prevents the method from being used in training of complex models that depend on GPU acceleration, such as hyperbolic neural networks [60, 67, 28, 106].

In this work, we propose a **simple, feasible-on-GPUs**, and easy-to-understand solution for general numerically accurate float computations. We use multi-component floating-point (MCF) [138, 139, 77, 152] to represent exact numbers as the unevaluated sum of multiple ordinary (low-precision) floats. MCF allows roundoff error to occur when performing exact arithmetic, then accounts for it afterwards by adding more components (in the same precision). We make the following contributions:

- We propose representing hyperbolic space with the Poincaré upper-half space model for optimization, and show how to represent it with MCF to eliminate the 'NaN' problem.

- We provide algorithms for computing with these MCF that are adapted to hyperbolic space, where we prove numerical errors can be reduced to any degree by simply increasing the number of components of the MCF.
- We experimentally measure our model on embedding tasks across several datasets, and show that one can gain more learning capacity (compared to models that use ordinary floating-point numbers) with only a mild computational slowdown on GPUs, meanwhile significantly faster than prior high precision hyperbolic models.

3.3.1 Computing with MCF

Numerical error analysis. Section 3.2 shows that in the Lorentz hyperboloid model of the hyperbolic space, the error of representing an exact point $\boldsymbol{x}$ with ordinary floating-point $\mathrm{fl}(\boldsymbol{x})$ is unbounded and proportional to the distance to the origin $d(\boldsymbol{x}, \boldsymbol{O})$. Here, as a baseline to compare our MCF methods to, we show that the same sort of representation error $d(\boldsymbol{x}, \mathrm{fl}(\boldsymbol{x}))$ happens when representing points in the Poincaré upper-half space model with ordinary floating-point numbers.

Theorem 3.3.1. *The representation error of storing a particular point $\boldsymbol{x} \in \mathcal{U}^n$ using floating-points* fl *is $\delta_{\mathrm{fl}}(\boldsymbol{x}) = d_u(\boldsymbol{x}, \mathrm{fl}(\boldsymbol{x}))$, and the* worst case representation error *defined as a function of the distance-to-origin d in the Poincaré upper-half space model is*

$$\delta_d := \max_{\boldsymbol{x} \in \mathcal{U}^n, d_u(\boldsymbol{x}, \boldsymbol{O}) \leq d} \delta_{\mathrm{fl}}(\boldsymbol{x}) = \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cosh^2(d)).$$

where $\epsilon_{machine}$ is the machine epsilon of the underlying floating-point arithmetic. This becomes $\delta_d = 2\epsilon_{machine} + o(\epsilon_{machine})$ if $d < \log(1/\epsilon_{machine})$ and $\delta_d = 2d + 2\log(\epsilon_{machine}) +$

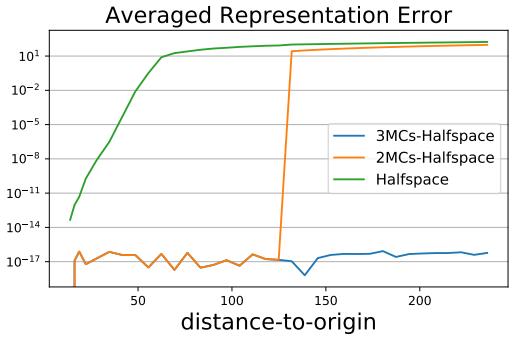


Figure 3.4: Averaged representation error $d(\boldsymbol{x}, \text{fl}(\boldsymbol{x}))$ as $\boldsymbol{x}$ moves away from the origin in halfspace models.

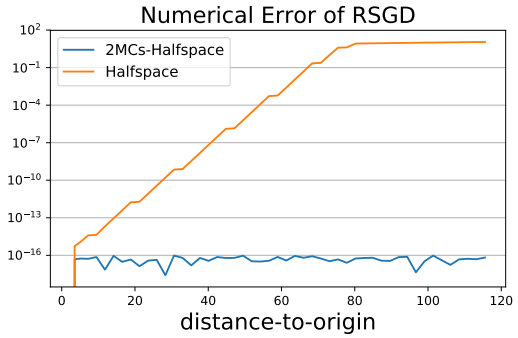


Figure 3.5: Numerical error $d(\boldsymbol{y}, \boldsymbol{y}')$ of the RSGD algorithm as $\boldsymbol{x}$ moves away from the origin in halfspace models.

$o(\epsilon_{machine}^{-1} \exp(-2d))$ *if* $d \geq \log(1/\epsilon_{machine})$.

We plot the averaged representation error and the 'NaN' problem practically in Figure 3.4, the distance error $d(\boldsymbol{x}, \text{fl}(\boldsymbol{x}))$ when $\boldsymbol{x}$ is represented as $\text{fl}(\boldsymbol{x})$ using floating-point. The error explodes at at distance of around 55 from the origin, which will then potentially cause NaNs in subsequent computations.

Optimizing $\boldsymbol{x}$ in the Poincaré upper-half space model will typically push $\boldsymbol{x}$ away from the origin, with $x_i, i \neq n$ being large floating-point numbers and x_n being floating-point numbers close to 0. Hence, the subtraction of two big floating-point numbers in $\|\boldsymbol{x} - \boldsymbol{y}\|^2$ of the distance calculation leads to catastrophic cancellation and accounts for a large part of the numerical errors. We can localize this part of the numerical errors easily in simple subtractions in the Poincaré upper-half space model. (For comparison, in the Lorentz model Section 2.1, the multiplication in Lorentzian scalar product accounts for the major numerical errors in distance computations.)

We can derive a $\boldsymbol{y}'$ using ordinary floating-point arithmetic and compute the distance error between the exact $\boldsymbol{y}$ and $\boldsymbol{y}'$. In Figure 3.5, we show how this error

Table 3.4: Overview of MCF algorithms.

Operation	Function	Example Usage
Two-Sum	sum of two floats	most MCF algorithms
Grow-Expansion	add a float to an expansion	RSGD, update parameters
Add-Expansion	sum of two expansions	distance calculation
Scale-Expansion	multiply a float to an expansion	gradient calculation, RSGD
Renormalize	reduce # components of an expansion	RSGD, reduce # components

Algorithm 4 Two-Sum

Input: p-bit floats a, b, where $p \geq 3$
$x \leftarrow \mathbf{fl}(a + b)$
$b_{\text{virtual}} \leftarrow \mathbf{fl}(x - a)$
$a_{\text{virtual}} \leftarrow \mathbf{fl}(x - b_{\text{virtual}})$
$b_{\text{roundoff}} \leftarrow \mathbf{fl}(b - b_{\text{virtual}})$
$a_{\text{roundoff}} \leftarrow \mathbf{fl}(a - a_{\text{virtual}})$
$y \leftarrow \mathbf{fl}(a_{\text{roundoff}} + b_{\text{roundoff}})$
Return: (x, y)

Algorithm 5 Grow-Expansion

Input: m p-bits expansion e, p-bit float b
initialize $Q_0 \leftarrow b$
for $i = 1$ to m **do**
$(Q_i, h_i) \leftarrow \textbf{Two-Sum}(Q_{i-1}, e_i)$
end for
$h_{m+1} \leftarrow Q_m$
Return: h

$d_u(\boldsymbol{y}, \boldsymbol{y}')$ varies as the starting point $\boldsymbol{x}$ gradually moves away from the origin. At a distance of around 60, the halfspace RSGD algorithm will be numerically unstable with numerical errors large enough to cause NaNs.

Hence, we propose to use multiple-component floating-point numbers in the Poincaré upper-half space model to avoid the numerical errors in addition and subtraction. We provide some basic numerical operations for computing with multiple-component floating-point numbers. To start with, we consider the sum of two p-bit floating-point numbers to form a non-overlapping expansion.

Theorem 3.3.2. *[93, 152]. For floating point numbers a, b, Alg. 4 produces a non-overlapping expansion (x, y) such that $a + b = x + y$, where x is an approximation to $a + b$ and y is the roundoff error. Particularly, when exact rounding is adopted, the roundoff error y is bounded as*

$$|y| \leq \frac{1}{2}\mathbf{ulp}(\text{fl}(a + b)), \quad |y| \leq \min(|a|, |b|).$$

where **ulp**(x) *is the unit of the least precision of* x.

This theoretical bound on the roundoff error y serves an important role in computations involving Alg. 4 (most MCF algorithms). Now we can derive an expansion by adding two p-bit floating-point values with this algorithm, next, we need to add a single floating-point number to an expansion. Alg. 5 shows how to add a single p-bit value to any arbitrary precision value expressed as an expansion.

Algorithm 6 Renormalize [139]

Input: $(m+1)$-components expansion $(a_0, a_1, \cdots, a_m)$ in decreasing order.
initialize $s \leftarrow a_m, k \leftarrow 0$
for $i = m$ to 1 **do**
  $(s, t_i) \leftarrow$ **Two-Sum**(a_{i-1}, s)
end for
for $i = 1$ to m **do**
  $(s, e) \leftarrow$ **Two-Sum**(s, t_i)
  if $e \neq 0$ **then**
    $b_k \leftarrow s$
    $s \leftarrow e$
    $k \leftarrow k + 1$
  end if
end for
Return: $(b_0, b_1, \cdots, b_{m-1})$

Theorem 3.3.3. *[152] Let* $e = \sum_{i=1}^{m} e_i$ *be a non-overlapping* m $p(\geq 3)$*-bit components expansion, sorted in increasing order, except that any of the* e_i *may be zero. Let* b *be a* $p(\geq 3)$*-bit value, then Alg. 5 produces a non-overlapping* $m+1$ *cmponents expansion* $h = \sum_{i=1}^{m+1} h_i = e + b$*, where the components are also in increasing order, except that any of them may be zero. Notably,* Q_i *in Alg. 5 can serve as an approximate sum of* b *and the first* i *components of* e.

As a result of Alg. 5, addition of a value to an m-components expansion will grow the expansion and outputs an expansion with $m+1$ components. However, if there are no constraints on the number of components, then different values will be highly irregular and vary a lot, which will slow down the computations and make the algorithms much more complicated. Hence, in our implementation, rather to grow the number of components without any constraints, we

would like to fix the number of components for all expansions we may use. With Alg. 6, we can renormalize an $m + 1$ expansion back to an m components expansion.

[139] proves that if the input expansion of Alg. 6 does not overlap by more than 51 bits, then the algorithm works correctly. This condition holds when ordinary floating-point numbers are used. Table 3.4 summarizes MCF algorithms used in this paper, some of them detailed in the Appendix. Note that firstly these MCF algorithms can be easily vectorized to apply on high dimensional vectors, secondly, they are linear in the number of components (quadratic for **Scale-Expansion**) and dimension, hence enjoy a great speedup.

3.3.2 Learning using MCF

We solve the 'NaN' problem in the Poincaré upper-half space model using multiple-component floating-point arithmetic in the addition & subtraction computations. We only need to use m-multiple-component floating-point numbers for the first $n - 1$ coordinates of the model, denoted as m**-xMC-Halfspace**. We will mostly compute with MCF arithmetic (in Table 3.4) for the addition & subtraction involved in the computations, and leave the rest of computations to ordinary floating-point arithmetic. For example, we compute the distance between $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{U}^n$ as:

$$d_u(\boldsymbol{x}, \boldsymbol{y}) = \operatorname{arcosh}\left(1 + \frac{\textbf{Add-Expansion}(\boldsymbol{x}, -\boldsymbol{y})_1^2}{2x_n y_n}\right)$$

where we add two expansions $\boldsymbol{x}, -\boldsymbol{y}$ to get the summed expansion and then take the first, largest component as an approximation of the scalar $\|\boldsymbol{x} - \boldsymbol{y}\|$. Further more, in Appendix we show how to do RSGD using the multi-component

floating-point numbers in the *m*-**xMC-Halfspace** model.

When we compute with the introduced multi-components floating-point numbers in the model, we show theoretically the worse case numerical representation error in Theorem 3.3.4.

Theorem 3.3.4. *The worst case representation error of storing an exact point* $\boldsymbol{x} \in \mathcal{U}^n$ *with m-multi-component floating-point expansion* $(\boldsymbol{x}^{(m)}, \cdots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ *defined as a function of the distance-to-origin d is*

$$\delta_d = \operatorname{arcosh}\left(1 + \epsilon_{machine}^2 + \frac{\epsilon_{machine}^{2m}(1+\epsilon_{machine})}{2^{2m}}\cosh^2(d)\right),$$

where $\epsilon_{machine}$ *is the machine epsilon. This becomes* $\delta_d = 2\epsilon_{machine} + o(\epsilon_{machine})$ *if* $d < m\log(1/\epsilon_{machine})$ *and* $\delta_d = 2d + 2m\log(\epsilon_{machine}/2) + o(\epsilon_{machine}^{-2m}\exp(-2d))$ *if* $d \geq m\log(1/\epsilon_{machine})$.

Hence, there's a region around the origin whose radius is proportional to the number of components *m*, where we can be confident there are no numerical issues. This is the scaling we'd expect, and is in line with the results from [147]. In Figure 3.4, we plot the averaged representation error $d(\boldsymbol{x}, \mathbf{MCF}(\boldsymbol{x}))$ when $\boldsymbol{x}$ is represented using 2-component floating-points. The error will not explode until a distance of 125 from the origin simply by using 2 components, compared to the distance of 55 for vanilla halfspace.

Particularly, we also show the numerical errors of the adapted RSGD algorithm in the 2-**xMC-Halfspace** model in Figure 3.5, with way smaller error compared to the increasing numerical errors in Poincaré upper-half space model using ordinary floating numbers. Practitioners can choose the number of components according to the desired precision requirements, yet many applications would get full benefits from using merely a small multiple of (such as two or three) components.

Table 3.5: Dataset Statistics.

Datasets	Nodes	Edges
WordNet[57]	74374	75834
↳ Nouns	82115	769130
↳ Verbs	13542	35079
↳ Mammals	1181	6541
Gr-QC[104]	4158	13422

Table 3.6: Embedding performances on Mammals

Model	MAP	MR
Halfspace	92.07%±1.01%	1.600±0.79
2-xMC-Halfspace	**93.85**%±0.57%	**1.430**±0.16
Lorentz	85.75%±2.30%	1.857±4.78
L-Tiling	91.49%±1.42%	1.645±1.28
H-Tiling	91.60%±1.60%	1.559±0.44

As a matter of fact, MCF can be more generally adopted in many operations, in all hyperbolic models and different generic loss functions for better performance. Hence we also implement the model with all coordinates represented with MCF, denoted as *m*-**MC-Halfspace**. We defer the discussion of this model to Appendix since it is more sophisticated by making use of the **Scale-Expansion** algorithm.

3.3.3 Experiments

Firstly proposed in [122], hyperbolic embedding was used to embed data with hierarchical structure such as taxonomies, producing results that outperform Euclidean embeddings. Motivated by this, we conduct embedding experiments on datasets including the Gr-Qc and commonly used WordNet Nouns, Verbs and Mammals dataset whose statistics are shown in Table 3.5. We are interested in two aspects of the proposed MCF-based models in practice: firstly the capacity of the models, and secondly the running speed of model training and optimization.

Reconstruction. We focus on the tasks which can show the capacity of the models. Given an observed transitive closure $\mathcal{D} = \{(u, v)\}$ of all objects, we em-

bed all objects into hyperbolic space and then reconstruct it from the embedding. Hence, the error from the reconstructed embedding to the ground truth will be a reasonable measure for the capacity of the model and reflect how the underlying floating-point arithmetic may affect the performance. Specifically, we learn embeddings of the transitive closure $\mathcal{D}$ such that related objects are close in the embedding space. To do this, we aim to minimize the loss introduced in [122]:

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

where $\mathcal{N}(u) = \{v \mid (u,v) \notin \mathcal{D}\} \cup \{u\}$ is the set of negative examples for u (including u). We do negative sampling randomly per positive example during training for different datasets.

We implemented the upper-half space model and **xMC-Halfspace** models trained with RSGD in our experiments, as for baselines, we consider the current state-of-the-art L-tiling models, H-tiling models [199] trained with RSGD and the Lorentz model [123]. Models were trained using ordinary float64 with the same hyper-parameters starting from same initializations of parameters, see Appendix for more implementation details. To evaluate the quality of the embeddings, we make use of the standard graph embedding metrics in [20, 121]. For an observed relationship (u, v), we rank the distance $d(\boldsymbol{u}, \boldsymbol{v})$ among the set $\{d(u, v')|(u, v') \in \mathcal{D})\}$, then we evaluate the ranking on all objects in the dataset and record the mean rank (MR, smaller⇒better) as well as the mean average precision (MAP, larger⇒better) of the ranking.

We firstly evaluate different 2-dimensional embeddings on the light-scale Mammals dataset. As shown in Table 3.6, we choose 2 components for the MCF-based models. The proposed upper-half space model already outperforms all

Dimension	Models	WordNet Nouns		WordNet Verbs		Gr-QC	
		MAP	MR	MAP	MR	MAP	MR
2	Lorentz	15.61±0.37%	55.59±0.53	56.76±0.52%	4.18±0.07	55.93±0.32%	71.81±1.24
	L-tiling	23.39±0.31%	34.22±1.03	63.08±0.31%	3.28±0.03	56.07±0.21%	72.53±2.08
	H-tiling	23.34±0.19%	34.30±0.49	64.26±0.40%	3.23±0.05	55.95±0.30%	**71.78**±1.02
	Halfspace	23.31±0.17%	34.81±0.26	63.65±0.32%	3.36±0.07	55.60±0.35%	74.29±1.32
	2-xMC-Halfspace	**24.42**±0.11%	**34.01**±0.17	**64.88**±0.25%	**3.18**±0.05	**56.35**±0.38%	71.81±1.12
5	Lorentz	74.82±0.38%	7.528±0.23	89.72±0.04%	1.52±0.00	71.16±0.33%	33.66±1.23
	H-tiling	74.90±0.08%	7.529±0.12	90.13±0.08%	1.46±0.00	70.89±0.15%	32.99±0.52
	Halfspace	74.89±0.14%	7.388±0.08	89.99±0.21%	1.51±0.01	71.10±0.19%	32.29±0.75
	2-xMC-Halfspace	**75.05**±0.12%	**7.377**±0.12	**90.25**±0.09%	**1.41**±0.01	**71.21**±0.24%	**31.89**±0.71
10	Lorentz	78.43±0.42%	6.148±0.15	**90.71**±0.13%	1.41±0.00	**72.84**±0.13%	28.53±0.27
	H-tiling	78.40±0.22%	6.229±0.13	90.62±0.02%	1.40±0.00	72.57±0.09%	**28.50**±0.35
	Halfspace	78.39±0.12%	6.227±0.21	**90.71**±0.11%	**1.39**±0.02	72.66±0.11%	28.56±0.24
	2-xMC-Halfspace	**78.46**±0.08%	**6.136**±0.14	**90.71**±0.04%	**1.39**±0.00	72.69±0.09%	28.57±0.32

Table 3.7: Embedding learning experiments on different datasets. Results are averaged over 5 runs and reported in mean+std style. **Bold** numbers show the best performance among all models.

previously proposed baseline models. In particular, the **xMC-Halfspace** model further improves the embedding performance and achieves a 2.25% MAP improvement on Mammals compared to the best baseline models.

We present the large scale experiments on other three datasets in Table 3.7. These results show that MCF-based models generally perform better than baseline models in various dimensions with respect to MAP and MR, particularly in 2 dimensional hyperbolic space, since higher dimension can alleviate the 'NaN' problem to some degree. On 2 dimensional embeddings of the largest WordNet Nouns dataset, MCF-based models gain as large as a 1.03% MAP improvement compared to the *L*-tiling model and a 1.11% MAP improvement compared to the upper-half space model.

The limitation of the work is that in some cases of embedding problems that stay close to the origin, MCF will give no improvement over plain half space because it was already numerically accurate enough, also our method will eventually break down at very large distances due to overflow or underflow of the floating point exponent.

Dataset	Models	Batchsize=32			Batchsize=64			Batchsize=128		
		2D	5D	10D	2D	5D	10D	2D	5D	10D
WordNet Nouns	Lorentz	57.37	60.00	53.42	27.38	27.28	26.74	9.37	9.08	9.14
	Halfspace	118.7	119.9	119.7	40.2	40.3	39.3	17.4	17.4	20.4
	L-tiling (CPU)	205	**203**	203	185	185	182	161	158	158
	H-tiling (CPU)	**146**	**188**	238	107	156	212	93	129	177
	2-xMC-Halfspace	272	284	296	118	136	**121**	**54**	**58**	**68**
WordNet Verbs	Lorentz	3.21	3.19	3.25	0.93	0.93	0.96	0.42	0.41	0.41
	Halfspace	5.65	5.68	5.71	1.69	1.66	1.66	0.78	0.75	0.77
	H-tiling (CPU)	**7.46**	**8.87**	**10.76**	**5.56**	7.85	8.54	5.06	6.63	8.59
	2-xMC-Halfspace	12.38	12.14	12.83	6.19	**5.86**	6.42	1.80	**1.84**	2.09
Gr-QC	Halfspace	4.29	4.36	4.31	2.13	1.91	1.88	0.59	0.63	0.67
	Lorentz	2.45	2.61	2.44	0.67	0.76	0.67	0.36	0.36	0.36
	H-tiling (CPU)	**5.99**	**7.22**	**10.12**	**4.39**	6.45	8.12	4.16	5.41	7.32
	2-xMC-Halfspace	9.52	9.96	10.41	5.42	5.57	5.94	**1.46**	1.63	**1.79**

Table 3.8: Training time (seconds per epoch) of models on various datasets. Results are averaged over 10 epochs. Performances of low precision models (Lorentz & Halfspace) are separated from that of high precision models in the table.

Training time cost. Running the model on GPUs confers important training advantage for modern machine learning tasks. In order to provide a comprehensive comparison result, we train most models in our experiments on GPUs including MCF-based half space models, upper-half space model and the Lorentz model. As for the *L*-tiling and *H*-tiling model, since big-integer matrix multiplications are not supported in GPUs, we train both models with CPUs. For all timing results within this section, we use GPU: GeForce RTX 2080 Ti and CPU: Intel(R) Xeon(R) Gold 6240 @2.60GHZ.

Note that negative sampling size and batch size can make a great difference to the timing performance over CPUs and GPUs. Herein, we fix the negative sampling size to be 50 and only vary the batchsize in 32, 64, 128. Readers may question the performances of the model as the batchsize changes, we note that with a simple scale of the learning rate, one can achieve comparable results for different batchsize, detailed in the Appendix. We report the training time (seconds per epoch) for different models on three different datasets in Table 3.8.

We can conclude that:

- On large-scale datasets or large batchsize (e.g. 128) training, MCF-based models (high precision) can be trained efficiently, with timing close to that of models using ordinary floating-point.
- MCF-based models (high precision) can run on GPUs, enjoying a great training speedup compared to other high precision models such as the *L*-tiling and *H*-tiling models which can only run over CPUs. As a matter of fact, on WordNet Nouns, relatively it takes MCF-based models around 2/3 less time and saves 30.27 hours to finish 1000 epochs training (batchsize = 128), compared to the H-Tiling model with best MAP performance.

3.3.4 MCTensor: A High-Precision Deep Learning Library

We further develop the MCF float representation approach into a multiple-component floating-point library, *MCTensor*, for general-purpose, fast, and high-precision deep learning. We build MCTensor on top of PyTorch and it can be used in the same way as the PyTorch Tensor object.

Our MCTensor library can benefit applications in the following areas: (1) high precision training with low precision numbers. Training with low precision numbers is an emerging deep learning area on tasks like mobile computer vision [169] and leveraging hardware accelerator like Google TPU [87], where the deep learning model computes with low precision numbers such as float8 and float16. A large quantization error using low precision arithmetic would affect the convergence [184] and may degrade the performance of the model; (2) numerical accurate and stable hyperbolic deep learning. For example, graph

embedding [122, 123] and many developed hyperbolic networks, including hyperbolic neural networks [60] and hyperbolic GCN [28, 196]. Our main contributions are as follows:

- We implement MCTensor in the same way as PyTorch Tensor with corresponding basic and matrix-level operations using MCF.
- We enable learning with MCTensor by developing the MCModule layers, MCOptimizers and etc with the same programming interface as PyTorch's counterparts.
- Beyond the usage of MCF/MCTensor on hyperbolic tasks (Section 3.3.3), we further demonstrate the performance of MCTensor for high precision training with low precision numbers.

Methodologies

Here we introduce some basics of our MCTensor library, built on top of PyTorch [130] [3] that employs multi-component floating-point as its underlying tensor representation. Each MCTensor x is represented as an expansion, an unevaluated sum of multiple tensors as follows:

$$x = (x_0, x_1, \cdots, x_{nc-1}) = x_0 + x_1 + ... + x_{nc-1} \tag{3.2}$$

where each x_i, as a component of x, can be a PyTorch floating-point Tensor in any precision, and nc is the number of components for MCTensor x. It's required that all components to be ordered in a decreasing magnitude (with x_0 being the largest and x_{nc-1} being the smallest). In this way, MCTensor allows roundoff

[3] Our PyTorch version is 1.11.0

error to be propagated to the later components and thus offers better precision compared to a standard PyTorch Tensor[4].

We first implement basic operators `add`, `subtract`, `multiply`, `divide`, ... for MCTensor with MCF arithmetic and further vectorize them to matrix-level operators `dot`, `mm`, ... with same semantics as their PyTorch counterparts. These operators then allow us to implement higher-level **MCModule, MCOptim** as the counterpart for `torch.nn.Module` and `torch.optim` so that we can use them for any deep learning applications.

MCTensor Object with Basic Operators

[MCTensor object] A MCTensor x can be abstracted as an object with specification `x{fc, tensor, nc}`. Specifically, `x.tensor` has nc components of PyTorch tensors $x_0, x_1, \cdots, x_{nc-1}$ in the last dimension, and it has shape as $(*x_0.\text{shape}, nc)$. The $x.fc$ data term in MCTensor is a view of x_0, keeps track of the gradient for x, and if needed, serves as an approximate tensor representation of x.

[Gradient] Because a MCTensor is an unevaluated sum of Tensors, then the gradient of a function f w.r.t. x is

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial (x_0 + x_1 + \cdots + x_{nc-1})} = \frac{\partial f}{\partial x_i},$$

which is same as the the gradient of f w.r.t. any component x_i. So we only keep track of the gradient information of `x.fc` for a MCTensor `x`, which can then be computed naturally by PyTorch's auto-differentiation engine to get `x.fc.grad` as `x.grad`.

[4] unless otherwise specified, the PyTorch tensor, or "Tensor", is referred to a PyTorch tensor with an arbitrary floating point data type in this paper

Algorithm 7 Grow-ExpN

Input: nc-MCTensor x, PyTorch Tensor v
initialize $Q \leftarrow v$
for $i = 1$ to nc **do**
  $k \leftarrow nc + 1 - i$
  $(Q, h_k) \leftarrow$ **Two-Sum**(x_{k-1}, Q)
end for
$h \leftarrow (Q, h_1, \cdots, h_{nc})$
Return: Simple-Renorm(h, nc)

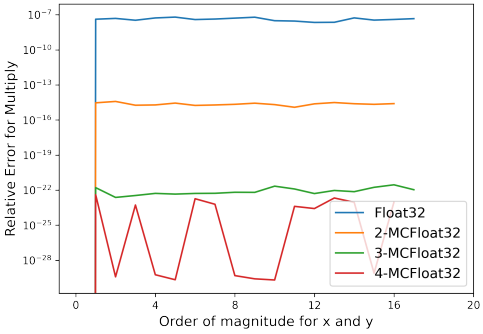


Figure 3.6: Relative Errors for Multiplication `Mul−MCN(x,y)`, Compared with High Precision Julia BigFloat (3000 bits precision). Order of magnitudes for `x` and `y` are the same.

[Basic Operators] We develop binary MCTensor operators to support input types (1) a MCTensor and a Tensor, (2) a Tensor and a MCTensor, and (3) a MCTensor and a MCTensor. For unary operators, we only accept a MCTensor as an input. The output for all these operators is a MCTensor with the same nc as the input(s). We implement MCF algorithms for basic arithmetic, with the full list of their algorithm statements available in the appendix:

- **MCTensor:** `Exp−MCN` (exp), `Square−MCN` (square)
- **MCTensor and Tensor:** `Grow−ExpN` (add), `ScalingN` (multiply), `DivN` (divide)
- **MCTensor and MCTensor:** `Add−MCN` (add), `Div−MCN` (divide), `Mul−MCN` (multiply)

For example, the addition between a MCTensor and a Tensor, or `Grow−ExpN` (Grow Expansion with Normalization), is given in Alg. 7. `Grow−ExpN` takes a nc-component MCTensor x and a normal Tensor v as input. The approximated result Q of this addition, is initialized to v. The Grow Expansion happens first in the for loop, starting with the last component of x, x_{nc-1}, to the first component

x_0. The algorithm `Two−Sum` sums Q and a component x_{k-1}, resulting in the updated approximation Q and the error term h_k. The resulting h is therefore grown into $nc + 1$ components, which are naturally ordered in a decreasing manner, except that there could be some intermediate zero components. Hence, in order to meet the requirements of a MCTensor, we use `Simple−Renorm` algorithm to move zeros backwards and output a nc components MCTensor.

We also implement two different algorithms for `Mul−MCN`: a fast version that uses two `Div−MCN` operations to perform multiplication, and a slower version that have better error bounds. However, in practice we see little difference between them, so we run all our experiments with the fast version. Details of these basic MCF algorithms are provided in the appendix.

To demonstrate how a MCTensor's precision increases with the number of components nc, we plot the relative numerical errors for MCTensor with different nc and PyTorch Tensor. We set the data type for both MCTensors and Tensors to be Float32, and compute the errors w.r.t. the results derived by high precision Julia BigFloat with number of bits of the significand set to 3000. As can be seen from Figure 3.6, even with $nc = 2$, the relative error is orders of magnitude smaller than PyTorch Tensors.

MCTensor Matrix Operators

After defining the basic MCF arithmetic, we are able to implement commonly used matrix level operators for MCTensor including `AddMM−MCN` (torch.addmm), `Dot−MCN` (torch.dot), `MV−MCN` (torch.mv), `MM−MCN` (torch.mm), `BMM−MCN` (torch.bmm) and `Matmul−MCN` (torch.matmul). Details of them are provided in the appendix.

Operators	Inputs sizes	FloatTensor	1-MCTensor	2-MCTensor	3-MCTensor
Dot-MCN	$5000, 5000$	$1.61\mu s \pm 3.29ns$	$442\mu s \pm 5.61\mu s$	$656\mu s \pm 1.16\mu s$	$858\mu s \pm 12.2\mu s$
MV-MCN	$(5000 \times 500), 500$	$157\mu s \pm 4.32\mu s$	$320ms \pm 5.78ms$	$460ms \pm 10.7ms$	$580ms \pm 12.1ms$
Matmul-MCN	$(500\times 200), (200\times 50)$	$97.3\mu s \pm 1.1\mu s$	$495ms \pm 10.8ms$	$735ms \pm 21.7ms$	$934ms \pm 28ms$

Table 3.9: MCTensor matrix operators running time (mean ± std)

For matrix operators, we leverage the broadcastability and vectorization from native PyTorch operations embedded in our basic MCF operators across each nc, and then apply sequential error propagation. For example, for `MV-MCN` with input as x and v, we first use `ScalingN` to compute the broadcasted product of x and v for each nc, and then we sequentially sum up the results and propagate errors with the `Add-MCN` operator. In this way, we can make our multiplication part (`ScalingN`) independent of the input size and only employ for-loop for addition part (`Add-MCN`) since error propagation in the addition is sequential by the algorithm. Theoretically, accurate addition of N MCTensors is of order at least $O(nc \cdot \log N)$.

MCTensor operators will be slower because of the need to propagate errors in computation, and have more memory burden than PyTorch operators because of the nature of MCF representation. In Table 3.9, we can see a tradeoff between program speed and precision. However, we would like to note that there is still much space to optimize these algorithms for better timing in practice. This work aims to provide ML community with the possibility to do high-precision computations for learning with MCTensor over GPUs. More details can be found in Appendix B.4.4.

```
class MCLinear(MCModule):
    def __init__(self, in_features, out_features, nc, bias=True):
        super(MCLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.nc = nc
        self.weight = MCTensor(out_features, in_features, nc=nc, requires_grad=True)
        if bias:
            self.bias = MCTensor(out_features, nc=nc, requires_grad=True)
        else:
            self.bias = None

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)
```

Figure 3.7: An example for implementing MCLinear

MCModule, MCActivation, and MCOptim

We enable learning with MCTensor by developing neural network basic modules (MCModule), activation functions (MCActivation) and optimizers (MCOptim), in the exact programming interfaces as their PyTorch counterparts in (`torch.nn.Module`, `torch.nn.functional` and `torch.optim`). The semantics are identical to that of PyTorch's module and optimizer, except that we are using MCTensor arithmetic. Specifically, we give some examples:

- **MCModule:** `MCLinear`, `MCEmbedding`, `MCSequential`
- **MCOptim:** `MC-SGD`, `MCAdam`
- **MCActivation:** `MCSoftmax`, `MCReLU`, `MC-GELU`

Here Figure 3.7 is the MCTensor implementation of linear layer, where weights and biases are represented by MCTensors. Similar to its PyTorch peer, a `MCLinear` takes input of size of input shape, in_features, size of output shape, out_features, learn with or without biases (boolean: bias) and number of component *nc* for setting up the underlying MCTensors. To handle the operations within MCLinear layer, we override PyTorch's `nn.functional.linear` to perform

```
mc_model = MCModel(nc=nc)
mc_optimizer = MCSGD(mc_model.parameters())

for x, y in train_dataset:
    mc_optimizer.zero_grad()
    y_hat = mc_model(x)
    loss = loss_fn(y_hat, y)
    loss.backward()
    mc_optimizer.step()
```

Figure 3.8: MCTensor model optimization programming paradigm

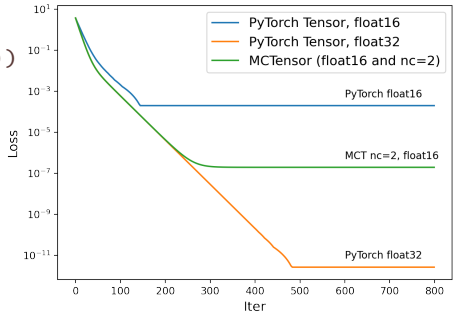


Figure 3.9: Loss curves on linear regression task.

matrix level multiplication (Algo. 39) between a MCTensor weight and the Tensor input, then the addition of the product with bias (Algo.11) if needed. The output of the MCLinear layer is a MCTensor with the same *nc*.

Since we implemented MCModule, MCActivation, and MCOptimizer to follow the same specification as their PyTorch counterparts, building a MCTensor model is the same way as one would build a PyTorch module. The only difference to the users is the need to specify the number of components *nc*. An demonstration of our MCTensor model optimization programming paradigm can be found in Fig. 3.8.

Error Analysis

Principally, one can achieve arbitrary precision using MCTensor by simply increasing the number of components, however, note that each component of a MCTensor is a standard float, which has a natural range. Take Float16 for example, the minimum representable strictly positive value is $2^{-24} \approx 5.96 \times 10^{-8}$, i.e., the smallest error that can be captured by MCFloat16 is 2^{-24}. Hence in practice, 2-MCFloat16 is usually sufficient and similar performances are observed when

more components of MCFloat16 were adopted. While simply adding an appropriate scale factor 2^{-k} (depending on the precision requirements) to smaller components can help capture even smaller errors, it is out of scope of this paper.

To validate improved precision of MCTensor models, we consider the linear regression task since it is possible to obtain loss arbitrarily close to zero. We use a single `MCLinear` without bias term and the Mean Squared Error (MSE) loss on fully observed synthetic data: $\mathcal{L}(W) = \text{MSELoss}(y, XW^T)$, where X is a (10000×2) matrix with each entries sampled from $x \sim \mathcal{N}(-0.5, 0.5^2)$, and y is calculated from $y = XW^{*T}$ where W^* is the (10000×1) target weight sampled from $w^* \sim \mathcal{N}(-0.5, 0.5^2)$. We use gradient descent with $lr = 0.05$ for optimization.

In Figure 3.9, we plot the training loss curves for the model with the same structure and initialization, but with MCTensor or Tensor as data structure. The comprehensive results can be seen in Table B.9. The final train loss for 2-MCFloat16 is orders of magnitudes smaller than Float16.

Since by just using 2-MCFloat16, we can achieve much better precision than Float16 Tensor (HalfTensor), we run all following experiments in Float16 with nc =2 or 3 expect for Hyperbolic MCEmbedding.

High-precision Computations with Low Precision

A MCTensor is able to achieve improved precision compared with a PyTorch Tensor with the same data type. If higher precision implies better results under the same experiment settings, we would expect our n-MCTensor ($n \geq 2$) model can achieve better performance than a native PyTorch model with same precision. We demonstrate the increasing precision under the same data type for

MCTensor by carrying out experiments in the setting: high precision learning with low precision numbers, this include logistic regression model and multi-layer perceptrons (MLP) in Section 3.3.4; For all these experiments, MCTensor models use the same initialization as the Tensor models, and use `MCModule` as its PyTorch's `nn.Module` counterpart, with MCTensors as layer weights. Our code and models are open-source and available at github [5].

Low precision machine learning employs models that compute with low precision numbers (e.g. float8 and float16), which become popular in many edge-applications [82, 209]. However, the quantization error inherent with low precision arithmetic could affect the convergence and performance of the model [184, 46]. As a matter of fact, most current low precision learning frameworks including [39, 46] adopt high precision numbers during gradient computations and optimizations, but pursue a better way to convert the results to low-precision numbers with less quantization errors. In comparison, with MCTensor, one can get *accurate* update with purely low-precision numbers thoroughly without even touching high precision arithmetics. This is particularly helpful on devices where only low-precision arithmetics are supported.

Logistic Regression. We conduct a logistic regression task on a synthetic dataset and the cancer dataset, both are datasets with binary labels. The synthetic dataset consists of $1,000$ data points, where each data point contains two features. This dataset is constructed through the `make_classification` [70] function from scikit-learn package with both `n_informative` and `n_clusters_per_class` are set to 1. The breast cancer dataset [120] consists of 569 data points and each data point contains 30 features. More details can be found in Appendix B.4.2.

[5] `https://github.com/ydtydr/MCTensor`

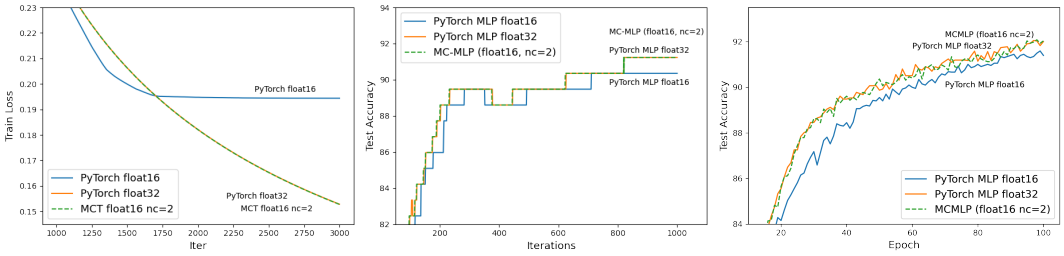


Figure 3.10: **(Left):** Training Losses of Logistic Regression on the Breast Cancer Dataset; Test Accuracy for MC-MLP on the Breast Cancer (**Middle**) and Reduced MNIST (**Right**). Note that the curve for MC-MLP model with nc=2, float16 essentially overlaps with the curve for PyTorch MLP model with float32.

Multi-layer Perceptron. We also construct a Multi-layer Perceptron using MCTensor (MC-MLP) and evaluate it on classification tasks on the breast cancer dataset and a reduced MNIST, which has only 10,000 data points in total, with 1,000 images sampled randomly per class [50]. The MC-MLP consists of three `MCLinear` layers with model weight in MCFloat16. After each `MCLinear` layer, the resulting MCTensor is transformed into a normal tensor, passed it to activation function and fed to the next layer.

We experiment on both the Breast Cancer dataset and the Reduced MNIST dataset, and as demonstrated in Table 3.10 and Figure 3.10, in both experiments, MC-MLP outperforms PyTorch float16 models, and arrive at a lower training loss and a higher test accuracy (same as PyTorch float32 and float64 models). Notice that for both datasets, after *nc* exceeds certain value ($nc = 2$), adding extra more *nc* would not lead to further improvement. More details can be found in Section B.4.2.

Model	Training Loss	Testing accuracy
MLP Float16	0.144	90.35
MLP Float32	**0.124**	**91.23**
MLP Float64	**0.124**	**91.23**
MC-MLP (nc=1)	0.144	90.35
MC-MLP (nc=2)	**0.124**	**91.23**
MC-MLP (nc=3)	**0.124**	**91.23**

Table 3.10: MC-MLP models on Breast Cancer dataset with MCSGD

3.4 Collage: Light-Weight Low-Precision Strategy for LLM Training

Overview

In this work, we elucidate that in the setting of low-precision (for example, 16-bit or lower) for floating point, using alternative representations such as multiple-component float (MCF) helps in making reduced precision accurate in computations, particularly for **low-precision training of large language models** (LLMs), which usually suffers from reduced numerical accuracy and renders less useful models.

We propose COLLAGE[6], a new approach to deal with floating-point errors in low-precision to make LLM training accurate and efficient. Our primary objective is to develop a training loop with storage strict in low-precision without a need to maintain high-precision clones. We realize that when dealing with low-precision floats (such as Bfloat16), the "standard" representation is not sufficient to avoid rounding errors which *should not be ignored*. To solve these issues, we rather apply an existing technique of MCF to represent floats which (i) either en-

[6]Inspired from the multi-component nature of the algorithm.

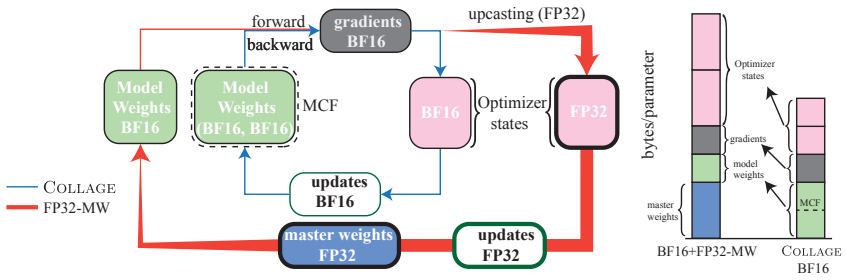


Figure 3.11: **Left:** COLLAGE uses a strict low-precision floating-point (such as BF16) optimization loop without ever needing to upcast to FP32 like in the mixed-precision with master weights (red thick loop). The model weights in COLLAGE are represented as multi-component float (MCF) instead of "standard float". **Right:** Total bytes/parameter savings for COLLAGE without taking the FP32 upcasting route. The memory savings and uncompromising use of low-precision results in speed-up as seen in Table 3.17.

counters drastic rounding effects, (ii) the scale of the involved floats has a wide range such that arithmetic operations were lost. We implemented COLLAGE as a plugin to be easily integrated with the well-known optimizers such as AdamW [108] (extensions to SGD [145] are straight-forward) using low-precision storage & computations. By turning the optimizer to be more *precision-aware*, even with additional low-precision components in MCF, we obtain faster training (upto 3.7× better train throughput on 6.7B GPT model, Table 3.17) and also have less memory foot-print due to strict low-precision floats (see Figure 3.11 right), compared to the most advanced mixed precision baseline.

We have developed a novel metric called "effective descent quality" to trace the lost information in the optimizer model update step. Due to rounding and lost arithmetic (see definition in Section 3.4.2), the effective update applied to the model is different from the intended update from optimizer, thus distracting the model training trajectory. Tracing this metric during the training enables to compare different precision strategies at a fine-grained level (see Figure 3.13 right).

In this work, we answer the critical question of where (which computation)

with low-precision during training is severely impacting the performance and why? The main contributions are outlined as follows.

- We provide COLLAGE as a **plugin** which could be easily integrated with existing optimizer such as AdamW for low-precision training and make it *precision-aware* by replacing critical floating-points with MCF. This avoids the path of high-precision master-weights and upcasting of variables, achieving memory efficiency (Figure 3.11 right).
- By proposing the metric effective descent quality, we measure loss in the information at model update step during the training process and provide **better understanding** of the impact of precisions and **interpretation** for comparing precision strategies.
- COLLAGE offers wall-clock time speedups by storing all variables in low-precision without upcasting. For GPT-6.7B and OpenLLaMA-7B, COLLAGE using bfloat16 has **up to 3.7×** speedup in the training throughput in comparison with mixed-precision strategy with FP32 master weights while following a similar training trajectory. The peak memory savings for GPTs (125M - 6.7B) is on average of **22.8%/14.9%** for COLLAGE formations (light/plus), respectively.
- COLLAGE **trains accurate** models using only low-precision storage compared with FP32 master-weights counterpart. For RoBERTa-base, the average GLUE accuracy scores differ by **+0.85%** among the best baseline in Table 3.14. Similarly, for GPT of sizes 125M, 1.3B, 2.7B, 6.7B, COLLAGE has **similar validation perplexity** as FP32 master weights in Table 3.15.

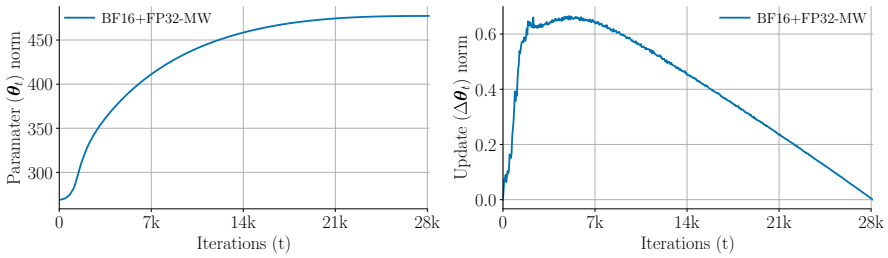


Figure 3.12: Bert-base-uncased phase-1 pretraining with settings as described in Section 3.4.4. **Left:** Model parameter L2 norm vs iterations for BF16 and FP32 master weights strategy. **Right:** update $\Delta\theta_t$ L2 norm across iterations. The model parameter norm and update norm are at different scales, for example, $\sim$ 450 vs $\sim$ 0.5 at 14k iterations, which is a factor of 900 and causes lost arithmetic.

3.4.1 Background

We provide a survey on using different floating-points precision strategies for training LLM. We also introduce necessary background information on floating-point representations using multi-component float.

