Statistics Innovation

ARTICLE

Open Access

https://doi.org/10.48130/stati-0025-0005

Statistics Innovation **2025**, 2: e005

Reinforcement learning in hyperbolic space for multi-step reasoning

Tao Xu[1], Dung-Yang Lee[2] and Momiao Xiong[2,3*]

[1] *Department of Immunology and Molecular Microbiology, School of Medicine, Texas Tech University Health Science Center, Lubbock, TX 79430, USA*
[2] *Department of Biostatistics and Data Science, School of Public Health, The University of Texas Health Science Center at Houston, Houston, TX 77030, USA*
[3] *Society of Artificial Intelligence Research, Houston, TX 77030, USA*
* Corresponding author, E-mail: momiao.xiong@gmail.com, momiao.xiong@uth.tmc.edu

Abstract

Multi-step reasoning is a fundamental challenge in artificial intelligence, with applications ranging from mathematical problem-solving to decision-making in dynamic environments. Reinforcement learning (RL) has shown promise in enabling agents to perform multi-step reasoning by optimizing long-term rewards. However, conventional RL methods struggle with complex reasoning tasks due to issues such as credit assignment, high-dimensional state representations, and stability concerns. Recent advancements in transformer architectures and hyperbolic geometry have provided novel solutions to these challenges. This paper introduces a new framework that integrates hyperbolic transformers into RL for multi-step reasoning. The proposed approach leverages hyperbolic embeddings to model hierarchical structures effectively. Theoretical insights, algorithmic details, and experimental results are presented, including Frontier Math and nonlinear optimal control problems. Compared to RL with vanilla transformer, the hyperbolic RL largely improves accuracy by 32%–44% on FrontierMath benchmark, 43%–45% on nonlinear optimal control benchmark, while achieving impressive reduction in computational time by 16%–32% on FrontierMath benchmark, 16%–17% on nonlinear optimal control benchmark. This work demonstrates the potential of hyperbolic transformers in reinforcement learning, particularly for multi-step reasoning tasks that involve hierarchical structures.

Citation: Xu T, Lee D-Y, Xiong M. 2025. Reinforcement learning in hyperbolic space for multi-step reasoning. *Statistics Innovation* 2: e005 https://doi.org/10.48130/stati-0025-0005

Introduction

Despite great progress in artificial intelligence, e.g., OpenAI's o1 and o3, DeepSeek-V3, and Alibaba's QwQ, solving reasoning tasks – particularly multi-step complex reasoning problems – remains a fundamental challenge due to high costs, proprietary nature, and complex architectures[1]. Multi-step reasoning refers to the ability of AI systems to make logical connections between different pieces of context or different sources of information. Multi-step reasoning moves toward more human-like understanding and decision-making for AI systems. Its ability to interact with context, combine different information sources, and make logical connections is essential for artificial general intelligence (AGI)[2]. AGI is the new frontier in artificial intelligence, where the aim is to create human-like cognitive abilities. Recently, OpenAI announced the Deep Research AI system, which combines advanced multi-step reasoning capabilities with extensive internet search and synthesis functions, marking a significant milestone on the path to AGI[3].

Reinforcement learning (RL) that incorporates multi-step reasoning is widely seen as one of the promising components on the long road toward AGI, though it is not a silver bullet on its own. However, high costs, proprietary nature, scalability, integration of reasoning mechanisms, and complex architectures present great challenges[4−9].

RL is a Markov decision process (MDP) and provides a mathematical formalism for sequential decision-making[10,11]. It is observed that RL can acquire intelligent behaviors automatically. Decision actions are selected by the agent's optimal policy to maximize the expected cumulative reward. Policy and reward are often approximated by neural networks. However, standard neural network architectures cannot efficiently deal with long-standing problems in RL, including partial observability[12], high-dimensional state and action spaces[13]. Recently, the transformer architecture demonstrated its superior performance. The essential idea behind the transformer architecture is to use a self-attention mechanism to capture long-range relationships within the data. The remarkable feature of the transformer is its excellent scalability. Transformers can be used to learn representations, model transition functions, learn reward functions, and learn policies[14,15].