Floats in LLM Training

In LLM training, weights, activation, gradients are usually stored in low precision floating-points such as 16-bit BF16 [114] for enhanced efficiency and optimized memory utilization. The low-bits floating point units (FPUs) are appealing because of its low memory foot-print and computational efficiency. Due to numerical inaccuracies, popular choices of training strategies using FPUs are as follows.

Mixed-precision refers to operations executed in low precision (16-bit) with minimal interactions with high precision (32-bits) floats, thus offering wall-clock

speedups. For example, in GEMM (Generalized Matrix Multiplication), matrix multiplication is performed in 16-bit while accumulation in done in 32-bit through tensor cores in NVIDIA A100 [86] and V100 [85].

Mixed-precision with Master Weights. Mixed-precision computations of the activations and gradients are not sufficient to ensure a stable training due to encountered numerical inaccuracies, especially, when gradients and model parameters are at different scale, which is the case with large models (see Figure 3.12). A standard workaround is to use the master weight (MW), which refers to maintaining an additional high-precision version (such as 32-bit float) copy of the model (Figure 3.11 left) and then performing model update (optimizer step) in high-precision to the master weight [115]. To our knowledge, this approach has the state-of-the-art performance among mixed-precision strategies.

Note that, we also use mixed-precision for GEMM (activations and gradients) in our work. In addition to "standard single float" representation which is used in the above strategies, an alternate form is discussed below in Section 3.4.1.

Multiple-Component Floating-point. Exact representations of real numbers such as 0.999 is usually muddled in low-precision, such as BF16, with rounding-to-the-nearest (RN); $0.999 \xrightarrow[\text{BF16}]{\text{RN}} 1.0$, but can be represented accurately as a length-2 expansion $(1.0, -0.001)$ in MCF with two BF16 components. The first component serves as an approximation to the value, while the second accounts for the roundoff error. This problem is further aggravated in weighted averaging (see Section 3.4.3), such that instead of the average, a monotonic increasing

sum is produced causing reduced step size and poor learning. We aim to alleviate such problems by using expansions to represent numbers and parameters accurately (e.g., Table 3.11). Since speed and scalability is critical for LLM training, we are particularly interested in utilizing low-precision MCF (e.g., BF16 and FP16) as low-bit FPUs are faster than their high-bit counterparts such as FP32. For rest of the work, we consider only length-2 expansion for MCF as it suffices for our purpose.

3.4.2 Imprecision Issues

To motivate the work, in this section, we formalize the issue of imprecision in floating point units. Afterwards, we introduce a novel metric to monitor the information loss. Next, we show its impact via a case study on BERT-like models [52, 107]. Unless specified otherwise, the low-precision FPU is referred to bfloat16, and the same analogy can be easily extended for other low-precision FPUs such as float16, float8.

Imprecision with Bfloat16

A commonly encountered problem of computations using low-precision arithmetic is *imprecision*, where an exact representation of a real-number x either requires more mantissa bits (see Appendix B.3.1 for definitions) beyond the limit (for example, 7 bits in bfloat16), or is not possible (for example, $x = 0.1$, is rounded to 0.1001 in BF16). As a result, the given number x will be rounded to a representable floating-point value, causing numerical quantization errors. An important concept for FPU rounding is unit in the last place (ulp), which is

the spacing between two consecutive representable floating-point numbers, i.e., the value the least significant (rightmost) bit represents if it is 1.

Definition 3.4.1 (ulp [119]). *In radix 2 with precision P, if $2^e \leq |x| < 2^{e+1}$ for some integer e, then* $\text{ulp}(x) = 2^{\max(e, e_{\min})-P}$, *where $e_{\min}$ is the zero offset in the IEEE 754 standard.*

Broadly speaking, two numbers for a given FPU are separated by its ulp, hence the worst case rounding error for any given x is $\text{ulp}(x)/2$ [63] assumed rounding-to-the-nearest is used. Next, lets denote $\mathcal{F}^{\text{BF16}}(a \varkappa b)$ as bfloat16 floating-point operation between a, b, where $\varkappa$ could be $\oplus$ addition, $\odot$ multiplication, etc. Such operations can be computationally inaccurate and as a consequence, we identify below a problematic behavior with RN.

Definition 3.4.2 (Lost Arithmetic). *Given the input floating-point numbers a, b and precision P. A floating operation $\mathcal{F}^P(a \varkappa b)$ is lost if*

$$|\mathcal{F}^P(a \varkappa b) - a| \leq \frac{\text{ulp}(a)}{2}, \text{ or } |\mathcal{F}^P(a \varkappa b) - b| \leq \frac{\text{ulp}(b)}{2}.$$

Consequently, $\mathcal{F}^P(a \varkappa b) = a$, or b, respectively.

Remark 2. *For any non-zero bfloat16 number, if $|b| \leq ulp(a)/2$, then $\mathcal{F}^{BF16}(a \oplus b) = a$. As an example, if $a = 200, b = 0.1$, then $\mathcal{F}^{BF16}(200 \oplus 0.1) = 200$, since $ulp(200) = 1$. Next, we discuss these concepts in the context of LLM training.*

Loss of Information in LLM Training

The situation of 'adding two numbers at different scale' is very common in LLM training. See Figure 3.12 , where due to different scales of model parameter

and updates, $\oplus$ in bfloat16 becomes an *lost arithmetic*. A pseudocode of model parameter ($\boldsymbol{\theta}$) update using bfloat16 at iteration t is written as

$$\boldsymbol{\theta}_t \leftarrow \mathcal{F}^{\text{BF16}}(\boldsymbol{\theta}_{t-1} \oplus \Delta\boldsymbol{\theta}_t), \tag{3.3}$$

where, $\Delta\boldsymbol{\theta}_t$ is the aggregated update from an optimizer (for example, including learning rate, momentum, etc.) at step t. With a possibility of lost arithmetic in Equation (3.3), the actual updated parameter could be different from expected. Hence, we define the effective update at step t as

$$\widehat{\Delta\boldsymbol{\theta}}_t = \mathcal{F}^{\text{BF16}}(\boldsymbol{\theta}_{t-1} \oplus \Delta\boldsymbol{\theta}_t) - \boldsymbol{\theta}_{t-1}. \tag{3.4}$$

Note that in the event of no lost arithmetic, $\widehat{\Delta\boldsymbol{\theta}}_t = \Delta\boldsymbol{\theta}_t$. While, when $\widehat{\Delta\boldsymbol{\theta}}_t \neq \Delta\boldsymbol{\theta}_t$ which is usually the case with low-precision FPUs, there is a loss in information as $\leq$ ulp /2 values are simply ignored (see Figure 3.13(a)). To better capture this information loss, we introduce a novel metric.

Definition 3.4.3 (**Effective Descent Quality**)**.** *Given the current parameter, aggregated update at step t as $\boldsymbol{\theta}_t$, $\Delta\boldsymbol{\theta}_t$, respectively. The effective descent quality for a given floating-pint precision is defined as*

$$\text{EDQ}(\Delta\boldsymbol{\theta}_t, \widehat{\Delta\boldsymbol{\theta}}_t; \boldsymbol{\theta}_t, P) = \Big\langle \frac{\Delta\boldsymbol{\theta}_t}{||\Delta\boldsymbol{\theta}_t||}, \widehat{\Delta\boldsymbol{\theta}}_t \Big\rangle, \tag{3.5}$$

where, $\widehat{\Delta\boldsymbol{\theta}}_t$ is defined in eq. (3.4) *for a given precision P.*

In other words, EDQ in eq. (3.5) is projection of the effective update along the desired update. In the absence of any imprecision, EDQ will be simply the norm of original update. We show in Section 3.4.4 and Figure 3.13 how EDQ relates to the learning and helps understanding impacts of different precision strategies.

To remedy the imprecision and lost arithmetic in the model parameter update step (Equation (3.3)), works such as Kahan summation [205, 127] exist (see Appendix B.3.1), however, we see in Figure 3.13 (Middle) that although Kahan-based BF16 approach improves over 'BF16' training but it still could not match with the commonly used FP32 master weights approach.

3.4.3 COLLAGE: Low-Precision MCF Optimizer

In this section, we present COLLAGE, a low precision strategy & optimizer implementation to solve aforementioned imprecision and lost arithmetic issues in Section 3.4.2 without upcasting to a higher precision, using the multiple-component floating-point (MCF) structure.

Computing with MCF

Precise computing with exact numbers stored as MCF expansions is easy with some basic algorithms[7]. For example, Fast2Sum captures the roundoff error for the float addition $\oplus$ and outputs an expansion of length 2.

Theorem 3.4.1 (Fast2Sum [48])**.** *Let two floating-point numbers a, b be $|a| \geq |b|$, Fast2Sum produces a MCF expansion (x, y) such that $a+b = x+y$, where $x \leftarrow \mathcal{F}^P(a \oplus b)$ is the floating-point sum with precision P, $y \leftarrow \mathcal{F}^P\left(b \ominus \mathcal{F}^P(x \ominus a)\right) = a + b - \mathcal{F}^P(a \oplus b)$ is the rounding error. Also, y is upper-bounded such that $|y| < \text{ulp}(x)/2$.*

Note that, particularly for LLM training, we are able to add using Fast2Sum without any sorting since parameter weights $\boldsymbol{\theta}$ are usually larger than the gra-

[7]The correctness of algorithms presented herein rely on the assumption that standard rounding-to-the-nearest is used.

Algorithm 8 Grow

Input: an expansion (x, y) and a float a with $|x| \geq |a|$
$(u, v) \leftarrow$ **Fast2Sum**(x, a)
$(u, v) \leftarrow$ **Fast2Sum**$(u, y + v)$
Return: (u, v)

dients and updates $\boldsymbol{\Delta\theta}$ in absolute value at the parameter update step Equation (3.3) (See Figure 3.12). Similar basic algorithms exist for the multiplication of two floats, which produces in the same way a length-2 expansion. Using the basic algorithms, an exhaustive set of advanced algorithms are developed [201]. We refer the reader to Appendix B.3.2 for more details. Particularly, for the optimizer update step (3.3), a useful algorithm to introduce is Grow (see Algorithm 8) which adds a float to a MCF expansion of length 2.

Algorithm 9 COLLAGE: Bfloat16 MCF AdamW Optimization

Given α (learning rate), $\beta_1, \beta_2, \epsilon, \lambda \in \mathbb{R}$
Initialize time step: $t \leftarrow 0$, BF16 parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, BF16 first moment vector: $\boldsymbol{m}_{t=0} \leftarrow \mathbf{0}$, BF16 second moment vector: $\boldsymbol{v}_{t=0} \leftarrow \mathbf{0}$
Initialize 2nd component $\boldsymbol{\delta\theta}_{t=0} \leftarrow \mathbf{0}$ in BF16 for parameter
(optional) Represent β_2 as expansion $(\hat{\beta}_2, \delta\beta_2)$,
+ initialize 2nd component $\boldsymbol{\delta v}_{t=0} \leftarrow \mathbf{0}$ in BF16 for second moment

repeat
  $t \leftarrow t + 1$
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$
  $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$
  $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1-\beta_2) \cdot \boldsymbol{g}_t^2 \implies (\boldsymbol{v}_t, \boldsymbol{\delta v}_t) \leftarrow \mathbf{Grow}(\mathbf{Mul}(\hat{\beta}_2, \delta\beta_2), (\boldsymbol{v}_{t-1}, \boldsymbol{\delta v}_{t-1})), (1 - \beta_2) \cdot \boldsymbol{g}_t^2)$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$
  $\boldsymbol{\Delta\theta}_t \leftarrow -\alpha(\hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t + \epsilon}) + \lambda\boldsymbol{\theta}_{t-1})$
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t \implies (\boldsymbol{\theta}_t, \boldsymbol{\delta\theta}_t) \leftarrow \mathbf{Grow}((\boldsymbol{\theta}_{t-1}, \boldsymbol{\delta\theta}_{t-1}), \boldsymbol{\Delta\theta}_t)$
until stopping criterion is met
return: optimized parameters $\boldsymbol{\theta}_t$

COLLAGE: Bfloat16 MCF AdamW

Using the basic components from Section 3.4.3 and Appendix B.3.2, we now provide plugins to modify a given optimizer such as AdamW [108] to be *precision-aware* and **store entirely with low-precision** floats, specifically bfloat16 in Algorithm 9. Note that, mixed-precision is still used in GEMM for obtaining gradients and activations but are stored in bfloat16 only. The required changes are highlighted in pink, and are discussed individually as follows.

Model Parameters We substitute the bfloat16 model parameter $\boldsymbol{\theta}_t$ with a length-2 MCF expansion $(\boldsymbol{\theta}_t, \boldsymbol{\delta\theta}_t)$ by appending an additional bfloat16 variable $\boldsymbol{\delta\theta}_t$ in line-3 which does not require any gradients. Next, to update the model parameter expansion, we use Grow in line-13 to add a float $\Delta\boldsymbol{\theta}_t$ to the expansion.

Optimizer States With Adam-like algorithms, unlike the first moment $\boldsymbol{m}_t$, the second moment $\boldsymbol{v}_t$ update suffers from severe imprecision and lost arithmetic due to smaller accumulation, $\boldsymbol{g}_t$ vs $\boldsymbol{g}_t^2$. To make the matter worse, default choice of β_2 such as 0.999 [52] are simply rounded to 1.0 in bfloat16, thus resulting in a monotonic increase in second momentum. This in turn makes the update $\Delta\boldsymbol{\theta}_t$ smaller and hence slower learning as we see in Figure 3.13. To alleviate this issue, we propose switching β_2 from standard single float to a MCF expansion as $(\beta_2, \delta\beta_2)$, and also for second momentum as $(\boldsymbol{v}_t, \boldsymbol{\delta v}_t)$. Doing so, we have an exact representation of β_2 as shown in Table 3.11. We then perform a multiplication of two expansions

Table 3.11: Length-2 expansions for β_2 in Bfloat16.

β_2	BF16 MCF
0.999	$(1, -0.001)$
0.99	$(0.9893, 0.0017)$
0.95	$(0.9492, 0.0008)$

Table 3.12: Precision breakdown of various training strategies applied to the given optimizer. The strategies are ranked from top to bottom in the order of byte/parameter occupancy.

Precision Option	Stages & Components			Memory (bytes/parameter)
	Parameter & Gradient	Optimizer States	MCF or Master Weight	
A (BF16)	BF16 × 2	BF16 × 2	NA	8
B (COLLAGE-light) (ours)	BF16 × 2	BF16 × 2	BF16 × 1	10
C (COLLAGE-plus) (ours)	BF16 × 2	BF16 × 2	BF16 × 2	12
D (BF16 + FP32Optim + FP32MW)	BF16 × 2	FP32 × 2	FP32 × 1	16

Table 3.13: Pre-training perplexity of BERT (both phases) and RoBERTa for all precision strategies as listed in Table 3.12. Lower values are better, with the best results in bold. D^{-MW} with FP32Optim with same bytes/parameter as COLLAGE could not match its performance.

Precision Option	$\beta_2 = 0.999$				$\beta_2 = 0.98$
	BERT-base		BERT-large		RoBERTa-base
	Phase-1	Phase-2	Phase-1	Phase-2	
A	8.67	7.61	6.05	5.47	3.82
B (COLLAGE-light)	5.99	5.26	4.39	3.90	3.49
C (COLLAGE-plus)	**5.26**	**4.66**	**3.94**	**3.53**	3.49
D^{-MW} (BF16 + FP32Optim)	6.23	5.64	4.66	4.22	3.82
D	**5.26**	4.71	4.06	3.63	**3.46**

using Mul (see Appendix B.3.2).

For the sake of simplicity in notations, we denote COLLAGE-light as using MCF expansions only for model parameters and COLLAGE-plus for both model parameters and optimizer states. It's worthy to note that imprecision and lost arithmetic are common and sometimes hard to notice. We only identify places when they hurt training accuracies. A rule of thumb is to do as many scalar computations in high precision as possible before casting them to low precision (e.g., PyTorch BFloat16 Tensor). Worthy to note, existing Kahan-based optimizers are special cases of COLLAGE-light under a magnitude assumption, we defer this discussion and other places of imprecision and lost arithmetic such as weight decay that exist in the algorithm to Appendix B.3.2.

3.4.4 Empirical Evaluation

We evaluate COLLAGE formations against the existing precision strategies on pretraining LLMs at different scales, including BERT [52], RoBERTa [107], GPT [10], and OpenLLaMA [166]. Specifically, we compared the following precision strategies in our experiments, which are ordered in an increasing number of byte/parameter (see Table 3.12).

- Option A: Bfloat16 parameters
- Option B: Bfloat16 + COLLAGE-light
- Option C: Bfloat16 + COLLAGE-plus
- Option D: Bfloat16 + FP32 Optimizer states + FP32 master weights

Since option D is the best-known baseline with state-of-the-art quality among mixed-precision strategies, we aim to outperform, or at least match the quality of option D with COLLAGE throughout our experiments. We show that COLLAGE matching the quality of option D, has **orders-magnitude higher** performance (speed, see Table 3.17). All strategies are evaluated using AdamW [108] optimizer with standard $\beta_1 = 0.9$ while varying β_2 as per different experiments. We use *aws.p4.24xlarge* compute instances for all of our experiments.

Pre-training BERT & RoBERTa

We demonstrate that BF16-COLLAGE can be used to obtain an accurate model, comparable to heavy-weighted FP32 master weights strategy.

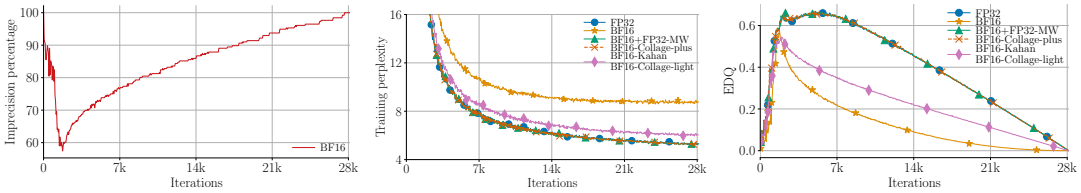


Figure 3.13: BERT phase-1 pre-training (see Appendix B.3.3 for details). **Left:** Imprecision percentage (%) measured as the percentage of lost arithmetic for all model parameters, i.e., not updated, vs iterations for BF16. **Middle:** Training perplexity vs iterations for various precision strategies (see Table 3.12). Additionally, we evaluate "FP32" as 32-bit counterpart of option A, and BF16-Kahan as Kahan-sum [205] with BF16 parameters. **Right:** Effective descent quality (EDQ) in (3.5) vs iterations to measure loss in information at the optimizer step for different precision strategies. BF16-COLLAGE-plus training perplexity and EDQ **overlaps** with the best "FP32", and "BF16 + FP32 MW" with less bytes/parameter.

Precision options. In addition to options A, B, C, D, we further augment our experiments with another baseline strategy D^{-MW}, where we disabled the FP32 master weights but only used FP32 optimizer states. This strategy saves 4 bytes/parameter in comparison to Option D and has the same bytes/parameter as option C (COLLAGE-plus).

Model and Dataset. We first pre-train the BERT-base-uncased, BERT-large-uncased, and RoBERTa-base model with HuggingFace (HF) [182] configuration on the Wikipedia-en corpus [11], preprocessed with BERT Wordpiece tokenizer. We execute the following pipeline to pretrain, i) BERT in two phases with phase-1 on 128 sequence length, and then phase-2 with 512 sequence length; and ii) RoBERTa with sequence length 512. We adopt $\beta_2 = 0.999$ for BERT and $\beta_2 = 0.98$ for RoBERTa following the configs from HF. We defer more training details to Appendix B.3.3.

Table 3.14: GLUE benchmark for BERT-base-uncased and RoBERTa-base pre-trained using different precision strategies. See Appendix B.3.3 for experimental details. BF16-COLLAGE training strategy matches/exceeds the finetuning quality over several metrics.

Model	Precision	MRPC	QNLI	SST-2	CoLA	RTE	STS-B	QQP	MNLI	Avg
BERT-base	A	0.8210	0.8832	0.8890	0.3522	0.6462	0.8666/0.8618	0.8973	0.7993	0.7796
	B (ours)	0.8431	0.8974	0.9071	0.4149	0.6606	0.8837/0.8785	0.9031	0.8184	0.8007
	C (ours)	0.8602	**0.9090**	**0.9128**	**0.4314**	0.6698	0.8851/0.8821	**0.9069**	**0.8330**	**0.8100**
	D	**0.8651**	0.9071	0.9036	0.4212	**0.6714**	**0.8890**/**0.8849**	0.9064	**0.8330**	0.8090
RoBERTa-base	A	0.8504	0.8914	0.9000	0.3866	0.6281	0.8636/0.8625	0.8981	0.8155	0.7884
	B (ours)	0.8455	0.9000	0.9025	0.4460	0.6281	0.8636/0.8635	0.9002	0.8182	0.7964
	C (ours)	**0.8529**	**0.9040**	**0.9048**	**0.4588**	0.6137	**0.8658**/**0.8647**	**0.9005**	**0.8230**	**0.7986**
	D	0.8406	0.8993	0.9002	0.3870	**0.6389**	0.8622/0.8631	0.8999	0.8203	0.7901

Results. The final pretraining perplexity of various precision strategies are summarized in Table 3.13 and for BERT-base, the complete phase-1 training loss trajectory is shown in Figure 3.13 middle. Additionally, we did finetuning of the pre-trained models on the GLUE benchmark [177] for eight tasks in Table 3.14 with the same configurations specified in Appendix B.3.3. COLLAGE-plus although using only BF16 parameters, outperforms the vanilla BF16 option A and matches/exceeds option D for both pre-training and finetuning experiments. For BERT-base COLLAGE-plus exceeds on **5/8** tasks with **+0.1**% lead in average, while for roberta-base its exceeds on **7/8** tasks with **+0.85**% in average. Note that, although D^{-MW} has FP32 optimizer states and same/more byte/parameter complexity as COLLAGE-plus/light, respectively, it could not match the quality **showing the importance of MCF** in the AdamW through COLLAGE. This shows that simply having higher-precision is not enough to obtain better models but requires a *careful consideration of the floating errors*.

Interestingly, COLLAGE-light suffices to closely match the option D in the RoBERTa pretraining experiments with $\beta_2 = 0.98$, while lagging to match with the $\beta_2 = 0.999$ BERT pretraining experiments. Our proposed metric, the effective descent quality (EDQ) provides a nuanced understanding of this phenomenon

Table 3.15: **Left:** Train | Validation perplexity of pre-trained GPT with $\beta_2 = 0.95$. **Right:** OpenLLaMA-7B with $\beta_2 = 0.95$ and 0.99.

Model	GPT				OpenLLaMA-7B	
Precision Option	125M	1.3B	2.7B	6.7B	$\beta_2 = 0.95$	$\beta_2 = 0.99$
A (BF16)	14.73 | 15.64	10.28 | 12.43	9.97 | 12.18	9.87 | 12.18	6.36 | 4.81	15.96 | 12.55
B (COLLAGE-light)	14.01 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.53	8.00 | 5.99
C (COLLAGE-plus)	14.01 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.57	6.11 | 4.62
D (BF16 + FP32Optim + FP32MW)	13.87 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.57	8.58 | 6.42

in Figure 3.13 (Right). COLLAGE-light and Kahan-based approach improve EDQ upon BF16 option A at the parameter update step, yet cannot achieve the optimal EDQ due to lost arithmetic at the exponential moving averaging step. In contrast, COLLAGE-plus achieves better EDQ by taking it into considerations and thereby outperforms the best-known baseline, Option D.

Pretraining multi-size GPTs & OpenLLaMA 7B

Model and Dataset. We conduct following pretraining experiments; 1) GPT with different sizes ranging from 125M, 1.3B, 2.7B to 6.7B, and 2) OpenLLaMA-7B using NeMo Megatron [97] with the provided configs. The GPTs are trained on the Wikipedia corpus [11] with GPT2 BPE tokenizer, and OpenLLaMA-7B on the LLaMA tokenizer, respectively. Additional training and hyerparameter details are described in Appendix B.3.3.

Results. Using the recommended $\beta_2 = 0.95$ [10], Table 3.15 summarizes the train & validation perplexity after pre-training GPT models and OpenLLaMA-7B under various options. Our COLLAGE formations are able to **match** the quality of the **best-known** baseline, FP32 MW option D, most of the time *for all models* with the only exception on the smallest GPT-125M, while having the same validation perplexity.

Table 3.16: Train | Validation perplexity of GPT-125M pre-trained with $\beta_2 \in \{0.95, 0.99, 0.999\}$ and Global BatchSize $\in \{1024, 2048\}$.

Precision Option	Global BatchSize = 1024			Global BatchSize = 2048		
	$\beta_2 = 0.95$	$\beta_2 = 0.99$	$\beta_2 = 0.999$	$\beta_2 = 0.95$	$\beta_2 = 0.99$	$\beta_2 = 0.999$
A (BF16)	14.73 | 15.64	14.88 | 15.80	17.29 | 18.17	14.73 | 15.18	14.88 | 15.33	17.64 | 15.33
B (COLLAGE-light)	14.01 | 15.03	14.01 | 15.03	14.88 | 15.80	13.87 | 14.44	13.87 | 14.44	14.59 | 15.18
C (COLLAGE-plus)	14.01 | 15.03	14.01 | 15.03	14.15 | 15.18	13.87 | 14.44	13.87 | 14.44	14.01 | 14.59
D (BF16 + FP32Optim + FP32MW)	13.87 | 15.03	14.01 | 15.03	14.01 | 15.03	13.87 | 14.44	13.87 | 14.44	14.01 | 14.59

Ablation: Impact of β_2. We conduct ablation experiments to illustrate the impact of β_2 on the quality of precision strategies by further pre-training the GPT-125M model using $\beta_2 = 0.99$ and 0.999, with a global batchsize 1024, 2048 and the same micro-batchsize 16, as summarized in Table 3.16. Similar to the BERT and RoBERTa pre-training experiments, COLLAGE-light is able to closely match Option D when $\beta_2 = 0.95$ or 0.99 and remain unaffected by changes in the global batchsize.

However, with $\beta_2 = 0.999$, COLLAGE-light underperforms option D while COLLAGE-plus is still able to closely match option D. As analyzed in Section 3.4.3, low precision (Bfloat16) arithmetic fails to represent and compute with $\beta_2 = 0.999$ due to rounding errors. In fact, we observed the same phenomenon as pre-training BERT & RoBERTa in Section 3.4.4, including i) a high imprecision percentage of lost additions with low-precision BF16 arithmetic; ii) a reduced EDQ for COLLAGE-light and a better EDQ for COLLAGE-plus. These together rationalize the utility and significance of our proposed metric EDQ and the necessity of COLLAGE-plus for quality models. We defer figures of these metrics for GPTs to Appendix B.3.3.

We also pretrain OpenLLaMA-7B with $\beta_2 = 0.99$ in Table 3.15 (right), where both COLLAGE formations outperform option D. In fact, we observe that $\beta_2 = 0.99$ can easily lead to gradient explosion (see Figure B.4 right in Ap-

Table 3.17: Relative speed-up compared to the option D.

Precision		GPT		OpenLlama
Option	1.3B	2.7B	6.7B	7B
A	1.78×	2.59×	3.82×	3.15×
B (ours)	1.74×	2.57×	3.74×	3.14×
C (ours)	1.67×	2.48×	3.57×	3.05×
D	1×	1×	1×	1×

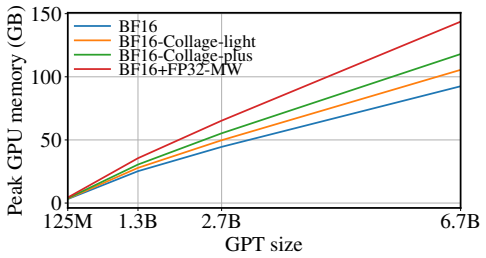


Figure 3.14: GPU peak memory in GB vs model size. GPT-125M is hosted on 1 NVIDIA A100 40GB, while all other models were hosted on 8× A100 40GB using tensor-parallelism 8.

pendix B.3.3), while COLLAGE-plus provides stable training. The training perplexity trajectories in Figure B.3,B.4 (in Appendix B.3.3) show that COLLAGE-plus effectively solves the imprecision issue and produces quality models.

Remark 3. *The optimal choice of β_2 differs case-by-case. To our best knowledge, there is no clear conclusion between β_2 and the converged performance of the pre-trained models. Showing* COLLAGE *works with different β_2's, enable LLM training to be not limited by such precision issues.*

Performance and Memory

Throughput. We record the mean training throughput of precision strategies for pre-training GPTs and OpenLLaMA-7B in a simple setting for fair comparisons: one *aws.p4.24xlarge* node with sequence parallel [95] turned off[8], and present relative speed-up in Table 3.17. Both COLLAGE formations are able to maintain the efficiency of option A. The speed factor for COLLAGE increases with an increase in the model size, obtaining up to **3.74×** for GPT-6.7B model.

[8]We observed similar throughputs for precision strategies when sequence parallel is turned on

Table 3.18: Memory compatibility of pre-training GPT-NeoX-30B using precision options with different micro batchsize (UBS) and sequence length.

Precision option / SeqLen	UBS= 1		UBS= 2	
	1,024	2,048	1,024	2,048
A (BF16)	✓	✓	✓	✓
B (COLLAGE-light)	✓	✓	✓	OOM
C (COLLAGE-plus)	✓	✓	✓	OOM
D (BF16 + FP32Optim + FP32MW)	✓	OOM	OOM	OOM

Memory. We probe the peak GPU memory of training precision strategies during practical runs on 8×NVIDIA A100s (40GB) with the same hyper-parameters for a fair comparison: sequence length 2048, global batchsize 128 and micro (per-device) batchsize 1. Figure 3.14 visualizes the peak memory usage of GPTs vs model sizes. During real runs, on average, COLLAGE formations (light/plus) use **23.8%/15.6%** less peak memory compared to option D. The best savings are for the largest model OpenLLaMA-7B, with savings **27.8%/18.5%**, respectively.

Increased Sequence Length and Micro BatchSize. We study the benefits of COLLAGE's reduced memory foot-print (as shown in Figure 3.14), with a demonstration on pre-training a large GPT-30B model with tensor-parallelism=8, pipeline-parallelism=2 on two *aws.p4.24xlarge* (8×A100s 40GB) instances. Specifically, we identify the maximum sequence length and micro batchsize for all precision strategies to be able to run without OOM, as summarized in Table 3.18. COLLAGE enables training with an increased sequence length and micro batchsize compared to option D, thus providing a smooth trade-off between quality and performance.

Remark 4. *Further improvements on throughput and memory can be achieved for* COLLAGE *with specialized fused kernels.*

APPENDIX A

EXPRESSIVE HIERARCHICAL DATA MODELING WITH HYPERBOLIC GEOMETRY

A.1 Partial Orders Embedding in Hyperbolic Shadow Cones

A.1.1 Shadow Cones' Boundary Characterization

In this part, we explain intuitively the boundary computation of shadow cones and the reason why Umbral cones are not geodesically convex.

Penumbral shadow, which results from partial eclipse of a light source with volume, is produced by a point object. Therefore, the boundaries of penumbral shadow are geodesics passing through u while being tangent to the light source. Note that the Penumbral Shadow(v) $\subset$ Penumbral Shadow(u) for any $v \in$ Penumbral Shadow(u), therefore, all possible children of u, i.e., penumbral cone of u coincides with the penumbral shadow of u, with geodesics boundary described above.

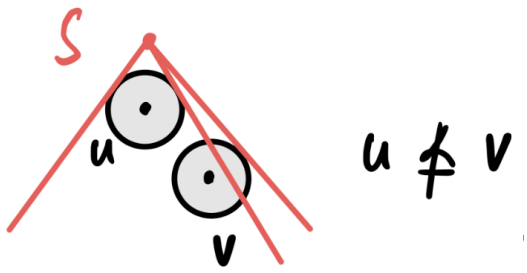


Figure A.1: An example illustrating the necessity for v's ball to be entirely immersed in u's umbral shadow, in order that $u \preceq v$. In the marginal case shown here, $u \not\preceq v$ even though the point v is in u's umbral shadow. This is because part of v's ball and corresponding umbral shadow is outside of u's umbral shadow.

On the other hand, Umbral shadow, which results from total eclipse of a

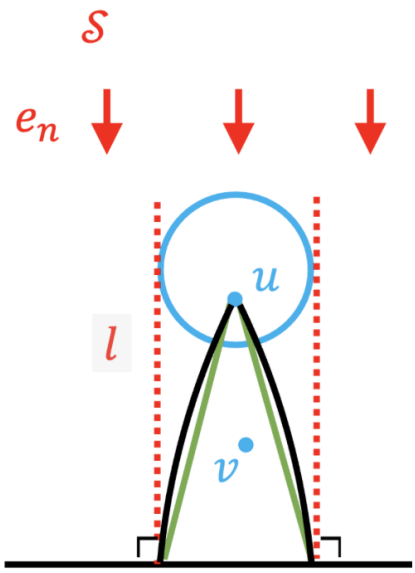


Figure A.2: Umbral shadow cone (green) of u in half-space model with light source $\mathcal{S}$ at infinity, where e_n is the propagation direction of lights. l (dotted red) denotes the boundary of umbral shadow casted by u and its ball. Solid green lines denote the boundary of the umbral cone of u, while solid black lines delineate the geodesically convex hull of u's umbral cone. This figure shows that umbral cones are not geodesically convex, because they are strict subsets of their geodesically convex hulls.

point light source, is produced by an object with volume (a ball around u). Therefore, the boundary of umbral shadow is geodesics passing through the point light source, while being tangent to the ball around u. Now, let's look at all possible children v of u, i.e., umbral cone of u. Since the objects u and v have volumes, then for $u \preceq v$ to be true, it must be that v and its ball are entirely immersed in u's shadow. Otherwise, the portion of v's ball outside of u's shadow can cast shadows outside. In Fig A.1, we sketch a case where v's center is in u's shadow, but nevertheless $u \not\preceq v$, because part of v's ball is outside.

Hence, the boundary of umbral cones are the set of points equidistant from the umbral shadow, aka, hypercycles in hyperbolic space, a well-studied geometric object. In Euclidean space, the set of points equidistant from a straight

line forms another, parallel straight line. In contrast, in hyperbolic space, the set of points equidistant from a geodesic do *not* necessarily form a geodesic, and is instead referred to as a hypercycle. In Fig A.2, we illustrate the non-convexity of umbral cones in Poincaré half-space. In green are the umbral cone boundaries, while in black are its convex hull - the minimum convex set - that contains the umbral cone. The boundaries of the convex hull are geodesics. We can see that the umbral cone is a strict subset of its convex hull, which means the umbral cone is not geodesically convex.

A.1.2 Proofs for Theorems/Claims in Section 2.2

Isometry Mapping Light Source to Origin

For the Poincaré ball model, when the light source of shadow cones is in the space but not at the origin, we utilize the following isometry to map the light source to the origin.

Definition A.1.1 (Isometry [197])**.** *Let* $\text{Inv}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{k\|\boldsymbol{x}\|^2}$*, then the map*

$$T_{\mathcal{S}}(\boldsymbol{x}) = -\mathcal{S} + (1 - \|\mathcal{S}\|^2)\,\text{Inv}(\text{Inv}(\boldsymbol{x}) - \mathcal{S})$$

is an isometry of the Poncaré ball, which maps $\mathcal{S}$ *to the origin* $\boldsymbol{O}$*, i.e.,* $T_{\mathcal{S}}(\mathcal{S}) = \boldsymbol{O}, T_{\mathcal{S}}^{-1}(\boldsymbol{O}) = \mathcal{S}$*.*

Since isometries preserve distance, geodesics and also equal distance curves (i.e., hypercycles), then both the umbral and penumbral cones are preserved under isometries. Therefore, when the light source $\mathcal{S}$ is not at the origin, we apply the isometry T_S to all embeddings, then the energy function can be computed accordingly as the light source at the origin case.

Proof of Transitivity & Geodesic Convexity

In this section, we provide proofs to the transitivity of the partial orders induced by umbral and penumbral cones, together with the proof of penumbral cones' convexity. We first give several equivalent definitions to umbral and penumbral cones

1. $\boldsymbol{u} \preceq \boldsymbol{v}$ iff $\boldsymbol{v}$ (and its ball) is in the shadow of $\boldsymbol{u}$ (and its ball).
2. $\boldsymbol{u} \preceq \boldsymbol{v}$ if the shadow of $\boldsymbol{v}$ (and its ball) is a subset of the shadow of $\boldsymbol{u}$ (and its ball).
3. $\boldsymbol{u} \preceq \boldsymbol{v}$ if every geodesic between the light source $\mathcal{S}$ and $\boldsymbol{v}$ (and its ball) passes through $\boldsymbol{u}$ (or its ball).

Wherein, the parentheses refer to the umbral cone case. We will adopt the 3rd definition within this section.

Proof of transitivity. We start with the umbral cone case, suppose that $\boldsymbol{x} \preceq \boldsymbol{y}$ and $\boldsymbol{y} \preceq \boldsymbol{z}$, then every geodesic between $\mathcal{S}$ and $\boldsymbol{y}$'s ball passes through $\boldsymbol{x}$'s ball, and every geodesic between $\mathcal{S}$ and $\boldsymbol{z}$'s ball passes through $\boldsymbol{y}$'s ball. Now consider any geodesic between $\mathcal{S}$ and $\boldsymbol{z}$'s ball, which must passes through the $\boldsymbol{y}$'s ball, but then it is also a geodesic between $\mathcal{S}$ and $\boldsymbol{y}$'s ball, which must pass through $\boldsymbol{x}$'s ball, that is $\boldsymbol{x} \preceq \boldsymbol{z}$.

For penumbral cones, suppose that $\boldsymbol{x} \preceq \boldsymbol{y}$ and $\boldsymbol{y} \preceq \boldsymbol{z}$. Consider the geodesic submanifold (isometric to the hyperbolic plane) passing through $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$. Since $\boldsymbol{x} \preceq \boldsymbol{y}$, then the geodesic from $\boldsymbol{y}$ through $\boldsymbol{x}$ intersects the light source. Similarly, the geodesic from $\boldsymbol{z}$ through $\boldsymbol{y}$ intersects the light source. Denote these

intersection points on the boundary of $\mathcal{S}$ as $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively, then consider the geodesic ray from $\boldsymbol{z}$ passing through $\boldsymbol{x}$, as it passes through $\boldsymbol{z}$, it either enters or exits the triangle $\triangle \boldsymbol{aby}$. Since $\boldsymbol{x}$ is on the line segment $\boldsymbol{ya}$ now, it can't be exiting the triangle, because $\boldsymbol{y}$ is on the line segment $\boldsymbol{zb}$, so $\boldsymbol{z}$ was already outside the triangle. Therefore, it must be entering the triangle and it must exit the triangle at some point along one of the other sides.

Note that it can't exit along the side $\boldsymbol{yb}$, because $\boldsymbol{z}$ is already on that line, and it can't intersect the line twice, so it must exit along the side $\boldsymbol{ab}$, but that side is entirely within the light source, because $\boldsymbol{a}$ and $\boldsymbol{b}$ are on the light source and the light source is convex. therefore $\boldsymbol{x} \preceq \boldsymbol{z}$.

Worthy to mention, in the proof of penumbral cones, we used the fact that $\mathcal{S}$ is convex, so is the intersection of $\mathcal{S}$ with any geodesic submanifold of dimension 2. In fact, we only need the intersection with any geodesic submanifold of dimension 2 to be connected, but convexity of $\mathcal{S}$ suffices. □

Proof of geodesic convexity for penumbral cones. This can be proved following a similar pattern as last proof. Suppose that $\boldsymbol{x} \succeq \boldsymbol{y}$ and $\boldsymbol{x} \succeq \boldsymbol{z}$. Let $\boldsymbol{w}$ be any point on the geodesic line segment $\boldsymbol{yz}$, again we consider the geodesic submanifold passing through $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$, which also passes through $\boldsymbol{w}$. The geodesic from $\boldsymbol{y}$ through $\boldsymbol{x}$ intersects the light source, so is the geodesic from $\boldsymbol{z}$ through $\boldsymbol{x}$. Denote these intersection points as $\boldsymbol{a}, \boldsymbol{b}$ respectively. Now consider the geodesic from $\boldsymbol{w}$ through $\boldsymbol{x}$, which is contained between geodesics $\boldsymbol{xy}$ and $\boldsymbol{xz}$, so it will also be contained at the other side of them, i.e., $\boldsymbol{xa}$ and $\boldsymbol{xb}$. Also $\boldsymbol{x}$ is one vertex of the triangle $\triangle \boldsymbol{abx}$, so the geodesic $\boldsymbol{wx}$ enters the triangle $\triangle \boldsymbol{abx}$, then it must exit the triangle at some point along one of its sides. Since $\boldsymbol{wx}$ intersects $\boldsymbol{xa}$ and $\boldsymbol{xb}$ at $\boldsymbol{x}$, so it can't exit along $\boldsymbol{xa}$ and $\boldsymbol{xb}$ sides, because it can't intersect a line

twice. Therefore, it must exit along the side $\boldsymbol{ab}$, which is entirely within the light source, because the light source is convex. Therefore, $\boldsymbol{wx}$ intersects with the light source at some point, i.e., $\boldsymbol{w} \geq \boldsymbol{x}$. □

Equivalence of Penumbral-Poincaré-ball and [60]'s entailment cone construction

Proof. [60] defines the entailment cones in the Poincaré ball model (of curvature -1) as

$$\{y \in \mathbb{B}^n | \phi(y, x) \leq \arcsin(K \frac{1 - ||x||^2}{||x||})\},$$

where $\phi(y, x)$ is the angle between the half-line $(xy$ and $(ox$.

On the other hand, Penumbral-poincaré-ball is defined as

$$\{y \in \mathbb{B}^n | \phi(y, x) \leq \theta_x = \arcsin(\frac{\sinh r}{\sinh d_{\mathbb{H}}(x, O)})\}.$$

With

$$d_{\mathbb{H}}(x, O) = \operatorname{arcosh}\left(1 + \frac{2||x||^2}{1 - ||x||^2}\right),$$

we get

$$\theta_x = \arcsin(\frac{\sinh r}{\sinh \operatorname{arcosh}\left(1 + \frac{2||x||^2}{1-||x||^2}\right)}) = \arcsin(\frac{\sinh r}{2} \frac{1 - ||x||^2}{||x||}),$$

then the penumbral-Poincaré-ball cone is

$$\{y \in \mathbb{B}^n | \phi(x, y) \leq \theta_x = \arcsin(\frac{\sinh r}{2} \frac{1 - ||x||^2}{||x||})\},$$

set $K = \frac{\sinh r}{2}$ (since r is a user-defined hyperparameter), the entailment cone reduces to penumbral-Poincaré-ball cone, a special case of shadow cones. □

More Details on Shortest Hyperbolic Distance to Shadow Cones

Umbral-half-space Cones

In the main paper, we provide the shortest hyperbolic distance to the umbral-half-space cone, here we give a detailed derivation of the formulas. We start by giving the logarithm map in the Poincaré half-space model [200], let $\boldsymbol{v} = \log_{\boldsymbol{x}}(\boldsymbol{y})$, then

$$v_i = \frac{x_n}{y_n}\frac{s}{\sinh s}(y_i - x_i)$$
$$v_n = \frac{s}{\sinh s}(\cosh s - \frac{x_n}{y_n})x_n,$$

where $s = d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y})$.

Note that the hyperbolic ball of radius r centered at $\boldsymbol{u}$ in Poincaré half-space corresponds to an Euclidean ball with center $\boldsymbol{c_u} = (u_1, \ldots, u_{n-1}, u_n \cosh r)$ and radius $r_e = u_n \sinh r$. Note that boundaries of the umbral cone induced by $\boldsymbol{u}$ are hypercycles with axis l, where l belongs to the set of light paths that are tangent to the boundary of $\boldsymbol{u}$'s ball: $\{(x_1, \ldots, x_{n-1}, t) | \sum_{i=1}^{n-1}(x_i - u_i)^2 = r_e^2, t > 0\}$. In order to derive the signed hyperbolic distance from $\boldsymbol{v}$ to the boundary of the umbral cone, it suffices to compute the signed distance of $\boldsymbol{v}$ to such a l since hypercycles are equal-distance curves.

Now we derive the shortest signed hyperbolic distance from $\boldsymbol{v}$ to such an l. Let $\boldsymbol{w}$ be a point on l such that $d_{\mathbb{H}}(\boldsymbol{v}, l) = d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{w})$, then clearly the geodesic from $\boldsymbol{w}$ through $\boldsymbol{v}$ is orthogonal to l at $\boldsymbol{w}$, i.e.,

$$(\log_{\boldsymbol{w}}(\boldsymbol{v}))_n = 0 \implies \cosh s = w_n/v_n \implies d_{\mathbb{H}}(\boldsymbol{v}, l) = d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{w}) = \operatorname{arcosh}(w_n/v_n).$$

Note that the geodesic from $\boldsymbol{w}$ through $\boldsymbol{v}$ is a half-circle-style geodesic with center on the 0-hyperplane. Since it's orthogonal to l (a vertial line), hence the center

of the geodesic is in fact $(w_1, \ldots, w_{n-1}, 0)$, then we have

$$w_n^2 = \sum_{i=1}^{n-1} (w_i - v_i)^2 + v_n^2,$$

by computing the radius to $\boldsymbol{v}$ and $\boldsymbol{w}$ respectively. Meanwhile, we have the following ratio between coordinates:

$$\frac{v_i - w_i}{v_i - u_i} = 1 - \frac{r_e}{\sqrt{\sum_{i=1}^{n-1} (v_i - u_i)^2}}.$$

From both equations we can derive that

$$\begin{aligned} w_n^2 &= v_n^2 + (1 - \frac{r_e}{\sqrt{\sum_{i=1}^{n-1} (v_i - u_i)^2}})^2 \sum_{i=1}^{n-1} (v_i - u_i)^2 \\ &= v_n^2 + (\sqrt{\sum_{i=1}^{n-1} (v_i - u_i)^2} - r_e)^2. \end{aligned}$$

Hence, the signed shortest distance from $\boldsymbol{v}$ to the boundary Cone($\boldsymbol{u}$) is

$$r + \operatorname{arcosh}(w_n / v_n) = r + \operatorname{arcosh}(\sqrt{1 + (\sqrt{\sum_{i=1}^{n-1} (v_i - u_i)^2} - r_e)^2 / v_n^2})$$

Note that $\operatorname{arcosh}(\sqrt{1 + t^2}) = \operatorname{arsinh}(t),\ \forall t \geq 0$, where the latter is a desired signed distance, then let

$$t = \left(\sqrt{\sum_{i=1}^{n-1} (u_i - v_i)^2} - u_n \sinh r \right) / v_n$$

be a temperature function, we derive the signed shortest distance from $\boldsymbol{v}$ to the boundary Cone($\boldsymbol{u}$) is

$$r + \operatorname{arsinh}(t).$$

In order to derive the relative altitude function of $\boldsymbol{v}$ respect to $\boldsymbol{u}$, consider when the shortest geodesic from $\boldsymbol{v}$ to the boundary of Cone($\boldsymbol{u}$) is attained at $\boldsymbol{u}$, which will be orthogonal to the boundary at $\boldsymbol{u}$, using the property of half-circle-stlye geodesic, we have that

$$v_n^2 + (\sqrt{\sum_{i=1}^{n-1} (u_i - v_i)^2} - r_e)^2 = r_e^2 + u_n^2 = u_n^2 (1 + \sinh^2 r) = u_n^2 \cosh^2 r,$$

that is, $v_n^2(1+t^2) = u_n^2 \cosh^2 r$, hence, a natural choice of the relative altitude function is $H(\boldsymbol{v}, \boldsymbol{u}) = v_n^2(1+t^2) - u_n^2 \cosh^2 r$. Then the shortest signed distance from $\boldsymbol{v}$ to Cone($\boldsymbol{u}$) is $d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})$ when $H(\boldsymbol{v}, \boldsymbol{u}) > 0$ and $r + \text{arsinh}(t)/$ when $H(\boldsymbol{v}, \boldsymbol{u}) \leq 0$.

Umbral-Poincaré-ball Cones

Similarly for umbral-Poincaré-ball cone, we have

Lemma A.1.1 (Umbral-Poincaré-ball)**.** *Denote α as the angle between $\boldsymbol{u}, \boldsymbol{v}$, and β as the maximum angle spanned by the hyperbolic ball of radius r associated with $\boldsymbol{u}$, then*

$$\alpha = \arccos \frac{\boldsymbol{u}^\intercal \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}, \quad \beta = \arcsin \frac{r}{\sinh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{O})\right)} = \arcsin \frac{2r\|\boldsymbol{u}\|}{1 - \|\boldsymbol{u}\|^2}.$$

Set the temperature as

$$t = \sinh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right) \sin(\alpha - \beta) = \frac{2\|\boldsymbol{v}\|}{1 - \|\boldsymbol{v}\|^2} \sin(\alpha - \beta),$$

then the relative altitude function of $\boldsymbol{v}$ with respect to $\boldsymbol{u}$ is

$$\begin{aligned} H(\boldsymbol{v}, \boldsymbol{u}) &= \text{arcosh}\left(\frac{\cosh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right)}{\sqrt{1+t^2}}\right) - \text{arcosh}\left(\cosh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{O})\right) \cosh(r)\right), \\ &= \text{arcosh}\left(\frac{1}{\sqrt{1+t^2}} \frac{1 + \|\boldsymbol{v}\|^2}{1 - \|\boldsymbol{v}\|^2}\right) - \text{arcosh}\left(\cosh(r) \frac{1 + \|\boldsymbol{u}\|^2}{1 - \|\boldsymbol{u}\|^2}\right). \end{aligned}$$

In fact, boundaries of the umbral cone induced by $\boldsymbol{u}$ are hypercycles with axis l, where l belongs to the set of light paths that are tangent to the boundary of $\boldsymbol{u}$'s ball. We compute the signed distance of $\boldsymbol{v}$ to such a l in the Poincaré ball model, which is easier since the line $\boldsymbol{Ov}$ and l are geodesics, where hyperbolic laws of sines can be applied.

Denote α as the angle between $\boldsymbol{u}, \boldsymbol{v}$, and β as the maximum angle spanned by the hyperbolic ball of radius r associated with $\boldsymbol{u}$, then

$$\alpha = \arccos \frac{\boldsymbol{u}^\intercal \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}, \quad \beta = \arcsin \frac{r}{\sinh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{O})\right)} = \arcsin \frac{2r\|\boldsymbol{u}\|}{1 - \|\boldsymbol{u}\|^2}.$$

where the equation of β is a result of hyperbolic laws of sines. Again using the hyperbolic laws of sines, we derive the signed distance from $\boldsymbol{v}$ to l as

$$d_{\mathbb{H}}(\boldsymbol{v}, l) = \operatorname{arsinh}\left(\sinh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right)\sin(\alpha - \beta)\right),$$

therefore, we set the temperature as

$$t = \sinh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right)\sin(\alpha - \beta) = \frac{2\,\|\boldsymbol{v}\|}{1 - \|\boldsymbol{v}\|^2}\sin(\alpha - \beta),$$

then the signed shortest distance to the boundary of $\mathrm{Cone}(\boldsymbol{u})$ is simply $\operatorname{arsinh}(t) + r$. In order to derive the relative altitude function, we consider the altitude of $\boldsymbol{v}$, which is simply the projection of $\boldsymbol{v}$ to l, using hyperbolic laws of cosines, we have

$$\cosh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right) = \cosh\left(d_{\mathbb{H}}(\boldsymbol{v}, l)\right)\cosh\left(H(\boldsymbol{v})\right),$$

Similarly for the altitude of $\boldsymbol{u}$,

$$\cosh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{O})\right) = \cosh\left(d_{\mathbb{H}}(\boldsymbol{u}, l)\right)\cosh\left(H(\boldsymbol{u})\right) = \cosh(r)\cosh\left(H(\boldsymbol{u})\right),$$

combining them togeterm, the relative altitude function $H(\boldsymbol{v}, \boldsymbol{u}) = H(\boldsymbol{v}) - H(\boldsymbol{u})$ is

$$\begin{aligned} H(\boldsymbol{v}, \boldsymbol{u}) &= \operatorname{arcosh}\left(\frac{\cosh\left(d_{\mathbb{H}}(\boldsymbol{v}, \boldsymbol{O})\right)}{\sqrt{1 + t^2}}\right) - \operatorname{arcosh}\left(\cosh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{O})\right)\cosh(r)\right), \\ &= \operatorname{arcosh}\left(\frac{1}{\sqrt{1 + t^2}}\frac{1 + \|\boldsymbol{v}\|^2}{1 - \|\boldsymbol{v}\|^2}\right) - \operatorname{arcosh}\left(\cosh(r)\frac{1 + \|\boldsymbol{u}\|^2}{1 - \|\boldsymbol{u}\|^2}\right). \end{aligned}$$

Then the shortest signed distance from $\boldsymbol{v}$ to $\mathrm{Cone}(\boldsymbol{u})$ is $d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})$ when $H(\boldsymbol{v}, \boldsymbol{u}) > 0$ and $r + \operatorname{arsinh}(t)$ when $H(\boldsymbol{v}, \boldsymbol{u}) \leq 0$.

Penumbral Cones

For penumbral cones, things are simpler since the boundary of penumbral cones are geodesics, where we can freely apply the hyperbolic laws of sines to hyperbolic triangles.

Theorem A.1.2 (Shortest Distance to Penumbral Cones)**.** *For penumbral cones, the temperature is defined as* $t = \phi(\boldsymbol{v}, \boldsymbol{u}) - \theta_{\boldsymbol{u}}$*, where* $\phi(\boldsymbol{v}, \boldsymbol{u})$ *is the angle between the cone central axis and the geodesic connecting* $\boldsymbol{u}, \boldsymbol{v}$ *at* $\boldsymbol{u}$*,* $\theta_{\boldsymbol{u}}$ *is half-aperture of the cone. The relative altitude function is* $H(\boldsymbol{v}, \boldsymbol{u}) = t - \pi/2$*, and the shortest distance from* $\boldsymbol{v}$ *to the penumbral cone induced by* $\boldsymbol{u}$ *is*

$$d(\boldsymbol{v}, \mathit{Cone}(\boldsymbol{u})) = \begin{cases} d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v}) & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) > 0, \\ \operatorname{arsinh}\left(\sinh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})\right) \sin t\right) & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) \leq 0, \end{cases}$$

where the second formula represents the signed-distance-to-boundary.

We derive the relative altitude function first, note that $H(\boldsymbol{v}, \boldsymbol{u}) = 0$ when the geodesic from $\boldsymbol{v}$ through $\boldsymbol{u}$ is orthogonal to one boundary of the cone at $\boldsymbol{u}$, with simple geometry, the relative altitude function is $H(\boldsymbol{v}, \boldsymbol{u}) = t - \pi/2$. We derive the signed-distance-to-boundary $d_{\mathbb{H}}(\boldsymbol{v}, l)$ by simply applying the hyperbolic laws of sines:

$$\frac{\sinh d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})}{\sin(\pi/2)} = \frac{\sinh d_{\mathbb{H}}(\boldsymbol{v}, l)}{\sin t},$$

then we get that $d_{\mathbb{H}}(\boldsymbol{v}, l) = \operatorname{arsinh}\left(\sinh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})\right) \sin t\right)$. In summary, the shortest distance from $\boldsymbol{v}$ to the penumbral cone induced by $\boldsymbol{u}$ is

$$d(\boldsymbol{v}, \mathrm{Cone}(\boldsymbol{u})) = \begin{cases} d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v}) & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) > 0, \\ \operatorname{arsinh}\left(\sinh\left(d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})\right) \sin t\right) & \text{if } H(\boldsymbol{v}, \boldsymbol{u}) \leq 0, \end{cases}$$

where the second formula represents the signed-distance-to-boundary.

A.1.3 Training Details and Additional Experiments

Data pre-processing and statistics. WordNet data are directly taken from [60], which was already a DAG. MCG and Hearst are taken from [181, 185] and [75]

Table A.1: Statistics of partial order datasets

	Mammal	Noun	MCG	Hearst
Depth	8	18	31	56
# of Nodes	1,179	82,114	22,665	35,545
# of Components	4	2	657	2,133
# of Basic Relations	1,176	84,363	38,288	42,423
# of All Relations	5,361	661,127	1,134,348	6,846,245

respectively. Since these datasets are orders of magnitudes larger than WordNet, we take the $50,000$ relations with the highest confidence score. We note that there are numerous cycles even in the truncated MCG and Hearst graphs. To obtain DAGs from these graphs, we randomly remove 1 relation from a detected cycle until no more cycles are found. The resulting four DAGs have vastly different hierarchy structures. To roughly characterize these structures, we use longest path length as a proxy for the depth of DAG and number of components (disconnected sub-graphs) as the width.

While the WordNet Noun and MCG datasets are trained with a maximum of 50% non-basic edges, we limit Hearst training to 5%. This is due to Hearst's transitive closure being more than $10\times$ larger than that of WordNet Noun, despite having comparable numbers of basic relations. We note that a data set's complexity is better reflected by the size of its basic edges than that of the transitive closure, as the latter scales quadratically with depth. For instance, although Hearst possesses only half as many basic relations as Noun, its transitive closure is tenfold larger. This can be attributed to its depth - 56 compared to Noun's 18 and MCG's 31. Full data statistics can be found in A.1.

Negative sampling. Negative samples for testing: For each positive pair in the transitive closure (u, v), we create 10 negative pairs by randomly selecting 5

nodes v', and 5 nodes u' such that the corrupted pairs (u, v') and (u', v) are not present in the transitive closure. As a result, this 'true negative set' is ten times the size of the transitive closure. We choose negative set size equals 10 as a result of following [60].

Negative samples for training are generated in a similar fashion: For each positive pair (u, v), randomly corrupt its edges to form 10 negative pairs (u, v') and (u', v), while ensuring that these negative pairs do not appear in the training set. We remark that since the training set is not the full transitive closure, these dynamically generated negative pairs are impure as they might include non-basic edges.

Burnin. Following [122, 123], we adopt a burnin stage for 20 epochs at the beginning of training for better initialization, where a smaller learning rate (burnin-multiplier $0.01\times$) is used. After the burnin stage, the specified learning rate is then used ($1\times$).

Hyper-parameters and Optimization. On WordNet Noun and Mammal, we train shadow cones and entailment cones for 400 epochs, following [60]. On MCG and Hearst, we train shadow cones and entailment cones for 500 epochs since due to their increased hierarchy depth. A training batchsize of 16 is used for all datasets and models. For the margin parameters in shadow loss, we use $\gamma_2 = 0$ consistently for all experiments. We tune γ_1 and the learning rate in $\{0.01, 0.001, 0.0001\}$. For umbral cones, we tune the source radius r in $\{0.01, 0.05, 0.1, 0.2, 0.3\}$, empirically $r = 0.05$ during training gives the optimal performance when evaluated under a slightly larger radius $r = 0.1$. For penumbral-half-space cones, we tune the exponentiated height e^h in $\{2, 5, 10, 20\}$,

where empirically 20 during training gives the optimal performance, which validates our assumption that the height of penumbral-half-space cones can limit its performance. Note that for shadow cones, when $H(\boldsymbol{v}, \boldsymbol{u}) > 0$, the shortest distance to the cone is $d_{\mathbb{H}}(\boldsymbol{u}, \boldsymbol{v})$, which will only pull $\boldsymbol{v}$ to be close to $\boldsymbol{u}$, apex of the shadow cone, but not pull $\boldsymbol{v}$ into the shadow cone. To solve this issue, we use $d_{\mathbb{H}}(\boldsymbol{u}', \boldsymbol{v})$ when $H(\boldsymbol{v}, \boldsymbol{u}) > 0$, where $\boldsymbol{u}'$ is derived by pushing $\boldsymbol{u}$ into its shadow cone along the central axis for a distance γ_3. We set $\gamma_3 = 0.0001$ consistently for all shadow cones. We use HTorch [202] for optimization in various models of hyperbolic space. We use RiemannianSGD for Poincaré half-space model, and RiemannianAdam for Poincaré ball model due to the ill-conditioned initialization results from the hole around the origin, where RiemannianAdam offers a much faster convergence than RiemannianSGD.

Ease of Training. Our training routine is less complicated than that of the Entailment Cone [60]: 1) While the entailment cone uses pretrained 100-epoch embedding as initialization in order to avoid the ϵ-hole issue in the Poincaré-ball, we simply use random initialization around the origin as a one-stage approach. 2) Our half-space based cones and embeddings have no ϵ-hole issue at the origin, and need no extra steps to project the "stray" points out of the hole.

Additional Experiments

We would also like to supplement with a simpler experiment that demonstrates how partial order embeddings capture meaningful structures in data. Given a shadow cone embedding of mammal taxonomy, we are interested in predicting the order of a species. For example, the order of American beaver is rodents,

and the order of Bonobo is primates. To construct the classification task, we first find all the nodes in the mammal graph that represent order names. This yields 10 categories, including ungulate, rodent, cetacean, etc. Then, we identify all the species names as the leaf nodes with no outgoing links. Lastly, we predict the category of each species using a softmax probability distribution based on distance. Specifically, we measure the distance between the cone central axis of the species node and those of the 10 category nodes. In the Poincaré-half space model, this distance is simply:

$$d(x, y) = \text{arcosh}(1 + \frac{||\mathbf{x} - \mathbf{y}||^2}{2x_n y_n}),$$

where we x_n and y_n are both fixed to be 1. Therefore, this distance could also be thought of as the hyperbolic distance confined to the $x_n = 1$ plane.

We note that the shadow cone embedding optimizes for the entailment relations among nodes, and does not directly optimize this classification task and its loss. Yet a simple distance-based classification achieved an accuracy 94.80% with a 5D Umbral half-space cone and 73.23% with a 2D Umbral half-space cone.

Interpretation of light source radius

We modified our existing framework to learn light source radius from data. Fig A.3 and A.4 show the learned embedding as well as light source radius. For the mammal dataset, the source radius quickly expands from its initial size, resulting in a converged radius that is very large compared to the distribution of learned embeddings. We speculate that this might be a result of the connectivity of the underlying data: the mammal graph is highly interconnected, featuring

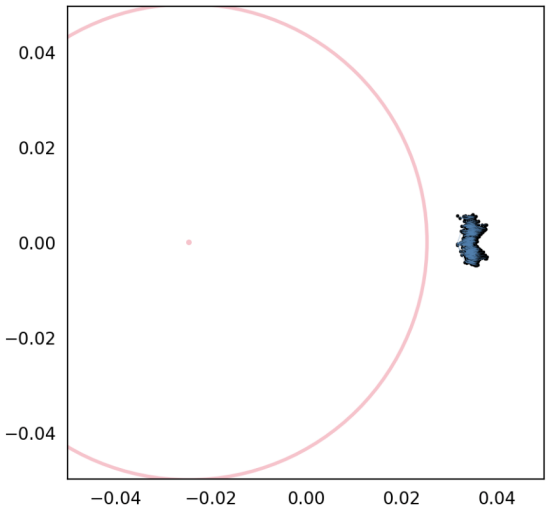


Figure A.3: A global view of learned Mammal subgraph embeddings with a trainable light source radius, using Penumbral-Poincaré-ball.

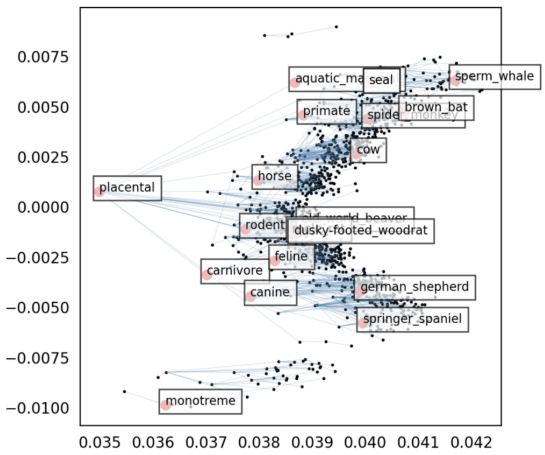


Figure A.4: A zoomed-in view of learned Mammal subgraph embeddings with a trainable light source radius, using Penumbral-Poincaré-ball.

only four disconnected components. A large light source can cast large penumbral shadows, making it easier to model highly connected DAGs. We would be interested in exploring whether different data can induce radius of different sizes, and quantify if there is a relation between source radius and graph connectivity.

A.2 HyLa: Random Hyperbolic Laplacian Features

A.2.1 Proofs for Theorems in Section 2.3

Theorem A.2.1 ([76, 206]). *All smooth eigenfunctions of the Euclidean Laplacian* Δ *on* $\mathbb{R}^n$ *are*

$$f(\boldsymbol{x}) = \int_{S^{n-1}} e^{i\lambda\langle\omega,\boldsymbol{x}\rangle} dT(\omega),$$

where $\lambda \in \mathbb{C} - \{0\}$ and T is an analytic functional (or hyperfunction), i.e., an element of the dual space of the space of analytic functions on S^{n-1}.

Theorem A.2.2 ([76, 206])**.** *All smooth eigenfunctions of the Hyperbolic Laplacian $\mathcal{L}$ on $\mathcal{B}^n$ are*

$$f(z) = \int_{\partial \mathcal{B}^n} \exp((i\lambda + \frac{n-1}{2})\langle \omega, z \rangle_H)\, dT(\omega),$$

where $\lambda \in \mathbb{C}$ and T is an analytic functional (or hyperfunction).

Lemma A.2.3 (Expression of $k(\boldsymbol{x}, \boldsymbol{O})$)**.** *Denote $\zeta_{\lambda,\omega}(z) = \exp\left((\frac{n-1}{2} + i\lambda)\langle \omega, z \rangle_H\right)$, then the corresponding kernel $k_\lambda(\boldsymbol{x}, \boldsymbol{O})$ for a particular value of λ defined as*

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \mathbb{E}\left[\mathrm{HyLa}_{\lambda,b,\omega}(\boldsymbol{x})\, \mathrm{HyLa}_{\lambda,b,\omega}(\boldsymbol{y})\right] = \frac{1}{2} \cdot \mathop{\mathbb{E}}_{\omega}\left[\zeta_{\lambda,\omega}(\boldsymbol{O})^* \zeta_{\lambda,\omega}(\boldsymbol{x})\right] = \frac{1}{2} \cdot \mathop{\mathbb{E}}_{\omega}\left[\zeta_{\lambda,\omega}(\boldsymbol{x})\right] \tag{A.1}$$

takes the form

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{O}))\right)\right),$$

where the expectation in Equation A.1 is taken over $\boldsymbol{\omega}$ drawn uniformly from the n-dimensional unit sphere.

Proof. Recall that for $\boldsymbol{x}$ in the n-dimensional Poincare ball $\mathcal{B}^n$ and $\boldsymbol{\omega} \in \partial\mathcal{B}^n$,

$$\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle_H = \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{\|\boldsymbol{x} - \boldsymbol{\omega}\|^2}\right).$$

Let $\gamma = \frac{n-1}{2} + i\lambda$, expanding Equation A.1 out,

$$\begin{aligned}
k_\lambda(\boldsymbol{x}, \boldsymbol{O}) &= \frac{1}{2} \cdot \mathop{\mathbb{E}}_{\omega}\left[\exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{\|\boldsymbol{x} - \boldsymbol{\omega}\|^2}\right)\right)\right] \\
&= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \mathop{\mathbb{E}}_{\omega}\left[\exp\left(\gamma \log\left(\frac{1 + \|\boldsymbol{x}\|^2}{\|\boldsymbol{x} - \boldsymbol{\omega}\|^2}\right)\right)\right] \\
&= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \mathop{\mathbb{E}}_{\omega}\left[\left(\frac{\|\boldsymbol{x} - \boldsymbol{\omega}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)^{-\gamma}\right] \\
&= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \mathop{\mathbb{E}}_{\omega}\left[\left(\frac{\|\boldsymbol{x}\|^2 - 2\boldsymbol{\omega}^T\boldsymbol{x} + \|\boldsymbol{\omega}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)^{-\gamma}\right],
\end{aligned}$$

and if we let

$$u = \frac{-2\,\|\boldsymbol{x}\|}{1 + \|\boldsymbol{x}\|^2},$$

and let $\hat{\boldsymbol{x}}$ denote the unit vector in the direction of $\boldsymbol{x}$, then this becomes

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\left(1 + u\hat{\boldsymbol{x}}^T\boldsymbol{\omega}\right)^{-\gamma}\right].$$

Expanding this out using the Binomial Formula (which is valid here because $|u| < 1$), we get

$$\begin{aligned} k_\lambda(\boldsymbol{x}, \boldsymbol{O}) &= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\sum_{k=0}^{\infty} \binom{-\gamma}{k} u^k \left(\hat{\boldsymbol{x}}^T\boldsymbol{\omega}\right)^k\right] \\ &= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \binom{-\gamma}{k} u^k \mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\left(\hat{\boldsymbol{x}}^T\boldsymbol{\omega}\right)^k\right]. \end{aligned}$$

Since this expected value is 0 for odd k, this becomes

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \binom{-\gamma}{2k} u^{2k} \mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\left(\hat{\boldsymbol{x}}^T\boldsymbol{\omega}\right)^{2k}\right].$$

But $\hat{\boldsymbol{x}}^T\boldsymbol{\omega}$ has the same the distribution as the inner product of two uniform random unit vectors in n dimensions. The square of this is well known to be Beta-distributed with parameters $(\frac{1}{2}, \frac{n-1}{2})$. So we can write this as

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \binom{-\gamma}{2k} u^{2k} \mathop{\mathbb{E}}_{w}\left[w^k\right],$$

where $w \sim \text{Beta}(\frac{1}{2}, \frac{n-1}{2})$. Of course, the moments of the Beta distribution are well-known to be

$$\mathop{\mathbb{E}}_{w}\left[w^k\right] = \frac{\left(\frac{1}{2}\right)^{(k)}}{\left(\frac{n}{2}\right)^{(k)}},$$

where $x^{(k)}$ denotes the Pochhammer symbol representing the rising factorial. On the other hand,

$$\binom{-\gamma}{2k} = \frac{(-\gamma)_{(2k)}}{(2k)!}$$

where $x_{(k)}$ denotes the Pochhammer symbol representing the falling factorial. Since $x_{(k)} = (-1)^k(-x)_{(k)}$, we can write this in terms of rising factorials as

$$\binom{-\gamma}{2k} = \frac{(\gamma)^{(2k)}}{(2k)!}.$$

So substituting everything in, we have

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \frac{(\gamma)^{(2k)}}{(2k)!} \cdot u^{2k} \cdot \frac{\left(\frac{1}{2}\right)^{(k)}}{\left(\frac{n}{2}\right)^{(k)}}.$$

Next, observe that

$$\left(\frac{1}{2}\right)^{(k)} = \prod_{m=0}^{k-1}\left(\frac{1}{2} + m\right) = 2^{-k} \prod_{m=0}^{k-1}(2m+1) = 4^{-k} \cdot \frac{(2k)!}{k!}.$$

So this becomes

$$k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \frac{(\gamma)^{(2k)}}{k!} \cdot u^{2k} \cdot 4^{-k} \cdot \frac{1}{\left(\frac{n}{2}\right)^{(k)}}.$$

Next, we leverage the famous identity that

$$\gamma^{(2k)} = 4^k \left(\frac{\gamma}{2}\right)^{(k)} \left(\frac{\gamma+1}{2}\right)^{(k)}$$

to get

$$\begin{aligned} k_\lambda(\boldsymbol{x}, \boldsymbol{O}) &= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \sum_{k=0}^{\infty} \frac{1}{k!} \cdot \left(\frac{\gamma}{2}\right)^{(k)} \cdot \left(\frac{\gamma+1}{2}\right)^{(k)} \cdot u^{2k} \cdot \frac{1}{\left(\frac{n}{2}\right)^{(k)}} \\ &= \frac{1}{2} \cdot \exp\left(\gamma \log\left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)\right) \cdot {}_2F_1\left(\frac{\gamma+1}{2}; \frac{\gamma}{2}; \frac{n}{2}; u^2\right). \end{aligned}$$

Since

$$u^2 = \frac{4\|\boldsymbol{x}\|^2}{1 + 2\|\boldsymbol{x}\|^2 + \|\boldsymbol{x}\|^4},$$

it follows that

$$1 - u^2 = \frac{1 - 2\|\boldsymbol{x}\|^2 + \|\boldsymbol{x}\|^4}{1 + 2\|\boldsymbol{x}\|^2 + \|\boldsymbol{x}\|^4} = \left(\frac{1 - \|\boldsymbol{x}\|^2}{1 + \|\boldsymbol{x}\|^2}\right)^2$$

and

$$\frac{\sqrt{1-u^2} - 1}{2\sqrt{1-u^2}} = \frac{-\|\boldsymbol{x}\|^2}{1 - \|\boldsymbol{x}\|^2}.$$

Recall the classic formula that

$$ {}_2F_1\left(a, a+\frac{1}{2}; c; z\right) = (1-z)^{-a}\, {}_2F_1\left(2a, 2c-2a-1; c; \frac{\sqrt{1-z}-1}{2\sqrt{1-z}}\right). $$

Substituting $z = u^2$, $a = \frac{\gamma}{2}$, and $c = \frac{n}{2}$ yields

$$ \begin{aligned} k_\lambda(\boldsymbol{x}, \boldsymbol{O}) &= \frac{1}{2} \cdot {}_2F_1\left(\gamma, n-\gamma-1; \frac{n}{2}; \frac{-\|\boldsymbol{x}\|^2}{1-\|\boldsymbol{x}\|^2}\right) \\ &= \frac{1}{2} \cdot {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{-\|\boldsymbol{x}\|^2}{1-\|\boldsymbol{x}\|^2}\right). \end{aligned} $$

Therefore,

$$ \frac{-\|\boldsymbol{x}\|^2}{1-\|\boldsymbol{x}\|^2} = \frac{-\tanh^2(D/2)}{1-\tanh^2(D/2)} = \frac{-\tanh^2(D/2)}{\operatorname{sech}^2(D/2)} = -\sinh^2(D/2) = \frac{1}{2}\left(1-\cosh(D)\right), $$

where $D = d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{O})$. So we get a final expression

$$ k_\lambda(\boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2} \cdot {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1-\cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{O}))\right)\right). $$

The proof is done here. We further make a remark here. Recall that for the Poincaré ball model,

$$ \begin{aligned} d(\boldsymbol{x}, \boldsymbol{O}) &= 2\log\left(\frac{\|\boldsymbol{x}\|+1}{\sqrt{1-\|\boldsymbol{x}\|^2}}\right) \\ &= \log\left(\frac{(\|\boldsymbol{x}\|+1)^2}{1-\|\boldsymbol{x}\|^2}\right) \\ &= \log\left(\frac{1+\|\boldsymbol{x}\|}{1-\|\boldsymbol{x}\|}\right) \\ &= 2\,\operatorname{artanh}(\|\boldsymbol{x}\|). \end{aligned} $$

Therefore,

$$ \begin{aligned} \log\left(\frac{1-\|\boldsymbol{x}\|^2}{1+\|\boldsymbol{x}\|^2}\right) &= \log\left(\frac{1-\tanh^2(D/2)}{1+\tanh^2(D/2)}\right) \\ &= \log\left(\frac{\operatorname{sech}^2(D/2)}{1+\tanh^2(D/2)}\right) \\ &= \log\left(\frac{1}{\sinh^2(D/2)+\cosh^2(D/2)}\right) \\ &= \log\left(\frac{1}{\sinh^2(D/2)+\cosh^2(D/2)}\right) \\ &= -\log\left(\cosh(D)\right). \end{aligned} $$

And on the other hand,

$$u = \frac{-2\tanh(D/2)}{1+\tanh^2(D/2)} = \tanh(D).$$

then we can also write the kernel as

$$k_\lambda(\boldsymbol{x},\boldsymbol{O}) = \frac{1}{2}\cdot\exp\left(-\left(\frac{n-1}{2}+i\lambda\right)\log\left(\cosh(D)\right)\right)\cdot {}_2F_1\left(\frac{n+1}{4}+i\frac{\lambda}{2},\frac{n-1}{4}+i\frac{\lambda}{2};\frac{n}{2};\tanh(D)^2\right),$$

where $D = d_{\mathbb{H}}(\boldsymbol{x},\boldsymbol{O})$. □

Lemma A.2.4 (Isometry-invariance)**.** *The kernel defined by*

$$k_\lambda(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\omega}\left[\zeta_{\lambda,\omega}(\boldsymbol{x})^*\zeta_{\lambda,\omega}(\boldsymbol{y})\right] = \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\omega}\left[\exp\left((\frac{n-1}{2}-i\lambda)\langle\boldsymbol{x},\boldsymbol{\omega}\rangle_H + (\frac{n-1}{2}+i\lambda)\langle\boldsymbol{y},\boldsymbol{\omega}\rangle_H\right)\right]$$

is isometry-invariant.

Proof. Let g be any isometry of the space, then observe the geometric identity [76]:

$$\langle g\circ\boldsymbol{x}, g\circ\boldsymbol{\omega}\rangle = \langle\boldsymbol{x},\boldsymbol{\omega}\rangle + \langle g\circ\boldsymbol{O}, g\circ\boldsymbol{\omega}\rangle. \tag{A.2}$$

Take $\boldsymbol{\omega} = g^{-1}\circ\boldsymbol{\omega}$ in Equation A.2, it follows that

$$\langle g\circ\boldsymbol{x},\boldsymbol{\omega}\rangle = \langle\boldsymbol{x}, g^{-1}\circ\boldsymbol{\omega}\rangle + \langle g\circ\boldsymbol{O},\boldsymbol{\omega}\rangle.$$

Take $\boldsymbol{x} = g^{-1}\circ\boldsymbol{O}$ in Equation A.2, it follows that

$$0 = \langle\boldsymbol{O},\boldsymbol{\omega}\rangle = \langle g^{-1}\circ\boldsymbol{O},\boldsymbol{\omega}\rangle + \langle g\circ\boldsymbol{O}, g\circ\boldsymbol{\omega}\rangle,$$

i.e.,

$$\langle g^{-1}\circ\boldsymbol{O},\boldsymbol{\omega}\rangle = -\langle g\circ\boldsymbol{O}, g\circ\boldsymbol{\omega}\rangle,$$

replace g^{-1} with g, then

$$\langle g\circ\boldsymbol{O},\boldsymbol{\omega}\rangle = -\langle g^{-1}\circ\boldsymbol{O}, g^{-1}\circ\boldsymbol{\omega}\rangle.$$

By definition,

$$k_\lambda(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\exp\left((\frac{n-1}{2}-i\lambda)\langle\boldsymbol{x},\boldsymbol{\omega}\rangle_H + (\frac{n-1}{2}+i\lambda)\langle\boldsymbol{y},\boldsymbol{\omega}\rangle_H\right)\right]. \tag{A.3}$$

Now assume, $g\circ\boldsymbol{y} = O$, then consider

$$\begin{aligned}
k_\lambda(g\circ\boldsymbol{x},\boldsymbol{O}) &= \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\zeta_{\lambda,\omega}(g\circ\boldsymbol{x})^*\zeta_{\lambda,\omega}(\boldsymbol{O})\right] = \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\zeta_{\lambda,\omega}(g\circ\boldsymbol{x})^*\right]\\
&= \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\langle g\circ\boldsymbol{x},\boldsymbol{\omega}\rangle\right)\right]\\
&= \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\left(\langle\boldsymbol{x},g^{-1}\circ\boldsymbol{\omega}\rangle + \langle g\circ\boldsymbol{O},\boldsymbol{\omega}\rangle\right)\right)\right]\\
&= \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\left(\langle\boldsymbol{x},g^{-1}\circ\boldsymbol{\omega}\rangle - \langle g^{-1}\circ\boldsymbol{O},g^{-1}\circ\boldsymbol{\omega}\rangle\right)\right)\right]\\
&= \frac{1}{2}\cdot\mathop{\mathbb{E}}_{\boldsymbol{\omega}}\left[\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\left(\langle\boldsymbol{x},g^{-1}\circ\boldsymbol{\omega}\rangle - \langle\boldsymbol{y},g^{-1}\circ\boldsymbol{\omega}\rangle\right)\right)\right]\\
&= \frac{1}{2}\cdot\int_{\mathbb{S}^{n-1}}\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\left(\langle\boldsymbol{x},g^{-1}\circ\boldsymbol{\omega}\rangle - \langle\boldsymbol{y},g^{-1}\circ\boldsymbol{\omega}\rangle\right)\right)\rho_1(\boldsymbol{\omega})d\boldsymbol{\omega},
\end{aligned}$$

where $\rho_1(\boldsymbol{\omega})$ is a uniform distribution over the sphere, use $\hat{\boldsymbol{\omega}} = g^{-1}\circ\boldsymbol{\omega}$ as a change of variable, then

$$\begin{aligned}
k_\lambda(g\circ\boldsymbol{x},\boldsymbol{O}) &= \int_{\mathbb{S}^{n-1}}\exp\left(\left(\frac{n-1}{2}-i\lambda\right)\left(\langle\boldsymbol{x},g^{-1}\circ\boldsymbol{\omega}\rangle - \langle\boldsymbol{y},g^{-1}\circ\boldsymbol{\omega}\rangle\right)\right)\rho_1(\boldsymbol{\omega})d\boldsymbol{\omega}\\
&= \int_{\mathbb{S}^{n-1}}\exp\left(\left(\frac{n-1}{2}-i\lambda\right)(\langle\boldsymbol{x},\hat{\boldsymbol{\omega}}\rangle - \langle\boldsymbol{y},\hat{\boldsymbol{\omega}}\rangle)\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})d(g\circ\hat{\boldsymbol{\omega}}).
\end{aligned}$$

We claim that the mapping g acts on the boundary with the Jacobian given by

$$\frac{d(g\circ\hat{\boldsymbol{\omega}})}{d(\hat{\boldsymbol{\omega}})} = \frac{1}{2}\cdot\exp((n-1)\cdot\langle g^{-1}\circ\boldsymbol{O},\hat{\boldsymbol{\omega}}\rangle) = \frac{1}{2}\cdot\left(\frac{1-\left\|g^{-1}\circ\boldsymbol{O}\right\|^2}{\left\|g^{-1}\circ\boldsymbol{O}-\hat{\boldsymbol{\omega}}\right\|^2}\right)^{n-1}, \tag{A.4}$$

then

$$
\begin{aligned}
k_\lambda(g\circ \boldsymbol{x}, \boldsymbol{O}) &= \frac{1}{2}\cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} - i\lambda\right)(\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle - \langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle)\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})d(g\circ\hat{\boldsymbol{\omega}}) \\
&= \frac{1}{2}\cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} - i\lambda\right)(\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle - \langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle)\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})\exp((n-1)\cdot\langle g^{-1}\circ\boldsymbol{O}, \hat{\boldsymbol{\omega}}\rangle)d(\hat{\boldsymbol{\omega}}) \\
&= \frac{1}{2}\cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} - i\lambda\right)(\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle - \langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle) + (n-1)\cdot\langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})d(\hat{\boldsymbol{\omega}}) \\
&= \frac{1}{2}\cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} - i\lambda\right)\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle + \left(n-1-\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})d(\hat{\boldsymbol{\omega}}) \\
&= \frac{1}{2}\cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} - i\lambda\right)\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle + \left(\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle\right)\rho_1(g\circ\hat{\boldsymbol{\omega}})d(\hat{\boldsymbol{\omega}}).
\end{aligned}
$$

Since ρ_1 is a uniform distribution, this is

$$
k_\lambda(g\circ \boldsymbol{x}, \boldsymbol{O}) = \frac{1}{2}\cdot \mathop{\mathbb{E}}_{\hat{\omega}}\left[\exp\left(\left(\frac{n-1}{2} - i\lambda\right)\langle \boldsymbol{x}, \hat{\boldsymbol{\omega}}\rangle + \left(\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{y}, \hat{\boldsymbol{\omega}}\rangle\right)\right],
$$

compared with Equation A.3, it follows that $k_\lambda(g\circ\boldsymbol{x}, \boldsymbol{O}) = k_\lambda(\boldsymbol{x}, \boldsymbol{y})$. Since $k_\lambda(g\circ\boldsymbol{x}, \boldsymbol{O})$ only depends on $d_{\mathbb{H}}(g\circ\boldsymbol{x}, \boldsymbol{O}) = d_{\mathbb{H}}(\boldsymbol{x}, g^{-1}\circ\boldsymbol{O}) = d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y})$ from Lemma A.2.3, then $k_\lambda(\boldsymbol{x}, \boldsymbol{y})$ is distance-invariant, and hence isometry-invariant.

It suffices to prove Equation A.4, i.e.,

$$
\frac{d(g\circ\boldsymbol{\omega})}{d(\boldsymbol{\omega})} = \left(\frac{1 - \left\|g^{-1}\circ\boldsymbol{O}\right\|^2}{\left\|g^{-1}\circ\boldsymbol{O} - \boldsymbol{\omega}\right\|^2}\right)^{n-1},
$$

clearly it holds when g is an rotation, it suffices to show this for a translation isometry. Denote $\mathrm{Inv}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|^2}$, then in the Poincarè Ball model, all translation isometry takes the form

$$
T_{\boldsymbol{a}}(\boldsymbol{x}) = -\boldsymbol{a} + (1 - \|\boldsymbol{a}\|^2)\,\mathrm{Inv}(\mathrm{Inv}(\boldsymbol{x}) - \boldsymbol{a}),
$$

where both $\boldsymbol{x}, \boldsymbol{a}$ in the Poincarè Ball model and $T_{\boldsymbol{a}}(\boldsymbol{a}) = \boldsymbol{O}, T_{\boldsymbol{a}}^{-1}(\boldsymbol{O}) = \boldsymbol{a}$. Thus,

$$
T_{\boldsymbol{a}}(\boldsymbol{\omega}) = -\boldsymbol{a} + (1-\|\boldsymbol{a}\|^2)\,\mathrm{Inv}(\mathrm{Inv}(\boldsymbol{\omega}) - \boldsymbol{a}) = -\boldsymbol{a} + (1-\|\boldsymbol{a}\|^2)\,\mathrm{Inv}(\boldsymbol{\omega} - \boldsymbol{a}) = -\boldsymbol{a} + (1-\|\boldsymbol{a}\|^2)\frac{\boldsymbol{\omega} - \boldsymbol{a}}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2},
$$

where we use the fact $\|\boldsymbol{\omega}\| = 1$. Since the integral is taken over the unit sphere with $\|\boldsymbol{\omega}\| = 1, \|T_{\boldsymbol{a}}(\boldsymbol{\omega})\| = 1$, we consider only the mapping of $T_{\boldsymbol{a}}$ restricted to the

first $n-1$ (free) dimensions, with an abuse of notation, regard $\boldsymbol{\omega}$ as an $n-1$ dimensional vector. Then the Jacobian of $T_{\boldsymbol{a}}(\boldsymbol{\omega})$ with respect to $\boldsymbol{\omega}$ is

$$dT_{\boldsymbol{a}}(\boldsymbol{\omega}) = (1 - \|\boldsymbol{a}\|^2) d(\frac{\boldsymbol{\omega} - \boldsymbol{a}}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2}),$$

we can calculate that

$$\begin{aligned} d(\frac{\boldsymbol{\omega} - \boldsymbol{a}}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2}) =& \frac{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2 \, d\boldsymbol{\omega} - (\boldsymbol{\omega} - \boldsymbol{a}) \cdot 2(\boldsymbol{\omega} - \boldsymbol{a})^\intercal d\boldsymbol{\omega}}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^4} \\ =& \frac{1}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \cdot \frac{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2 - 2(\boldsymbol{\omega} - \boldsymbol{a}) \cdot (\boldsymbol{\omega} - \boldsymbol{a})^\intercal}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} d\boldsymbol{\omega} \\ =& \frac{d\boldsymbol{\omega}}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \cdot \left(\mathbf{I}_{n-1,n-1} - \frac{2(\boldsymbol{\omega} - \boldsymbol{a}) \cdot (\boldsymbol{\omega} - \boldsymbol{a})^\intercal}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \right), \end{aligned}$$

then

$$\frac{dT_{\boldsymbol{a}}(\boldsymbol{\omega})}{d\boldsymbol{\omega}} = \frac{1 - \|\boldsymbol{a}\|^2}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \left(\mathbf{I}_{n-1,n-1} - \frac{2(\boldsymbol{\omega} - \boldsymbol{a}) \cdot (\boldsymbol{\omega} - \boldsymbol{a})^\intercal}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \right),$$

note the relation that

$$\det(\mathbf{I} + \boldsymbol{x}\boldsymbol{y}^\intercal) = 1 + \boldsymbol{x}^\intercal \boldsymbol{y},$$

then

$$\det(\mathbf{I}_{n-1,n-1} - \frac{2(\boldsymbol{\omega} - \boldsymbol{a}) \cdot (\boldsymbol{\omega} - \boldsymbol{a})^\intercal}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2}) = 1 - \frac{2(\boldsymbol{\omega} - \boldsymbol{a})^\intercal \cdot (\boldsymbol{\omega} - \boldsymbol{a})}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} = 1 - 2 = -1,$$

then the absolute value of determinant of the Jacobian is

$$\begin{aligned} |\det(\frac{dT_{\boldsymbol{a}}(\boldsymbol{\omega})}{d\boldsymbol{\omega}})| =& |\det\left(\frac{1 - \|\boldsymbol{a}\|^2}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} (\mathbf{I}_{n-1,n-1} - \frac{2(\boldsymbol{\omega} - \boldsymbol{a}) \cdot (\boldsymbol{\omega} - \boldsymbol{a})^\intercal}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2}) \right)| \\ =& \left(\frac{1 - \|\boldsymbol{a}\|^2}{\|\boldsymbol{\omega} - \boldsymbol{a}\|^2} \right)^{n-1} \\ =& \left(\frac{1 - \left\| g^{-1} \circ \boldsymbol{O} \right\|^2}{\left\| \boldsymbol{\omega} - g^{-1} \circ \boldsymbol{O} \right\|^2} \right)^{n-1}, \end{aligned}$$

with which a change of variable would give

$$\frac{d(g \circ \boldsymbol{\omega})}{d(\boldsymbol{\omega})} = \left(\frac{1 - \left\| g^{-1} \circ \boldsymbol{O} \right\|^2}{\left\| g^{-1} \circ \boldsymbol{O} - \boldsymbol{\omega} \right\|^2} \right)^{n-1},$$

which finishes the proof. □

Proof of Theorem 2.3.2. As a result of Lemma A.2.3 and Lemma A.2.4, the expression for $k_\lambda(\boldsymbol{x}, \boldsymbol{y})$ follows:

$$k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \cdot {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right),$$

where ${}_2F_1$ is the hypergeometric function, we can also apply the Euler transformation to get

$$k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \cdot \left(\frac{1}{2}\left(1 + \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right)^{1-\frac{n}{2}} \cdot {}_2F_1\left(\frac{1}{2} + i\lambda, \frac{1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right).$$

If we let

$$z = \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)$$

then this is written more succinctly as

$$k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \cdot (1-z)^{1-\frac{n}{2}} \cdot {}_2F_1\left(\frac{1}{2} + i\lambda, \frac{1}{2} - i\lambda; \frac{n}{2}; z\right).$$

We can also write this as a Legendre function,

$$\begin{aligned} k_\lambda(\boldsymbol{x}, \boldsymbol{y}) &= \frac{1}{2} \cdot (1-z)^{1-\frac{n}{2}} \cdot \Gamma\left(\frac{n}{2}\right) \cdot z^{\frac{2-n}{4}} \cdot (1-z)^{\frac{n-2}{4}} \cdot P^{1-\frac{n}{2}}_{-\frac{1}{2}+i\lambda}(1-2z) \\ &= \frac{1}{2} \cdot (z(1-z))^{\frac{2-n}{4}} \cdot \Gamma\left(\frac{n}{2}\right) \cdot P^{1-\frac{n}{2}}_{-\frac{1}{2}+i\lambda}(1-2z) \end{aligned}$$

Observe that

$$z(1-z) = \frac{1}{4}\left(1 - \cosh^2(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right) = \frac{1}{4}\left(-\sinh^2(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)$$

and similarly

$$1 - 2z = \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y})).$$

This is manifestly real because

$$\begin{aligned} k_\lambda(\boldsymbol{x}, \boldsymbol{y}) &= \frac{1}{2} \cdot \sum_{k=0}^{\infty} \prod_{m=0}^{k-1} \frac{\left(\frac{n-1}{2} + m + i\lambda\right)\left(\frac{n-1}{2} + m - i\lambda\right)}{\left(\frac{n}{2} + m\right)(1+m)} \cdot \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right) \\ &= \frac{1}{2} \cdot \sum_{k=0}^{\infty} \prod_{m=0}^{k-1} \frac{\left(\frac{n-1}{2} + m\right)^2 + \lambda^2}{\left(\frac{n}{2} + m\right)(1+m)} \cdot \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right). \end{aligned}$$