RL with transformers can perform single reasoning very well. It is also reported that either transformer or RL can perform multi-step reasoning[16,17]. However, very few papers about using RL incorporating transformers for multi-step reasoning have been published.

Reasoning problems such as mathematical operations, coding, and logical reasoning involve a chain of thought, a tree of thought, and a graph of thought. Reason data are tree-like structured data. Embedding tree-like data, from hierarchies to taxonomies, is a well-defined problem for representing graph knowledge. Hyperbolic geometry provides a natural solution for embedding tree-like and hierarchical data, with demonstrated superior performance over Euclidean embeddings[18]. Hypformer, developed last year, is a novel and complete hyperbolic transformer[19]. It includes well-defined modules in hyperbolic space, such as linear transformation layers, LayerNorm layers, activation functions, dropout operations, and addresses the issue of quadratic time complexity of the existing hyperbolic self-attention module. Despite the impressive performance of various hyperbolic transformers, the papers integrating RL with hyperbolic transformers for multi-step reasoning are very limited.

The purpose of this paper is to introduce a novel multi-step reasoning large language model (LLM) with RL incorporating a complete hyperbolic transformer into it. The proposed hyperbolic transformer is designed to encode diverse sequences, such as entities, agents, and stacks of historical information, and to serve as an expressive predictor for the dynamics model. On the other hand, the developed hyperbolic transformers will integrate all subroutines into RL and act as a sequential decision-maker. To facilitate the development of the complete hyperbolic transformer-based RL for multi-step reasoning, applications are outlined in robotics, medicine, multi-step reasoning language modeling, combinatorial optimization, environmental sciences, and hyperparameter optimization. Limitations and challenges for future research are also addressed. This work is intended to stimulate discussions on hyperbolic transformer-based RL for multi-step reasoning, inspire further research, and facilitate the development of RL approaches for real-world applications.

Methods

Model framework

Many complex reasoning tasks – from robotic navigation and manipulation to mathematical problem solving and language reasoning – require planning over several sequential decisions. In such settings, the 'credit assignment problem' (i.e., what past actions contributed to current rewards) is acute: a reward might be received only at the end of a long sequence, yet every step in the chain of decisions may be crucial. Standard RL algorithms (such as one-step temporal-difference (TD) methods) can struggle because they update only with information from the immediate next reward. Multi-step reasoning in RL aims to address this by propagating rewards over several steps using the observed value function, thereby providing better learning signals for tasks with delayed or sparse rewards[17].

Major features of multi-step reasoning RL are:

• **Delayed reward propagation:** When rewards are delayed, multi-step (or *n*-step) methods allow the agent to use a longer 'lookahead' window so that early decisions are more directly informed by later outcomes.

• **Hierarchical structure:** Many real-world tasks naturally decompose into sub-tasks. Hierarchical RL (HRL) approaches – using options or subpolicies – can help hierarchically organize multi-step reasoning by assigning higher-level 'goals' that guide lower-level behavior.

• **Improved sample efficiency:** By aggregating several reward signals before updating the value estimate, multi-step methods help in situations where rewards are sparse.

Methods for multi-step reasoning in RL

There are several complementary approaches to enable multi-step reasoning in RL. They can be used for both on- and off-policy learning. All these methods are often described in the simple one-step case, but they can also be extended across multiple time steps.

Merging transformer reasoning with multi-step RL

Integrating transformer architectures with RL is an emerging field[10]. The integration of transformer architecture with RL represents a significant advancement in artificial intelligence. An extended framework is introduced that 'plugs in' the reasoning mechanisms of transformer models into an RL formulation for multi-step reasoning. In this framework, the transformer functions not only as a powerful sequence generator (the policy) but also as a mechanism to provide rich, intermediate representations (the 'chain-of-thought') that can serve as states in an MDP. These representations, combined with the transformer's inherent self-attention and contextual integration, can then be leveraged to guide multi-step RL updates.

Representation learning high-dimensional data in RL

Good high-dimensional data representations are critical for efficient RL. They can enhance performance, convergence speed, and policy stability. The data in large reason models often include mathematical and logical symbol data, coding data, and science such as physics, chemistry, and biology data, sensory signals, and vision image data. Of course, data also includes natural language data. In general, data is represented as a sequence of data. The goal of representation learning is to map high-dimensional data to a compact representation for learning RL.