If we draw HyLa features using a distribution ρ over λ, then the resulting approximated kernel will be

$$\begin{aligned} k(\boldsymbol{x}, \boldsymbol{y}) &= \frac{1}{2} \cdot \int_{-\infty}^{\infty} k_\lambda(\boldsymbol{x}, \boldsymbol{y}) \cdot \rho(\lambda)\, d\lambda \\ &= \frac{1}{2} \cdot \int_{-\infty}^{\infty} {}_2F_1\left(\frac{n-1}{2} + i\lambda, \frac{n-1}{2} - i\lambda; \frac{n}{2}; \frac{1}{2}\left(1 - \cosh(d_{\mathbb{H}}(\boldsymbol{x}, \boldsymbol{y}))\right)\right) \cdot \rho(\lambda)\, d\lambda. \end{aligned}$$

□

Proof of Theorem 2.3.3. Denote

$$\phi_\lambda(\boldsymbol{z}) = \int_{\mathbb{S}^{n-1}} \zeta_{\lambda,\boldsymbol{\omega}}(\boldsymbol{z}) d\boldsymbol{\omega} = \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{\omega}, \boldsymbol{z}\rangle_H\right) d\boldsymbol{\omega},$$

which are basic spherical functions. For any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{B}^n$, assume g_y is an isometry that maps $\boldsymbol{y}$ to the origin, i.e., $g_y \circ \boldsymbol{y} = O$, denote $\hat{\boldsymbol{x}} = g_y \circ \boldsymbol{x}$, then $k_\lambda(\boldsymbol{x}, \boldsymbol{y}) = k_\lambda(\hat{\boldsymbol{x}}, \boldsymbol{O})$ for any λ, note that

$$\begin{aligned} k_\lambda(\hat{\boldsymbol{x}}, \boldsymbol{O}) &= \frac{1}{2} \cdot \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{\omega}, \hat{\boldsymbol{x}}\rangle_H\right) \rho_1(\boldsymbol{\omega}) d\boldsymbol{\omega} \\ &= \frac{1}{2} \cdot \frac{1}{\text{Area}(\mathbb{S}^{n-1})} \int_{\mathbb{S}^{n-1}} \exp\left(\left(\frac{n-1}{2} + i\lambda\right)\langle \boldsymbol{\omega}, \hat{\boldsymbol{x}}\rangle_H\right) d\boldsymbol{\omega} \\ &= \frac{1}{2} \cdot \frac{1}{\text{Area}(\mathbb{S}^{n-1})} \phi_\lambda(\hat{\boldsymbol{x}}), \end{aligned}$$

where we use the fact that $\rho_1(\boldsymbol{\omega})$ is a uniform distribution over the sphere. As-

sume the existence of an associated density $\rho(\lambda)$ with the kernel, then

$$\begin{aligned}
k(\boldsymbol{x},\boldsymbol{y}) = k(d_{\mathbb{H}}(\boldsymbol{x},\boldsymbol{y})) &= k(d_{\mathbb{H}}(\hat{\boldsymbol{x}},\boldsymbol{O})) \\
&= \int_{-\infty}^{\infty} k_\lambda(\boldsymbol{x},\boldsymbol{y})\cdot\rho(\lambda)\,d\lambda \\
&= \int_{-\infty}^{\infty} k_\lambda(\hat{\boldsymbol{x}},\boldsymbol{O})\cdot\rho(\lambda)\,d\lambda \\
&= \frac{1}{2}\cdot\int_{-\infty}^{\infty}\frac{1}{\text{Area}(\mathbb{S}^{n-1})}\phi_\lambda(\hat{\boldsymbol{x}})\cdot\rho(\lambda)\,d\lambda \\
&= \frac{1}{2}\cdot\frac{1}{\text{Area}(\mathbb{S}^{n-1})}\int_{-\infty}^{\infty}\phi_\lambda(\hat{\boldsymbol{x}})\cdot\rho(\lambda)\,d\lambda \\
&= \frac{1}{\pi\text{Area}(\mathbb{S}^{n-1})}\int_{-\infty}^{\infty}\frac{\rho(\lambda)}{\lambda\tanh(\frac{\pi\lambda}{2})}\cdot\phi_\lambda(\hat{\boldsymbol{x}})\cdot|c(\lambda)|^{-2}\,d\lambda
\end{aligned}$$

where $|c(\lambda)|^{-2} = \frac{\pi\lambda}{2}\tanh\frac{\pi\lambda}{2}$ when $\lambda\in\mathbb{R}$. Note that the last integral is exactly the inverse spherical transform [76] of $\frac{\rho(\lambda)}{\lambda\tanh(\frac{\pi\lambda}{2})}$, hence it can be derived in the reverse direction by taking the spherical transform of $k(d_{\mathbb{H}}(\hat{\boldsymbol{x}},\boldsymbol{O}))$, i.e.,

$$\frac{\rho(\lambda)}{\lambda\tanh(\frac{\pi\lambda}{2})} = \int_{\mathcal{B}^n} k(d_{\mathbb{H}}(\hat{\boldsymbol{x}},\boldsymbol{O}))\phi_{-\lambda}(\hat{\boldsymbol{x}})d\hat{\boldsymbol{x}}.$$

Hence $\rho(\lambda)$ can be derived as

$$\begin{aligned}
\rho(\lambda) &= \lambda\tanh(\frac{\pi\lambda}{2})\int_{\mathcal{B}^n} k(d_{\mathbb{H}}(\hat{\boldsymbol{x}},\boldsymbol{O}))\phi_{-\lambda}(\hat{\boldsymbol{x}})d\hat{\boldsymbol{x}} \\
&= \lambda\tanh(\frac{\pi\lambda}{2})\int_{\mathcal{B}^n}\int_{\partial\mathcal{B}^n} k(d_{\mathbb{H}}(\boldsymbol{z},\boldsymbol{O}))\exp\left((\frac{n-1}{2}-i\lambda)\langle\boldsymbol{\omega},\boldsymbol{z}\rangle_H\right)d\boldsymbol{\omega}d\boldsymbol{z}.
\end{aligned}$$

Therefore, given an isometry-invariant positive semidefinite kernel $k(\boldsymbol{x},\boldsymbol{y}) = k(d_{\mathbb{H}}(\boldsymbol{x},\boldsymbol{y}))$, we can compute $\rho(\lambda)$ following the above expression if it exists, then the rest of the theorem follows. □

A.2.2 Concentration of the Kernel Estimation

The readers may wonder whether there is a concentration behavior using the random variable $\langle\phi(\boldsymbol{x}),\phi(\boldsymbol{y})\rangle$ to approximate $k(\boldsymbol{x},\boldsymbol{y})$. Unfortunately, the eigen-

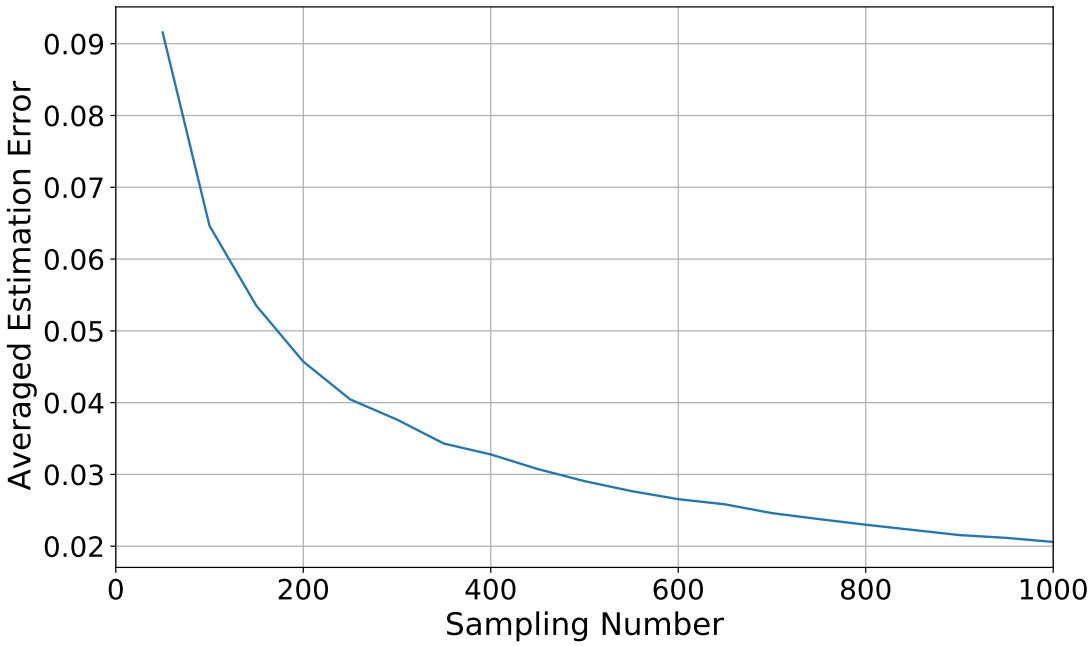


Figure A.5: Averaged estimation error of HyLa to the kernel.

function $\phi(\boldsymbol{x})$ itself is not a sub-Gaussian so as to derive a concentration bound in a straightforward way, but we do provide a numerical experiment to measure the estimation behavior. For the estimation in Figure 2.13, we sampled $1,000$ different eigenfunctions.

We first fix a set of $1,000$ points x_i by uniformly sampling over 2-dimensional hyperbolic space, then approximate the kernel $k(\boldsymbol{o}, \boldsymbol{x_i})$ using $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{o}) \rangle$ by sampling with an increasing number of eigenfunctions, ranging from 50 to $1,000$. At last, we compute the mean absolute error of the estimation $\langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{o}) \rangle$ to the true kernel value $k(\boldsymbol{o}, \boldsymbol{x_i})$. We plot the mean estimation error in Figure A.5, which seems to be an exponentially decay, it's an interesting future work to investigate this estimation error.

A.2.3 Discussions on the Methodology

Node Embedding vs Feature Embedding

When HyLa is adopted at node level, i.e., each vertex/node v_i in the graph is associated with a hyperbolic embedding parameter $z_i \in \mathcal{B}^{d_0}$. Then the inner product of HyLa features $\langle \phi(z_i), \phi(z_j) \rangle$ of vertex v_i, v_j approximates some kernel $k(z_i, z_j)$. The optimization of z_i encourages learning of the kernel on the hyperbolic space to solve the task.

When HyLa is adopted at feature level, i.e., each column dimension of the node feature $\mathbf{X} \in \mathbb{R}^{n \times d}$ is associated with a hyperbolic embedding parameter $z_i \in \mathcal{B}^{d_0}$. The HyLa feature associated to each vertex/node v_i is then computed as $\sum_{k=1}^{d} \mathbf{X}_{ik} \phi(z_k)$, where $\sum_{k=1}^{d} \mathbf{X}_{ik} = 1$ if a row-normalization is applied on the original node features.

Therefore, the inner product of two node HyLa features is

$$\langle \sum_{k=1}^{d} \mathbf{X}_{ik} \phi(z_k), \sum_{l=1}^{d} \mathbf{X}_{jl} \phi(z_l) \rangle = \sum_{k,l=1}^{d} \mathbf{X}_{ik} \mathbf{X}_{jl} \langle \phi(z_k), \phi(z_l) \rangle,$$

which in expectation equals a linear combination of kernels, i.e., $\sum_{k,l=1}^{d} \mathbf{X}_{ik} \mathbf{X}_{jl} k(z_k, z_l)$. Therefore, it captures a much more complicated kernel relation on the hyperbolic space than directly embedding nodes.

Two-Step Approach

For the purpose of end-to-end learning, in our experiments, we jointly learn the embedding parameter $\mathbf{Z}$ and weight $\mathbf{W}$ in SGC during the training time, as detailed in A.2.4. It's possible to adopt a two-step approach, i.e., first pretrain a hy-

perbolic embedding, then fix the embedding and train the graph learning model only. In the first step, for example, optimization-based methods [122, 123] and combinatorial construction methods [147, 156] can be adopted by supervising the graph connectivity. However, these methods only utilize the graph structure information, but ignore the node feature information $\mathbf{X}$, which leads to a natural performance degradation. In comparison, as shown in experiments and analyzed in A.2.3, our end-to-end learning of HyLa can be used to embed features and enables learning a complex kernel representation to avoid this shortcoming. Intuitively, the graph connectivity information can be too general for downstreaming tasks which rely more on semantic information. It's not clear to us how to encode the semantic information (node features) into embedding following e.g., [122, 123].

Another way [28] of deriving a pretrained hyperbolic embedding that might take semantic information into consideration is to train a link prediction model, however, this method is not efficient as HGCN, shown in Table A.6 and Figure 2.15.

A.2.4 Experiment Details

Task and Dataset

We provide a detailed description/table of used datasets in Table A.2 and Table A.3.

1. **Citation Networks.** Cora, Citeseer and Pubmed [151] are standard citation network benchmarks, where nodes represent papers, connected to each

Table A.2: Node classification dataset statistics.

Setting	Dataset	# Nodes	# Edges	Classes	Features
Trans-ductive	Cora	2,708	5,429	7	1,433
	Citeseer	3,327	4,732	6	3,703
	Pubmed	19,717	44,338	3	500
	Disease	1,044	1,043	2	1,000
	Airport	3,188	18,631	4	4
Inductive	Reddit	233K	11.6M	41	602

Table A.3: Text classification dataset statistics.

Dataset	# Docs	# Words	Average Length	Classes
R8	7,674	7688	65.72	8
R52	9,100	8892	69.82	52
Ohsumed	7400	14157	135.82	23
MR	10662	18764	20.39	2

other via citations. We follow the standard splits [92] with 20 nodes per class for training, 500 nodes for validation and 1000 nodes for test.

2. **Disease propagation tree** [28]. This is tree networks simulating the SIR disease spreading model [9], where the label is whether a node was infected or not and the node features indicate the susceptibility to the disease. We use dataset splits of 30/10/60% for train/val/test set.

3. **Airport.** We take this dataset from [28]. This is a transductive dataset where nodes represent airports and edges represent the airline routes as from OpenFlights. Airport contains 3,188 nodes, each node has a 4 dimensional feature representing geographic information (longitude, latitude and altitude), and GDP of the country where the airport belongs to. For node classification, labels are chosen to be the population of the country where the airport belongs to. We use dataset splits of 524/524 nodes for val/test set.

4. **Reddit.** This is a much larger graph dataset built from Reddit posts, where the label is the community, or "subreddit", that a post belongs to. Two nodes are connected if the same user comments on both. We use a dataset split of 152K/24K/55K follows [72, 30], similarly, we evaluate HyLa inductively by following [183]: we train on a subgraph comprising only training nodes and test with the original graph.

Training details.

We use HyLa together with SGC model as softmax($\mathbf{A}^K\overline{\mathbf{X}}\mathbf{W}$), where the HyLa feature matrix $\overline{\mathbf{X}} \in \mathbb{R}^{n\times d_1}$ is derived from the hyperbolic embedding $\mathbf{Z} \in \mathbb{R}^{n\times d_0}$ using Algorithm 1. Specifically, we randomly sample constants of HyLa features $\overline{\mathbf{X}}$ by sampling the boundary points ω uniformly from the boundary $\partial\mathcal{B}^n$, eigenvalue constants λ from a zero-mean s-standard-deviation Gaussian and biases b uniformly from $[0, 2\pi]$. These constants remain fixed throughout training.

We use cross-entropy as the loss function and jointly optimize the low dimensional hyperbolic embedding $\mathbf{Z}$ and linear weight $\mathbf{W}$ simultaneously during training. Specifically, Riemannian SGD optimizer [18] (of learning rate lr_1) for $\mathbf{Z}$ and Adam [91] optimizer (of learning rate lr_2) for $\mathbf{W}$. RSGD naturally scales to very large graph because the graph connectivity pattern is sufficiently sparse. We adopt early-stopping as regularization. We tune the hyper-parameter via grid search over the parameter space. Each hyperbolic embedding is initialized around the origin, by sampling each coordinate at random from $[-10^{-5}, 10^{-5}]$.

Table A.4: Hyper-parameters for node classification.

Dataset	d_0	d_1	K	s	lr_1	lr_2	# Epochs
Disease	16	100	5	1.0	0.05	0.0001	100
Airport	16	1000	2	0.01	0.5	0.1	100
Pubmed	16	100	5	0.1	0.5	0.001	200
Citeseer	16	500	5	1.0	0.1	0.001	100
Cora	16	100	2	1.0	0.1	0.01	100
Reddit	50	1000	2	0.5	0.1	0.001	100

Table A.5: Hyper-parameters for text classification.

	Transductive Setting					**Inductive Setting**				
Dataset	d_0	d_1	s	lr_1	lr_2	d_0	d_1	s	lr_1	lr_2
R8	50	500	0.5	0.01	0.0001	50	500	0.5	0.001	0.0001
R52	50	500	0.5	0.1	0.0001	50	1000	0.5	0.008	0.0001
Ohsumed	50	500	0.5	0.01	0.0001	50	1000	0.1	0.001	0.0001
MR	30	500	0.5	0.1	0.0001	50	500	0.5	0.01	0.0001

Hyper-parameters

We provide the detailed values of hyper-parameters for **node classification** and **text classification** in Table A.4 and Table A.5 respectively. Particularly, we fix $K = 2$ for the text classification task and train the model for a maximum of 200 epochs without using any regularization (e.g. early stopping). Also note that in the transductive text classification setting, HyLa is used at node level, hence the size of parameters will be proportional to the size of graph, in which case, d_0 and d_1 can not be too large so as to avoid OOM. In the inductive text classification setting, there is no such constraint as the dimension of lower level features is not very large itself.

Timing

We show the specific training timing statistics of different models on Pubmed dataset in Table A.6. Particularly for HGCN model, in order to achieve the report performances, we follow the same training procedure using public code, which is divided into two stages: (1) a link prediction task on the dataset to learn hyperbolic embeddings, and (2) use the pretrained embeddings to train a MLP classifier. Hence, we add the timing of both stages as the timing for HGCN.

Table A.6: Training time on Pubmed.

Model	Timing (seconds)
SGC	0.37
GCN	1.25
GAT	12.52
HNN	2.41
HGCN	15.41
LGCN	10.93
HyLa-SGC	3.51

A.3 Coneheads: Hierarchy Aware Attention

A.3.1 Penumbral & Umbral Attention Derivation

Penumbral Attention Derivation

Recall the definition of Penumbral Attention in the infinite setting shadow cone construction:

$$K(u,v) = \exp\left(-\gamma \max\left(u_d, v_d, \sqrt{r^2 - \left(\frac{\sqrt{r^2-u_d^2} + \sqrt{r^2-v_d^2} - \|u_{:-1} - v_{:-1}\|}{2}\right)^2}\right)\right) \quad \text{(A.5)}$$

when there exists a cone that contains u and v, and

$$K(u,v) = \exp\left(-\gamma \sqrt{\left(\frac{\|u_{:-1} - v_{:-1}\|^2 + u_d^2 - v_d^2}{2\|u_{:-1} - v_{:-1}\|}\right)^2 + v_d^2}\right) \quad \text{(A.6)}$$

otherwise. There exists a cone that contains u and v when

$$\left(\|u_{:-1} - v_{:-1}\| - \sqrt{r^2 - u_d^2}\right)^2 + v_d^2 < r^2 \qquad \text{or} \qquad \|u_{:-1} - v_{:-1}\| \leq \sqrt{r^2 - u_d^2} \quad \text{(A.7)}$$

Below, we give a derivation of this definition in the infinite penumbral cone construction. From theorem A.3.2, $\sup_2(u,v)$ is the lowest cone that contains u and v in the plane containing u, v, and the ideal point of $\mathcal{S}$. This lets us derive the height of $\sup_2(u,v)$ in that plane, which is given by the intersection of Euclidean semicircle geodesics with Euclidean radius r.

First, we check if there exists a cone that contains u and v in this plane. Note that the region of points v s.t. $\exists$ a cone containing u and v is the region contained by the two geodesics forming the cone of u (see Figure A.6). Assume $u_x = 0$. Then, in the infininte setting penumbral construction, these geodesics are the two semicircles centered at $\left(\pm\sqrt{r^2-u_y^2}, 0\right)$ with radius r. We can check if a point is in this region by checking if it is in the quarter circle bounded by $\left(-\sqrt{r^2-u_y^2}, 0\right)$ and the arc from $\left(-\sqrt{r^2-u_y^2} - r, 0\right)$ to $\left(-\sqrt{r^2-u_y^2}, r\right)$, the quarter circle bounded by $\left(\sqrt{r^2-u_y^2}, 0\right)$ and the arc from $\left(\sqrt{r^2-u_y^2}r, 0\right)$ to $\left(\sqrt{r^2-u_y^2}, r\right)$, or the rectangular region between the two. This is given by

$$\left(\left(\left(v_x - \sqrt{r^2-u_y^2}\right)^2 + (v_y - 0)^2 < r^2\right) \wedge v_x > \sqrt{r^2-u_y^2}\right) \vee v_x \leq \sqrt{r^2-u_y^2}$$

which reduces to equation A.7 since $u_x = 0 \implies v_x = \|u_{:-1} - v_{:-1}\|$.

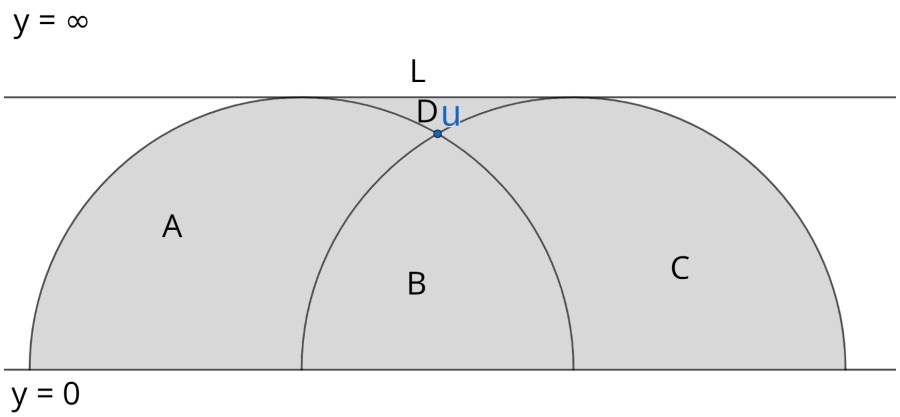


Figure A.6: The gray area ($A \cup B \cup C \cup D$) is the region of all points that share a cone with u. B is the cone of u, and D is the region of points whose cones contain u (ancestors of u).

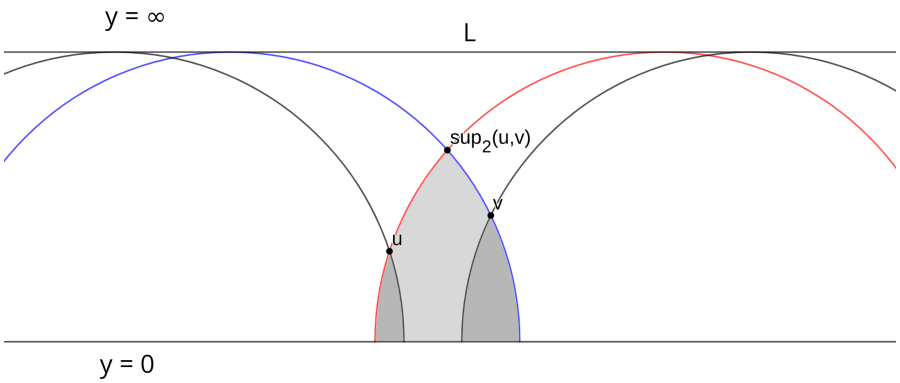


Figure A.7: If $v_x \geq u_x$, then $\sup_2(u, v)$ is the intersection of the "right" geodesic of u (red) and the "left" geodesic of v (blue).

Now, we derive the height of $\sup_2(u, v)$ when there exists a cone containing u and v. Assume WLOG that $v_x \geq u_x$, and normalize u and v s.t. $u_x = -\|u_{:-1} - v_{:-1}\|/2$ and $v_x = \|u_{:-1} - v_{:-1}\|/2$. If $u \not\prec v$ and $v \not\prec u$, $\sup_2(u, v)$ is intersection of the "right" geodesic of u and the "left" geodesic of v (see Figure A.7). The center of the right geodesic of u is given by $\left(u_x + \sqrt{r^2 - u_y^2}, 0\right)$, and the center of the left geodesic of v is given by $\left(v_x - \sqrt{r^2 - v_y^2}\right)$. Since $\sup_2(u, v)$ is equidistant from these two centers,

$\sup_2(u,v)_x = \left(\sqrt{r^2 - u_y^2} - \sqrt{r^2 - v_y^2}\right)/2$. Then,

$$\begin{aligned}
\sup_2(u,v)_y &= \sqrt{r^2 - \left(\frac{\sqrt{r^2 - u_y^2} - \sqrt{r^2 - v_y^2}}{2} - \left(u_x + \sqrt{r^2 - u_y^2}\right)\right)^2} \\
&= \sqrt{r^2 - \left(\frac{\|u_{:-1} - v_{:-1}\| - \sqrt{r^2 - u_y^2} - \sqrt{r^2 - v_y^2}}{2}\right)^2} \\
&= \sqrt{r^2 - \left(\frac{\sqrt{r^2 - u_y^2} + \sqrt{r^2 - v_y^2} - \|u_{:-1} - v_{:-1}\|}{2}\right)^2}
\end{aligned} \tag{A.8}$$

If $u \prec v$ or $v \prec u$, then $\sup_2(u,v)$ is equal to u or v, respectively. In this situation, equation A.8 is less than u_y or v_y, repsectively. Thus, we take the max of u_y, v_y, and equation A.8 to get equation A.5.

When there does not exist a cone containing u and v, we return the height of the lowest light source s.t. there exists such a cone. This corresponds to the height of the highest point in the extended geodesic through u and v. Since geodesics are Euclidean semicircles, this height is the radius of the Euclidean semicircle through u and v. Let the center of this semicircle be $(x, 0)$, and normalize u and v s.t. $u_x = 0$ and $v_x = \|u_{:-1} - v_{:-1}\|$. Then, we have

$$\begin{aligned}
x^2 + u_y^2 &= (v_x - x)^2 + v_y^2 \\
2v_x x &= v_x^2 + v_y^2 - u_y^2 \\
2\|u_{:-1} - v_{:-1}\|x &= \|u_{:-1} - v_{:-1}\|^2 + v_y^2 - u_y^2 \\
x &= \frac{\|u_{:-1} - v_{:-1}\|^2 + v_y^2 - u_y^2}{2\|u_{:-1} - v_{:-1}\|}
\end{aligned}$$

The radius of the semicircle through u and v is then

$$\sqrt{\left(\frac{\|u_{:-1} - v_{:-1}\|^2 + v_y^2 - u_y^2}{2\|u_{:-1} - v_{:-1}\|}\right)^2 + u_d^2}$$

It is easy to verify that this is symmetric in u and v and gives equation A.6. Furthermore, it is also easy to verify that when

$$\left(\|u_{:-1} - v_{:-1}\| - \sqrt{r^2 - u_d^2}\right)^2 + v_d^2 = r^2$$

the in-cone and out-of-cone heights are both r, and equations A.5 and A.6 both equal $\exp(-\gamma r)$, making K continuous with respect to the positions of u and v.

Umbral Attention Derivation

Recall the definition of Umbral Attention in the infinite setting shadow cone construction:

$$K(u, v) = \exp\left(-\gamma \max\left(u_d, v_d, \frac{\|u_{:-1} - v_{:-1}\|}{2\sinh(r)} + \frac{u_d + v_d}{2}\right)\right) \tag{A.9}$$

As shown in [203], cone regions for infinite setting umbral cones are given by Euclidean triangles with angle apertures of $2\arctan(\sinh(r))$. WLOG, assume that $v_x \geq u_x$ and normalize u and v s.t. $u_x = 0$ and $v_x = \|u_{:-1} - v_{:-1}\|$. When $u \not\prec v$ and $v \not\prec u$, $\sup_2(u, v)$ is given by the intersection of the line with slope $1/\sinh(r)$ through u and the line with slope $-1/\sinh(r)$ through v. This gives

$$\begin{aligned} y - u_y &= \frac{x - u_x}{\sinh(r)} \\ y - v_y &= \frac{x + v_x}{\sinh(r)} \\ 2y &= \frac{v_x - u_x}{\sinh(r)} + v_y + u_y \\ y &= \frac{\|u_{:-1} - v_{:-1}\|}{2\sinh(r)} + \frac{v_y + u_y}{2} \end{aligned} \tag{A.10}$$

where $\sup_2(u, v) = (x, y)$. When $u \prec v$ or $v \prec u$, equation A.10 is less than u_y or v_y, respectively. Thus, we take the max of u_y, v_y, and equation A.10, giving us equation A.9. When $v_x < u_x$, $\sup_2(u, v)$ becomes the intersection of the line with slope $1/\sinh(r)$ through v and the line with slope $-1/\sinh(r)$ through u, which gives us the same result.

A.3.2 Proofs for Theorems in Section 2.4

Theorem A.3.1. *In the infinite shadow cone construction, the cone with root farthest away from $\mathcal{S}$ that contains u and v is the minimum height cone that contains u and v.*

Proof. We prove the umbral and penumbral cases separately.

Penumbral Cones: The distance from boundary of $\mathcal{S}$ to a point x is the length of geodesic orthogonal to S through x. In the infinite penumbral construction, S is a horosphere, so this geodesic is the vertical Euclidean line from x to $\mathcal{S}$. Clearly, the longer this line is, the lower x is.

Umbral Cones: Here $\mathcal{S}$ is a point, and geodesics through $\mathcal{S}$ are vertical Euclidean lines orthogonal to the "x-axis" (using the definition of "x-axis" from the main text). As with the penumbral case, since the geodesic from a point x to $\mathcal{S}$ is vertical Euclidean line, the longer this line is the lower x is. □

Theorem A.3.2. *In the infinite shadow cone construction, the root of the minimum height cone that contains u and v lies on the plane containing u, v, and $\mathcal{S}$ (or the ideal point of $\mathcal{S}$).*

Proof. We prove the umbral and penumbral cases separately.

Penumbral Cones: Consider the region of points x s.t. $x \prec u$. This is the region of geodesics through u that intersect $\mathcal{S}$ and is axially symmetric around the vertical Euclidean line A_u through u [203]. Denote this region C_u and its boundary $\mathcal{B}_u$. Note that the ideal point of $\mathcal{S}$, $x_d = \infty$, is the intersection of vertical line geodesics, so all planes through u and $x_d = \infty$ contain A_u. Since C_u is axially symmetric w/r.t. A_u, it follows that it is reflection-symmetric across such planes.

Now, consider the intersection of C_u and C_v, which has boundary $\mathcal{B}_{u\cap v}$. $C_u \cap C_v$ is clearly reflection-symmetric along the plane P containing u, v, and $x_d = \infty$. As such, for any plane P' orthogonal to P, all points on $\mathcal{B}_{u\cap d} \cap P'$ are either on $\mathcal{B}_u$ or $\mathcal{B}_v$ but not both. Since the geodesics that form B_u are monotonically increasing in height in the direction away from A_u (and likewise for v), the minimum height cone that contains u and v must have root in $\mathcal{B}_{u\cap v}$. Furthermore, the root of minimum height cone on $\mathcal{B}_{u\cap v} \cap P'$ is the closest point on $\mathcal{B}_{u\cap v} \cap P'$ to A_u. Since the set of closest points on a plane to a parallel line is the intersection of the orthogonal plane through the line, the minimum height cone on $\mathcal{B}_{u\cap v} \cap P'$ is on P. Thus, the minimum height cone that contains u and v, which is the minimum over P' of minimum height cones on $\mathcal{B}_{u\cap d} \cap P'$, is on P.

Umbral Cones: The proof for the umbral construction is identical to the penumbral construction, except that $\mathcal{S}$ is a point at $x_d = \infty$. □

A.3.3 Implementation and Experiment Details

All experiments were run on a shared GPU cluster with various machine configurations. All GPUs in the cluster were NVIDIA Volta or newer cards, and all machines had Intel Skylake or newer CPUs or AMD Milan or newer CPUs. Most experiments used PyTorch 2.0 with `torch.compile` and TF32 turned on for matrix multiplications. We provide PyTorch implementations of the infinite-setting penumbral and umbral attention operators at `https://github.com/tsengalb99/coneheads`. Below, we give implementation details for each tested model. For penumbral attention, we set the height of $\mathcal{S}$ to $h = 1$. For umbral attention, we set the radius of the ball around each point to $r = 0.1$

Graph Attention Networks. We use the Pytorch-GAT repository (`https://github.com/gordicaleksa/pytorch-GAT`) for our experiments. In this repository, we modified the `//models/definitions/GAT.py` file to implement various attention mechanisms. We use the provided training commands and hyperparameters to train models. We experimented with different hyperparameters, which did not have a significant effect on the final results. We report the default attention results from the original GAT paper.

Neural Machine Translation (NMT) Transformers. For NMT experiments, we used the fairseq repository available at `https://github.com/facebookresearch/fairseq`. We primarily modified `//fairseq/modules/multihead`
We used the commands provided at `https://github.com/facebookresearch/fairseq/blob/main/examples/translation/README.md` to download and preprocess the IWSLT'14 De-En dataset and to train models.

Vision Transformers. For DeiT-Ti experiments, we use the Facebook DeiT repository available at `https://github.com/facebookresearch/deit/blob/main/README_deit.md`. We observed that all methods performed better at 500 epochs than at 300 epochs, and we report our experimental results for 300 and 500 epochs. The DeiT repository relies on the Hugging Face timm repository, which is available at `https://huggingface.co/timm`. We primarily modify `//models/vision_transformer.py` in the timm.

Adaptive Input Representations for Transformers. For adaptive inputs experiments, we use the same fairseq repository as the NMT experiments. The `transformer_lm_wiki103` architecture in the fairseq repository uses 8 attention heads per module, which does not match 16 attention heads per module in in [12]. Thus, our results and those from other papers that use this repository,

such as [212], are not directly comparable to the results presented in [12].

Diffusion Transformers. For DiT-B/4 experiments, we use the code and training instructions available at `https://github.com/facebookresearch/DiT`. We use the Pytorch version of the codebase, and train with seed 42 to match the experiments presented in [131] and the repository. The DiT codebase also uses timm, so we make the same modifications as in DeiT. We report the DiT-B/4 seed 42 PyTorch result for dot product attention from the DiT Github repository.

Runtime Comparison

Table A.7 shows the training speed of the 4 tested transformers in iterations per second. Like dot product attention, cone attention requres $O(n^2d)$ operations. However, cone attention requires more operations, since dot product attention can be computed with a batch matrix multiply. Inside a transformer, cone attention generally results in a 10-20% performance decrease when implemented with `torch.compile`. However, `torch.compile` is not perfect, and an optimized raw CUDA implementation would likely narrow the speed gap between dot product and cone attention. `torch.compile` was not used for the NMT transformer since the training speed was sufficiently fast and, as of writing, `torch.cdist` has issues with dynamic-shaped tensors. We expect this to be fixed in future PyTorch releases and performance to be similar to the Adaptive Inputs transformer.

Table A.7: Training speed in iterations per second of various attention models across the 4 tested transformers. When used in transformers, cone attention is slightly slower than dot product attention. As mentioned in the main text, `torch.compile` is not optimal at fusing operations, and the performance gap can likely be narrowed with a custom CUDA implementation. `torch.compile` was not used for the NMT transformer since the training speed was sufficiently fast and, as of writing, `torch.cdist` has issues with dynamic-shaped tensors. We expect this to be fixed in future PyTorch releases and the performance to be similar to the Adaptive Inputs transformer. All numbers were obtained on a single NVIDIA Tesla V100 SXM2 GPU.

Method	NMT (4096 tokens/it)	Adaptive Inputs (4096 tokens/it)	DeiT-Ti (bs=256)	DiT-B/4 (bs=64)
Dot Product	9.09	0.526	62.9	1.09
Penumbral	7.10 (-21.9%)	0.490 (-6.86%)	51.3 (-18.6%)	1.08 (-0.92%)
Umbral	7.48 (-17.7%)	0.488 (-7.32%)	56.0 (-11.0%)	1.07 (-1.83%)
`torch.compile`?	No	Yes	Yes	Yes

A.3.4 α-approximate rank

Here, we discuss an interesting and potentially useful connection between the problem of encoding hierarchies in dot product attention and the α-approximate rank problem. This connection gives us some intuition as why larger models with large token embedding dimensions perform well, even with hierarchical data. We note that we have not extensively studied this connection and how it relates to training dynamics, such as in the context of learning attention with gradient based optimizers, and leave that for future work.

Consider the following formulation of encoding a partial ordering in attention:

$$K(x, y) = \exp(\gamma P(x, y)) \qquad P(x, y) \in \begin{cases} [1, \alpha] & \text{if } x \preceq y \\ [-\alpha, -1] & \text{if } x \not\preceq y \end{cases} \tag{A.11}$$

where $1 \leq \alpha \leq \infty$. Essentially, for a set of keys and a single query, the attention matrix should give higher "weight" to keys who are descendants of the query,

which is similar to cone attention. If α is close to 1, then this gives a tighter margin on the attention matrix, which gives a "higher quality" attention after softmax normalization. We wish to characterize the number of dimensions d needed in dot product attention to encode a partial ordering with equation A.11. Now, consider the α-approximate rank of a sign matrix A:

Definition A.3.1. ***α-approximate rank:*** *For a sign matrix $A \in \{-1, +1\}^{n \times n}$, the α-approximate rank is defined as the minimum rank of a matrix $A' \in \mathbb{R}^{n \times n}$ such that $J \leq A \circ A' \leq \alpha J$, where J is the all one's matrix, $\alpha \geq 1$, and $\circ$ is the elementwise product.*

In dot product attention, $P = qk^\top$, where $q, k \in \mathbb{R}^{n \times d}$. Note that P has rank d. Furthermore, if the partial ordering is encoded in a sign matrix S s.t. $S_{ij} = +1$ if $i \preceq j$ and -1 otherwise, finding the minimum d s.t. $\exists P$ that satisfies equation A.11 reduces to finding the α-approximate rank of S. A number of works have focused on characterizing the α-approximate rank of arbitrary sign matrices. Below, we summarize some results from [4] and [103]. [103] give the following bounds on the α-approximate rank of a $n \times m$ sign matrix A, denoted $\text{rk}_\alpha(A)$.

$$\frac{1}{\alpha^2}\gamma_2^\alpha(A)^2 \leq \text{rk}_\alpha(A) \leq \frac{8192\alpha^6}{(\alpha-1)^6}\ln^3(4mn)\gamma_2^\alpha(A)^6 \tag{A.12}$$

where $\gamma_2^\alpha(A)$ is defined as

$$\gamma_2^\alpha(A) = \min_{A' : J \leq A \circ A' \leq \alpha J} \gamma_2(A')$$

and $\gamma_2(A)$ is defined as

$$\gamma_2(A) = \max_{u,v : \|u\| = \|v\| = 1} \|A \circ vu^\top\|_{tr},$$

and

$$\text{rk}_{\frac{\alpha+t}{1-t}}(A) \leq \frac{4\gamma_2^\alpha(A)^2 \ln(4mn)}{t^2} \tag{A.13}$$

for $0 < t < 1$. These bounds depend on γ_2^α, which, to the best of our knowledge, has not been well characterized for partial ordering matrices. However, γ_2^α can be computed in polynomial time with a semidefinite program, which may be useful [103]. Equation A.12 gives some indication of how α changes with $d = \text{rk}_\alpha$. From equation A.12

$$\alpha \leq \frac{1}{1 - \sqrt[6]{\frac{8192 \ln^3(4n^2)\gamma_2^\alpha(S)^6}{d}}}$$

For fixed S, as d increases, this upper bound on the achievable α margin decreases with $O(1/(1 - \sqrt[6]{1/d}))$. Finally, if we go in the other direction and set $\alpha = \infty$, finding d reduces to finding the sign-rank of S. From [5], the sign-rank of S is bounded by $SC(S)+1$, where SC is the minimum over all column permutations of S of the maximum number of sign changes in a row of S. If S encodes a hierarchy, and the nodes of that hierarchy are numbered depth first in S, then $SC(S) \leq 2$, and so the sign-rank is at most 3.

APPENDIX B

ROBUST AND EFFICIENT NUMERICAL COMPUTATIONS

B.1 Numerically Accurate Hyperbolic Embeddings Using Tiling-Based Models

B.1.1 Learning and Experiment Details

Learning Details

Efficient Computation in L-tiling Model. For two points $(U, u), (V, v)$ in L-tiling model, the formula to compute distance is

$$\begin{aligned} d((U,u),(V,v)) &= \operatorname{arcosh}\left(h(u)^T L^{-T} Q L^{-1} h(v)\right) \\ &= \operatorname{arcosh}(Q_{11} \cdot d_c) \\ &= \log(Q_{11}) + \log\left(d_c + \sqrt{d_c^2 - Q_{11}^{-2}}\right) \end{aligned}$$

where $Q = -U^T L^T g_{lt} L V, \hat{Q} = \frac{Q}{Q_{11}}, d_c = h(u)^T L^{-T} \hat{Q} L^{-1} h(v)$. Since Q_{11} can be super large, then we extract Q_{11} out here to avoid potential overflow, also there is no underflow problem since Q_{11} is a positive integer. Then the corresponding formula for the gradient of this distance is

$$\begin{cases} \nabla_u d((U,u),(V,v)) = \frac{\nabla h(u)^T L^{-T} \hat{Q} L^{-1} h(v)}{\sqrt{(h(u)^T L^{-T} \hat{Q} L^{-1} h(v))^2 - Q_{11}^{-2}}} = \frac{\nabla h(u)^T L^{-T} \hat{Q} L^{-1} h(v)}{\sqrt{d_c^2 - Q_{11}^{-2}}} \\ \nabla_v d((U,u),(V,v)) = \frac{\nabla h(v)^T L^{-T} \hat{Q}^T L^{-1} h(u)}{\sqrt{(h(u)^T L^{-T} \hat{Q} L^{-1} h(v))^2 - Q_{11}^{-2}}} = \frac{\nabla h(v)^T L^{-T} \hat{Q}^T L^{-1} h(u)}{\sqrt{d_c^2 - Q_{11}^{-2}}} \end{cases}$$

where

$$\nabla h(u) = \left[\frac{u}{\sqrt{1 + \|u\|^2}}, \quad I\right], \nabla h(v) = \left[\frac{v}{\sqrt{1 + \|v\|^2}}, \quad I\right]$$

We provide the error bound for distance and gradient computation in L-tiling model using float arithmetic in Theorem B.1.3, B.1.4, where errors are independent of how far points are from the origin and solves the "NaN" problem.

SGD in the L-tiling model. We offer SGD algorithm below, with the addition of a normalization when the parameter goes out of the L-tiling model, which is performed using Algorithm 2, whose convergence and complexity were shown in Theorem 3.2.3.

Algorithm 10 SGD using group representation

Require: Objective function f, fuchsian group G with fundamental domain $F \subset \mathbb{R}^2$
Require: Tuple $(\theta_t, U_t) \in F \times G$
Require: Number of epochs T, and learning rate α
  for $t =$ to $T - 1$ **do**
    $l_t \Leftarrow \nabla_{\theta_t} f(LU_tL^{-1}h(\theta_t))$ ▷ Euclidean gradient w.r.t. θ_t
    $\theta_{t+1} \Leftarrow \theta_t - \alpha l_t$ ▷ Update $\theta_t \in F$
    if $\theta_{t+1} \notin F$ **then**
      $W \Leftarrow \arg\min_{W \in G} d(LW^{-1}L^{-1}h(\theta_{t+1}), 0)$
      $U_{t+1} \Leftarrow U_t \cdot W$ ▷ Normalize if $\theta_{t+1} \notin F$
      $\theta_{t+1} \Leftarrow LW^{-1}L^{-1}h(\theta_{t+1})$
    else
      $U_{t+1} \Leftarrow U_t$
    end if
  end for
  Output (θ_{t+1}, U_{t+1})

RSGD in the L-tiling model. We show RSGD in the L-tiling model here, which is in correspondence to Algorithm 2. Equivalence of this algorithm to that in Lorentz model is shown in Proof B.1.3 For (U_t, u_t) in the L-tiling model, let f be the objective function, denote $\nabla_{u_t} f$ to be the Euclidean gradient of f w.r.t. u_t. To do RSGD in the L-tiling model, firstly transform the Euclidean gradient to

Riemannian gradient using the pull-back hyperbolic metric:

$$h_t = g_{u_t}^{-1} \nabla_{u_t} f,$$

then project it into the tangent space at u_t,

$$\text{grad}_{u_t} f = h_t + \langle u_t, h_t \rangle_L u_t.$$

then the RSGD algorithm in the L-tiling model updates u^t as follows:

$$u_{t+1} = \exp_{u_t}(v) = \cosh(\|v\|_L) u_t + \sinh(\|v\|_L) \frac{v}{\|v\|_L},$$

where $v = -\eta \cdot \text{grad}_{u_t} f$ and $\|v\|_L = \sqrt{\langle v, v \rangle_L}$.

Efficient Computation in H-tiling Model. For points (j_1, k_1, z_1) and (j_2, k_2, z_2) in H-tiling model, directly computing distance as follows works in most cases,

$$d_h((j_1, \boldsymbol{k}_1, \boldsymbol{z}_1), (j_2, \boldsymbol{k}_2, \boldsymbol{z}_2)) = \text{arcosh}(1 + \frac{\|2^{j_1}(z_1 + k_1) - 2^{j_2}(z_2 + k_2)\|^2}{2^{j_1 + j_2 + 1} z_{1n} z_{2n}})$$

However, we do consider situations where overflows may happen and provide an alternative way to compute distance. Suppose without loss of generality that $j_2 \geq j_2$, then we can write distance as

$$d_h((j_1, \boldsymbol{k}_1, \boldsymbol{z}_1), (j_2, \boldsymbol{k}_2, \boldsymbol{z}_2)) = \text{arcosh}(1 + 2^{j_1 - j_2} \frac{\|z_1 - 2^{j_2 - j_1} z_2 + k_1 - 2^{j_2 - j_1} k_2)\|^2}{2 z_{1n} z_{2n}})$$

let $k_1 - 2^{j_2 - j_1} k_2 = 2^s I$ where s is some natural number scale factor such that $\|I\| < 1$. This is easy to compute exactly using integer arithmetic. (Note that $k_1 - 2^{j_2 - j_1} k_2$ is an integer vector, choose $s = \lceil \log_2(\|k_1 - 2^{j_2 - j_1} k_2\|^2)/2 \rceil$, where the values inside

the $\log_2$ are all integers). Then we get

$$\begin{aligned} d_h((j_1, \boldsymbol{k}_1, z_1), (j_2, \boldsymbol{k}_2, z_2)) &= \operatorname{arcosh}(1 + 2^{j_1 - j_2} \frac{\|2^s I + z_1 - 2^{j_2 - j_1} z_2\|^2}{2 z_{1n} z_{2n}}) \\ &= \operatorname{arcosh}(1 + 2^{2s + j_1 - j_2} \frac{\|I + 2^{-s} z_1 - 2^{j_2 - j_1 - s} z_2\|^2}{2 z_{1n} z_{2n}}) \\ &= \operatorname{arcosh}(1 + 2^{2s + j_1 - j_2} X) \\ &= \log(1 + 2^{2s + j_1 - j_2} X + \sqrt{(1 + 2^{2s + j_1 - j_2} X)^2 - 1}) \\ &= (2s + j_1 - j_2)\log(2) + \log(2^{-2s - j_1 + j_2} + X + \sqrt{X^2 + 2^{1 - 2s - j_1 + j_2} X}) \end{aligned}$$

where

$$X = \frac{\|I + 2^{-s} z_1 - 2^{j_2 - j_1 - s} z_2\|^2}{2 z_{1n} z_{2n}}$$

then we can get following gradients,

$$\begin{cases} \dfrac{\partial d}{\partial X} = \dfrac{1 + \frac{X + 2^{-2s - j_1 + j_2}}{\sqrt{X^2 + 2^{1 - 2s - j_1 + j_2} X}}}{2^{-2s - j_1 + j_2} + X + \sqrt{X^2 + 2^{1 - 2s - j_1 + j_2} X}} \\ \nabla_{z_{1i}} X = \dfrac{I_i + 2^{-s} z_{1i} - 2^{j_2 - j_1 - s} z_{2i}}{2^s z_{1n} z_{2n}} \\ \nabla_{z_{2i}} X = -\dfrac{I_i + 2^{-s} z_{1i} - 2^{j_2 - j_1 - s} z_{2i}}{2^{s + j_1 - j_2} z_{1n} z_{2n}} \\ \nabla_{z_{1n}} X = \dfrac{I_n + 2^{-s} z_{1n} - 2^{j_2 - j_1 - s} z_{2n}}{2^s z_{1n} z_{2n}} - \dfrac{\|I + 2^{-s} z_1 - 2^{j_2 - j_1 - s} z_2\|^2}{2 z_{1n}^2 z_{2n}} \\ \nabla_{z_{2n}} X = -\dfrac{I_n + 2^{-s} z_{1n} - 2^{j_2 - j_1 - s} z_{2n}}{2^{s + j_1 - j_2} z_{1n} z_{2n}} - \dfrac{\|I + 2^{-s} z_1 - 2^{j_2 - j_1 - s} z_2\|^2}{2 z_{1n} z_{2n}^2} \end{cases}$$

Using chain rule, Euclidean gradients of d_h w.r.t. z_1, z_2 can be derived. We provide the error bound for this distance computation in H-tiling model using float arithmetic in Theorem B.1.5. we neglect the error bound for gradient computation here which can also be derived similarly. In H-tiling model, these computation errors are independent of how far points are from the origin and solves the "NaN" problem.

RSGD in the H-tiling model. For (j_t, k_t, z^t) in the H-tiling model, let f be the objective function, denote $\nabla_{z^t} f$ to be the Euclidean gradient of f w.r.t. z^t. To do

RSGD in the H-tiling model, firstly transform the Euclidean gradient to Riemannian gradient using the pull-back metric:

$$\text{grad}_{z^t} f = z_n^t \nabla_{z^t} f$$

take the learning rate η into consideration, denote $v = -\eta \cdot \text{grad}_{z^t} f$, firstly compute its norm as $s = \sqrt{v^T v}$, then the RSGD algorithm in the H-tiling model updates z^t as follows:

$$\begin{cases} z_i^{t+1} = z_i^t + \dfrac{z_n^t}{\frac{s}{\tanh s} - v_n} \cdot v_i \\ z_n^{t+1} = \dfrac{z_n^t}{\cosh s - \frac{\sinh s}{s} v_n} \end{cases}$$

RSGD for the multi-components halfspace model. For two points x_i, y_i stored in the multi-components halfspace model, which represent $u = \sum_{i=0}^{n} 2^{ik} x_i$, $v = \sum_{i=1}^{n} 2^{ik} y_i$ respectively, where k is a constant ($k = 20$). Here $|x_i| < 2^k$ and x_i are integers for $i \geq 1$. To calculate the distance between u, v, we use

$$\begin{aligned} d_u(\boldsymbol{u}, \boldsymbol{v}) &= \text{arcosh}\left(1 + \frac{\|\boldsymbol{u} - \boldsymbol{v}\|^2}{2u_n v_n}\right) \\ &= \text{arcosh}\left(1 + \frac{\|\sum_{i=0}^{n} 2^{ik}(x_i - y_i)\|^2}{2u_n v_n}\right) \end{aligned}$$

programmingly, we calculate $w = \sum_{i=0}^{n} 2^{ik}(x_i - y_i), u_n = \sum_{i=0}^{n} 2^{ik} x_{in}, v_n = \sum_{i=0}^{n} 2^{ik} y_{in}$, and compute the distance as

$$d_u(\boldsymbol{u}, \boldsymbol{v}) = \text{arcosh}\left(1 + \frac{\|w\|^2}{2u_n v_n}\right)$$

for simplicity, denote $t = 1 + \frac{\|w\|^2}{2u_n v_n}$, then

$$d_u(\boldsymbol{x}, \boldsymbol{y}) = \text{arcosh}\,(t) = \log\left(t + \sqrt{t^2 - 1}\right)$$

With these formulas, we can get the derivatives of the distance to u:

$$\begin{cases} \nabla_{u_{0:n-1}} d = \dfrac{1}{\sqrt{t^2 - 1}} \dfrac{w_{0:n-1}}{u_n v_n} \\ \dfrac{\partial d}{\partial u_n} = \dfrac{1}{\sqrt{t^2 - 1}} \left(\dfrac{w_n}{u_n v_n} - \dfrac{\|w\|^2}{2u_n^2 v_n}\right) \end{cases}$$

with $\nabla_{u^t} d$ as this Euclidean gradient of d w.r.t. u^t. We perform RSGD in the multi-components halfspace model as follows, firstly transform the Euclidean gradient to Riemannian gradient using the pull-back metric:

$$\text{grad}_{u^t}\ d = u_n^t \nabla_{u^t} d$$

take the learning rate η into consideration, denote $g = -\eta \cdot \text{grad}_{u^t} d$, firstly compute its norm as $s = \sqrt{g^T g}$, to update the u^t using RSGD algorithm, we update the stored parameters x_i^t as follows, rather than directly update all x_i^t, we firstly only update x_0^t as follows:

$$\begin{cases} x_{0i}^{t+1} = x_{0i}^t + \dfrac{u_n^t}{\frac{s}{\tanh s} - g_n} \cdot g_i \\ x_{0n}^{t+1} = \dfrac{u_n^t}{\cosh s - \frac{\sinh s}{s} g_n} \end{cases}$$

then we normalize and update all x_i such that $|x_i| < 2^k$ and x_i are integers for $i \geq 1$.

Experiment Details

Compression. Compression experiments were implemented in Julia. We compress L-tiling model using storage methods mentioned in Section 3.2.4, round Poincaré ball model towards zero since it is bounded in the Euclidean unit ball, round Lorentz model to the nearest to compress baseline models.

Learning. We implemented learning experiments in PyTorch using float64. Notably, for tiling-based models, we also use float64 to store the integers, in order to avoid potential numerical imprecision when the integers overflow and are out of the expressible range of float64, we developed a secure method to

express a integer matrix U and do accurate integer arithmetic using two float64 type matrices. Specifically, we express it as $U = U_1 + U_2$, where $2^t|U_1, |U_2| < 2^t$, and U_1, U_2 are float64 type. Alternatively, we can similarly use n float64 type matrices to express the integer matrix and pick suitable t to prevent overflow. In our experiments, we found that two float matrices and $t = 20$ are sufficient to prevent overflow and get exact computation of integer arithmetic using float64.

We initialize embeddings randomly from the uniform distribution $U(-0.0001, 0.0001)$, except from embeddings in the H-tiling model, whose last elements z_n were initialized from the uniform distribution $U(1, 1.0001)$ in order to make the division to z_n stable. Matrices $g \in G$ in L-tiling model were initialized to be identity matrices, integer vectors and the exponential integer in H-tiling model were initialized to be zeros. Then we project those embedding to the manifold accordingly before training.

Second, similar as [122], to get a good initial angular layout which is helpful to find good embeddings, we train during an initial "burn-in" phase (20 epochs) with a reduced learning rate $\eta/100$. We train the embedding using multi-threads N to speed up convergence. Those hyperparameters together with batch size b for different datasets were summarized in Table B.1.

Hyperparameters	η	b	N
Gr-QC	0.3	10	2
WordNet Mammals	0.3	10	2
WordNet Verbs	0.5	10	5
WordNet Nouns	0.5	50	5

Table B.1: Hyperparameters for Tiling embedding Learning

B.1.2 More Experiments

Embedding in hyperbolic space reaches better performance compared to Euclidean space, as a simple experiment, consider embedding of a simple tree in Figure B.1, where the lengths of edges are in different scale. When embedded into Euclidean space, the global distortion is 0.1395, the worst-case distortion is 2.15. However, when embedded into Poincaré ball model of hyperbolic space, the global distortion is just 0.0007 and the worst-case distortion is 1.025, which is far better than that in Euclidean space.

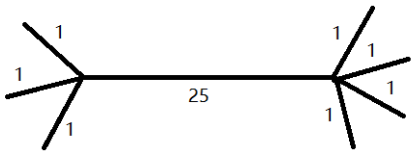


Figure B.1: A simple tree.

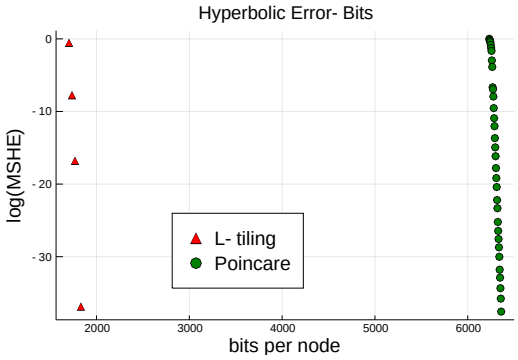


Figure B.2: hyperbolic error for bio-yeast dataset.

Compression Experiments For compression of Bio-yeast dataset mentioned in section 3.2.4, we plot the scatter of the relationship between log(MSHE) and bits to store per node in Figure B.2, under the same MSHE, *L*-tiling model store per node with approximately 3/4 less bits compared to that of Poincaré model.

We also consider compressing embeddings that were trained using optimization algorithms like SGD and RSGD, to learn 2-dimensional Poincaré disk embedding with great performance, but this method fails to learn a great embedding for larger dataset like WordNet Nouns using 2 dimension model, so we

use this method to derive 2-dimensional embeddings for Mammals (Tree-like) and Gr-QC (Dense) dataset. We show the MSHE of different compressions in Table B.2, and it leads to the same conclusion that those compression will not hurt the performance of the embedding such as MAP and MR while largely shrinking the size of embeddings.

Table B.2: Compression performance on Mammals and Gr-QC

Models	Mammals			Gr-QC		
	MSHE	MAP	MR	MSHE	MAP	MR
Poincaré(128b)	0	0.7936	2.36	0	0.5382	73.88
Poincaré(64b)	1.59e-2	0.7935	2.36	1.65e-1	0.5382	73.88
Lorentz(128b)	1.51e-11	0.7935	2.36	1.39e-10	0.5382	73.88
Lorentz(64b)	9.44e-3	0.7935	2.36	1.77e-1	0.5382	73.88
L-tiling-f32	5.17e-08	0.7935	2.36	5.29e-8	0.5382	73.88
L-tiling-f16	4.16e-04	0.7935	2.36	4.19e-4	0.5382	73.88

Learning Experiments. We further include the embedding learning results where all models were trained using float32 in PyTorch as shown in Table B.3.

B.1.3 Mathematical Background and Proofs for Theorems in Section 3.2

Definition 3.2.1. *[Representation error] We are concerned with representing points in hyperbolic space $\mathbb{H}^n$ using floating-points* fl. *Define the* representation error *of a particular point $x \in \mathbb{H}^n$ as $\delta_{\mathrm{fl}}(x) = d_{\mathbb{H}^n}(x, \mathrm{fl}(x))$, and the* worst case representation error *of floating-points representation as a function of the distance-to-origin d, which*

Table B.3: Embedding experiments with float32

Dimension	Models	WordNet Nouns		WordNet Verbs		Gr-QC	
		MAP	MR	MAP	MR	MAP	MR
2	Poincaré	0.092	95.01	0.478	6.24	0.566	69.11
	Lorentz	0.371	19.07	0.701	2.35	0.556	**63.62**
	L-tiling-rsgd	**0.390**	**17.52**	0.721	2.36	0.564	71.36
	L-tiling-sgd	0.341	21.53	0.726	**2.10**	**0.582**	65.19
	H-tiling	0.385	17.70	**0.741**	2.28	0.568	65.84
4	2*L-tiling-rsgd	/	/	0.858	1.37	0.717	12.32
5	Poincaré	0.850	4.76	**0.953**	1.23	0.712	34.77
	Lorentz	0.851	3.78	0.935	1.19	0.712	33.95
	H-tiling	**0.869**	**3.62**	**0.953**	**1.17**	**0.716**	**32.57**
6	3*L-tiling-rsgd	/	/	0.935	1.13	**0.852**	**4.45**
10	Poincaré	0.875	3.88	**0.954**	1.23	**0.730**	29.86
	Lorentz	0.872	3.45	0.951	**1.14**	0.724	29.50
	H-tiling	**0.894**	**3.25**	**0.954**	1.15	0.726	**29.45**

is the maximum representation error of any point with a distance-to-origin at most d,

$$\delta_{\mathrm{fl}}^{d} = \max_{x \in \mathbb{H}^n,\, d_{\mathbb{H}^n}(x,O) \le d} \delta_{\mathrm{fl}}(x).$$

Definition B.1.1 ([38, 45, 158, 137])**.** *A tiling of the plane is a collection of sets ("tiles") whose union is the entire plane, but the interiors of different tiles are disjoint. A uniform tiling is an edge-to-edge filling of the hyperbolic plane, which has regular congruent polygons as faces and is vertex-transitive (there is an isometry mapping any vertex onto any other).*

Definition B.1.2 ([90, 15, 176])**.** *A Fuchsian group G is a discrete subgroup of the 2×2 projective special linear group over $\mathbb{R}$, $PSL(2, \mathbb{R})$.*

Definition B.1.3 ([176])**.** *The Dirichlet domain for G centered at $z_0 \in \mathbb{H}^2$ is*

$$\square(G; z_0) = \{z \in \mathbb{H}^2 : d(z, z_0) \le d(gz, z_0), \forall g \in G\}$$

Definition B.1.4 ([176, 175])**.** *A fundamental domain for G is a closed set $F \subset \mathbb{H}^2$ such that*

- $\{gx|\forall g \in G, x \in F\} = \mathbb{H}^2$
- $\{gx|\forall x \in F^{\circ}\} \cap F^{\circ} = \emptyset, \forall g \in G \setminus \{1\}$, *where* $^{\circ}$ *denotes the interior.*

Theorem 3.2.1. *The worst-case representation error (Definition 3.2.1) in the Lorentz model using floating-point arithmetic (with machine epsilon* ϵ_m*) is* $\delta_l^d = \operatorname{arcosh}(1 + \epsilon_m(2\cosh^2(d) - 1))$*, where* d *is the hyperbolic distance to origin. This becomes* $\delta_l^d = 2d + \log(\epsilon_m) + o(\epsilon_m^{-1}\exp(-2d))$ *if* $d = O(-\log\epsilon_m)$.

Theorem 3.2.4. *The representation error (Definition 3.2.1) in L-tiling model is bounded as* $\delta_{lt}^d \leq \sqrt{5\epsilon_m} + 15\epsilon_m/4 + o(\epsilon_m)$*, where* ϵ_m *is the machine error.*

Proof of Theorem 3.2.1 and Theorem 3.2.4. For a real point $(g, \boldsymbol{x})$ in L-tiling model, where $g \in G$, $\boldsymbol{x} \in F$, we represent it as $(g, \mathrm{fl}(\boldsymbol{x}))$, then we get the representation error as follows:

$$\delta_{lt}^d = d_{lt}((g,\boldsymbol{x}),(g,\mathrm{fl}(\boldsymbol{x}))) = \operatorname{arcosh}(-\boldsymbol{x}^T g^T g_{lt} g\mathrm{fl}(\boldsymbol{x})) = \operatorname{arcosh}(-\boldsymbol{x}^T g_{lt}\mathrm{fl}(\boldsymbol{x}))$$

Note that $|x_i - \mathrm{fl}(x_i)| \leq \epsilon_m x_i$, so we have

$$\begin{aligned}\delta_{lt}^d &= \operatorname{arcosh}\left(-(x_1,x_2,x_3)\, g_{lt}\begin{pmatrix}(1+\epsilon_1)x_1\\(1+\epsilon_2)x_2\\(1+\epsilon_3)x_3\end{pmatrix}\right)\\ &= \operatorname{arcosh}((1+\epsilon_1)x_1^2 - (1+\epsilon_2)x_2^2 - (1+\epsilon_3)x_3^2)\\ &= \operatorname{arcosh}(1+\epsilon_1 x_1^2 - \epsilon_2 x_2^2 - \epsilon_3 x_3^2)\end{aligned}$$

since $x_1^2+x_2^2+x_3^2 \leq B_f$, then we derive that $\delta_{lt}^d \leq \operatorname{arcosh}(1+\epsilon_m B_f) = \sqrt{\epsilon_m B_f} + 3\epsilon_m B_f/4 + o(\epsilon_m)$, simple calculation will lead to $B_f = 5$, then $\delta_{lt}^d \leq \sqrt{5\epsilon_m} + 15\epsilon_m/4 + o(\epsilon_m)$. If we consider the representation error in Lorentz model, the only difference is that x is not bounded in the fundamental domain any more. Then we can get that $\delta_l^d = \operatorname{arcosh}(1+\epsilon_m\|x\|^2)$, noticed that $\cosh d = -\boldsymbol{x}^T g_{lt}\boldsymbol{O} = x_1$, where d is the distance

to origin, then

$$\delta_l^d = \operatorname{arcosh}(1+\epsilon_m(x_1^2+x_2^2+x_3^2)) = \operatorname{arcosh}(1+\epsilon_m(2x_1^2-1)) = \operatorname{arcosh}(1+\epsilon_m(2\cosh^2(d)-1)),$$

which becomes $\delta_l^d = 2d + \log(\epsilon_m) + o(\epsilon_m^{-1}\exp(-2d))$ if $d = O(-\log\epsilon_m)$, this error also generalize similarly to high dimensional Lorentz model. □

Theorem 3.2.2. *$F = \{(x_1, x_2, x_3) \in \mathcal{L}^2 | \max(2x_2^2 - x_3^2, 2x_3^2 - x_2^2) < 1\}$ is a fundamental domain of G. Any point in $\mathcal{L}^2$ can be mapped by G to one unique point in F or to a point on its boundary.*

Proof. To begin with, we prove that F is the Dirichlet domain for G centered at $O \in \mathcal{L}^2$ denoted as $\square(G)$. Firstly, we show that $F \subset \square(G)$, that is, for any $z \in F$, we have $d(z, O) \leq d(Uz, O)$ for all $U \in G$. It suffices to show

$$z_1 \leq z_1u_{11} + z_2u_{12} + z_3u_{13},$$

where $z_1^2 = 1 + z_2^2 + z_3^2$. Also note that $U \in G$, then $U^T g_l U = g_l$, from which we can derive that $u_{11}^2 = 1 + u_{12}^2 + u_{13}^2$. Further, from the construction of G, we can write $u_{11} = t_{11}, u_{12} = \sqrt{3}t_{12}, u_{13} = \sqrt{3}t_{13}$, where t_{1i} is an integer, then $t_{11}^2 = 1 + 3t_{12}^2 + 3t_{13}^2$.

Consider following:

$$
\begin{aligned}
& z_1 \leq z_1 u_{11} + z_2 u_{12} + z_3 u_{13} \\
\iff & z_1 \leq z_1 t_{11} + \sqrt{3} z_2 t_{12} + \sqrt{3} z_3 t_{13} \\
\iff & -\sqrt{3}(z_2 t_{12} + z_3 t_{13}) \leq z_1 (t_{11} - 1) \\
\impliedby & 3(z_2 t_{12} + z_3 t_{13})^2 \leq z_1^2 (t_{11} - 1)^2 && \triangleright\ z_1, t_{11} \geq 1 \\
\iff & 3(z_2^2 t_{12}^2 + z_3^2 t_{13}^2 + 2 t_{12} t_{13} z_2 z_3) \leq (1 + z_2^2 + z_3^2)(1 + 3t_{12}^2 + 3t_{13}^2 - 2t_{11} + 1) \\
\iff & 6 t_{12} t_{13} z_2 z_3 \leq (3t_{12}^2 + 3t_{13}^2 - 2t_{11} + 2) + z_2^2 (3t_{13}^2 - 2t_{11} + 2) + z_3^2 (3t_{12}^2 - 2t_{11} + 2) \\
\impliedby & 2 t_{11} z_1^2 \leq (3t_{12}^2 + 3t_{13}^2) + 2 z_1^2 && \triangleright\ z_2^2 t_{13}^2 + z_3^2 t_{12}^2 \geq 2 z_2 z_3 t_{13} t_{12} \\
\iff & 2 t_{11} z_1^2 \leq (t_{11}^2 - 1) + 2 z_1^2 \\
\iff & 2 z_1^2 (t_{11} - 1) \leq t_{11}^2 - 1 \\
\iff & 2 z_1^2 \leq t_{11} + 1 && \triangleright\ t_{11} - 1 \geq 0 \\
\impliedby & 5 \leq t_{11} && \triangleright\ z_1 \leq \sqrt{3} \impliedby z \in F
\end{aligned}
$$

Hence, if $t_{11} \geq 5$, then the inequality is proven. If $t_{11} < 5$, since t_{11} is an integer, then $t_{11} = 1, 2, 3, 4$.

- If $t_{11} = 1$, then U is identity matrix, the inequality is satisfied.
- If $t_{11} = 2$, then the integer solutions to $t_{11}^2 = 1 + 3t_{12}^2 + 3t_{13}^2$ is $\{g_a, g_b, g_a^{-1}, g_b^{-1}\}$, the inequality is satisfied by simply checking one by one.
- If $t_{11} = 3$, then there is no integer solutions to $t_{11}^2 = 1 + 3t_{12}^2 + 3t_{13}^2$.
- If $t_{11} = 4$, then the solutions to $t_{11}^2 = 1 + 3t_{12}^2 + 3t_{13}^2$ is $(t_{11}, t_{13}, t_{13}) = (4, 2, 1), (4, 1, 2)$. Manually check will find that the inequality is satisfied in both cases.

Then $F \subset \square(G)$, also note that with Algorithm 1, for any $z \in \square(G)$, it will be mapped by some $V \in G$ such that $Vz \in F$ and $d(Vz, O) \leq d(z, O)$. From the

definition of $\square(G)$, $d(Vz, O) \geq d(z, O)$, then $(Vz)^T g_l O = d(Vz, O) = d(z, O) = z^T g_l O$, which leads to $V_{11} = 1$, then $V = I$ considering $V^T g_l V = g_l$, thus, $z = Vz \in F$ and $\square(G) \subset F$, which shows that $F = \square(G)$.

For the second part of the proof, we show that F is a fundamental domain for G. According to Theorem 37.1.10 in [176], it suffices to show that $Stab_G(O) = \{1\}$. Consider $TO_H = O_H$, where $T \in G$. From $(T - I)O_H = 0$, we can get that

$$T = \begin{bmatrix} 1 & 0 \\ 0 & B \end{bmatrix}.$$

where $B^T = B$. Also note $T \in G$, then $B^2 = I$, these two conditions lead to that $B = I$. Hence, $Stab_G(O) = \{1\}$, then F is a fundamental domain for G. Since fundamental domain F only contains one element in the orbit, then for any point in the space, it can only be mapped to one unique point in F. □

Theorem 3.2.3. *For any point in the Lorentz model, Algorithm 2 converges and stops within* $1 + 7d$ *steps, where* $d = d(\boldsymbol{x}, \boldsymbol{O})$ *denotes the distance from* $\boldsymbol{x}$ *to the origin.*

Proof. We just consider $x_2 \leq -|x_3|$ case in the Algorithm (other cases can be proved in the same way). Here we have

$$\frac{\cosh d(L \cdot g_a \cdot L^{-1} \cdot \boldsymbol{x}, O)}{\cosh d(\boldsymbol{x}, \boldsymbol{O})} = 2 + \frac{\sqrt{3}x_2}{\sqrt{1 + x_2^2 + x_3^2}} < 1$$

then the distance to the origin in the space is monotonically decreasing as Algorithm 1 goes, note that this distance is bounded by 0, then it will converge.

To see the steps required for the algorithm to finish, we may assume that $\max\{|x_2|, |x_3|\} \geq C_0 > 1$, due to the symmetry of this algorithm, also consider the case $x_2 \leq -|x_3|$, then $|x_2| \geq C_0$, we have

$$\frac{\cosh d(L \cdot g_a \cdot L^{-1} \cdot \boldsymbol{x}, \boldsymbol{O})}{\cosh d(\boldsymbol{x}, \boldsymbol{O})} \leq 2 - \sqrt{\frac{3C_0}{2C_0 + 1}} \leq 1$$

Hence,

$$\begin{aligned}
&d(\boldsymbol{x},\boldsymbol{O}) - d(L\cdot g_a\cdot L^{-1}\cdot\boldsymbol{x},\boldsymbol{O})\\
&\geq \operatorname{arcosh}(\sqrt{1+x_2^2+x_3^2}) - \operatorname{arcosh}\left((2-\sqrt{\frac{3C_0}{2C_0+1}})\sqrt{1+x_2^2+x_3^2}\right)\\
&\geq -\log(2-\sqrt{\frac{3C_0}{2C_0+1}})
\end{aligned}$$

so $d(\boldsymbol{x},\boldsymbol{O})$ will decrease monotonically for at most $s_0(C_0)$ steps, where

$$s_0(C_0) = \frac{d(\boldsymbol{x},\boldsymbol{O})}{-\log(2-\sqrt{\frac{3C_0}{2C_0+1}})}$$

Consider $\max\{|x_2|,|x_3|\}$ at the boundary between F and its neighborhood tiles:

$$\min_{x\in L\{g_a,g_b,g_a^{-1},g_b^{-1}\}L^{-1}F} \max\{|x_2|,|x_3|\} = \frac{5}{2}\sqrt{2}$$

Hence, we can choose $C_0 = \frac{5}{2}\sqrt{2}$, then $d(\boldsymbol{x},\boldsymbol{O})$ will decrease monotonically until $\max\{|x_2|,|x_3|\} < C_0$ within $s_0(5\sqrt{2}/2)$ steps, which means x lies either in F or its 4 neighborhood tiles, so totally it will cost at most $s_0(5\sqrt{2}/2)+1 \leq 1+7d$ steps. □

Lemma B.1.1. *If the integer matrix U is given, then corresponding VBW encoding can be derived using Algorithm 2. Further, if only U_{21}, U_{31} are given, then U can be reconstructed using Algorithm 2.*

Proof. If U is given, consider the point (U,O) in the L-tiling model, which is in correspondence to $x = LUL^{-1}O$ in the Lorentz model, then we can map x to (U',u') with Algorithm 2, where we choose a generator at each step to get a generator order string, with which we can reconstruct U'. Since each point in the Lorentz model will be mapped to a unique point in F as Theorem 3.2.2 states, also x can be mapped to (U,O) and (U',u'), then $u' = O$. The question is whether $U = U'$, consider $LUL^{-1}O = LU'L^{-1}O$, which leads to $(LU'^{-1}UL^{-1} - I)O = 0$, since $Stab_G(O) = \{1\}$ as the second part in the proof of Theorem 3.2.2 proved, then

$LU'^{-1}UL^{-1} = I$ to get $U = U'$. Hence, given U, we can get its generator order string, which can be then used to get the VBW encoding accordingly. Further, note that $U^T M_3 U = M_3$, then we have

$$U_{11}^2 = 1 + \frac{U_{21}^2 + U_{31}^2}{3}.$$

Therefore, we can compute U_{11} if only U_{21}, U_{31} were given to get the first column of U. Since $x = LUL^{-1}O$ and $O = (1, 0, 0)$, then the first column of U suffices to get x, then we can reconstruct U out using Algorithm 2. □

Lemma B.1.2. *Q_{11} has the largest absolute value in $Q = U^T M_3 V$, where $M_3 = -L^T g_{lt} L$.*

Proof. Note that $Q = U^T M_3 V = M_3 U^{-1} V = M_3 T$, where T is an integer matrix generated by g_a and g_b, so we have $T^T M_3 T = M_3$, using this relation, we can get following equations:

$$\begin{aligned} t_{11}^2 &= 1 + \frac{t_{21}^2 + t_{31}^2}{3} \\ t_{11}^2 &= 1 + 3(t_{12}^2 + t_{13}^2) \\ 3t_{12}^2 &= (t_{22}^2 + t_{32}^2) - 1 \\ 3t_{13}^2 &= (t_{23}^2 + t_{33}^2) - 1 \end{aligned}$$

since $Q_{11} = 3t_{11}$, from first formula, we get that $Q_{11}^2 = 9t_{11}^2 \geq 3t_{11}^2 > t_{21}^2 + t_{31}^2 = Q_{21}^2 + Q_{31}^2$, so Q_{11} has the largest absolute value in the first column of Q. From the second formula, we get that $Q_{11}^2 \geq t_{11}^2 > t_{12}^2 + t_{13}^2$, so Q_{11} has the largest absolute value in the first row of Q. Then combine formulas 2,3,4 and we get that

$$1 + t_{11}^2 = t_{22}^2 + t_{32}^2 + t_{23}^2 + t_{33}^2$$

From first formula, we know that $t_{11} \geq 1$, then we have $Q_{11}^2 \geq 2t_{11}^2 \geq 1 + t_{11}^2 = t_{22}^2 + t_{32}^2 + t_{23}^2 + t_{33}^2$ Combine above results, clearly that Q_{11} has the largest absolute value in Q, which finishes our proof □

Theorem 3.2.5. *The representation error (Definition 3.2.1) in H-tiling model is bounded as* $\delta_{ht}^d = \sqrt{(n+3)\epsilon_m}/2 + (n+3)\epsilon_m/4 + o(\epsilon_m)$*, where* ϵ_m *is the machine error.*

Proof. For a real point $(j, \boldsymbol{k}, \boldsymbol{x}) \in \mathbb{Z}\times(\mathbb{Z}^{n-1}\times\{0\})\times S$ in H-tiling model, we represent it as $(j, \boldsymbol{k}, \mathrm{fl}(\boldsymbol{x}))$, then we get the representation error as follows:

$$\delta_{ht}^d = d_{ht}((j, \boldsymbol{k}, \boldsymbol{x}), (j, \boldsymbol{k}, \mathrm{fl}(\boldsymbol{x}))) = \mathrm{arcosh}(1 + \frac{\|\boldsymbol{x} - \mathrm{fl}(\boldsymbol{x}))\|^2}{2x_n \mathrm{fl}(x_n)})$$

Note that $|x_i - \mathrm{fl}(x_i)| \leq \epsilon_m x_i$, so we have

$$\delta_{ht}^d = \mathrm{arcosh}(1 + \frac{\sum_{i=1}^{n} \epsilon_i x_i^2}{2(1+\epsilon_n)x_n^2})$$

since $0 \leq x_1, \cdots, x_{n-1} < 1 \leq x_n < 2$, then we derive that $\delta_{ht}^d \leq \mathrm{arcosh}(1 + \frac{(n+3)\epsilon_m}{2(1-\epsilon_m)}) = \sqrt{(n+3)\epsilon_m}/2 + (n+3)\epsilon_m/4 + o(\epsilon_m)$. □

Proof of RSGD algorithm. Here we show the equivalence between the RSGD algorithm of L-tiling model described in appendix B.1.1 and that in Lorentz model. To begin with, consider the RSGD algorithm in Lorentz model. Let $x_t, y_t \in \mathcal{H}^2$, then we have $d(x_t, y_t) = \mathrm{arcosh}(-x_t^T g_l y_t)$, the Euclidean gradient of x_t can be computed as

$$\nabla_{x_t} d(x_t, y_t) = \frac{-g_l y_t}{\sqrt{(x_t^T g_l y_t)^2 - 1}},$$

to get the Riemannian gradient in the model, we make use of the pull-back metric as follows,

$$h_t = g_{l,x_t}^{-1} \nabla_{x_t} d(x_t, y_t) = \frac{-y_t}{\sqrt{(x_t^T g_l y_t)^2 - 1}},$$

further we project this Riemannian gradient into the tangent space at x_t,

$$\mathrm{grad}_{x_t} d = h_t + \langle x_t, h_t \rangle_L x_t = -\frac{y_t + (x_t^T g_l y_t)x_t}{\sqrt{(x_t^T g_l y_t)^2 - 1}},$$

then we make use of the exponential map in Lorentz model to update,

$$x_{t+1} = \exp_{x_t}(v) = \cosh(\|v\|_L)x_t + \sinh(\|v\|_L)\frac{v}{\|v\|_L},$$

where

$$v = -\eta \cdot \mathrm{grad}_{x_t} d = \eta \frac{y_t + (x_t^T g_l y_t) x_t}{\sqrt{(x_t^T g_l y_t)^2 - 1}}.$$

Now consider the norm of v under hyperbolic metric,

$$\begin{aligned}
\|v\|_L^2 =& v^T g_l v \\
=& \frac{\eta^2}{(x_t^T g_l y_t)^2 - 1} (y_t^T g_l y_t + (x_t^T g_l y_t)^2 + (x_t^T g_l y_t)^2 + (x_t^T g_l y_t)^2 x_t^T g_l x_t) \\
=& \frac{\eta^2}{(x_t^T g_l y_t)^2 - 1} (-1 + (x_t^T g_l y_t)^2 + (x_t^T g_l y_t)^2 - (x_t^T g_l y_t)^2) \\
=& \eta^2
\end{aligned}$$

hence we derived the RSGD algorithm in the Lorentz model as

$$x_{t+1} = \exp_{x_t}(v) = \cosh(\eta) x_t + \sinh(\eta) \frac{y_t + (x_t^T g_l y_t) x_t}{\sqrt{(x_t^T g_l y_t)^2 - 1}} \tag{B.1}$$

For the second part, we turn to L-tiling model, let $x_t = LUL^{-1}u_t, y_t = LVL^{-1}v_t$, the distance is

$$d(x_t, y_t) = \mathrm{arcosh}(u_t^T L^{-T} Q L^{-1} v_t),$$

then the Euclidean gradient of u_t can be computed as

$$\nabla_{u_t} d(x, y) = \frac{L^{-T} Q L^{-1} v_t}{\sqrt{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1}}.$$

In the same way, make use of the pull-back by the metric matrix, we derived the Riemannian gradient

$$h_t = g_{lt,u_t}^{-1} \nabla_{u_t} d(x_t, y_t) = \frac{g_{lt} L^{-T} Q L^{-1} v_t}{\sqrt{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1}},$$

also project it into the tangent space at u_t,

$$\mathrm{grad}_{u_t} d = h_t + \langle u_t, h_t \rangle_L u_t = \frac{g_{lt} L^{-T} Q L^{-1} v_t + (v_t^T L^{-T} Q^T L^{-1} u_t) u_t}{\sqrt{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1}}.$$

then the update rule in the tiling-based model is

$$u_{t+1} = \exp_{u_t}(v),$$

where

$$v = -\eta \cdot \mathrm{grad}_{u_t} d = -\eta \frac{g_{lt} L^{-T} Q L^{-1} v_t + (v_t^T L^{-T} Q^T L^{-1} u_t) u_t}{\sqrt{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1}},$$

then consider the norm of v under hyperbolic metric,

$$\begin{aligned}
\|v\|_L^2 =& v^T g_{lt} v \\
=& \frac{\eta^2}{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1} [g_{lt} L^{-T} Q L^{-1} v_t + (v_t^T L^{-T} Q^T L^{-1} u_t) u_t]^T \cdot \\
& \qquad g_{lt} [g_{lt} L^{-T} Q L^{-1} v_t + (v_t^T L^{-T} Q^T L^{-1} u_t) u_t] \\
=& \frac{\eta^2}{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1} [v_t^T L^{-T} Q^T L^{-1} g_{lt} L^{-T} Q L^{-1} v_t \\
& \qquad + 2(v_t^T L^{-T} Q^T L^{-1} u_t)^2 + (v_t^T L^{-T} Q^T L^{-1} u_t) u_t^T g_{lt} u_t] \\
=& \frac{\eta^2}{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1} [v_t^T L^{-T} Q^T L^{-1} g_{lt} L^{-T} Q L^{-1} v_t \\
& \qquad + 2(v_t^T L^{-T} Q^T L^{-1} u_t)^2 - (v_t^T L^{-T} Q^T L^{-1} u_t)] \\
=& \frac{\eta^2}{(u_t^T L^{-T} Q L^{-1} v_t)^2 - 1} [(L^{-T} Q L^{-1} v_t)^T g_{lt} (L^{-T} Q L^{-1} v_t) + (v_t^T L^{-T} Q^T L^{-1} u_t)^2]
\end{aligned}$$

Since $U \in G_0$, then it follows

$$Q = U^T M_3 V = M_3 U^{-1} V = M_3 W, \quad W \in G.$$

So we get

$$L^{-T} Q L^{-1} v_t = L^{-T} M_3 W L^{-1} v_t,$$

hence

$$\begin{aligned}
(L^{-T} Q L^{-1} v_t)^T g_{lt} (L^{-T} Q L^{-1} v_t) =& v_t^T L^{-T} W^T M_3^T L^{-1} g_{lt} L^{-T} M_3 W L^{-1} v_t \\
=& - v_t^T L^{-T} W^T M_3 W L^{-1} v_t \\
=& - v_t^T L^{-T} M_3 L^{-1} v_t \\
=& v_t^T g_{lt} v_t \\
=& - 1
\end{aligned}$$

so $\|v\|_L^2 = \eta^2$, then the RSGD algorithm in L-tiling model is

$$u_{t+1} = \exp_{u_t}(v) = \cosh(\eta)u_t - \sinh(\eta)\frac{g_{lt}L^{-T}QL^{-1}v_t + (v_t^T L^{-T}Q^T L^{-1}u_t)u_t}{\sqrt{(u_t^T L^{-T}QL^{-1}v_t)^2 - 1}} \tag{B.2}$$

For the third part, again consider the RSGD algorithm in Lorentz model, from B.1, we have that

$$LUL^{-1}u_{t+1} = \cosh(\eta)LUL^{-1}u_t + \sinh(\eta)\frac{LVL^{-1}v_t - (v_t^T L^{-T}Q^T L^{-1}u_t)LUL^{-1}u_t}{\sqrt{(v_t^T L^{-T}Q^T L^{-1}u_t)^2 - 1}},$$

so RSGD algorithm in Lorentz model is equivalent to:

$$u_{t+1} = \cosh(\eta)u_t + \sinh(\eta)\frac{LU^{-1}VL^{-1}v_t - (v_t^T L^{-T}Q^T L^{-1}u_t)u_t}{\sqrt{(v_t^T L^{-T}Q^T L^{-1}u_t)^2 - 1}}, \tag{B.3}$$

note that $U^T M_3 U = M_3$, then $U^{-1} = M_3^{-1}U^T M_3$, with simple computation, we get that

$$LU^{-1}VL^{-1} = LM_3^{-1}U^T M_3 VL^{-1} = LM_3^{-1}QL^{-1} = -gL^{-T}QL^{-1},$$

hence, RSGD algorithm in L-tiling model (Equation B.2) becomes the same as RSGD algorithm in Lorentz model (Equation B.3), which finishes our proof. □

Error for Computing in L-tiling Model. We approximate $(U, u), (V, v)$ with $(U, \mathrm{fl}(u))$ and $(V, \mathrm{fl}(v))$, here we provide the error of computing in L-tiling model together with that in Lorentz model.

Theorem B.1.3. *The worst case error of computing distance in Lorentz model using float is*

$$|d_l^{fl}(fl(x), fl(y)) - d_l(x, y)| = \frac{2\cosh(d(x, O))\cosh(d(y, O))}{\sinh(d(x, y))}\epsilon_m + \epsilon_m d(x, y) + o(\epsilon_m),$$

the error of computing distance in L-tiling model using float is

$$|d_{lt}^{fl}((U, fl(u)), (V, fl(v))) - d_{lt}((U, u), (V, v))| = \begin{cases} d\epsilon_m + A_1(C_0)\epsilon_m + o(C_0^{-2} + \epsilon_m), d \geq C_0 \\ d\epsilon_m + [A_2(C_0) + \dfrac{A_3(C_0)}{\tanh(d)}]\epsilon_m + o(\epsilon_m), d < C_0 \end{cases}$$

where d is the real distance between two points, $A_i(C_0)$ are constants only depends on C_0, d^{fl} means that inside computation like multiplication are performed with machine error ϵ_m.

Remark: The worst case error of distance computation in Lorentz model using float is dominated by $d(x, O), d(y, O)$, this will cause the "NaN" problem when two points are far away from the origin. However, in L-tiling model, the error only depends on d, i.e., how far two points are to each other, also tanh term is bounded, which controls the distance error and solves the "NaN" problem.