Transformer architecture

A transformer consists of the following main components (a list of symbols is summarized in Supplementary Data 1):

Input token embedding and positional encoding:

Each input token $x_t \in \mathbb{R}^{In}$ is mapped to a continuous vector using an embedding matrix *AND* and then combined with a positional encoding p_t. The embedding vector of the token x_t is given by

$$AND(x_t) = Ex_t \in \mathbb{R}^d$$

A fixed or learned positional encoding $p_t \in \mathbb{R}^d$ is added:

$$X_t = Ex_t + p_t$$

Attention block:

Consider *n* input token. Define input embedding matrix:

$$X = \begin{bmatrix} X_1^T \\ \vdots \\ X_n^T \end{bmatrix} \in \mathbb{R}^{n\times d}$$

Define query, key, and value matrices: $Q \in \mathbb{R}^{n\times d_k}, K \in \mathbb{R}^{n\times d_k}$ and $V \in \mathbb{R}^{n\times d_v}$:

$$Q = XW_Q\ ,\ \ K = XW_k\ ,\ \ V = XW_v$$

where, $W_Q \in \mathbb{R}^{d\times d_k}, W_k \in \mathbb{R}^{d\times d_k}$, and $W_v \in \mathbb{R}^{d\times d_v}$ are learnable weight matrices for query, key, and value.

Attention scores are defined as:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Multi-head attention combines multiple attention heads:

$$\text{MultiHead}(X) = \text{Concat}(\text{Head}_1,\ldots,\text{Head}_h)\,W_O \in \mathbb{R}^{n\times d}$$

where, *h* is the number of heads,

$$\text{Head}_i = \text{Attention}\left(XW_Q^i, XW_k^i, XW_v^i\right), W_O \in \mathbb{R}^{hd_v\times d}$$

LayerNorm:

Layer normalization (usually called LayerNorm) is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training.

Define:

$$\mu_i = \frac{1}{n}\sum_{j=1}^{n} x_{ji}, \sigma_i = \sqrt{\frac{\sum_{j=1}^{n}\left(x_{ji}-\mu_i\right)^2}{n}}, x_i = \begin{bmatrix} x_{1i} \\ \vdots \\ x_{ni} \end{bmatrix} \text{ and } \hat{x}_i = \frac{x_i-\mu_i}{\sigma_i}$$

Then, LayerNorm is defined as:

$$\text{LayerNorm}(x_i) = \gamma\hat{x}_i + \beta$$

Feed-forward network (FFN):

The feed-forward layer in a transformer architecture is one of the main components and is present in each of the encoder and decoder blocks. Here is a high-level description of its architecture and function.

(1) Position-wise FFNs: Each layer in the encoder and decoder contains a fully connected FFN, which is applied to each position separately and identically.

(2) Linear transformations: The FFN consists of two linear transformations (i.e., fully connected layers) with a Rectified Linear Unit (ReLU) activation in between. This can be described by the equations:

$\mathrm{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2$, where $W_1 \in \mathbb{R}^{d_{model} \times d_f}, W_2 \in \mathbb{R}^{d_f \times d_{model}}$ are weight matrices, and $b_1 \in \mathbb{R}^{d_f}, b_2 \in \mathbb{R}^{d_{model}}$, are bias vectors.

Now describing how to complete attention blocks. The input representations are processed through *L* layers. Each layer *l* includes multi-head self-attention and an FFN with residual connections and layer normalization. A simplified update at layer *l* is given by:

$$\tilde{Z}^{(l)} = \mathrm{LayerNorm}\left(Z^{(l-1)} + \mathrm{MultiHead}\left(Z^{(l-1)}\right)\right)$$

$$Z^{(l)} = \tilde{Z}^{(l)} + FFN\left(\tilde{Z}^{(l)}\right), l = 1, 2, \ldots, L, \text{ where } Z^{(0)} = X$$

Final state representation:

The output of the final layer is then used as the state: $s_t = Z^{(L)}$. In the RL framework, the sequence $\{s_1, s_2, ..., s_n\}$ represents the agent's trajectory (its chain-of-thought), where each s_t encapsulates the contextual information accumulated up to time *t*. These states can then be used by the RL algorithm (e.g., to compute Q-values or to determine the next action).