Proof. We consider the Lorentz model at first, let $z = x^T My, \hat{z} = \text{fl}(x)^T M\text{fl}(y)$, then

$$\begin{aligned}|\hat{z} - z| \leq |(1+\epsilon_m)^6 z - z| &= 6\epsilon_m |x|^T |y| + o(\epsilon_m) \leq 2\epsilon_m x_0 y_0 + o(\epsilon_m) \\ &= 2\epsilon_m \cosh(d_x)\cosh(d_y) + o(\epsilon_m) = \delta_z\end{aligned}$$

thus,

$$\begin{aligned}|d_l^{\text{fl}}(\text{fl}(x), \text{fl}(y)) - d_l(x, y)| &= |(1+\epsilon)\,\text{arcosh}\,(x^T My + \delta_z) - \text{arcosh}\,(x^T My)| \\ &= \frac{\delta_z}{\sqrt{(x^T My)^2 - 1}} + \epsilon d_l + o(\delta_z) = \frac{2\cosh(d_x)\cosh(d_y)}{\sinh(d_l)}\epsilon_m + \epsilon_m d_l + o(\epsilon_m)\end{aligned}$$

As for the distance error in L-tiling model, here in the same way denote $z = h(u)^T L^{-T} \hat{Q} L^{-1} h(v)$, since $h(u), h(v)$ are in the fundamental domain, which is bounded by B_f, so $||h(u)||, ||h(v)|| \leq \sqrt{B_f} = \sqrt{5}$, also $\hat{Q}$ is bounded, then using Cauchy inequality, we have

$$|z| \leq ||h(u)||(\frac{\sqrt{7}}{3}|h(v)_1| + \sqrt{\frac{7}{3}}|h(v)_2| + \sqrt{\frac{7}{3}}|h(v)_3|) \leq \frac{7}{3}||h(u)||||h(v)|| \leq \frac{7}{3}B_f^2 = 35/3$$

so the distance is

$$d_{lt}((U,u),(V,v)) = \log(Q_{11}) + \log\left(z + \sqrt{z^2 - Q_{11}^{-2}}\right)$$

then $\log(Q_{11}) \leq d$, further note that

$$\begin{aligned}
\operatorname{arcosh}(Q_{11}/3) =& d(L^{-T}UL^{-1}O, L^{-T}VL^{-1}O) \\
\leq& d(L^{-T}UL^{-1}O, L^{-T}UL^{-1}h(u)) + d(L^{-T}UL^{-1}h(u), L^{-T}VL^{-1}h(v)) \\
&+ d(L^{-T}VL^{-1}O, L^{-T}VL^{-1}h(v)) \\
=& d(O, h(u)) + d(L^{-T}UL^{-1}h(u), L^{-T}VL^{-1}h(v)) + d(O, h(v)) \\
\leq& d + 2\operatorname{arcosh}(\sqrt{3}) = d + 2\log(\sqrt{3} + \sqrt{2})
\end{aligned}$$

in this way, we get

$$\begin{aligned}
z^2 = z^2 - Q_{11}^{-2} + Q_{11}^{-2} =& Q_{11}^{-2}(\cosh^2(d) - 1) + Q_{11}^{-2} \geq Q_{11}^{-2} + \frac{\sinh^2(d)}{9\cosh^2(d + 2B_f)} \\
z =& Q_{11}^{-1}\cosh(d) \geq \frac{\cosh(d)}{3\cosh(d + 2\log(\sqrt{3} + \sqrt{2}))}
\end{aligned}$$

Now, we consider the first term of calculating distance $\log(Q_{11})$, in order to avoid overflow, we computed with following formula.

$$\log(Q_{11}) = \frac{\log(3)}{2} + \log(U^{11}) + \log(V_{11}) + \log(1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}})$$

then

$$\begin{aligned}
\mathrm{flc}(\log(Q_{11})) &= \left(\frac{\log(3)}{2} + \log(U^{11}) + \log(V_{11})\right)(1+\epsilon) \\
&\quad + \mathrm{fl}\left(\log(1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}}(1+\epsilon)^2 + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}(1+\epsilon)^2)\right) \\
&= \left(\frac{\log(3)}{2} + \log(U^{11}) + \log(V_{11})\right)(1+\epsilon) \\
&\quad + \left(\log(1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}}(1+\epsilon)^2 + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}(1+\epsilon)^2)\right)(1+\epsilon) \\
&= \left(\frac{\log(3)}{2} + \log(U^{11}) + \log(V_{11})\right)(1+\epsilon) \\
&\quad + (1+\epsilon)\log(1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}}(1+2\epsilon) + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}(1+2\epsilon) + o(\epsilon)) \\
&= \left(\frac{\log(3)}{2} + \log(U^{11}) + \log(V_{11})\right)(1+\epsilon) + (1+\epsilon)\log(1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}) \\
&\quad + (1+\epsilon)\frac{2\epsilon(\frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}) + o(\epsilon)}{1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}} \\
&= \log(Q_{11})(1+\epsilon) + (1+\epsilon)\left(\frac{2\epsilon(\frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}) + o(\epsilon)}{1 + \frac{U^{12}}{U^{11}}\frac{V_{12}}{V_{11}} + \frac{U^{13}}{U^{11}}\frac{V_{13}}{V_{11}}}\right) \\
&= \log(Q_{11})(1+\epsilon) + 2\epsilon\frac{U^{12}V_{12} + U^{13}V_{13}}{U^{11}V_{11} + U^{12}V_{12} + U^{13}V_{13}} + o(\epsilon)
\end{aligned}$$

Here flc means calculating with float arithmetic, hence, the error of computing the first term is

$$\delta_Q = |\mathrm{flc}(\log(Q_{11})) - \log(Q_{11})| \le \log(Q_{11})\epsilon_m + \frac{1}{2}\epsilon_m + o(\epsilon_m)$$

Then we consider the error of computing $z = h(u)^T L^{-T}\hat{Q}L^{-1}h(v)$, given by following formula:

$$\begin{aligned}
|\mathbf{flc}(z) - z| &\le |(1+\epsilon)^7 h(u)^T L^{-T}\hat{Q}L^{-1}h(v) - z| \le 7\epsilon_m |h(u)|^T L^{-T}|\hat{Q}|L^{-1}|h(v)| + o(\epsilon_m) \\
&\le \frac{245}{3}\epsilon_m + o(\epsilon_m) = \delta_z
\end{aligned}$$

based on this error, we consider the error for the second term of distance

$$
\begin{aligned}
\delta_2 =& \text{flc}\left(\log(z+\sqrt{z^2-Q_{11}^{-2}})\right) - \log(z+\sqrt{z^2-Q_{11}^{-2}}) \\
=&(1+\epsilon_1)(\log((1+\epsilon_2)(z+\delta_z+(1+\epsilon_3)\sqrt{(1+\epsilon_4)((1+\epsilon_5)(z+\delta_z)^2-(1+\epsilon_6)Q_{11}^{-2})})) \\
&-\log(z+\sqrt{z^2-Q_{11}^{-2}}) \\
=&(1+\epsilon_1)(\log(z+\delta_z+(1+\epsilon_3)\sqrt{(1+\epsilon_4)((1+\epsilon_5)(z+\delta_z)^2-(1+\epsilon_6)Q_{11}^{-2})})) \\
&-\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(\epsilon_m) \\
=&(1+\epsilon_1)(\log(z+\delta_z+(1+\epsilon_3)\sqrt{(1+\epsilon_4)}\cdot\sqrt{(z+\delta_z)^2-Q_{11}^{-2}+\epsilon_7((z+\delta_z)^2+Q_{11}^{-2})})) \\
&-\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(\epsilon_m) \\
=&(1+\epsilon_1)(\log(z+\frac{245}{3}\epsilon_m+(1+\frac{2}{3}\epsilon_8)\sqrt{(z+\frac{245}{3}\epsilon_m)^2-Q_{11}^{-2}+\epsilon_7((z+\delta_z)^2+Q_{11}^{-2})})) \\
&-\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(\epsilon_m)
\end{aligned}
$$

We divide it into two cases, firstly, consider Taylor expansion here when $Q_{11}z \geq$

C_0, where $C_0 \geq 1$ is a large constant, then

$$\begin{aligned}
&\log(z + \frac{245}{3}\epsilon_m + (1 + \frac{2}{3}\epsilon_8)\sqrt{(z + \frac{245}{3}\epsilon_m)^2 - Q_{11}^{-2} + \epsilon_7((z + \frac{245}{3}\epsilon_m)^2 + Q_{11}^{-2})}) \\
&= \log(z + (1 + \frac{2}{3}\epsilon_8)\sqrt{z^2 - Q_{11}^{-2} + \epsilon_7(z^2 + Q_{11}^{-2})}) + \frac{245\epsilon_m}{3\sqrt{z^2 - Q_{11}^{-2} + \epsilon_7(z^2 + Q_{11}^{-2})}} + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2} + \epsilon_7(z^2 + Q_{11}^{-2})}) + \frac{2}{3}\epsilon_8 - \frac{2\epsilon_8 z}{3[z + \sqrt{z^2 - Q_{11}^{-2} + \epsilon_7(z^2 + Q_{11}^{-2})}]} \\
&\quad + \frac{245\epsilon_m}{3\sqrt{z^2 - Q_{11}^{-2} + \epsilon_7(z^2 + Q_{11}^{-2})}} + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + \frac{(z^2 + Q_{11}^{-2})\epsilon_7}{2\sqrt{z^2 - Q_{11}^{-2}}(z + \sqrt{z^2 - Q_{11}^{-2}})} + \frac{2}{3}\epsilon_8 - \frac{2\epsilon_8 z}{3[z + \sqrt{z^2 - Q_{11}^{-2}}]} \\
&\quad + \frac{245\epsilon_m}{3\sqrt{z^2 - Q_{11}^{-2}}} + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + \frac{\epsilon_7}{4} + \frac{7\epsilon_7}{16(Q_{11}z)^2} + \frac{2}{3}\epsilon_8 - \frac{\epsilon_8}{3} - \frac{\epsilon_8}{12(Q_{11}z)^2} + \frac{245\epsilon_m}{3z} \\
&\quad + O(\epsilon_m Q_{11}^{-2} z^{-3}) + o((Q_{11}z)^{-3}) + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + \frac{\epsilon_7}{4} + \frac{7\epsilon_7}{16(Q_{11}z)^2} + \frac{1}{3}\epsilon_8 - \frac{\epsilon_8}{12(Q_{11}z)^2} + \frac{245\epsilon_m}{3z} \\
&\quad + O(\epsilon_m Q_{11}^{-2} z^{-3}) + o((Q_{11}z)^{-3}) + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + \frac{\epsilon_7}{4} + \frac{7\epsilon_7}{16(Q_{11}z)^2} + \frac{1}{3}\epsilon_8 - \frac{\epsilon_8}{12(Q_{11}z)^2} + \frac{245\epsilon_m}{3z} + o(C_0^{-2}) + o(\epsilon_m) \\
&= \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + (\frac{7}{12} + \frac{25}{48(Q_{11}z)^2})\epsilon_9 + \frac{245\epsilon_m}{3z} + o(C_0^{-2}) + o(\epsilon_m)
\end{aligned}$$

where $|\epsilon_i| \leq \epsilon_m$, the machine error, then the error is

$$
\begin{aligned}
&|(1+\epsilon_1)(\log(z+\sqrt{z^2-Q_{11}^{-2}})+(\frac{7}{12}+\frac{25}{48(Q_{11}z)^2})\epsilon_9+\frac{245\epsilon_m}{3z}+o(C_0^{-2})+o(\epsilon_m))\\
&-\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(\epsilon_m)|\\
=&|(1+\epsilon_1)\left((\frac{7}{12}+\frac{25}{48(Q_{11}z)^2})\epsilon_9+\frac{245\epsilon_m}{3z}+o(C_0^{-2})\right)+\epsilon_1\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(\epsilon_m)|\\
=&|(\frac{7}{12}+\frac{25}{48(Q_{11}z)^2})\epsilon_9+\frac{245\epsilon_m}{3z}+\epsilon_1\log(z+\sqrt{z^2-Q_{11}^{-2}})+\epsilon_2+o(C_0^{-2})+o(\epsilon_m)+\frac{7}{12}\epsilon_1|\\
\leq&\frac{277}{432}\epsilon_m+63\epsilon_m C_0 Q_{11}^{-1}+\epsilon_m\log 70/3+\frac{19}{12}\epsilon_m+o(C_0^{-2})+o(\epsilon_m)\\
\leq&(\frac{961}{432}+\frac{245C_0}{3}+\log 70/3)\epsilon_m+o(C_0^{-2})+o(\epsilon_m)=\delta_2
\end{aligned}
$$

On the other hand, if $Q_{11}z \leq C_0$, then notice that

$$
\begin{aligned}
\operatorname{arcosh}(Q_{11}/3) =&d(L^{-T}UL^{-1}O, L^{-T}VL^{-1}O)\\
\leq&d(L^{-T}UL^{-1}O, L^{-T}UL^{-1}h(u))+d(L^{-T}UL^{-1}h(u), L^{-T}VL^{-1}h(v))\\
&+d(L^{-T}VL^{-1}O, L^{-T}VL^{-1}h(v))\\
=&d(O,h(u))+d(L^{-T}UL^{-1}h(u), L^{-T}VL^{-1}h(v))+d(O,h(v))\\
\leq&\operatorname{arcosh}(Q_{11}z)+2\log(\sqrt{3}+\sqrt{2})\\
\leq&\operatorname{arcosh}(C_0)+2\log(\sqrt{3}+\sqrt{2})
\end{aligned}
$$

so we can get $Q_{11} \leq E(C_0)$, where $E(C_0)$ is a constant depending on C_0, then we

further have

$$
\begin{aligned}
&\log(z+\frac{245}{3}\epsilon_m+(1+\frac{2}{3}\epsilon_8)\cdot\sqrt{(z+\frac{245}{3}\epsilon_m)^2-Q_{11}^{-2}+\epsilon_7((z+\frac{245}{3}\epsilon_m)^2+Q_{11}^{-2})}) \\
&=\log(Q_{11}z+\frac{245}{3}\epsilon_m Q_{11}-\log(Q_{11}) \\
&\quad+(1+\frac{2}{3}\epsilon_8)\cdot\sqrt{(Q_{11}z+\frac{245}{3}\epsilon_m Q_{11})^2-1+\epsilon_7((Q_{11}z+\frac{245}{3}\epsilon_m Q_{11})^2+1)}) \\
&=\log(Q_{11}z+(1+\frac{2}{3}\epsilon_8)\sqrt{(Q_{11}z)^2-1+\epsilon_7((Q_{11}z)^2+1)})-\log(Q_{11})+o(\epsilon_m) \\
&\quad+\frac{245}{3}\epsilon_m Q_{11}[1+\frac{Q_{11}z}{\sqrt{(Q_{11}z)^2-1}}]\frac{1}{Q_{11}z+\sqrt{(Q_{11}z)^2-1}} \\
&=\log(Q_{11}z+\sqrt{(Q_{11}z)^2-1+\epsilon_7((Q_{11}z)^2+1)})-\log(Q_{11})+o(\epsilon_m) \\
&\quad+(\frac{245}{3}\epsilon_m Q_{11}+\frac{2}{3}\epsilon_8)[1+\frac{Q_{11}z}{\sqrt{(Q_{11}z)^2-1}}]\frac{1}{Q_{11}z+\sqrt{(Q_{11}z)^2-1}} \\
&=\log(Q_{11}z+\sqrt{(Q_{11}z)^2-1})-\log(Q_{11})+o(\epsilon_m) \\
&\quad+(\frac{245}{3}\epsilon_m Q_{11}+\frac{2}{3}\epsilon_8+\epsilon_7)\cdot[1+\frac{Q_{11}z}{\sqrt{(Q_{11}z)^2-1}}]\frac{1}{Q_{11}z+\sqrt{(Q_{11}z)^2-1}} \\
&=\log(z+\sqrt{z^2-Q_{11}^{-2}})+o(\epsilon_m) \\
&\quad+(\frac{245}{3}\epsilon_m Q_{11}+\frac{2}{3}\epsilon_8+\epsilon_7)\cdot[1+\frac{Q_{11}z}{\sqrt{(Q_{11}z)^2-1}}]\frac{1}{Q_{11}z+\sqrt{(Q_{11}z)^2-1}} \\
&=(\frac{245}{3}C_0E(C_0)+\frac{5}{3})\epsilon_m\cdot[1+\frac{Q_{11}z}{\sqrt{(Q_{11}z)^2-1}}]\frac{1}{Q_{11}z+\sqrt{(Q_{11}z)^2-1}}+o(\epsilon_m) \\
&\quad+\log(z+\sqrt{z^2-Q_{11}^{-2}})
\end{aligned}
$$

Hence, we get the error to be

$$\begin{aligned}
\delta_2 =&|\epsilon_1 \log(z + \sqrt{z^2 - Q_{11}^{-2}}) + \epsilon_2 + o(\epsilon_m) \\
&+ (\frac{245}{3} C_0 E(C_0) + \frac{5}{3})\epsilon_m \cdot [1 + \frac{Q_{11} z}{\sqrt{(Q_{11} z)^2 - 1}}] \frac{1}{Q_{11} z + \sqrt{(Q_{11} z)^2 - 1}}| \\
\leq&(\frac{245}{3} C_0 E(C_0) + \frac{5}{3}) \cdot [1 + \frac{Q_{11} z}{\sqrt{(Q_{11} z)^2 - 1}}] + 1)\epsilon_m + (\log(C_0 + \sqrt{C_0^2 - 1}) \\
&- \log(Q_{11}) + o(\epsilon_m) \\
\leq&[\log(C_0 + \sqrt{C_0^2 - 1}) - \log(Q_{11}) + (\frac{245}{3} C_0 E(C_0) + \frac{8}{3}) \cdot (1 + \frac{45}{8 \tanh(d)})]\epsilon_m \\
&+ o(\epsilon_m) \\
\leq&[\log(2C_0) + (\frac{245}{3} C_0 E(C_0) + \frac{8}{3}) \cdot (1 + \frac{45}{8 \tanh(d)})]\epsilon_m + o(\epsilon_m)
\end{aligned}$$

All in all, we can get the total error of computing distance

$$\delta = \delta_Q + \delta_2 = \log(Q_{11})\epsilon_m + \frac{1}{2}\epsilon_m + o(\epsilon_m) + \delta_2$$

Because $\log(Q_{11}) \leq d$, then we get that if $Q_{11} z \geq C_0$,

$$\delta \leq d\epsilon_m + (\frac{1825}{432} + \frac{245 C_0}{3} + \log 70/3)\epsilon_m + o(C_0^{-2}) + o(\epsilon_m)$$

if $Q_{11} z \leq C_0$,

$$\delta \leq d\epsilon_m + [\frac{1}{2} + \log(2C_0) + (\frac{245}{3} C_0 E(C_0) + \frac{8}{3}) \cdot (1 + \frac{45}{8 \tanh(d)})]\epsilon_m + o(\epsilon_m)$$

□

Also, in the same way, we give the error for computing gradient:

Theorem B.1.4. *The worst case error for computing gradient of distance in Lorentz model using float is*

$$\begin{aligned}
|\nabla_x d_l^{fl}(fl(x), fl(y)) - \nabla_x d_l(x, y)| =& \frac{2 \cosh(d(x, O)) \cosh(d(y, O))}{\sinh(d(x, y)) \tanh(d(x, y))} \epsilon_m \nabla_x d \\
&+ \frac{3}{4 \tanh^2(d(x, y))} \epsilon_m \nabla_x d + \frac{1}{2} \epsilon_m \nabla_x d + o(\epsilon_m)
\end{aligned}$$

the error of computing gradient of distance in L-tiling model using float is

$$|\nabla_u d_{lt}^{fl} - \nabla_u d_{lt}|_1 = \begin{cases} [(B_1(C_0) + B_2(C_0)\exp(d))|\nabla_x d|_1 + B_3(C_0)\exp(d)]\epsilon_m + o(\epsilon_m C_0^{-1}), & d \geq C_0 \\ [\dfrac{B_4(C_0)}{\tanh(d)} + (\dfrac{B_5(C_0)}{\tanh^2(d)} + \dfrac{B_6(C_0)}{\tanh(d)} + B_7(C_0))|\nabla_x d|_1]\epsilon_m + o(\epsilon_m), & d \leq C_0 \end{cases}$$

where d is the real distance between two points, $B_i(C_0)$ are constants only depends on C_0, B_f is a fixed constant, d^{fl} means that inside computation like multiplication are performed with machine error ϵ_m.

Remark: Similar to the worst case error of computing distance, the worst case error of computing gradient in Lorentz model using float is dominated by $d(x, O), d(y, O)$, this will also cause the "NaN" problem when two points are far away from the origin. In L-tiling model, the gradient error only depends on the gradient itself, i.e., also tanh term is bounded, which controls the error and solves the "NaN" problem.

Proof. We consider the Lorentz model at first, the gradient is

$$\nabla_x d = \frac{My}{\sqrt{(x^T My)^2 - 1}},$$

then we have the error to be

$$\begin{aligned} &|\text{flc}(\nabla_x d) - \nabla_x d| \\ &= |(1+\epsilon_1)\frac{(1+\epsilon_2)My}{(1+\epsilon_3)\sqrt{(1+\epsilon_4)[(1+\epsilon_5)(x^T My + \delta_z)^2 - 1]}} - \frac{My}{\sqrt{(x^T My)^2 - 1}}| \\ &= \frac{\epsilon_m My}{2\sqrt{(x^T My)^2 - 1}} + \frac{\delta_z (x^T My) My}{((x^T My)^2 - 1)^{3/2}} + \frac{3\epsilon_m (x^T My)^2 My}{4((x^T My)^2 - 1)^{3/2}} \\ &= \frac{1}{2}\epsilon_m \nabla_x d + \frac{3}{4\tanh^2(d(x,y))}\epsilon_m \nabla_x d + \frac{2\cosh(d(x,O))\cosh(d(y,O))}{\sinh(d(x,y))\tanh(d(x,y))}\epsilon_m \nabla_x d + o(\epsilon_m) \end{aligned}$$

Here flc means calculating with float arithmetic. As for the gradient error in L-tiling model, note that the gradient is

$$\nabla_u d_{lt}((U,u),(V,v)) = \frac{\nabla h(u)^T L^{-T} \hat{Q} L^{-1} h(v)}{\sqrt{z^2 - Q_{11}^{-2}}}$$

where $\nabla h(u) = \left[\frac{u}{\sqrt{1+\|u\|^2}}, \ I \right]$. First, consider

$$\begin{aligned}
&\text{flc}(\sqrt{(z+\delta_1)^2 - Q_{11}^{-2}}) \\
&=(1+\epsilon_1)\sqrt{(1+\epsilon_2)((1+\epsilon_3)(z+\frac{245}{3}\epsilon_m)^2 - (1+\epsilon_4)Q_{11}^{-2})} \\
&=(1+\epsilon_1)(1+\epsilon_2/2)\sqrt{((1+\epsilon_3)(z+\frac{245}{3}\epsilon_m)^2 - (1+\epsilon_4)Q_{11}^{-2})} + o(\epsilon_m) \\
&=(1+\frac{2}{3}\epsilon_5)\sqrt{(z+\frac{245}{3}\epsilon_m)^2 - Q_{11}^{-2} + \epsilon_6((z+\frac{245}{3}\epsilon_m)^2 + Q_{11}^{-2}))} + o(\epsilon_m) \\
&=(1+\frac{2}{3}\epsilon_5)[\sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})} + \frac{245\epsilon_m z}{3\sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})}}] + o(\epsilon_m) \\
&= \sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})} + \frac{245\epsilon_m z}{3\sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})}} \\
&\quad + \frac{2}{3}\epsilon_5\sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})} + o(\epsilon_m) \\
&= \sqrt{z^2 - Q_{11}^{-2} + \epsilon_6(z^2+Q_{11}^{-2})} + \frac{245\epsilon_m z}{3\sqrt{z^2 - Q_{11}^{-2}}} + \frac{2}{3}\epsilon_5\sqrt{z^2 - Q_{11}^{-2}} + o(\epsilon_m) \\
&= \sqrt{z^2 - Q_{11}^{-2}} + \frac{\epsilon_6(z^2+Q_{11}^{-2})}{2\sqrt{z^2 - Q_{11}^{-2}}} + \frac{245\epsilon_m z}{3\sqrt{z^2 - Q_{11}^{-2}}} + \frac{2}{3}\epsilon_5\sqrt{z^2 - Q_{11}^{-2}} + o(\epsilon_m)
\end{aligned}$$

In the same way, we divide this into two cases, if $Q_{11}z \geq C_0$, then the error of this term is

$$\begin{aligned}
\delta_4 &= |\mathbf{fl}(\sqrt{(z+\delta_z)^2 - Q_{11}^{-2}}) - \sqrt{z^2 - Q_{11}^{-2}}| \\
&= |\frac{\epsilon_6(z^2+Q_{11}^{-2})}{2\sqrt{z^2 - Q_{11}^{-2}}} + \frac{245\epsilon_m z}{3\sqrt{z^2 - Q_{11}^{-2}}} + \frac{2}{3}\epsilon_5\sqrt{z^2 - Q_{11}^{-2}} + o(\epsilon_m)| \\
&= |\frac{\epsilon_6 z}{2}(1 + \frac{3}{2(Q_{11}z)^2}) + \frac{245}{3}\epsilon_m(1 + \frac{1}{2(Q_{11}z)^2}) + \frac{2}{3}\epsilon_5\sqrt{z^2 - Q_{11}^{-2}} \\
&\quad + O(\epsilon_m(Q_{11}z)^{-4}) + o(\epsilon_m)| \leq E_1\epsilon_m + O(\epsilon_m C_0^{-2}) + o(\epsilon_m)
\end{aligned}$$

Also, if $Q_{11}z \leq C_0$, then the error of this term is

$$\begin{aligned}\delta_4 =&|\mathbf{fl}(\sqrt{(z+\delta_1)^2 - Q_{11}^{-2}}) - \sqrt{z^2 - Q_{11}^{-2}}| \\ =&|\frac{\epsilon_6(z^2+Q_{11}^{-2})}{2\sqrt{z^2-Q_{11}^{-2}}} + \frac{245\epsilon_m z}{3\sqrt{z^2-Q_{11}^{-2}}} + \frac{2}{3}\epsilon_5\sqrt{z^2-Q_{11}^{-2}} + o(\epsilon_m)| \\ \leq&\frac{E_2}{\tanh(d)}\epsilon_m + E_3\epsilon_m + o(\epsilon_m)\end{aligned}$$

For the numerator term $z_p = \nabla h(u)^T L^{-T}\hat{Q}L^{-1}h(v)$, where $\nabla h(u) = \left[\frac{u}{\sqrt{1+\|u\|^2}},\ \ I\right]$, also because $|z_p|$ is bounded, then we can easily get that $\|\mathrm{fl}(z_{pi}) - z_{pi}\| \leq E_4\epsilon_m + o(\epsilon_m)$. Hence, we have

$$\begin{aligned}\nabla_u d_{lt}((U,u),(V,v))_i &= (1+\epsilon_1)\frac{z_{pi} + E_4\epsilon_m}{\sqrt{z^2 - Q_{11}^{-2}} + \delta_4} \\ &= (1+\epsilon_1)[\frac{z_{pi}+E_4\epsilon_m}{\sqrt{z^2-Q_{11}^{-2}}} - \frac{z_{pi}\delta_4}{z^2-Q_{11}^{-2}}] \\ &= \frac{z_{pi}+E_4\epsilon_m}{\sqrt{z^2-Q_{11}^{-2}}} + \frac{z_{pi}\delta_4}{z^2-Q_{11}^{-2}} + \frac{z_{pi}\epsilon_1}{\sqrt{z^2-Q_{11}^{-2}}}\end{aligned}$$

Then, the error of gradient is

$$\begin{aligned}\delta_{gi} =&\frac{E_4\epsilon_m}{\sqrt{z^2-Q_{11}^{-2}}} + \frac{z_{pi}\delta_4}{z^2-Q_{11}^{-2}} + \frac{z_{pi}\epsilon_1}{\sqrt{z^2-Q_{11}^{-2}}} \\ =&\frac{E_4\epsilon_m}{\sqrt{z^2-Q_{11}^{-2}}} + \frac{\delta_4\nabla_u d_{lti}}{\sqrt{z^2-Q_{11}^{-2}}} + \epsilon_1\nabla_u d_{lti}\end{aligned}$$

So using this formula, we can get that if $Q_{11}z \geq C_0$, then the error of this term is

$$\begin{aligned}\delta_{gi} \leq&E_4Q_{11}\epsilon_m C_0^{-1} + [\delta_4 Q_{11}C_0^{-1} + \epsilon_m]\nabla_u d_{lti} + o(\epsilon_m C_0^{-1}) \\ \leq&E_4Q_{11}\epsilon_m C_0^{-1} + [E_1Q_{11}C_0^{-1} + 1]\epsilon_m\nabla_u d_{lti} + o(\epsilon_m C_0^{-1}) \\ \leq&E_4\epsilon_m C_0^{-1}\exp(d) + [E_1C_0^{-1}\exp(d) + 1]\epsilon_m\nabla_u d_{lti} + o(\epsilon_m C_0^{-1})\end{aligned}$$

If $Q_{11}z \leq C_0$, then the error of this term is

$$\begin{aligned}\delta_{gi} \leq&\frac{E_4}{\tanh(d)}\epsilon_m + [\frac{\delta_4}{\tanh(d)} + \epsilon_m]\nabla_u d_{lti} \\ \leq&\frac{E_4}{\tanh(d)}\epsilon_m + [\frac{E_2}{\tanh^2(d)} + \frac{E_3}{\tanh(d)} + 1]\epsilon_m\nabla_u d_{lti} + o(\epsilon_m)\end{aligned}$$

All in all, if $Q_{11}z \geq C_0$, then the error of gradient is

$$|\delta_g|_1 \leq 3E_4\epsilon_m C_0^{-1}\exp(d) + [E_1 C_0^{-1}\exp(d) + 1]\epsilon_m|\nabla_u d_{lt}|_1 + o(\epsilon_m C_0^{-1})$$

if $Q_{11}z \leq C_0$, then

$$|\delta_g|_1 \leq \frac{3E_4}{\tanh(d)}\epsilon_m + [\frac{E_2}{\tanh^2(d)} + \frac{E_3}{\tanh(d)} + 1]\epsilon_m|\nabla_u d_{lt}|_1 + o(\epsilon_m)$$

□

Error for Computing in H-tiling Model. Here we provide the error of computing in H-tiling model.

Theorem B.1.5. *The error of computing distance in H-tiling model using float is*

$$\begin{aligned}&|d_h^{fl}((j_1, \boldsymbol{k}_1, fl(z_1)), (j_2, \boldsymbol{k}_2, fl(z_2))) - d_h((j_1, \boldsymbol{k}_1, z_1), (j_2, \boldsymbol{k}_2, z_2))|\\ =&[C_3(n)(j_2 - j_1) + C_4(n)d]\epsilon_m + (3e^{-d} + \frac{1}{e^d \sinh d})[C_1(n)2^{2j_2-2j_1}(1 + e^{d/2})^2 + C_2(n)]\epsilon_m\end{aligned}$$

where d is the real distance between two points, $C_i(n)$ are constants only depends on n, d^{fl} means that inside computation like multiplication are performed with machine error ϵ_m.

Remark: Similar to the worst case error of distance computation in L-tiling model, the worst case error in H-tiling model using float only depends on the distance itself, rather than how far points are from the origin, and hence controls the error and solves the "NaN" problem.

Proof. Firstly, note the distance is

$$d = (2s + j_1 - j_2)\log(2) + \log(2^{-2s-j_1+j_2} + X + \sqrt{X^2 + 2^{1-2s-j_1+j_2}X}),$$

where

$$X = \frac{\|I + 2^{-s}z_1 - 2^{j_2-j_1-s}z_2\|^2}{2z_{1n}z_{2n}}, \;\; I = 2^{-s}(k_1 - 2^{j_2-j_1}k_2)$$

Note that $\|I\| \le 1, \|z_1\|, \|z_2\| \le \sqrt{n+3}$, then we can be bound X in following way:

$$\begin{aligned}
X =& \frac{\|I + 2^{-s}z_1 - 2^{j_2-j_1-s}z_2\|^2}{2z_{1n}z_{2n}} \\
\le& \frac{\|I\|^2 + 2^{-2s}\|z_1\|^2 + 2^{2j_2-2j_1-2s}\|z_2\|^2}{2z_{1n}z_{2n}} \\
&+ \frac{2^{1-2s}\|z_1\|\|I\| + 2^{1+2j_2-2j_1-2s}\|z_2\|\|I\| + 2^{1-2s+j_2-j_1}\|z_1\|\|z_2\|}{2z_{1n}z_{2n}} \\
\le& \frac{1 + 2^{-2s}(n+3) + 2^{2j_2-2j_1-2s}(n+3)}{2z_{1n}z_{2n}} \\
&+ \frac{2^{1-2s}\sqrt{n+3} + 2^{1+2j_2-2j_1-2s}\sqrt{n+3} + 2^{1-2s+j_2-j_1}(n+3)}{2} \\
\le& \frac{1}{2} + 2^{-2s-1}(n+3)(1 + 2^{2j_2-2j_1} + 2^{1+j_2-j_1}) + 2^{-2s}(1 + 2^{2j_2-2j_1})\sqrt{n+3} \\
\le& \frac{1}{2} + 2^{-2s+1}((n+3) + \sqrt{n+3}) \le 1 + 2^{-2s+2}(n+2)
\end{aligned}$$

now consider the error for computing I

$$|\hat{I}_i - I_i| = |(1+\epsilon_1)2^{-s}(k_{1i} - 2^{j_2-j_1}k_{2i}) - I_i| \le \epsilon_m|I_i| = \delta_{i,1}$$

here denote $X_u = I + 2^{-s}z_1 - 2^{j_2-j_1-s}z_2$, then

$$\begin{aligned}
|\hat{X}_{ui} - X_{ui}| =& |[(I_i + \delta_{i,1} + (1+\epsilon_1)2^{-s}z_{1i})(1+\epsilon) - (1+\epsilon_2)2^{j_2-j_1-s}z_{2i}](1+\epsilon) - X_{ui}| \\
=& |[I_i + 2^{-s}z_{1i} + \delta_{i,1} + \epsilon_1 2^{-s}z_{1i} + \epsilon(I_i + 2^{-s}z_{1i}) - (1+\epsilon_2)2^{j_2-j_1-s}z_{2i}](1+\epsilon) - X_{ui}| \\
\le& |[I_i + 2^{-s}z_{1i} - 2^{j_2-j_1-s}z_{2i} + \delta_{i,1} + \epsilon_m(I_i + 2^{1-s}z_{1i} + 2^{j_2-j_1-s}z_{2i})](1+\epsilon) - X_{ui}| \\
\le& |\epsilon_m X_{ui} + \delta_{i,1} + \epsilon_m(I_i + 2^{1-s}z_{1i} + 2^{j_2-j_1-s}z_{2i})| \\
\le& \epsilon_m(|X_{ui}| + 2|I_i| + 2^{1-s}|z_{1i}| + 2^{j_2-j_1-s}|z_{2i}|) = \delta_{i,2}
\end{aligned}$$

next, make use of float arithmetic to get the norm of X_u, the error becomes

$$
\begin{aligned}
&|\|\hat{X}_u\|^2 - \|X_u\|^2| \\
\leq &|(1+n\epsilon_m)\|X_u+\delta_2\|^2 - \|X_u\|^2| = |n\epsilon_m\|X_u\|^2 + \sum_{i=1}^{n}(\delta_{i,2}^2 + 2X_{ui}\delta_{i,2})| \\
\leq &|n\epsilon_m\|X_u\|^2 + \sum_{i=1}^{n} 2X_{ui}\delta_{i,2} + o(\epsilon_m)| \\
= &\sum_{i=1}^{n}(4|I_i| + 2^{2-s}|z_{1i}| + 2^{1+j_2-j_1-s}|z_{2i})|)\epsilon_m|X_{ui}| + (n+2)\epsilon_m\|X_u\|^2 + o(\epsilon_m) \\
\leq &(4\|z_1\| + 2^{1+j_2-j_1}\|z_2\|)\epsilon_m 2^{-s}\|X_u\| + 4\epsilon_m\|I\|\|X_u\| + (n+2)\epsilon_m\|X_u\|^2 + o(\epsilon_m) = \delta_3
\end{aligned}
$$

Note that $X = \frac{X_u^2}{2z_{1n}z_{2n}}$, then we bound the error of computing X here:

$$
\begin{aligned}
|\hat{X} - X| = &|\frac{\|\hat{X}_u\|^2}{2z_{1n}z_{2n}(1+2\epsilon)} - X| = |\frac{\|X_u\|^2+\delta_3}{2z_{1n}z_{2n}}(1-2\epsilon) - X| \leq |\frac{\delta_3}{2z_{1n}z_{2n}} - 2\epsilon X| \\
\leq &2|\epsilon_m X| + |\frac{\delta_3}{2z_{1n}z_{2n}}| \leq 2|\epsilon_m X| + |\frac{\delta_3}{2}| \\
\leq &2|\epsilon_m X| + (2\|z_1\| + 2^{j_2-j_1}\|z_2\|)\epsilon_m 2^{-s}\|X_u\| + 2\epsilon_m\|I\|\|X_u\| \\
&+(n/2+1)\epsilon_m\|X_u\|^2 + o(\epsilon_m) \\
\leq &2\epsilon_m X + (2+2^{j_2-j_1})\sqrt{n+3}\epsilon_m 2^{1-s}\sqrt{2X} + 4\epsilon_m\sqrt{2X} + 4(n+2)\epsilon_m X + o(\epsilon_m) \\
\leq &2\epsilon_m X + 2^{2+j_2-j_1-s}\epsilon_m\sqrt{2(n+3)X} + 4\epsilon_m\sqrt{2X} + 4(n+2)\epsilon_m X + o(\epsilon_m) \\
\leq &2\epsilon_m(1+2^{-2s+2}(n+2)) + 2^{3+j_2-j_1-s}\epsilon_m\sqrt{(n+3)(1+2^{-2s+2}(n+2))} \\
&+ 6\epsilon_m\sqrt{1+2^{-2s+2}(n+2)} + 4(n+2)(1+2^{-2s+2}(n+2))\epsilon_m = \delta_4
\end{aligned}
$$

After having these errors, consider the computation error for the second log in

distance:

$$
\begin{aligned}
&\log(2^{-2s-j_1+j_2}+(1+\epsilon)(\hat{X}+\sqrt{1+\epsilon}\sqrt{(1+\epsilon)\hat{X}^2+2^{1-2s-j_1+j_2}\hat{X}}))+\log(1+\epsilon)\\
=&\log(2^{-2s-j_1+j_2}+(1+\epsilon)(\hat{X}+\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\frac{1}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}\\
&+\frac{\delta_4(2^{-1-2s-j_1+j_2}+X/2)}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}}))+\epsilon\\
=&\log(2^{-2s-j_1+j_2}+X+\delta_4+\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\frac{1}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}\\
&+\frac{\delta_4(2^{-1-2s-j_1+j_2}+X/2)}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}}))+\epsilon(X+\sqrt{X^2+2^{1-2s-j_1+j_2}}X))+\epsilon\\
=&\log(2^{-2s-j_1+j_2}+X+\sqrt{X^2+2^{1-2s-j_1+j_2}X})+\epsilon\\
&+\frac{\delta_4+\frac{1}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\frac{\delta_4(2^{-1-2s-j_1+j_2}+X/2)}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}}))+\epsilon(X+\sqrt{X^2+2^{1-2s-j_1+j_2}X})}{2^{-2s-j_1+j_2}+X+\sqrt{X^2+2^{1-2s-j_1+j_2}X}}
\end{aligned}
$$

All in all, we combine all errors to derive the error for distance computation:

$$
\begin{aligned}
\delta_0=&(2s+j_1-j_2)\log(2)\epsilon+\log(1+\epsilon)\\
&+\frac{\delta_5+\frac{1}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\frac{\delta_5(2^{-1-2s-j_1+j_2}+X/2)}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}}))+\epsilon(X+\sqrt{X^2+2^{1-2s-j_1+j_2}X})}{2^{-2s-j_1+j_2}+X+\sqrt{X^2+2^{1-2s-j_1+j_2}X}}\\
=&(2s+j_1-j_2)\log(2)\epsilon+\epsilon\\
&+\frac{\delta_5+\frac{1}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\frac{\delta_5(2^{-1-2s-j_1+j_2}+X/2)}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}}))+\epsilon(X+\sqrt{X^2+2^{1-2s-j_1+j_2}X})}{2^{-2s-j_1+j_2}+X+\sqrt{X^2+2^{1-2s-j_1+j_2}X}}\\
=&(2s+j_1-j_2)\log(2)\epsilon+\epsilon+\frac{\delta_5(\frac{3}{2}+\frac{2^{-1-2s-j_1+j_2}}{\sqrt{X^2+2^{1-2s-j_1+j_2}X}})+\frac{3}{2}\epsilon\sqrt{X^2+2^{1-2s-j_1+j_2}X}+\epsilon X}{2^{-2s-j_1+j_2}+X+\sqrt{X^2+2^{1-2s-j_1+j_2}X}}\\
=&(2s+j_1-j_2)\log(2)\epsilon+\epsilon+\frac{3}{2}\epsilon+2^{2s+j_1-j_2}\frac{(\frac{3}{2}+\frac{1}{2\sinh d})}{\cosh d+\sinh d}\delta_5\\
=&(2s+j_1-j_2)\log(2)\epsilon+\frac{5}{2}\epsilon+2^{2s+j_1-j_2-1}\frac{(3+\frac{1}{\sinh d})}{\cosh d+\sinh d}[2(1+2^{-2s+2}(n+2))\\
&+2^{3+j_2-j_1-s}\sqrt{(n+3)(1+2^{-2s+2}(n+2))}+6\sqrt{1+2^{-2s+2}(n+2)}\\
&+4(n+2)(1+2^{-2s+2}(n+2))]\epsilon_m
\end{aligned}
$$

We simplify the constants with $C_i(n)$, also note that $k_1-2^{j_2-j_1}k_2$ is an integer vector, then $s=\lceil\log_2(\|k_1-2^{j_2-j_1}k_2\|^2)/2\rceil\geq 0$, also we have $j_2\geq j_1$, then the error

for distance computation is

$$\begin{aligned}
\delta_0 \leq & (3+2s+j_1-j_2)\epsilon+2^{2s+j_1-j_2-1}\frac{(3+\frac{1}{\sinh d})}{\cosh d+\sinh d}[C_1(n)+2^{-2s}C_2(n)+2^{j_2-j_1-2s}C_3(n)]\epsilon_m \\
\leq & (3+2s+j_1-j_2)\epsilon+\frac{(3+\frac{1}{\sinh d})}{\cosh d+\sinh d}[2^{2s+j_1-j_2}C_1(n)+2^{j_1-j_2}C_2(n)+C_3(n)]\epsilon_m \\
\leq & (3+2s)\epsilon+\frac{(3+\frac{1}{\sinh d})}{\cosh d+\sinh d}[2^{2s}C_1(n)+C_2(n)]\epsilon_m \\
= & \lceil \log_2(\|k_1-2^{j_2-j_1}k_2\|^2)\rceil\epsilon_m+\frac{(3+\frac{1}{\sinh d})}{\cosh d+\sinh d}[C_1(n)\|k_1-2^{j_2-j_1}k_2\|^2+C_2(n)]\epsilon_m
\end{aligned}$$

To bound the norm in the distance error, consider

$$\begin{aligned}
\|2^{j_1}k_1-2^{j_2}k_2\| = & \|2^{j_1}z_1-2^{j_2}z_2+2^{j_1}k_1-2^{j_2}k_2+2^{j_2}z_2-2^{j_1}z_1\| \\
\leq & \|2^{j_1}z_1-2^{j_2}z_2+2^{j_1}k_1-2^{j_2}k_2\|+\|2^{j_2}z_2-2^{j_1}z_1\| \\
\leq & 2^{\frac{j_1+j_2+1}{2}}\sqrt{(\cosh d-1)z_{1n}z_{2n}}|+2^{j_2}\|z_2\|+2^{j_1}\|z_1\| \\
\leq & 2^{\frac{j_1+j_2}{2}+2}\sinh(d/2)+(2^{j_2}+2^{j_1})\sqrt{n+3}
\end{aligned}$$

then scale it by 2^{-j_1}, we have

$$\|k_1-2^{j_2-j_1}k_2\| \leq 2^{\frac{j_2-j_1}{2}+2}\sinh(d/2)+(2^{j_2-j_1}+1)\sqrt{n+3}$$

Therefore, we bound the distance computation error as follows:

$$\delta_0 \leq [C_3(n)(j_2-j_1)+C_4(n)d]\epsilon_m+(3e^{-d}+\frac{1}{e^d\sinh d})[C_1(n)2^{2j_2-2j_1}(1+e^{d/2})^2+C_2(n)]\epsilon_m$$

□

B.2 Representing Hyperbolic Space Accurately using Multi-Component Floats (MCF)

B.2.1 Proofs for Theorems in Section 3.3

Here we first provide the proof of the worst case representation error in the Poincaré upper-half space model.

Theorem B.2.1. *The representation error of storing a particular point* $\boldsymbol{x} \in \mathcal{U}^n$ *using floating-points* fl *is* $\delta_{\mathrm{fl}}(\boldsymbol{x}) = d_u(\boldsymbol{x}, \mathrm{fl}(\boldsymbol{x}))$*, and the* worst case representation error *defined as a function of the distance-to-origin d in the Poincaré upper-half space model is*

$$\delta_d := \max_{\boldsymbol{x} \in \mathcal{U}^n, d_u(\boldsymbol{x}, \boldsymbol{O}) \leq d} \delta_{\mathrm{fl}}(\boldsymbol{x}) = \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cosh^2(d)).$$

where $\epsilon_{machine}$ *is the machine epsilon of the underlying floating-point arithmetic. This becomes* $\delta_d = 2\epsilon_{machine} + o(\epsilon_{machine})$ *if* $d < \log(1/\epsilon_{machine})$ *and* $\delta_d = 2d + 2\log(\epsilon_{machine}) + o(\epsilon_{machine}^{-1} \exp(-2d))$ *if* $d \geq \log(1/\epsilon_{machine})$.

Proof. Consider the error $\delta_{fl}(\boldsymbol{x})$ first as

$$\begin{aligned}
\delta_{fl}(\boldsymbol{x}) &= \operatorname{arcosh}(1 + \frac{\sum_{i=1}^n \epsilon_i^2 x_i^2}{2x_n \mathrm{fl}(x_n)}) \\
&= \operatorname{arcosh}(1 + \frac{\sum_{i=1}^n \epsilon_i^2 x_i^2}{2(1+\epsilon_n) x_n^2}) \\
&\leq \operatorname{arcosh}(1 + \frac{\epsilon_{machine}^2 \|x\|^2}{2(1+\epsilon_n) x_n^2}) \\
&= \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cdot \frac{\|x\|^2}{2x_n^2} + o(\epsilon_{machine}^3)).
\end{aligned}$$

On the other hand, note that

$$\begin{aligned}\cosh d(\boldsymbol{x}, \boldsymbol{O}) &= 1 + \frac{\sum_{i=1}^{n-1} x_i^2 + (x_n - 1)^2}{2x_n} \\ &= 1 + \frac{\|x\|^2 + 1 - 2x_n}{2x_n} \\ &= \frac{\|x\|^2 + 1}{2x_n}.\end{aligned}$$

Hence, $2x_n \cosh d = \|x\|^2 + 1$, where $d = d(\boldsymbol{x}, \boldsymbol{O})$, then we have

$$\begin{aligned}\delta_{fl}(\boldsymbol{x}) &= \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cdot \frac{\|x\|^2}{2x_n^2} + o(\epsilon_{machine}^3)). \\ &\leq \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cdot \frac{2x_n \cosh d - 1}{2x_n^2} + o(\epsilon_{machine}^3)) \\ &\leq \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cdot (\frac{\cosh d}{x_n} - \frac{1}{2x_n^2}) + o(\epsilon_{machine}^3)) \\ &= \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cdot (t \cosh d - \frac{1}{2}t^2) + o(\epsilon_{machine}^3))\end{aligned}$$

where $t = \frac{1}{x_n}$, note that $t \cosh d - \frac{1}{2}t^2$ attains maximum $\frac{1}{2}\cosh^2 d$ at $t = \cosh d$, therefore, $\delta_{fl}(\boldsymbol{x}) \leq \operatorname{arcosh}(1 + \frac{\epsilon_{machine}^2 \cosh^2 d}{2} + o(\epsilon_{machine}^3))$, then we can derive

$$\delta_d := \max_{\boldsymbol{x} \in \mathcal{U}^n, d_u(\boldsymbol{x}, \boldsymbol{O}) \leq d} \delta_{fl}(\boldsymbol{x}) = \operatorname{arcosh}(1 + \epsilon_{machine}^2 \cosh^2 d).$$

With taylor expansion when $d < \log(1/\epsilon_{machine})$ and $d \geq \log(1/\epsilon_{machine})$, the conclusion follows. □

Here we provide the proof of the worst case representation error in the ***m*-xMC-Halfspace** model, where MCF is only adopted for the first $n - 1$ axes.

Theorem B.2.2. *The worst case representation error of storing an exact point $\boldsymbol{x} \in \mathcal{U}^n$ with m-multi-component floating-point expansion $(\boldsymbol{x}^{(m)}, \cdots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ (increasing order) defined as a function of the distance-to-origin d is*

$$\delta_d = \operatorname{arcosh}\left(1 + \epsilon_{machine}^2 + \frac{\epsilon_{machine}^{2m}(1 + \epsilon_{machine})}{2^{2m}} \cosh^2(d)\right),$$

where $\epsilon_{machine}$ *is the machine epsilon. This becomes* $\delta_d = 2\epsilon_{machine} + o(\epsilon_{machine})$ *if* $d < m\log(1/\epsilon_{machine})$ *and* $\delta_d = 2d + 2m\log(\epsilon_{machine}/2) + o(\epsilon_{machine}^{-2m}\exp(-2d))$ *if* $d \geq m\log(1/\epsilon_{machine})$.