Converting Euclidean transformer to a hyperbolic transformer for general representation

Hyperbolic geometry has demonstrated that it can provide a powerful tool in modeling complex structured data, particularly reasoning data with underlying treelike and hierarchical structures[14,19–22]. Tree-like and hierarchical structures underlie human cognitive processes, making hyperbolic geometry modelling an intuitive approach to data representation. Despite the growing interest in hyperbolic representation, exploration of the transformer – a mathematical foundation in artificial intelligence model – within hyperbolic space remains limited. This section introduces an efficient and complete hyperbolic transformer involving hyperbolic transformation with curvatures and hyperbolic readjustment and refinement with curvatures. The key idea of the hyperbolic transformer is to take the Euclidean output $Z_t^{(L)}$ and map it into hyperbolic space. For simplicity, the Poincaré ball model of hyperbolic space with curvature −1 is assumed. The following steps detail this conversion and the necessary modifications to core operations.

Notation and setup

Before introducing procedures for hyperbolic transformers, the notations and basic setup are presented. The Poincaré ball $\mathbb{D}^d = \left\{x \in \mathbb{R}^d : ||x|| < 1\right\}$ is adopted as the hyperbolic geometry model, with curvature −1. For clarity, the focus on curvature −1; if curvature −*c* is desired, all operations can be scaled accordingly.

Key maps:

Exponential map $exp_0(\cdot)$ from the tangent space at the origin (Euclidean) to $\mathbb{D}^d$.

Logarithmic map $\log_0(\cdot)$ from $\mathbb{D}^d$ to the tangent space at the origin (Euclidean).

Möbius addition $\oplus$ in $\mathbb{D}^d$ and **Möbius scalar multiplication** $\lambda \otimes x$.

Also relied on are$<\cdot,>$ to denote the standard Euclidean inner product for intermediate calculations in the tangent space.

Exponential map to the Poincaré ball

The first step is to map a Euclidean vector to the Poincaré ball. Given a Euclidean vector$v \in \mathbb{R}^d$ (e.g., $Z_t^{(L)}$ or an intermediate representation), the exponential map at the origin projects *v* to a point in the Poincaré ball $D^d = \left\{x \in \mathbb{R}^d, ||x|| < 1\right\}$. This map, denoted by $\varnothing(v)$, is

$$\varnothing(v) = Exp_0(v) = \tanh(||v||)\frac{v}{||v||}, \text{ for } v \neq 0, \varnothing(0) = 0 \quad (1)$$

Therefore, the hyperbolic state is defined as

$$\tilde{s}_t = \varnothing\left(Z_t^{(L)}\right) \in D^d \quad (2)$$

Hyperbolic versions of basic operations

Once in hyperbolic space, standard operations are replaced by their Möbius analogues.

Möbius addition:

Similar to addition in Euclidean space, for two points $x, y \in D^d$, Möbius addition is defined as:

$$x \oplus y = \frac{\left(1 + 2 < x, y > + ||y||^2\right)x + \left(1 - ||x||^2\right)y}{1 + 2 < x, y > + ||x||^2||y||^2} \quad (3)$$

Möbius scalar multiplication:

Now the Möbius scalar multiplication in hyperbolic space is introduced.

For a scalar $\lambda \in \mathbb{R}$ and a point $x \in D^d$,

$$\lambda \otimes \mathrm{x} = \tanh\left(\lambda \tanh^{-1}(||x||)\right)\frac{x}{||x||}, \text{ for } x \neq 0, \lambda \otimes 0 = 0 \quad (4)$$

Hyperbolic linear transformation:

A hyperbolic linear transformation is defined as follows. It is typically implemented by mapping the hyperbolic point back to the tangent space at the origin via the logarithmic map, applying a Euclidean linear transformation, then mapping back.