Proof. Firstly, note due to the construction of multi-component floats, the error caused by the floating point only exists at the smallest component, since MCF accounts for the errors in previous components afterwards by adding more components. Due to the non-overlapping property of the MCF, we have $x^{(m+1)} \leq \frac{1}{2}\text{ulp}(x^{(m)}) = \frac{1}{2}\epsilon_{machine}x^{(m)}$, hence, we can sequentially derive that $x^{(m)} \leq 2^{-(m-1)}\epsilon_{machine}^{m-1}x^{(1)}$, then similarly, consider the error $\delta_{fl}(\boldsymbol{x})$ as

$$\begin{aligned}
\delta_{fl}(\boldsymbol{x}) &= \text{arcosh}(1 + \frac{\sum_{i=1}^{n-1}\epsilon_i^2(x_i^{(m)})^2 + \epsilon_n^2 x_n^2}{2x_n\text{fl}(x_n)}) \\
&\leq \text{arcosh}(1 + \frac{\sum_{i=1}^{n-1}\epsilon_i^2(2^{-(m-1)}\epsilon_{machine}^{m-1}x_i^{(1)})^2 + \epsilon_n^2 x_n^2}{2(1+\epsilon_n)x_n^2}) \\
&\leq \text{arcosh}(1 + \frac{2^{-2(m-1)}\epsilon_{machine}^{2m}\|x\|^2 + (1-2^{-2(m-1)})\epsilon_{machine}^2 x_n^2}{2(1+\epsilon_n)x_n^2}) \\
&\leq \text{arcosh}(1 + 2^{-2(m-1)}\epsilon_{machine}^{2m}(1+\epsilon_{machine})\cdot\frac{\|x\|^2}{2x_n^2} + \epsilon_{machine}^2(1+\epsilon_{machine})/2).
\end{aligned}$$

also we have $2x_n\cosh d = \|x\|^2 + 1$, where $d = d(\boldsymbol{x}, \boldsymbol{O})$, then we have

$$\begin{aligned}
\frac{\|x\|^2}{2x_n^2} &= \frac{2x_n\cosh d - 1}{2x_n^2} \\
&= \frac{\cosh d}{x_n} - \frac{1}{2x_n^2} \\
&= t\cosh d - \frac{1}{2}t^2
\end{aligned}$$

where $t = \frac{1}{x_n}$, note that this attains maximum $\frac{1}{2}\cosh^2 d$ at $t = \cosh d$, therefore, $\delta_{fl}(\boldsymbol{x}) \leq \text{arcosh}(1 + 2^{-2(m-1)}\epsilon_{machine}^{2m}(1+\epsilon_{machine})\cdot\cosh^2 d + \epsilon_{machine}^2(1+\epsilon_{machine})/2)$, then we can derive

$$\delta_d := \max_{\boldsymbol{x}\in\mathcal{U}^n, d_u(\boldsymbol{x},\boldsymbol{O})\leq d}\delta_{fl}(\boldsymbol{x}) = \text{arcosh}(1 + \epsilon_{machine}^2 + \frac{\epsilon_{machine}^{2m}(1+\epsilon_{machine})\cdot\cosh^2 d}{2^{2m}}).$$

With taylor expansion when $d < m\log(1/\epsilon_{machine})$ and $d \geq m\log(1/\epsilon_{machine})$, the conclusion follows. □

B.2.2 Gradient Computations & Numerical Stable Form of Exp

Here we show how to compute gradients in the halfspace model: assume two points $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$, we will compute the gradient $\boldsymbol{g_u} \in \mathbb{R}^d$ of the hyperbolic distance w.r.t. $\boldsymbol{u}$ as follows:

$$x = \frac{\|\boldsymbol{u} - \boldsymbol{v}\|^2}{2u_d v_d}, \quad z = \sqrt{x(x+2)}, \quad \boldsymbol{y} = \frac{\boldsymbol{u} - \boldsymbol{v}}{u_d v_d},$$
$$(\boldsymbol{g_u})_{1:d-1} = \frac{1}{z}\boldsymbol{y}_{1:d-1},$$
$$(\boldsymbol{g_u})_d = \frac{1}{z}(\boldsymbol{y}_d - \frac{x}{u_d}).$$

We can either choose to compute this gradients in ordinary floating-point arithmetic, or compute them with the adapted MCF arithmetic using the provided MCF algorithms in the *m*-**xMC-Halfspace** model. In our implementaions, we compute this gradients with the adapted MCF arithmetic.

Here we offer a numerical stable form of the aforementioned exponential map **Exp**. Firstly, for the first equation regarding the x-axes, i.e.,

$$z_i' = z_i + \frac{z_n}{\frac{s}{\tanh s} - v_n} \cdot v_i.$$

Note that if $v_n \geq 0$, then the subtraction in the denominator of $\frac{s}{\tanh s}$ (close to 1) to v_n (close to 0) is the major part to the numerical error, therefore, we'd like to avoid this subtraction with a different computation but in the same arithmetic as follows:

$$\begin{aligned}
z_i' =& z_i + \frac{z_n}{\frac{s}{\tanh s} - v_n} \cdot v_i \\
=& z_i + \frac{z_n}{s \coth s - v_n} \cdot v_i \\
=& z_i + \frac{s \coth s + v_n}{s^2 \coth^2 s - v_n^2} \cdot z_n \cdot v_i \\
=& z_i + \frac{s \coth s + v_n}{s^2 \operatorname{csc}^2 s + s^2 - v_n^2} \cdot z_n \cdot v_i \\
=& z_i + \frac{s \coth s + v_n}{s^2 \operatorname{csc}^2 s + r^2} \cdot z_n \cdot v_i,
\end{aligned}$$

where $s = \sqrt{\boldsymbol{v}^T\boldsymbol{v}}$, and $r^2 = \sum_{i=1}^{n-1} v_i^2$. In this way, we can avoid the numerical error caused by the subtraction when $v_n \geq 0$, note that if $v_n < 0$, then the 'subtraction' is actually an addition, hence we will keep the original formula.

For the second equation regarding the y-axis, i.e.,

$$z_n' = \frac{z_n}{\cosh s - \frac{\sinh s}{s} v_n}.$$

Again note that if $v_n \geq 0$, the subtraction in the denominator of $\cosh s$ (close to 1) to $\frac{\sinh s}{s} v_n$ (close to 0) is the major part to the numerical error, here we also provide a different computation but with the same arithmetic to avoid the subtraction when $v_n \geq 0$ as follows:

$$\begin{aligned}
z_n' =& \frac{z_n}{\cosh s - \frac{\sinh s}{s} v_n} \\
=& \frac{s \cdot z_n}{s \cosh s - v_n \sinh s} \\
=& \frac{s \cosh s + v_n \sinh s}{s^2 \cosh^2 s - v_n^2 \sinh^2 s} \cdot s \cdot z_n \\
=& \frac{s \cosh s + v_n \sinh s}{r^2 \cosh^2 s + v_n^2 \cosh^2 s - v_n^2 \sinh^2 s} \cdot s \cdot z_n \\
=& \frac{s \cosh s + v_n \sinh s}{r^2 \cosh^2 s + v_n^2} \cdot s \cdot z_n.
\end{aligned}$$

Similarly, we avoid the numerical error caused by the subtraction when $v_n \geq 0$, and if $v_n < 0$, then the 'subtraction' is actually an addition, hence we will keep the original formula.

B.2.3 More Algorithms Operating MCF

We intend to provide more algorithms to compute MCF expansions in this section, to begin with, the sum of two expansions in Alg. 11 **Add-Expansion**, differ-

Algorithm 11 Add-Expansion, modified from [152]

Input: m-components expansions $(a_1, \cdots, a_m)$ and $(b_1, \cdots, b_m)$, both in decreasing order.
initialize $e \leftarrow 0$
for $i = 1$ to m **do**
  $(h_p, e_1) \leftarrow$ **Two-Sum**(a_i, b_i)
  $(h_i, e_2) \leftarrow$ **Two-Sum**(h_p, e)
  $e \leftarrow \mathbf{fl}(e_1 + e_2)$
end for
$h_{m+1} \leftarrow e$
Return: $(h_1, \cdots, h_m, h_{m+1})$

ent from its version firstly proposed in [152], we modify the algorithm to output an expansion with $m + 1$.

Next, we aim to offer algorithms for multiplication of an expansion to a single p-bit floating-point number here. To begin with, we need an algorithm for multiplications between two p-bit floating-point numbers to form a non-overlapping expansion, termed as **Two-Prod**. We firstly show the following Lemma B.2.3 for the purpose, particularly designed for 53-bit IEEE double precision floating point numbers.

Algorithm 12 Split

Input: 53-bit double precision floats a
  $t \leftarrow \mathbf{fl}((2^{27} + 1) \cdot a)$
  $a_{hi} \leftarrow \mathbf{fl}(t - \mathbf{fl}(t - a))$
  $a_{lo} \leftarrow \mathbf{fl}(a - a_{hi})$
Return: (a_{hi}, a_{lo})

Lemma B.2.3. *[77] Alg. 12* ***Split*** *splits a 53-bit IEEE double precision floating point number into a_{hi} and a_{lo}, each with 26 bits of significand, such that $a = a_{hi} + a_{lo}$. a_{hi} contains the first 26 bits, while a_{lo} contains the lower 26 bits. Note that this algorithm can be easily generated to any p-bit floating-point number [152].*

With this, we show how to multiply two p-bit floating-point numbers to get

an non-overlapping expansion in Alg. 13 **Two-Prod**.

Algorithm 13 Two-Prod

Input: double precision floats a, b
$p \leftarrow \mathbf{fl}(a \cdot b)$
$(a_{hi}, a_{lo}) \leftarrow \mathbf{Split}(a)$
$(b_{hi}, b_{lo}) \leftarrow \mathbf{Split}(b)$
$err_1 \leftarrow \mathbf{fl}(x - \mathbf{fl}(a_{hi} \cdot b_{hi}))$
$err_2 \leftarrow \mathbf{fl}(err_1 - \mathbf{fl}(a_{lo} \cdot b_{hi}))$
$err_3 \leftarrow \mathbf{fl}(err_2 - \mathbf{fl}(a_{hi} \cdot b_{lo}))$
$y \leftarrow \mathbf{fl}(\mathbf{fl}(a_{lo} \cdot b_{lo}) - err_3)$
Return: (x, y)

Theorem B.2.4. *[77, 152] Alg. 13 computes* $p = \mathbf{fl}(a \cdot b)$ *and corresponding roundoff error* $e = \mathbf{err}(a \cdot b)$.

Herein, we provide the multiplication algorithm of an expansion to a single p-bit floating-point number in Alg. 14 **Scale-Expansion**. Note that both Alg. 14 **Scale-Expansion** and Alg. 11 **Add-Expansion** grows the expansion only one more to be $m + 1$ components, hence, we will need to apply the **Renormalize** algorithm to reduce the number of components.

Algorithm 14 Scale-Expansion, modified from [152]

Input: m-components expansion $(a_1, \cdots, a_m)$ in decreasing order, p-bit float b.

initialize $e \leftarrow 0$
for $i = 1$ to m **do**
  $(h_p, e_1) \leftarrow$ **Two-Prod**(a_i, b)
  $(h_i, e_2) \leftarrow$ **Two-Sum**(h_p, e)
  $e \leftarrow \mathbf{fl}(e_1 + e_2)$
end for
$h_{m+1} \leftarrow e$
Return: $(h_1, \cdots, h_m, h_{m+1})$

We also extends to develop Alg. 15 for the multiplication of expansions, where the for loop in actually a tree-like operation.

Algorithm 15 Mul-Expansion, modified from [152]

Input: m_1-components expansion $a = (a_1, \cdots, a_{m_1})$ and m_2-components expansion $b = (b_1, \cdots, b_{m_2})$ in decreasing order, assume $m_1 \leq m_2$ w.l.g.
Denote $a_i b$ as **Scale-Expansion**(b, a_i) for simplicity.
Compute $h_i^{(0)} = a_i b$ for $i = 1$ to m_1 in parallel
for $i = 1$ to $\lfloor \log_2 m_1 \rfloor$ **do**
  Compute $h_j^{(i)} \leftarrow$ **Add-Expansion**$(h_{2j-1}^{(i-1)}, h_{2j}^{(i-1)})$ for $j = 1$ to $\frac{m_1}{2^i}$ in parallel
end for
Return: $h_1^{(\lfloor \log_2 m_1 \rfloor)}$

B.2.4 RSGD & MCs-Halfspace Model

We provide the RSGD algorithm adapted in the ***m*-xMCs-Halfspace** model using the provided MCF algorithms in Alg. 16:

Algorithm 16 RSGD in the ***m*-xMCs-Halfspace** model

Require: Objective function f
Require: $z \in \mathcal{U}^n$, Epochs T, and learning rate η
  for $t = 0$ to $T - 1$ **do**
    $\text{grad}_z f \Leftarrow z_n \nabla_z f$, ▷ Riemannian gradient
    $\boldsymbol{v} = -\eta \cdot \text{grad}_z f$, ▷ learning rate
    $s \Leftarrow \sqrt{\boldsymbol{v}^T \boldsymbol{v}}$
    $\text{grad}_{z_{1:n-1}} \Leftarrow \frac{z_n}{\frac{s}{\tanh s} - v_n} \cdot v_i$, ▷ gradient of x-axis values
    $w \Leftarrow$ **Grow-Expansion**$(z_{1:n-1}, \text{grad}_{z_{1:n-1}})$
    $z'_{1:n-1} \Leftarrow$ **Renormalize**(w), ▷ Update x-axis values
    $z'_n = \frac{z_n}{\cosh s - \frac{\sinh s}{s} v_n}$, ▷ Update y-axis values
  end for
  Output z'

As mentioned in the main body of the paper, we can appy MCF on all coordinates to get the **m-MC-Halfspace** model. The distance computations within this model are consistent to the **m-xMC-Halfspace** model, while the key difference is the usage of the Alg. 14 **Scale-Expansion** in the **m-MC-Halfspace** model, since the last coordinate of the model is involved mostly in multiplications. More importantly, we show in Alg. 17 how to do RSGD in the ***m*-MCs-Halfspace** model.

Algorithm 17 RSGD in the *m*-**MCs-Halfspace** model

Require: Objective function f
Require: $z \in \mathcal{U}^n$, Epochs T, and learning rate η
  for $t = 0$ to $T - 1$ **do**
    $\text{grad}_z\ f \Leftarrow z_n \nabla_z f$, ▷ Riemannian gradient
    $\boldsymbol{v} = -\eta \cdot \text{grad}_z f$, ▷ learning rate
    $s \Leftarrow \sqrt{\boldsymbol{v}^T \boldsymbol{v}}$
    $\text{grad}_{z_{1:n-1}} \Leftarrow \frac{z_n}{\frac{s}{\tanh s} - v_n} \cdot v_i$, ▷ gradient of x-axis values
    $w_x \Leftarrow$ **Grow-Expansion**$(z_{1:n-1}, \text{grad}_{z_{1:n-1}})$
    $z'_{1:n-1} \Leftarrow$ **Renormalize**(w_x), ▷ Update x-axis values
    $w_y \Leftarrow$ **Scale-Expansion**$(z_n, \frac{1}{\cosh s - \frac{\sinh s}{s} v_n})$
    $z'_n \Leftarrow$ **Renormalize**(w_y), ▷ Update y-axis values
  end for
  Output z'

In this way, for the addition & subtraction occurring in the exponential map, between a potentially large floating-point coordinate number to a small floating-point gradient is done in MCF arithmetic. Notice that we only adopt MCF arithmetic in part of the distance, gradient and exponential map computations, and leave the rest of computations computed in ordinary floating-point arithmetic.

B.2.5 Experiment Details

We conducted our experiments based on the implementation of [199], with all learning experiments in PyTorch based on ordinary float64. For the initialization of all models, we drawn randomly from the uniform distribution $U(-1e-5, 1e-5)$. Particularly, for the initialized embedding of the upper-half space model and MCF-based models, the last axis is initialized with $1 + U(-1e-5, 1e-5)$, since the corresponding origin in halfspace model is $(0, \cdots, 0, 1)$.

We train embeddings of different dataset for 1000 epochs except for the largest WordNet-Nouns dataset with 500 epochs. At the start of the training, we train models with an initial "burn-in" phase firstly proposed in [122], which helps find a good initial angular layout and a good resulting embedding, simply by using a reduced learning rate $\eta/100$.

For most of the hyper-parameters in our experiment, we adopt the recommended values from the implementations of [199, 122, 123], the negative sampling size is 50 in the experiments. We vary batchsize within $\{32, 64, 128\}$ and use grid search to find the optimal learning rate in each case.

BATCHSIZE	LR	MAP (%)	MR
32	0.3	72.96	2.53
	1.0	87.65	1.99
	2.0	93.65	1.43
	3.0	90.95	1.81
64	0.3	29.34	18.45
	1.0	84.97	1.78
	2.0	89.90	1.76
	3.0	92.29	1.57
	4.0	92.36	1.66
128	1.0	61.79	3.98
	2.0	81.12	2.28
	3.0	87.45	1.89
	4.0	91.27	1.55
	5.0	92.07	1.50
	6.0	92.32	1.56
	20.0	87.80	2.79

Table B.4: Embedding performances of the half-space model on wordnet Mammal with different hyperparameters.

We mention an interesting tuning result here, take the training of the half-space model over the WordNet Mammal for example, we varies the learning rates for different batchsize as shown in Table. B.4. We found that, if trained

with a larger batchsize, when the learning rate is adjusted (increased) properly, the embedding performance of the converged model with a large batchsize can nearly match the best performance of the converged model with a smaller batchsize. Similar phenomenon was observed for the rest dataset in different dimensions for different models. Hence, we can safely choose batchsize=128 in our main experiments for its running time advantage, with a learning rate 5.0. We provide the code together with the parameters of our implementation in the supplementary material.

B.3 Collage: Light-Weight Low-Precision Strategy for LLM Training

B.3.1 Background on Floating Point Units and Related Work

Floating point units

Floating-point representation uses a sign bit to indicate positive or negative numbers, an exponent to determine scale, and a mantissa for significant digits, enabling efficient handling of a wide range of numbers with potential for precision errors. Different floating-point formats offer varying benefits and trade-offs. Single Precision (FP32) provides wide range and reasonable precision, while consuming more resources. Half Precision (FP16) reduces memory usage and improves efficiency, but sacrifices precision and range. Brain floating point (BF16) as another 16-bit format has a much bigger dynamic range (same as FP32), while having a worse precision than FP16. FP8 (two versions) could

further reduce resources, suitable for constrained environments, but with even more limited precision and range.

We present different formats referenced in the paper along with their exponent and mantissa bits.

Table B.5: Floating-Point precisions and ULPs

Precision	#Exponent bits	#Mantissa (significand) bits	ulp(1)
Single (FP32)	8	23	2^{-23}
Half (FP16)	5	10	2^{-10}
BF16	8	7	2^{-7}
FP8 E4M3	4	3	2^{-3}
FP8 E5M2	5	2	2^{-2}

Related Work

Low Precision and Quantization-aware Training. Fully quantized training attempts to downscale numerical precisions but not to compromise accuracy, mainly for large-scale training, using FP16 [114], BF16 [89], FP8 [179, 159], and other combination of integer types [14, 29]. [115] developed a mixed precision strategy that maintains master weight in FP32 only whereas others are in lower precision of FP16. [186] recently proposed a training method using INT4 but without customized data types, compatitable with contemporary hardwares. In parallel, [133] proposed a new mixed-precision strategy, gradually incorporating 8-bit gradients, optimizer states in an incremental manner, under distributed settings. When it comes to fine-tuning setings, LoRA [81] leverage structure of matrix to update in low-rank. [51, 191, 68] proposed various variants of LoRA more in memory and computationally efficient manners. Overall, these works

develop training strategies based on numerical structures like low-rank over attention matrices and/or sparsity over parameters in each layer, numerical scale of each variables used for gradient updates. However, they lack of thorough diagnosis on imprecision errors, which has been depriving potential algorithmic developments in numeric precision levels. [205] proposed to adopt Kahan summation [88] and stochastic rounding (SR) [42] to alleviate the influence of imprecision and lost arithmetic at the model parameter update step.

Pruning and Distillation. Pruning [73, 98] removes redundant parameters from the network. The goal is to maintain prediction quality of the model while shrinking its size, and therewith increasing its efficiency. distillation [80, 78, 80] transfers knowledge from a large model to a smaller one. Pruning can be combined with distillation approach to further reduce model parameters [149, 100, 188]. Structured pruning removes whole components of the network such as neurons, heads, and layers [195, 100, 213]. Unstructured pruning removes individual weights of the network with smaller magnitudes [58, 187]. Albeit these are useful in reducing computational overhead, distillation and pruning requires either the model already trained as post-hoc method, architecture change than original models or iterative procedures that potentially take longer in an end-to-end manner.

Post-training Quantization. Quantization compresses the representation of the parameters into low-precision data types, reducing the storage when loading the model in devices. Post-training quantization methods quantize the parameters of the pre-trained model [194, 189, 59] often with fine-tuning steps [99]. [84] emulates inference-time quantization, creating a model that can be

quantized later post-training . However, these works mostly focus on faster inference, rather than reducing end-to-end training time.

Kahan Summation. The Kahan summation is a standard algorithm in numerical analysis for accurate summation of floating-point numbers, just like the case of adding updates to the parameter. When incorporated with optimization algorithms such as SGD and AdamW, the Kahan algorithm introduces an auxiliary Kahan variable $\boldsymbol{c}$ (in the same precision) to track numerical errors at the parameter update step (i.e., $\boldsymbol{\theta}_t \leftarrow \mathcal{F}^P(\boldsymbol{\theta}_{t-1} \oplus \boldsymbol{\Delta\theta}_t)$) with $\boldsymbol{c} \leftarrow \mathcal{F}^P\left(\boldsymbol{\Delta\theta}_t \ominus \mathcal{F}^P(\boldsymbol{\theta}_t \ominus \boldsymbol{\theta}_{t-1})\right)$, and to compensate the addition results by adding $\boldsymbol{c}$ to the next iteration update: $\boldsymbol{\Delta\theta}_{t+1} \leftarrow \mathcal{F}^P(\boldsymbol{\Delta\theta}_{t+1} \oplus \boldsymbol{c})$. The Kahan variable $\boldsymbol{c}$ accumulates lost small updates until it grows large enough to be added with the model parameters. As pointed in [205], "16-bit-FPU training with Kahan summation for model weight updates have advantages in terms of throughput and memory consumption compared to 32-bit and mixed precision training", despite of the additional auxiliary value.

Stochastic Rounding. Different from the deterministic rounding-to-the-nearest behavior, stochastic rounding rounds the number up and down in a probabilistic way. For any $x \in \mathbb{R}$, assume $a_u, a_l \in \mathbb{R}$ be the closest upper and lower neighboring floating-point values of x, i.e., $a_l \leq x < a_u = a_l + \text{ulp}(a_l)$, then $\text{SR}(x) = a_l$ with probability $(a_u - x)/(a_u - a_l) = 1 - (x - a_l)/\text{ulp}(a_l)$ and otherwise rounds up to a_u. Stochastic rounding provides an unbiased estimate of the precise value: $\mathbb{E}[\text{SR}(x)] = x$ and alleviates the influence of imprecision by making the addition valid in expectation. Stochastic rounding for model weight updates adds minimal overhead for training and is supported in modern hardwares, such as AWS Trainium instances.

Algorithm 18 TwoSum

1: **Input:** P-bit floats a and b
2: $x \leftarrow \mathcal{F}^P(a \oplus b)$
3: $b_{\text{virtual}} \leftarrow \mathcal{F}^P(x \ominus a)$
4: $a_{\text{virtual}} \leftarrow \mathcal{F}^P(x \ominus b_{\text{virtual}})$
5: $b_{\text{roundoff}} \leftarrow \mathcal{F}^P(b \ominus b_{\text{virtual}})$
6: $a_{\text{roundoff}} \leftarrow \mathcal{F}^P(a \ominus a_{\text{virtual}})$
7: $y \leftarrow \mathcal{F}^P(a_{\text{roundoff}} \oplus b_{\text{roundoff}})$
8: **Return:** (x, y)

Algorithm 19 Split

1: **Input:** P-bit float a (with p-bit mantissa)
2: $c \leftarrow \lfloor \frac{p}{2} \rfloor$
3: $t \leftarrow \mathcal{F}^P(2^c \oplus 1) \cdot a$
4: $a_{hi} \leftarrow \mathcal{F}^P(t \ominus \mathcal{F}^P(t \ominus a))$
5: $a_{lo} \leftarrow \mathcal{F}^P(a \ominus a_{hi})$
6: **Return:** (a_{hi}, a_{lo})

Algorithm 20 TwoProd

1: **Input:** P-bit floats a and b
2: $x \leftarrow \mathcal{F}^P(a \odot b)$
3: $(a_{hi}, a_{lo}) \leftarrow$ **Split**(a)
4: $(b_{hi}, b_{lo}) \leftarrow$ **Split**(b)
5: $err_1 \leftarrow \mathcal{F}^P(p \ominus \mathcal{F}^P(a_{hi} \odot b_{hi}))$
6: $err_2 \leftarrow \mathcal{F}^P(err_1 \ominus \mathcal{F}^P(a_{lo} \odot b_{hi}))$
7: $err_3 \leftarrow \mathcal{F}^P(err_2 \ominus \mathcal{F}^P(a_{hi} \odot b_{lo}))$
8: $e \leftarrow \mathcal{F}^P(\mathcal{F}^P(a_{lo} \odot b_{lo}) \ominus err_3)$
9: **Return:** (x, e)

Algorithm 21 TwoProdFMA

1: **Input:** P-bit floats a and b
2: **Requires:** Machine supports FMA
3: $x \leftarrow \mathcal{F}^P(a \odot b)$
4: $e \leftarrow \mathcal{F}^P(a \odot b \ominus x)$ in FMA
5: **Return:** (x, e)

Algorithm 22 Scaling

1: **Input:** a float v and a length-2 expansion (a_1, a_2)
2: $(x, e) \leftarrow$ **TwoProdFMA**(a_1, v)
3: $e \leftarrow \mathcal{F}^P(a_2 \odot v \oplus e)$
4: $(x, e) \leftarrow$ **Fast2Sum**(x, e)
5: **Return:** (x, e)

Algorithm 23 Mul

1: **Input:** length-2 expansions (a_1, a_2) and (b_1, b_2)
2: $(x, e) \leftarrow$ **TwoProdFMA**(a_1, b_1)
3: $e \leftarrow \mathcal{F}^P(e \oplus ((a_1 \odot b_2) \oplus (a_2 \odot b_1)))$
4: $(x, e) \leftarrow$ **Fast2Sum**(x, e)
5: **Return:** (x, e)

B.3.2 MCF Algorithms and Further Discussions

MCF algorithms

As noted in Theorem 3.4.1, Fast2Sum requires $|a| > |b|$ so as to perform the arithmetic correctly. One can also derive the same length-2 expansion using TwoSum in Algorithm 18 for any floats a, b without sorting.

Another category of basic MCF algorithms is the multiplication, between i) a float and a float (with TwoProd Algorithm 20); ii) a float and a length-2 expansion (with Scaling Algorithm 22); iii) an expansion and an expansion (with Mul Algorithm 23), to produce length-2 expansions.

Case i). TwoProd computes the expansion using another basic algorithm Split (Algorithm 19), which takes a single P-bit floating point value and splits it into its high and low parts, both with $\frac{P}{2}$ bits. On a machine which supports the fused-multiply-add (FMA) instruction set, a much more efficient version TwoProdFMA Algorithm 21 can be adopted to give the same results. We utilized this efficient TwoProdFMA in our implementations as (Bfloat16) FMA is supported on CUDA, e.g., using torch.addcmul$(-x, a, b)$.

Case ii) and iii). Algorithm 22 Scaling describes the multiplication of a single float with a length-2 expansion and Algorithm 23 Mul the multiplication between 2 length-2 expansions. With FMA enabled, both algorithms run efficiently.

We refer the readers to [201] for a full list of MCF algorithms.

Further Discussions on Algorithms

Equivalence. The equivalence of using 'Kahan-sum in the optimizer' at the model-update step and COLLAGE-light is straightforward, realizing i) the Kahan variable $\boldsymbol{c}$ calculation is essentially Fast2Sum given $|\boldsymbol{\theta}_t| \geq |\Delta\boldsymbol{\theta}_{t-1}|$; and ii) next iteration update $\boldsymbol{\Delta\theta}_{t+1}$ has similar magnitude as $\boldsymbol{c}$ so that lost arithmetic doesn't happen. In contrast, COLLAGE-light doesn't have such concerns using Grow.

Weight Decay. [108] propose AdamW with the decoupled weight decay placed at line 12 in Algorithm 9 for a summed update $\Delta\boldsymbol{\theta}_t$, standard libraries including PyTorch and HuggingFace however implement the decoupled weight decay directly to the parameter:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha\lambda\boldsymbol{\theta}_{t-1}, \quad \text{or} \quad \boldsymbol{\theta}_t \leftarrow (1 - \alpha\lambda)\boldsymbol{\theta}_{t-1} \tag{B.4}$$

which works as expected using Float32, but is usually ineffective in Bfloat16 arithmetic due to imprecision and lost arithmetic. For example, a standard choice of the learning rate and weight decay hyper-parameter in GPT-6.7B pretraining is $\alpha = 1.2e-4$ and $\lambda = 0.1$, yielding $\alpha\lambda = 1.2e-5$ and causing lost arithmetic in Equation B.4 when Bfloat16 is used. In fact, the least $\alpha\lambda$ value to avoid invalid arithmetic is half ulp(1.0), i.e., $2^{-7}/2 \approx 0.0039$. Either decaying the parameter (expansion) with Grow or placing the decoupled weight decay term at line 13 following the original AdamW algorithm statement solves the issue, where we chose the latter option in our experiments.

Scalar and Bias Correction. A rule of thumb to avoid imprecision and lost arithmetic during low precision training is to do as many scalar computations in high precision as possible before casting them to low precision (e.g., PyTorch BFloat16 Tensor). For example, in BFloat16 AdamW, it's recommended to compute the bias correction scalar terms $1 - \beta_1^t$ and $1 - \beta_2^t$ in high precision before dividing the low precision momentums.

B.3.3 Additional Experiment Results

Experiment Details

BERT and RoBERTa. We pretrain the BERT-base-uncased, BERT-large-uncased and RoBERTa-base model from HuggingFace [182] on the Wikipedia-en corpus [11], preprocessed with BERT tokenizer. We follow the standard pipeline to pretrain BERT and RoBERTa with the same configs and hyper-parameters for all precision strategies. Note that we took these configs and hyper-parameters from open-sourced models in HuggingFace. We finetuned the pretrained BERT and RoBERTa models following [177] with BF16 mixed precision through HuggingFace and evaluated the final model on GLUE benchmarks. Particularly, we used $2e-5$ learning rate and a batch size of 32 evaluated on single Nvidia A100. All tasks were finetuned for 3 epochs, apart from MRPC which we ran for 5 epochs.

Multi-size GPTs & OpenLLaMA-7B. We conduct pretraining experiments of 1) GPT at different sizes ranging from 125M, 1.3B, 2.7B to 6.7B, and 2) OpenLLaMA-7B using NeMo Megatron [97] with provided standard configs, both on the Wikipedia corpus with HuggingFace GPT2 and LLaMA tokenizer, respectively. We split the dataset into train/val/test with the split ratio $980:10:10$. We trained all models consistently with disabled sequence parallelism, enabled flash attention, rotary positional embedding (of percentage 1.0) [157], disabled transformer engine, untied embeddings & output weights, sequence length of $2,048$, weight decay 0.1 and pipeline parallelism 1, for all GPT models and OpenLLaMA-7B in our experiments unless otherwise specified. All models were trained with the CosineAnnealing learning rate scheduler with 200

Table B.6: Pre-training hyperparameters used for BERT and RoBERTa.

Model	Phase	hyperparameters	Values
BERT-base	Phase-1	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	128
		global batch size	$16,384$
		learning rate	$4e-4$
		(β_1, β_2)	$(0.9, 0.999)$
	Phase-2	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	512
		global batch size	$32,768$
		learning rate	$2.8e-4$
		(β_1, β_2)	$(0.9, 0.999)$
RoBERTa-base	Phase-1	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	512
		global batch size	$8,192$
		learning rate	$lr = 6e-4$
		(β_1, β_2)	$(0.9, 0.98)$

warmup iterations. We trained all GPT models for 20k iterations and OpenLLaMA for 9k iterations due to timing constraints. The dafault value of βs are $\beta_1 = 0.9$ and $\beta_2 = 0.95$ unless otherwise specified, e.g., in ablation experiments. Note that we took these configs from EleutherAI/gpt-neox [10].

Table B.7: Some configs and hyper-parameters of GPT models and OpenLLaMA-7B.

Model	#Layers	HiddenSize	#AttentionHeads	Global BatchSize	TensorParallelism	lr
GPT-125M	12	768	12	$1,024$	1	$6e-4$
GPT-1.3B	24	2048	16	$1,024$	8	$2e-4$
GPT-2.7B	32	2560	32	512	8	$1.6e-4$
GPT-6.7B	32	4096	32	256	8	$1.2e-4$
OpenLLaMA-7B	32	4096	32	256	8	$3e-4$

GPT-30B. For the GPT-30B model used in Section 3.4.4, it has 56 layers, hidden size 7168 and 56 attention heads. We trained it with a global batchsize 256, tensor parallelism 8 and pipeline parallelism 2, then varied the micro batchsize

and sequence length to explore their maxium values without causing OOM on a NVIDIA A100 cluster with 2 nodes, 8 GPUs each.

Memory Statistics

Table B.8 summarizes the peak (total) memory of all training precision strategies during practical runs with the same hyper-parameters for a fair comparison: Sequence Length 2048, Global BatchSize 128 and Micro (per-device) BatchSize 1. We trained GPTs and OpenLLaMA with TensorParallelism 8 over 8×A100 40GB, except from GPT-125M with TensorParallelism 1 on 1×A100 40GB. In Table B.8, we report the saved memory compared to the mixed-precision option D with the percentage calculated. During real runs, on average, COLLAGE formations (light/plus) use **23.8%/15.6%** less peak memory compared to option D. The best savings are for largest model OpenLLaMA-7B, with savings **27.8%/18.5%**, respectively. The memory savings match the theoretical calculation in Table 3.12.

Table B.8: Peak (saved) pretraining memory (GB) of precision strategies compared to option D on GPTs and OpenLLaMA-7B.

Precision	GPT				OpenLLaMA
Option	125M	1.3B	2.7B	6.7B	7B
A	−1.1(−26.6%)	−10.3(−28.9%)	−20.8(−31.2%)	−51.2(−35.6%)	−65.7(−37.2%)
B (ours)	−0.8(−19.3%)	−7.6(−21.5%)	−15.6(−23.8%)	−38.2(−26.6%)	−49.2(−27.8%)
C (ours)	−0.5(−12.1%)	−5.0(−14.1%)	−10.1(−15.4%)	−25.7(−17.9%)	−32.7(−18.5%)
D	4.4	35.5	65.3	143.7	176.7

OpenLLama 7B pretraining

We provide the pretraining iterations progress for OpenLLama-7B (described in Section 3.4.4) in the Figure B.3, B.4 for $\beta_2 = 0.95, 0.99$, respectively. We observe a stable training using COLLAGE-plus when using $\beta_2 = 0.99$, where

other precision strategies show slow convergence. The gradient norm in Figure B.4 left show that COLLAGE-plus has stability while other precision strategies encounter transient errors causing blow-ups in gradients.

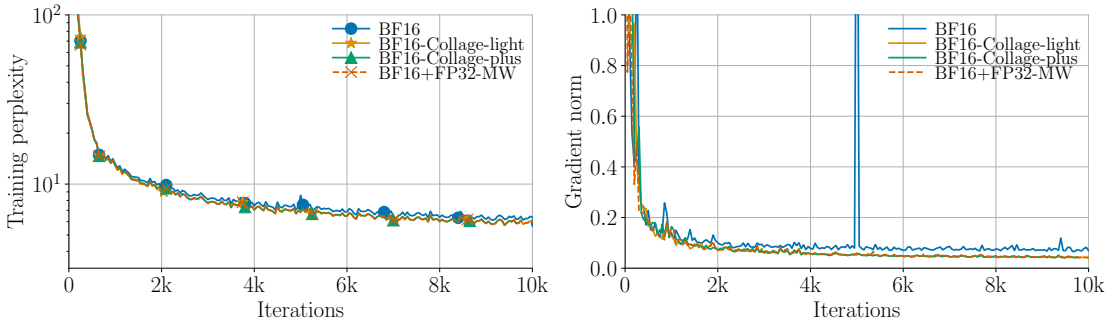


Figure B.3: Openllama 7B pretraining (see settings in Section 3.4.4) with $\beta_2 = 0.95$. **Left:** Training perplexity for different precision strategies listed in Table 3.12. **Right:** Model gradient L2 norm across iterations for different strategies. The COLLAGE formations overlap with heavy-weighted FP32 master weights strategy.

GPT pretraining

The pretraining progress of GPT 125M for various settings of β_2 and global batch sizes is provided in Figure B.5 B.6 B.7 B.8 B.9 B.10. For pretraining of GPT 1.3B, see Figure B.11. For pretraining of 2.7B, see Figure B.12. For pretraining of 6.7B, see Figure B.13.

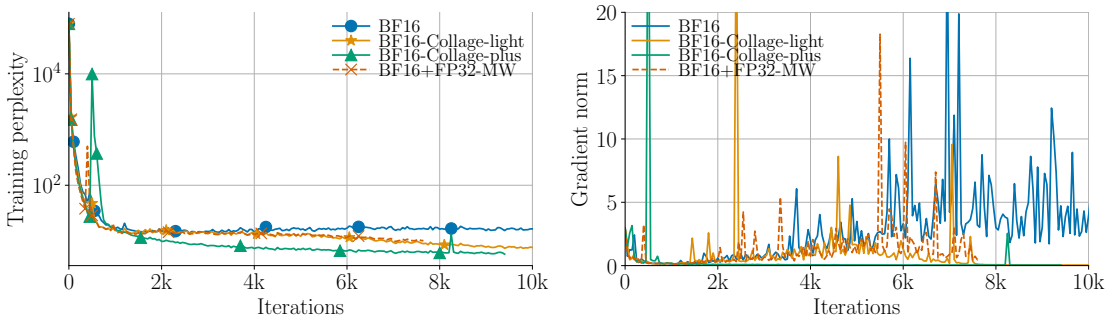


Figure B.4: Openllama 7B pretraining (see settings in Section 3.4.4) with $\beta_2 = 0.99$. **Left:** Training perplexity for different precision strategies listed in Table 3.12. **Right:** Model gradient L2 norm across iterations for different strategies. The COLLAGE-plus results in the best train perplexity over iterations while other approaches struggle. The gradient norm blows-up frequently but stays stable for COLLAGE-plus which suggest importance of using multi-components at critical locations.

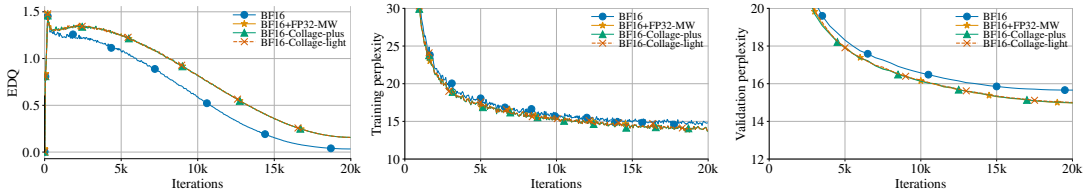


Figure B.5: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=1024, $\beta_2 = 0.95$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

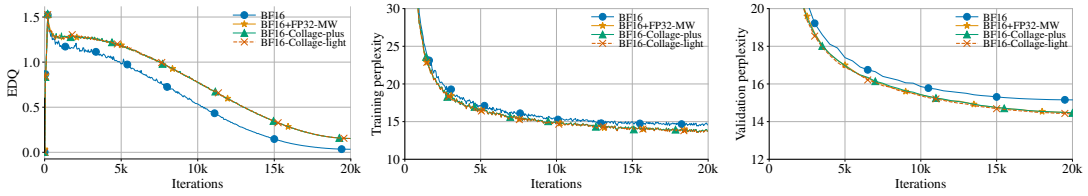


Figure B.6: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=2048, $\beta_2 = 0.95$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

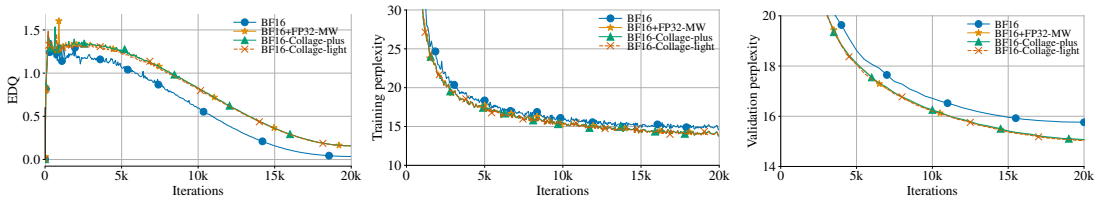


Figure B.7: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=1024, $\beta_2 = 0.99$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

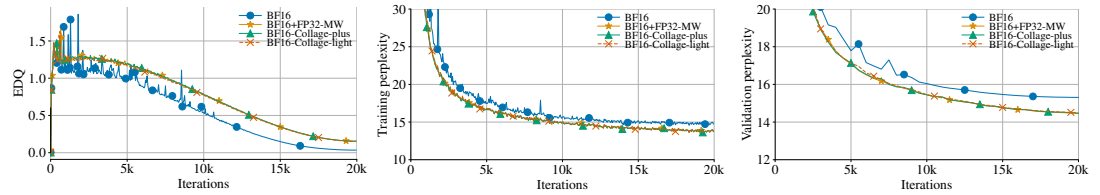


Figure B.8: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=2048, $\beta_2 = 0.99$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

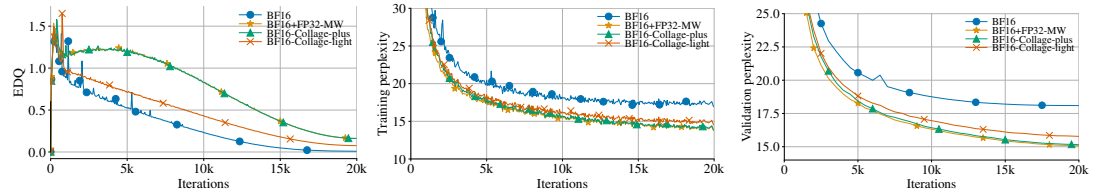


Figure B.9: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=1024, $\beta_2 = 0.999$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

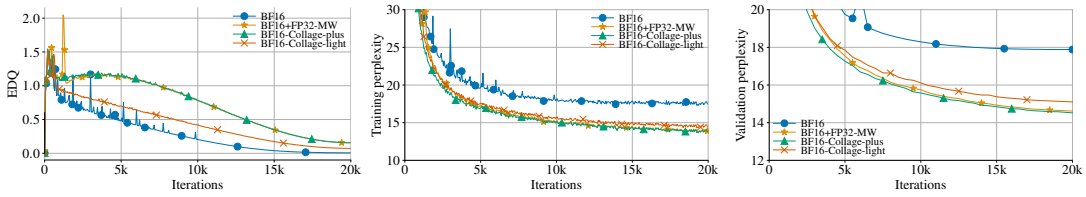


Figure B.10: Pretrainnig progress for GPT 125M with settings described in Section 3.4.4 and global batch-size=2048, $\beta_2 = 0.999$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

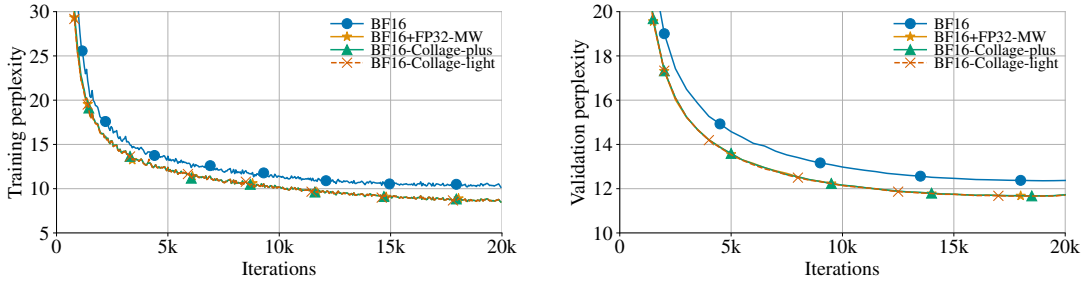


Figure B.11: Pretrainnig progress for GPT 1.3B with settings described in Section 3.4.4 and global batch-size=512, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

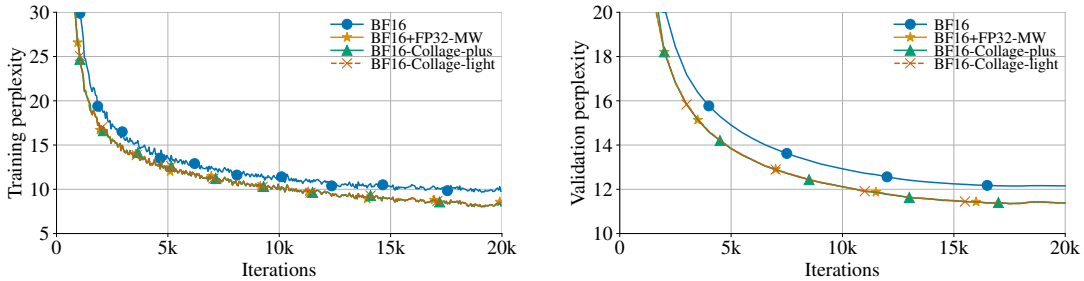


Figure B.12: Pretrainnig progress for GPT 2.7B with settings described in Section 3.4.4 and global batch-size=512, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

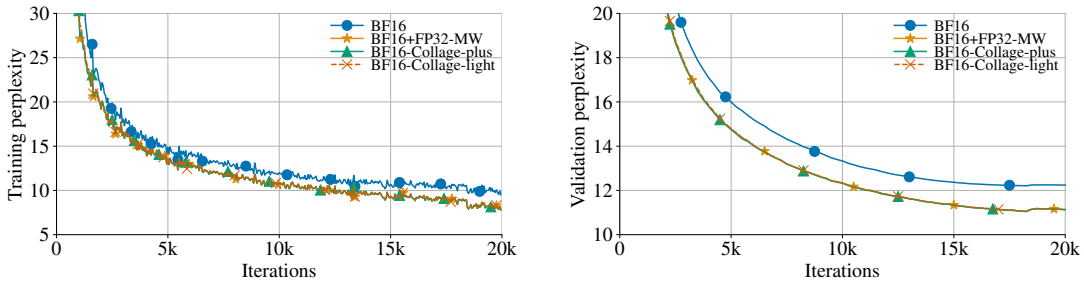


Figure B.13: Pretrainnig progress for GPT 6.7B with settings described in Section 3.4.4 and global batch-size=256, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 3.12. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

B.4 MCTensor: A High-Precision Deep Learning Library

B.4.1 Numerical Error

We further conduct experiments to evaluate the numerical errors of basic MCTensor arithmetic. Specifically, we first compute `Add_MCN(x,y)` of two random MCTensor `x,y` of roughly the same magnitude m. `x,y` are sampled by first sampling two random Julia BigFloat numbers with high precision (e.g. 3000 precision) using the equation $(10 - \mathcal{N}(0,1))^m$, then converted to their corresponding MCTensors in Float32. In order to get the *exact* numerical error, we transform the MCTensor result to a Julia BigFloat number, then compute the relative error of it to the high precision addition of `x,y` (and not the addition of the two BigFloat numbers initially sampled) in Julia. In the same way, we compute the numerical errors of `Mult_MCN(x,y)` and `ScalingN(x,y)` with `y` being MCTensor in the former case and standard (PyTorch) tensor in the later case. `x,y` are sampled in the same way throughout the three cases. The relative error is shown in the top row of Figure B.14.

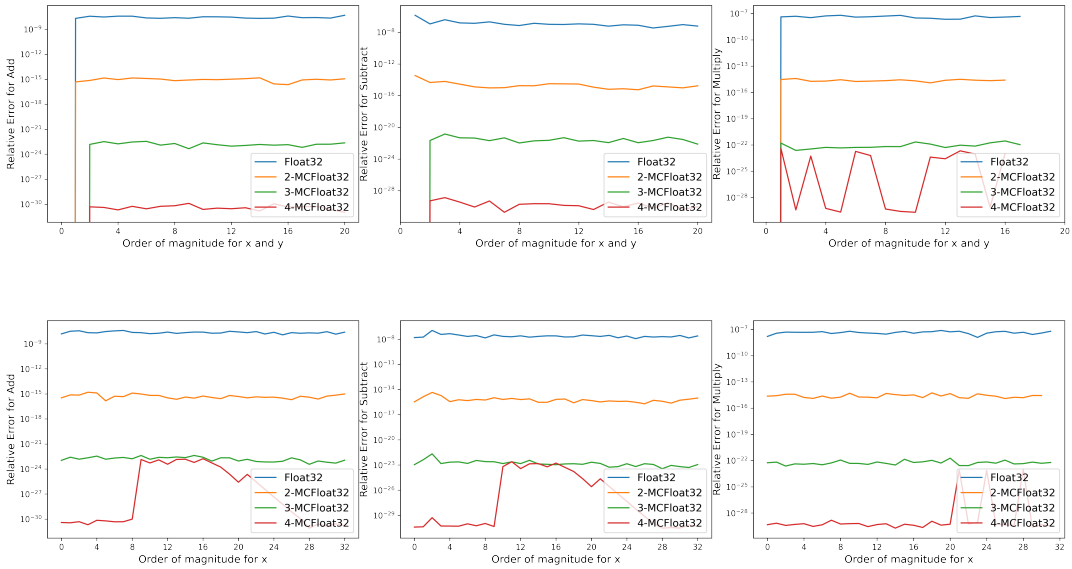


Figure B.14: Relative Error of MCTensor arithmetic with different number of components, compared with high precision Julia BigFloat results (i.e. 3000 precision). Left: `Add−MCN(x,y)`, middle: `ScalingN(x,y)` and right: `Mult−MCN(x,y)`. In the top row, order of magnitudes for `x` and `y` are kept the same. In the bottom row, order of magnitudes for `x` varies and order of magnitudes for `y` is kept at 2.

For a thorough comparison, we also derive the same numerical errors when the order of magnitudes for `x` varies and order of magnitudes for `y` is kept at 2, as shown in the bottom row of Figure B.14.

B.4.2 Detailed Results on Linear, Logistic Regression & MLP

Linear Regression. Table B.9 describes the final training loss of the linear regression task in Section 3.3.4.

Logistic Regression. We apply logistic regression on a synthetic dataset and a breast cancer dataset. The synthetic dataset consists of 1,000 data points, where each data point contains two features, while the breast cancer dataset consists

Model	Train Loss
PyTorch float16	1.99e-4
PyTorch float32	2.64e-12
PyTorch float64	8.02e-18
MCTensor float16, nc = 1	1.80e-4
MCTensor float32, nc = 2	1.95e-7
MCTensor float64, nc = 3	1.95e-7

Table B.9: Final Training Loss Results of Linear Regression Task

of 569 data points and each data point contains 30 numeric features. A single `MCLinear` layer with float16 and *nc* between 1, 2 and 3 are used for logistic regression with Binary Cross Entropy loss

$$\mathcal{L}(W) = \texttt{BCELoss}(y, \text{sigmoid}(XW^T))$$

For both datasets, we randomly split 80% of the data for training and 20% of the data for testing. As both datasets are small in scale, we run them in full batch with MC-SGD and SGD. For the synthetic dataset, we set learning rate to be 3e-3 and run for 4000 epochs. For the breast cancer dataset, we set learning rate to be 1e-4 and momentum as 0.9, and run for 3000 epochs. Below is the results for both dataset and figures for MCLinear results on breast cancer dataset with MC-SGD optimizer. As can be seen, even with as few as 2 components, MCTensor in float16 can match with the training loss of float32 Tensor.

Model	Training Loss	Testing Accuracy (%)
Tensor float16	0.1940	100
Tensor float32	0.1042	100
Tensor float64	0.1042	100
1-MCTensor float16	0.1941	100
2-MCTensor float16	**0.1041**	100
3-MCTensor float16	**0.1041**	100

Table B.10: Final training and testing results for logistic regression on synthetic dataset.

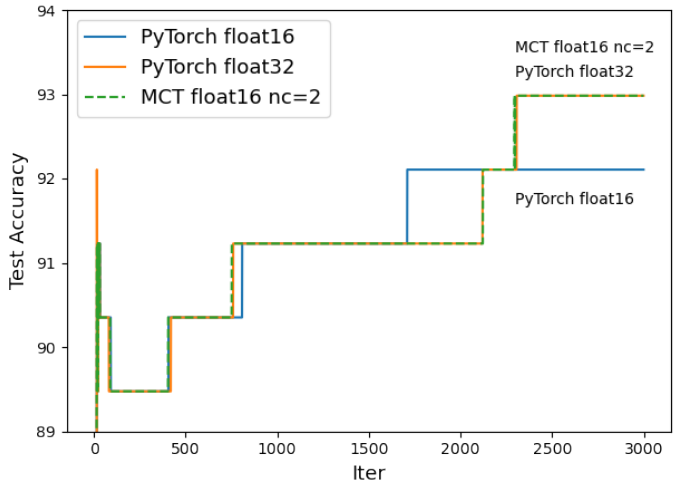


Figure B.15: Test accuracy of MCTensor and PyTorch Tensor for logistic regression on the Breast Cancer dataset.

Model	Training Loss	Testing Accuracy (%)
Tensor float16	0.1944	92.11
Tensor float32	0.1528	**92.98**
Tensor float64	0.1528	**92.98**
1-MCTensor float16	0.1944	92.11
2-MCTensor float16	**0.1527**	**92.98**
3-MCTensor float16	**0.1527**	**92.98**

Table B.11: Final training and testing results for logistic regression on Breast Cancer dataset.

MLP. Using MC-MLP, we perform multi-class classification task on the Reduced MNIST dataset and binary classification task on the Breast Cancer dataset. The training details for both tasks are shown in Table B.12 and Table B.13.

Figure B.16 shows the training loss curves for MC-MLP on the breast cancer and reduced MNIST dataset with MC-SGD optimizer. The hyperparameters used in training are shown in Table B.13, and the final results for testing accuracy is shown in Table B.14.

Parameter	Value
MLP first hidden layer dim	150
MLP second hidden layer dim	150
Batch size	full batch (569)
Optimizer	MC-GD
Learning rate	6e-3
Epoch	1000

Table B.12: Training details for the Breast Cancer dataset.

Parameter	Value
MLP first hidden layer dim	50
MLP second hidden layer dim	50
Batch size	128
Optimizer	MC-SGD
Learning rate	2e-3
Momentum	0.8
Epoch	100

Table B.13: Training details for the reduced MNIST dataset.

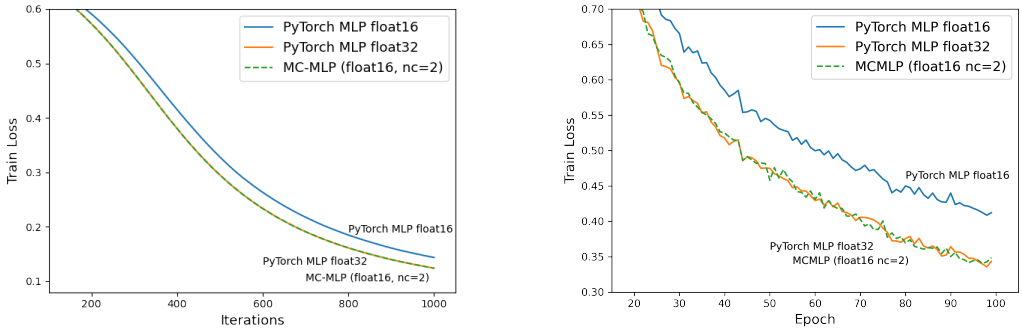


Figure B.16: Training loss curves for MLP on Breast Cancer (left) and reduced MNIST (right).

B.4.3 MCTensor Operators and Modules

Basic Operators

The input of `Two−Sum` is two PyTorch Tensors with same precision, a and b. Algorithm 4 returns the sum $s = \text{fl}(a+b)$ and the error, $err(a+b)$. Here in Algorithm 24 we provide a general version of the `Split` algorithm presented in Algorithm 12, which takes a standard PyTorch floating point value with p-bit significand and splits it into its high and low parts, both with $\frac{p}{2}$-bit of significand.

Model	Training Loss	Testing accuracy
MCMLP (f16 nc=1)	0.424	91.40
MCMLP (f16 nc=2)	0.349	**92.03**
MCMLP (f16 nc=3)	0.349	91.98
PyTorch MLP (f16)	0.412	91.40
PyTorch MLP (f32)	**0.343**	92.00
PyTorch MLP (f64)	**0.343**	92.00

Table B.14: MC-MLP models on reduced MNIST dataset with MCSGD

Algorithm 24 Split

Input: PyTorch Tensor a
if $a.dtype$ is HalfTensor (float16) **then**
  $constant \leftarrow 6$
else if $a.dtype$ is FloatTensor (float32) **then**
  $constant \leftarrow 12$
else if $a.dtype$ is DoubleTensor (float64) **then**
  $constant \leftarrow 26$
end if
$t \leftarrow \mathbf{fl}(2^{constant} + 1) \cdot a$
$a_{hi} \leftarrow \mathbf{fl}(t - \mathbf{fl}(t - a))$
$a_{lo} \leftarrow \mathbf{fl}(a - a_{hi})$
Return: (a_{hi}, a_{lo})

Based on the `Split` algorithm, the `TwoProd` Algorithm 13 computes and returns $p = \mathbf{fl}(a \times b)$ and $e = \text{err}(a \times b)$. Fuse multiply-add, or FMA, is a floating-point operation that performs multiplication and addition in one step. With proper hardware, this Algorithm 25 can speed up `TwoProd`.

Algorithm 25 Two-Prod-fma

Input: PyTorch Tensors a, b
Requires: Machine supports FMA instructions set
$p \leftarrow \mathbf{fl}(a \cdot b)$
$e \leftarrow$ torch.addcmul$(-p, a, b)$
Return: (p, e)

The constraint of decreasing magnitude and non-overlapping across nc might be temporarily violated in computation, so MCTensor must be renor-

malized during computation. Users can specify the target nc after renormalization, r_{nc}, but by default we keep them the same. The `Renormalize` Algorithm 6 is a variant of the Priest's algorithm [138]. We provide a simple and fast implementation here as `Simple−Renorm`, which extracts all non-zero values from a non-renormalized MCTensor and puts together a new MCTensor. Note that this `Simple−Renorm` does not have the same guarantee as `Renormalize`. Therefore in our `Mult−MCN`, we still use the original version of `Renormalize`. But for most other operations, we still utilize `Simple−Renorm` for fast computation.

Algorithm 26 Simple-Renorm

Input: nc-MCTensor x, r_{nc}
Requires: $r_{nc} < nc$
$k \leftarrow 0$; $(b_0, b_1, \cdots, b_{r_{nc}-1}) \leftarrow (0, 0, \cdots, 0)$
for $i = 0$ to $r_{nc} - 1$ **do**
  if $x_i \neq 0$ **then**
    $b_k \leftarrow x_i$
    $k \leftarrow k + 1$
  end if
end for
Return: $(b_0, b_1, \cdots, b_{r_{nc}-1})$

Algorithm 14 describes the multiplication of a MCTensor with a PyTorch Tensor. In our implementation, the user can specify whether the algorithm can return an expanded results with $nc + 1$, or a MCTensor with same nc.

`Add−MCN` (outlined in Algorithm 11), `Div−MCN, Mult−MCN` are operators for addition, division, and multiplication of two nc-MCTensors. Here we have two versions of multiplication, `Mult−MCN` and `Mult−MCN−Slow`. Algorithm 28 is implemented by taking the inverse of the second MCTensor first, and then the first MCTensor is divided by the second MCTensor's inverse. This division would give the result of multiplication. Algorithm 29 follows the same pattern of our definition

of Div−MCN and provides better error bounds, but it is rarely used as the computational cost is too high.

Algorithm 27 Div-MCN, modified from [152]

Input: nc-MCTensor x, y
initialize: $q \leftarrow \mathbf{fl}(x_0/y_0), h_0 \leftarrow q$
for $i = 1$ **to** nc **do**
  $r \leftarrow \mathbf{Add\text{-}MCN}(x, -\mathbf{ScalingN}(y, q, \mathbf{False}))$
  $x \leftarrow r$
  $q \leftarrow \mathbf{fl}(x_0/y_0)$
  $h_i \leftarrow q$
end for
$h \leftarrow (h_0, h_1, \cdots, h_{nc})$
Return: $\mathbf{Simple\text{-}Renorm}(h, nc)$

Algorithm 28 Mult-MCN

Input: nc-MCTensor x, y
initialize: $z_0 \leftarrow 1, z_1 = \cdots = z_{nc-1} \leftarrow 0$
$z \leftarrow (z_0, z_1, \cdots, z_{nc-1})$
$y^{-1} \leftarrow \mathbf{Renormalize}(\mathbf{Div\text{-}MCN}(z, y))$
$h \leftarrow \mathbf{Renormalize}(\mathbf{Div\text{-}MCN}(x, y^{-1}))$
Return: h

Algorithm 29 Mult-MCN-Slow

Input: nc-MCTensor x, y
initialize: $p \leftarrow \mathbf{fl}(x_0 \cdot y_0), h_0 \leftarrow p$
for $i = 1$ **to** nc **do**
  $e \leftarrow \mathbf{Add\text{-}MCN}(x, -\mathbf{DivN}(p, y))$
  $x \leftarrow e$
  $p \leftarrow \mathbf{fl}(x_0 \cdot y_0)$
  $h_i \leftarrow p$
end for
$h \leftarrow (h_0, h_1, \cdots, h_{nc})$
Return: $\mathbf{Simple\text{-}Renorm}(h, nc)$

Algorithm 30, DivN takes input of a PyTorch Tensor and a MCTensor and computes the Div−MCN results by appending zero-value components to a PyTorch Tensor and making it MCTensor.

Algorithm 30 DivN

Input: PyTorch Tensor x_0, nc-MCTensor y,
initialize: $x_1 = \cdots = x_{nc-1} \leftarrow 0$
$x \leftarrow (x_0, x_1, \cdots, x_{nc-1})$
Return: **Div-MCN**(x, y)

The following Algorithm 31 describes the exponential function for a MCTensor.

Algorithm 31 Exp-MCN

Input: nc-MCTensor x
initialize $h \leftarrow \exp(x_0)$
for $i = 1$ to $nc - 1$ **do**
  $h \leftarrow$ **ScalingN**$(h, \exp(h_i), \text{True})$
end for
Return: h

The following Algorithm 32 describes the square of a MCTensor.

Algorithm 32 Square-MCN

Input: nc-MCTensor x
initialize $h_0 \leftarrow \mathbf{fl}(2^{x_0}), h_1 = \cdots = h_{nc-1} \leftarrow 0$
$h \leftarrow (h_0, h_1, \cdots, h_{nc-1})$
$h \leftarrow$ **Grow-ExpN**$(h, \mathbf{fl}(2 \cdot x_0 \cdot x_1))$
Return: h

Matrix Operators

Based on the dimensions of input MCTensor and PyTorch Tensor, `Dot-MCN`, `MV-MCN`, `MM-MCN`, `BMM-MCN` and `4DMM-MCN` are implemented for calculating the matrix-level multiplication results. All operations are identical with the PyTorch implementations.

Algorithm 33 Dot-MCN

Input: *nc*-MCTensor x, PyTorch Tensor v
Requires: x and v both 1D array
$h \leftarrow$ **ScalingN**(x, v, False)
$h_{tensor} = h_0 + h_1 + \cdots + h_{nc-1}$
Return: h_{tensor}

Algorithm 34 MV-MCN

Input: *nc*-MCTensor x, PyTorch Tensor v
Requires: x is 2D matrix of size $(n \times m)$ and v is 1D array of size m
$scaled \leftarrow$ **ScalingN**(x, v, False)
$h \leftarrow scaled[..., 0]$
for $i = 1$ to $m - 1$ **do**
  $h \leftarrow$ **Add-MCN**$(h, scaled[..., i])$
end for
Return: h

Algorithm 35 MM-MCN

Input: *nc*-MCTensor x, PyTorch Tensor v
Requires: x is 2D matrix of size $(n \times m)$ and v is 2D Matrix of size $(m \times p)$
$x \leftarrow x$.**unsqueeze**(-1)
$v \leftarrow v$.**transpose**$(-1, -2)$
$scaled \leftarrow$ **ScalingN**(x, v, False)
$h \leftarrow scaled[..., 0]$
for $i = 1$ to $m - 1$ **do**
  $h \leftarrow$ **Add-MCN**$(h, scaled[..., i])$
end for
Return: h

Algorithm 36 BMM-MCN

Input: *nc*-MCTensor x, PyTorch Tensor v
Requires: x is 3D matrix of size $(b \times n \times m)$ and v is 3D Matrix of size $(b \times m \times p)$

$x \leftarrow x$.**unsqueeze**(-1)
$v \leftarrow v$.**unsqueeze**(1).**transpose**$(-1, -2)$
$scaled \leftarrow$ **ScalingN**(x, v, False)
$h \leftarrow scaled[..., 0]$
for $i = 1$ to $m - 1$ **do**
  $h \leftarrow$ **Add-MCN**$(h, scaled[..., i])$
end for
Return: h

Algorithm 37 4DMM-MCN

Input: *nc*-MCTensor x, PyTorch Tensor v
Requires: x is 4D matrix of size $(a \times b \times n \times m)$ and v is 4D Matrix of size $(c \times d \times m \times p)$; two sizes can be broadcasted in **torch.matmul**
$x \leftarrow x$.**unsqueeze**(-1)
$v \leftarrow v$.**unsqueeze**(2).**transpose**$(-1, -2)$
$scaled \leftarrow$ **ScalingN**(x, v, False)
$h \leftarrow scaled[..., 0]$
for $i = 1$ to $m - 1$ **do**
  $h \leftarrow$ **Add-MCN**$(h, scaled[..., i])$
end for
Return: h

The following Algorithm 38 computes the matrix multiplication of a *nc*-MCTensor with a PyTorch Tensor first, then times a constant α. Then the product of another *nc*-MCTensor times a constant β is added to the former multiplication result.

Algorithm 38 AddMM-MCN

Input: *nc*-MCTensor x, *nc*-MCTensor y, PyTorch Tensor v, β, α
Requires: x is 2D matrix of size $(n \times m)$, and v is 2D Matrix of size $(m \times p)$; y is 2D matrix of size $(n \times m)$ or y is 1D array of size m
$h \leftarrow$ **ScalingN**(**MM-MCN**$(x, v), \alpha)$
$bias \leftarrow$ **ScalingN**(y, β)
$h \leftarrow$ **Add-MCN**$(h, bias)$
Return: h

This Algorithm 39, `Matmul−MCN` is the central function for handling all matrix level multiplications of one *nc*-MCTensor and one PyTorch Tensor.

Algorithm 39 Matmul-MCN

Input: *nc*-MCTensor x, PyTorch Tensor y
$x_d, y_d \leftarrow x$.**dim()**, y.**dim()**
if $x_d = 1$ and $y_d = 1$ **then**
  Return: Dot-MCN(x, y)
else if $x_d = 2$ and $y_d = 2$ **then**
  Return: MM-MCN(x, y)
else if $x_d = 2$ and $y_d = 1$ **then**
  Return: MV-MCN(x, y)
else if $x_d > 2$ and $y_d = 1$ **then**
  Return: ScalingN(x, y)
else if $x_d = y_d$ and $x_d = 3$ **then**
  Return: BMM-MCN(x, y)
else if $x_d = y_d$ and $x_d = 4$ **then**
  Return: 4DMM-MCN(x, y)
end if

MCTensor Deep Learning models

As a demonstration for the ease of using MCTensor to build a MCTensor deep learning model, we provide an example code for MCTensor MLP (MC-MLP) model for the multi-class classification tasks. Essentially, the only differences visible to the users are the need to set the number of components, and the need to convert MCTensor output to Tensor approximation between different layers. The need for Tensor approximation is explained below.

In MCModule, all network layers take Tensor as inputs, keep MCTensor as their weights, and through MCTensor matrix operations, produces MCTensor as output. If the MCTensor outputs need to be taken as input for the next MCModule layer, there need to be new layers that will perform multiplication on two

```
class MC_MLP(MCModule):
    def __init__(self, input_dim, hidden1, hidden2, num_classes=10, nc=2):
        super(MCMLP, self).__init__()
        self.fc1 = MCLinear(input_dim, hidden1, nc=nc)
        self.fc2 = MCLinear(hidden1, hidden2, nc=nc)
        self.fc3 = MCLinear(hidden2, num_classes, nc=nc)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x.tensor.sum(-1))
        x = F.relu(self.fc2(x))
        x = self.dropout(x.tensor.sum(-1))
        x = self.fc3(x)
        return F.log_softmax(x.tensor.sum(-1), dim=1)
```

Figure B.17: MC-MLP code example.

MCTensor matrices. As can be seen from Table B.15, **ScalingN**, the multiplication between a MCTensor and a PyTorch Tensor, is more than a hundred times faster than **Mult-MCN**, the multiplication between two MCTensor. To avoid this forbidden cost of computation, we convert the unevaluated sums into an evaluated sum as shown above (`x.tensor.sum(-1)`). Although there might some losses of precision due to summation, the loss is marginal and we trade it for an orders of magnitude smaller execution time.

B.4.4 Running Time of MCTensor Operators

We run on a AMD Ryzen 7 5800X CPU with 64 GB memory. For all the basic operators, we repeat PyTorch addition for 7e3 runs and 1-, 2-, 3-MCTensor for 7 runs.

For vector-vector (`torch.dot`/**Dot-MCN**) product, we repeat PyTorch addition for 7e6 runs and 1-, 2-, 3-MCTensor for 7e3 runs. For matrix-vector (`torch.mv`/**MV-MCN**) product and batched matrix-matrix (`torch.bmm`/**BMM-MCN**) product, we repeat PyTorch addition for 7e3 runs and 1-, 2-, 3-MCTensor

for 7 runs. For matrix-matrix (`torch.matmul`/**Matmul-MCN**) product and bias-matrix-matrix (`torch.addmm`/**AddMM-MCN**) addition-product, we repeat PyTorch addition for 7e4 runs and 1-, 2-, 3-MCTensor for 7 runs.

The timing for both matrix and basic operators are given in Table B.15 and Table B.16.

Operators	Inputs sizes	FloatTensor	1-MCTensor	2-MCTensor	3-MCTensor
Add-MCN	(1000×1000)	497 μs ± 6.77 μs	26.7 ms ± 486 μs	44.7 ms ± 379 μs	64.4 ms ± 385 μs
ScalingN	(1000×1000)	490 μs ± 9.69 μs	33.7 ms ± 402 μs	57.5 ms ± 842 μs	84.7 ms ± 1.78 ms
Mult-MCN	(1000×1000)	514 μs ± 15.2 μs	218 ms ± 4.03 ms	667 ms ± 11.6 ms	1.4 s ± 21.6 ms
Div-MCN	(1000×1000)	510 μs ± 10.6 μs	80.3 ms ± 770 μs	243 ms ± 3.24 ms	498 ms ± 8.26 ms

Table B.15: MCTensor basic operators running Time (mean ± sd)

Operators	Inputs sizes	FloatTensor	1-MCTensor	2-MCTensor	3-MCTensor
Dot-MCN	$5000, 5000$	1.61 μs ± 3.29 ns	442 μs ± 5.61 μs	656 μs ± 1.16 μs	858 μs ± 12.2 μs
MV-MCN	$(5000 \times 500), 500$	157 μs ± 4.32μs	320 ms ± 5.78ms	460 ms ± 10.7ms	580 ms ± 12.1ms
Matmul-MCN	$(500\times 200), (200\times 50)$	97.3 μs ± 1.1 μs	495 ms ± 10.8 ms	735 ms ± 21.7 ms	934 ms ± 28 ms
AddMM-MCN	$100, (500\times 200), (200\times 100),$	153 μs ± 869 ns	750 ms ± 21.9 ms	1.12 s ± 28.4 ms	1.44 s ± 49.1 ms
BMM-MCN	$(16 \times 500\times 200), (16 \times 200\times 50)$	1.5 ms ± 9.32 μs	4.84 s ± 43.1 ms	8.03 s ± 68.4 ms	11.2 s ± 74.1 ms

Table B.16: MCTensor matrix operators running time (mean ± sd)

BIBLIOGRAPHY

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015.

[2] Shmuel Agmon. On the spectral theory of the laplacian on noncompact hyperbolic manifolds. *Journées Équations aux dérivées partielles*, pages 1–16, 1987.

[3] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[4] Noga Alon, Troy Lee, Adi Shraibman, and Santosh Vempala. The approximate rank of a matrix and its algorithmic applications: approximate rank. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 675–684, 2013.

[5] Noga Alon, Shay Moran, and Amir Yehudayoff. Sign rank versus vc dimension. In *Conference on Learning Theory*, pages 47–80. PMLR, 2016.

[6] J. Anderson. *Hyperbolic Geometry*. Springer Undergraduate Mathematics Series. Springer London, 2005.

[7] James W Anderson. *Hyperbolic geometry*. Springer Science & Business Media, 2006.

[8] James W. Anderson. *Hyperbolic geometry*. Springer, 2007.

[9] Roy M Anderson and Robert M May. *Infectious diseases of humans: dynamics and control*. Oxford university press, 1992.

[10] Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Jason Phang, Shivanshu Purohit, Hailey Schoelkopf, Dashiell Stander, Tri Songz, Curt Tigges, Benjamin Thérien, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023.

[11] Giusepppe Attardi. Wikiextractor. `https://github.com/attardi/wikiextractor`, 2015.

[12] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*, 2018.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[14] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks, 2018.

[15] Alan F Beardon. *The geometry of discrete groups*, volume 91. Springer Science & Business Media, 2012.

[16] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3), 2023.

[17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[18] Silvere Bonnabel. Stochastic gradient descent on riemannian manifolds. *IEEE Transactions on Automatic Control*, 58(9):2217–2229, 2013.

[19] Michael Boratko, Dongxu Zhang, Nicholas Monath, Luke Vilnis, Kenneth L Clarkson, and Andrew McCallum. Capacity and bias of learned geometric embeddings for directed graphs. *Advances in Neural Information Processing Systems*, 34:16423–16436, 2021.

[20] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.

[21] Brian H Bowditch. A course on geometric group theory. *Mathematical Society of Japan*, 16 of MSJ Memoirs, 2006.

[22] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[23] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097, 2016.

[24] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[25] James W. Cannon, William J. Floyd, Richard Kenyon, Walter, and R. Parry. Hyperbolic geometry. In *In Flavors of geometry*, pages 59–115. University Press, 1997.

[26] Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *Proceedings of the 13th international workshop on mining and learning from graphs held in conjunction with KDD*, 2017.

[27] Ines Chami, Adva Wolf, Da-Cheng Juan, Frederic Sala, Sujith Ravi, and Christopher Ré. Low-dimensional hyperbolic knowledge graph embeddings. *arXiv preprint arXiv:2005.00545*, 2020.

[28] Ines Chami, Zhitao Ying, Christopher Ré, and Jure Leskovec. Hyperbolic graph convolutional neural networks. *Advances in neural information processing systems*, 32:4868–4879, 2019.

[29] Jianfei Chen, Yu Gai, Zhewei Yao, Michael W Mahoney, and Joseph E Gonzalez. A statistical framework for low-bitwidth training of deep neural networks. *Advances in neural information processing systems*, 33:883–894, 2020.

[30] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[31] Wei Chen, Wenjie Fang, Guangda Hu, and Michael W Mahoney. On the hyperbolicity of small-world and treelike random graphs. *Internet Mathematics*, 9(4):434–491, 2013.

[32] Eli Chien, Puoya Tabaghi, and Olgica Milenkovic. Hyperaid: Denoising in hyperbolic spaces for tree-fitting and hierarchical clustering. *arXiv preprint arXiv:2205.09721*, 2022.

[33] Hyunghoon Cho, Benjamin DeMeo, Jian Peng, and Bonnie Berger. Large-margin classification in hyperbolic space. In *The 22nd international conference on artificial intelligence and statistics*, pages 1832–1840. PMLR, 2019.

[34] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and efficient 2-bit quantized neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 348–359, 2019.

[35] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[36] Fellbaum Christiane. Wordnet: an electronic lexical database. *Computational Linguistics*, pages 292–296, 1998.

[37] Aaron Clauset, Cristopher Moore, and Mark EJ Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98, 2008.

[38] J.H. Conway, H. Burgiel, and C. Goodman-Strauss. *The Symmetries of Things*. Ak Peters Series. Taylor & Francis, 2008.

[39] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[40] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[41] H. S. M. Coxeter. Regular honeycombs in hyperbolic space. In *III, Noordhoff, Groningen, and North-Holland*, page 155, 1956.

[42] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Theo Mary, and Mantas Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3):211631, 2022.

[43] Andrej Cvetkovski and Mark Crovella. Multidimensional scaling in the poincaré disk. *Applied Mathematics & Information Sciences*, abs/1105.5332, 05 2011.

[44] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[45] Basudeb Datta and Subhojoy Gupta. Uniform tilings of the hyperbolic plane. *arXiv e-prints*, page arXiv:1806.11393, Jun 2018.

[46] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[47] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29:3844–3852, 2016.

[48] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18(3):224–242, jun 1971.

[49] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[50] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[51] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[52] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[53] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

[54] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[55] Pengfei Fang, Mehrtash Harandi, and Lars Petersson. Kernel methods in hyperbolic spaces. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10665–10674, 2021.

[56] Marcello Federico, Sebastian Stüker, and François Yvon, editors. *Proceed-*

ings of the 11th International Workshop on Spoken Language Translation: Evaluation Campaign, Lake Tahoe, California, December 4-5 2014.

[57] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[58] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

[59] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[60] Octavian-Eugen Ganea, Gary Bécigneul, and Thomas Hofmann. Hyperbolic neural networks. *arXiv preprint arXiv:1805.09112*, 2018.

[61] David Gans. A new model of the hyperbolic plane. *The American Mathematical Monthly*, 73(3):291–295, 1966.

[62] David Gilbarg and Neil S Trudinger. *Elliptic partial differential equations of second order*, volume 224. springer, 2015.

[63] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991.

[64] Alexander Grigor'yan and Masakazu Noguchi. The heat kernel on hyperbolic space. *Bulletin of the London Mathematical Society*, 30(6):643–650, 1998.

[65] Mikhael Gromov. Hyperbolic groups. *Essays in group theory*, page 75–263, 1987.

[66] Albert Gu, Frederic Sala, Beliz Gunel, and Christopher Ré. Learning mixed-curvature representations in product spaces. In *International Conference on Learning Representations*, 2019.

[67] Caglar Gulcehre, Misha Denil, Mateusz Malinowski, Ali Razavi, Razvan Pascanu, Karl Moritz Hermann, Peter Battaglia, Victor Bapst, David Raposo, Adam Santoro, et al. Hyperbolic attention networks. *arXiv preprint arXiv:1805.09786*, 2018.

[68] Han Guo, Philip Greengard, Eric P Xing, and Yoon Kim. Lq-lora: Low-rank plus quantized matrix decomposition for efficient language model finetuning. *arXiv preprint arXiv:2311.12023*, 2023.

[69] Maosheng Guo, Yu Zhang, and Ting Liu. Gaussian transformer: A lightweight approach for natural language inference. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):6489–6496, Jul. 2019.

[70] Isabelle Guyon. Design of experiments of the nips 2003 variable selection benchmark. In *NIPS 2003 workshop on feature extraction and feature selection*, volume 253, page 40, 2003.

[71] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[72] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[73] Song Han, Huizi Mao, and William J Dally. Deep compression: Com-

pressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[74] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[75] Marti A Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING 1992 Volume 2: The 14th International Conference on Computational Linguistics*, 1992.

[76] Sigurdur Helgason. *Groups and geometric analysis: integral geometry, invariant differential operators, and spherical functions*, volume 83. American Mathematical Society, 2022.

[77] Yozo Hida, Xiaoye S Li, and David H Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, page 19, 2007.

[78] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[79] Peter D Hoff, Adrian E Raftery, and Mark S Handcock. Latent space approaches to social network analysis. *Journal of the american Statistical association*, 97(460):1090–1098, 2002.

[80] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes, 2023.

[81] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[82] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[83] Shyuichi Izumiya. Horospherical geometry in the hyperbolic space. In *Noncommutativity and Singularities: Proceedings of French–Japanese symposia held at IHÉS in 2006*, volume 55, pages 31–50. Mathematical Society of Japan, 2009.

[84] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[85] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

[86] Zhe Jia and Peter Van Sandt. Zhe jia and peter van sandt. dissecting the ampere gpu architecture via microbenchmarking. gpu technology conference, 2021. In *GTC*, 2021.

[87] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor pro-

cessing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[88] William Kahan. How futile are mindless assessments of roundoff in floating-point computation. *Preprint, University of California, Berkeley*, 2006.

[89] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[90] Svetlana Katok. *Fuchsian groups*. University of Chicago press, 1992.

[91] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[92] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[93] Donald Ervin Knuth. *The art of computer programming. 2. Seminumerical algorithms*, volume 2. Addison-Wesley, 1971.

[94] Benedikt Kolbe and Vanessa Robins. Tiling the euclidean and hyperbolic planes with ribbons. *arXiv preprint arXiv:1904.03788*, 2019.

[95] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[96] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-precision training for nlp and speech recognition with openseq2seq, 2018.

[97] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, and Jonathan M. Cohen. Nemo: a toolkit for building ai applications using neural modules, 2019.

[98] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models, 2022.

[99] Se Jung Kwon, Jeonghoon Kim, Jeongin Bae, Kang Min Yoo, Jin-Hwa Kim, Baeseong Park, Byeongwook Kim, Jung-Woo Ha, Nako Sung, and Dongsoo Lee. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. *arXiv preprint arXiv:2210.03858*, 2022.

[100] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[101] Matt Le, Stephen Roller, Laetitia Papaxanthos, Douwe Kiela, and Maximilian Nickel. Inferring concept hierarchies from text corpora via hyperbolic embeddings. *arXiv preprint arXiv:1902.00913*, 2019.

[102] Matthew Le, Apoorv Vyas, Bowen Shi, Brian Karrer, Leda Sari, Rashel

Moritz, Mary Williamson, Vimal Manohar, Yossi Adi, Jay Mahadeokar, and Wei-Ning Hsu. Voicebox: Text-guided multilingual universal speech generation at scale, 2023.

[103] Troy Lee and Adi Shraibman. An approximation algorithm for approximation rank. In *2009 24th Annual IEEE Conference on Computational Complexity*, pages 351–357. IEEE, 2009.

[104] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[105] Xiang Li, Luke Vilnis, and Andrew McCallum. Improved representation learning for predicting commonsense ontologies. *arXiv preprint arXiv:1708.00549*, 2017.

[106] Qi Liu, Maximilian Nickel, and Douwe Kiela. Hyperbolic graph neural networks. *arXiv preprint arXiv:1910.12892*, 2019.

[107] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[108] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[109] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[110] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. Mega: moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022.

[111] Emile Mathieu, Charline Le Lan, Chris J Maddison, Ryota Tomioka, and Yee Whye Teh. Continuous hierarchical representations with poincaré variational auto-encoders. *Advances in neural information processing systems*, 32, 2019.

[112] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, Jul 2000.

[113] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[114] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[115] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[116] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013.

[117] Shigeru Miyagawa, Robert C Berwick, and Kazuo Okanoya. The emergence of hierarchical structure in human language. *Frontiers in psychology*, 4:71, 2013.

[118] Alessandro Moschitti, Bo Pang, and Walter Daelemans. Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp). In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[119] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

[120] Ram MurtiRawat, Shivam Panchal, Vivek Kumar Singh, and Yash Panchal. Breast cancer detection using k-nearest neighbors, logistic regression and ensemble learning. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 534–540, 2020.

[121] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *Thirtieth Aaai conference on artificial intelligence*, 2016.

[122] Maximillian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. *Advances in neural information processing systems*, 30, 2017.

[123] Maximillian Nickel and Douwe Kiela. Learning continuous hierarchies in the lorentz model of hyperbolic geometry. In *International conference on machine learning*, pages 3779–3788. PMLR, 2018.

[124] Hee Oh. Euclidean traveller in hyperbolic worlds. *arXiv preprint arXiv:2209.01306*, 2022.

[125] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider,

Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher,

Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2023.

[126] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

[127] Hyunsun Park, Jun Haeng Lee, Youngmin Oh, Sangwon Ha, and Seungwon Lee. Training deep neural network in limited precision, 2018.

[128] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances*

in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[129] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[130] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[131] William Peebles and Saining Xie. Scalable diffusion models with transformers. *arXiv preprint arXiv:2212.09748*, 2022.

[132] Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A Smith, and Lingpeng Kong. Random feature attention. *arXiv preprint arXiv:2103.02143*, 2021.

[133] Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang, Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models, 2023.

[134] Wei Peng, Tuomas Varanka, Abdelrahman Mostafa, Henglin Shi, and Guoying Zhao. Hyperbolic deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(12):10023–10044, 2021.

[135] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[136] Peter Petersen. *Riemannian geometry*, volume 171. Springer, 2006.

[137] Melissa Potter and Jason M. Ribando. Isometries, tessellations and escher, oh my. *American Journal of Undergraduate Research*, 3, 03 2005.

[138] Douglas M Priest. Algorithms for arbitrary precision floating point arithmetic. *University of California, Berkeley*, 1991.

[139] Douglas M Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, Citeseer, 1992.

[140] Ali Rahimi, Benjamin Recht, et al. Random features for large-scale kernel machines. In *NIPS*, volume 3, page 5. Citeseer, 2007.

[141] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks, 2016.

[142] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Physical review E*, 67(2):026112, 2003.

[143] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[144] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with

interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[145] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[146] Walter Rudin. *Fourier analysis on groups*. Courier Dover Publications, 2017.

[147] Frederic Sala, Chris De Sa, Albert Gu, and Christopher Ré. Representation tradeoffs for hyperbolic embeddings. In *International conference on machine learning*, pages 4460–4469. PMLR, 2018.

[148] David Salomon. *Variable-length Codes for Data Compression*. Springer-Verlag, Berlin, Heidelberg, 2007.

[149] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[150] Rik Sarkar. Low distortion delaunay embedding of trees in hyperbolic plane. In *Graph Drawing: 19th International Symposium, GD 2011, Eindhoven, The Netherlands, September 21-23, 2011, Revised Selected Papers 19*, pages 355–366. Springer, 2012.

[151] Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008.

[152] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[153] Ryohei Shimizu, Yusuke Mukuta, and Tatsuya Harada. Hyperbolic neural networks++. *arXiv preprint arXiv:2006.08210*, 2020.

[154] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[155] Sho Sonoda, Isao Ishikawa, and Masahiro Ikeda. Fully-connected network on noncompact symmetric space and ridgelet transform based on helgason-fourier analysis. *arXiv preprint arXiv:2203.01631*, 2022.

[156] Rishi Sonthalia and Anna Gilbert. Tree! i am no tree! i am a low dimensional hyperbolic embedding. *Advances in Neural Information Processing Systems*, 33:845–856, 2020.

[157] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[158] Teruhisa Sugimoto. Convex pentagons for edge-to-edge tiling, ii. *Graphs and Combinatorics*, 31(1):281–298, 2015.

[159] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019.

[160] Yi Tay, Dara Bahri, Donald Metzler, Da-Cheng Juan, Zhe Zhao, and Che Zheng. Synthesizer: Rethinking self-attention for transformer models.

In *International conference on machine learning*, pages 10183–10192. PMLR, 2021.

[161] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

[162] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[163] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[164] Alexandru Tifrea, Gary Bécigneul, and Octavian-Eugen Ganea. Poincar\'e glove: Hyperbolic word embeddings. *arXiv preprint arXiv:1810.06546*, 2018.

[165] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021.

[166] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[167] Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4344–4353, Hong Kong, China, November 2019. Association for Computational Linguistics.

[168] Albert Tseng, Jennifer J. Sun, and Yisong Yue. Automatic synthesis of diverse weak supervision sources for behavior analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2211–2220, June 2022.

[169] Andrew Tulloch and Yangqing Jia. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427*, 2017.

[170] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[171] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[172] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[173] Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. *ICLR (Poster)*, 2(3):4, 2019.

[174] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-embeddings of images and language. *arXiv preprint arXiv:1511.06361*, 2015.

[175] John Voight. Computing fundamental domains for fuchsian groups. *Journal de théorie des nombres de Bordeaux*, 21(2):467–489, 2009.

[176] John Voight. The arithmetic of quaternion algebras. *preprint*, 2014.

[177] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

[178] Ming-Xi Wang. Laplacian eigenspaces, horocycles and neuron models on hyperbolic spaces, 2021. In *URL https://openreview. net/forum*, 2020.

[179] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.

[180] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[181] Zhongyuan Wang, Haixun Wang, Ji-Rong Wen, and Yanghua Xiao. An inference approach to basic level of categorization. In *Proceedings of the 24th acm international on conference on information and knowledge management*, pages 653–662, 2015.

[182] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan

Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[183] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

[184] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks, 2018.

[185] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q Zhu. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 481–492, 2012.

[186] Haocheng Xi, ChangHao Li, Jianfei Chen, and Jun Zhu. Training transformers with 4-bit integers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[187] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285*, 2023.

[188] Mengzhou Xia, Zexuan Zhong, and Danqi Chen. Structured pruning learns compact and accurate models. *arXiv preprint arXiv:2204.00408*, 2022.

[189] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

[190] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[191] Yuhui Xu, Lingxi Xie, Xiaotao Gu, Xin Chen, Heng Chang, Hengheng Zhang, Zhensu Chen, Xiaopeng Zhang, and Qi Tian. Qa-lora: Quantization-aware low-rank adaptation of large language models. *arXiv preprint arXiv:2309.14717*, 2023.

[192] Hanqi Yan, Lin Gui, and Yulan He. Hierarchical interpretation of neural text classification. *Computational Linguistics*, 48(4):987–1020, 2022.

[193] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7370–7377, 2019.

[194] Zhewei Yao, Xiaoxia Wu, Cheng Li, Stephen Youn, and Yuxiong He. Zeroquant-v2: Exploring post-training quantization in llms from comprehensive study to low rank compensation. *arXiv preprint arXiv:2303.08302*, 2023.

[195] Fang Yu, Kun Huang, Meng Wang, Yuan Cheng, Wei Chu, and Li Cui. Width & depth pruning for vision transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3143–3151, 2022.

[196] Tao Yu and Christopher De Sa. Hyla: Hyperbolic laplacian features for graph learning. *arXiv preprint arXiv:2202.06854*, 2022.

[197] Tao Yu and Christopher De Sa. Random laplacian features for learning with hyperbolic space. In *International Conference on Learning Representations*, 2023.

[198] Tao Yu and Christopher M De Sa. Numerically accurate hyperbolic embeddings using tiling-based models. *Advances in Neural Information Processing Systems*, 32, 2019.

[199] Tao Yu and Christopher M De Sa. Numerically accurate hyperbolic embeddings using tiling-based models. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[200] Tao Yu and Christopher M De Sa. Representing hyperbolic space accurately using multi-component floats. *Advances in Neural Information Processing Systems*, 34:15570–15581, 2021.

[201] Tao Yu, Went Guo, Jianan Canal Li, Tiancheng Yuan, and Christopher De Sa. Mctensor: A high-precision deep learning library with multi-component floating-point. *arXiv preprint arXiv:2207.08867*, 2022.

[202] Tao Yu, Wentao Guo, Jianan Canal Li, Tiancheng Yuan, and Christopher De Sa. Htorch: Pytorch-based robust optimization in hyperbolic space. GitHub Repository, 2023.

[203] Tao Yu, Toni J.B. Liu, Albert Tseng, and Christopher De Sa. Shadow cones: Unveiling partial orders in hyperbolic space. *arXiv preprint*, 2023.

[204] Robert Yuncken. Regular tessellations of the hyperbolic plane by fundamental domains of a fuchsian group. *Moscow Mathematical Journal*, 3, 03 2011.

[205] Pedram Zamirai, Jian Zhang, Christopher R Aberger, and Christopher De Sa. Revisiting bfloat16 training. *arXiv preprint arXiv:2010.06192*, 2020.

[206] Steve Zelditch. *Eigenfunctions of the Laplacian on a Riemannian manifold*, volume 125. American Mathematical Soc., 2017.

[207] Dongxu Zhang, Michael Boratko, Cameron Musco, and Andrew McCallum. Modeling transitivity and cyclicity in directed graphs via binary code box embeddings. *Advances in Neural Information Processing Systems*, 35:10587–10599, 2022.

[208] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[209] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. Qpytorch: A low-precision arithmetic simulation framework. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 10–13. IEEE, 2019.

[210] Yiding Zhang, Xiao Wang, Chuan Shi, Xunqiang Jiang, and Yanfang Fanny Ye. Hyperbolic graph attention network. *IEEE Transactions on Big Data*, 2021.

[211] Yiding Zhang, Xiao Wang, Chuan Shi, Nian Liu, and Guojie Song. Lorentzian graph convolutional networks. In *Proceedings of the Web Conference 2021*, pages 1249–1261, 2021.

[212] Lin Zheng, Jianbo Yuan, Chong Wang, and Lingpeng Kong. Efficient attention via control variates. *arXiv preprint arXiv:2302.04542*, 2023.

[213] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch, 2021.

[214] Shichao Zhu, Shirui Pan, Chuan Zhou, Jia Wu, Yanan Cao, and Bin Wang. Graph geometry interaction learning. *Advances in Neural Information Processing Systems*, 33:7548–7558, 2020.