Logarithmic map

$$\log_0(x) = \tanh^{-1}(||x||)\frac{x}{||x||} \quad (5)$$

Transformation: $y = \mathrm{W}\log_0(x) + b$.

Exponential map back: $f(x) = Exp_0(||y||) = \tanh(||y||)\frac{y}{||y||}$.

Combining the above three steps, the hyperbolic linear transformation is obtained:

$$f(x) = Exp_0(\mathrm{W}\log_0(x) + b)$$

Converting the input embedding

The procedure begins with an input Euclidean embedding and is then extended to an input hyperbolic embedding. In a standard transformer, an input token x_t (an integer index in {1, ..., *V*} is mapped to a Euclidean embedding vector $e_t \in \mathbb{R}^d$. For a hyperbolic transformer, the final representation is required to lie in $\mathbb{D}^d$. One popular approach includes the following steps.

Euclidean embedding (lookup): $e_t = x_t E, e_t \in \mathbb{R}^d$, where *E* is a learnable embedding matrix of size $V \times d$.

Positional encoding (optional): Let $p_t \in \mathbb{R}^d$. Combine them in Euclidean space: $u_t = e_t + p_t$.

Exponential map: To get a hyperbolic point $\tilde{u}_t \in \mathbb{D}^d$, apply the exponential map at the origin: $\tilde{u}_t = exp_0(u_t) = \tanh(||u_t||)\frac{u_t}{||u_t||}$, for $u_t \neq 0$. If $u_t = 0$ then $exp_0(u_t) = 0$.

Thus, combining the above three steps leads to each token's hyperbolic embedding:

$$\tilde{u}_t = exp_0(e_t + p_t)$$

Converting LayerNorm

A standard LayerNorm in Euclidean space for a vector $z \in \mathbb{R}^d$ is

$$\mathrm{LayerNorm}(z) = \frac{z - \mu}{\sigma} \odot \gamma + \beta \quad (6)$$

where, $\mu = \frac{1}{d}\sum_{i=1}^{d} z_i, \sigma = \sqrt{\frac{\sum_{i=1}^{d}(z_i - \mu)^2}{d}}$, γ, β are learnable parameters.

In hyperbolic space, the lack of a straightforward linear structure makes standard layer normalization (i.e., subtracting the mean and dividing by the standard deviation in a component-wise manner) valid. Therefore, in hyperbolic space, this cannot be directly done as

$\frac{z-\mu}{\sigma}$. A common approach is to first map from hyperbolic space to the tangent space at the origin using $\log_0$. Then, perform standard Euclidean layer normalization in that tangent space. Finally, map back to hyperbolic space via exp_0.

Therefore, let $\tilde{z} = \tilde{u}_t$, Hyperbolic LayerNorm is defined as:

$$\hat{z} = \text{HLayerNorm}(\tilde{z}) = exp_0\left(\frac{y-\mu_y}{\sigma_y}\odot\gamma+\beta\right)$$

where, $y = \log_0(\tilde{z})$, μ_y, σ_y.

Converting multi-head attention

The conversion of multi-head attention from Euclidean space to hyperbolic space is now considered. In hyperbolic space, multi-head attention consists of five steps:

Map tokens to tangent space:

In order to perform attention steps in Euclidean space, $\log_0$ is first used to map each hyperbolic token $\tilde{z}_t^{(l-1)} \in \mathbb{D}^d$ to Euclidean space: $z_t^{(l-1)} = \log_0\left(\tilde{z}_t^{(l-1)}\right) \in \mathbb{R}^d$.

Compute queries, keys, and values in tangent space:

$$Q_i = W_i^Q z_t^{(l-1)} \in \mathbb{R}^{d_v}, K_i = W_i^K z_t^{(l-1)} \in \mathbb{R}^{d_v}, V_i = W_i^V z_t^{(l-1)} \in \mathbb{R}^{d_v}$$

where, the weight matrices $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_v \times d}$ are learnable.

Compute attention scores:

$$A_{ij} = \frac{<Q_i, K_j>}{\sqrt{d}} \to \text{softmax}\left(A_{ij}\right) = \frac{exp\left(A_{ij}\right)}{\sum_{j'} exp\left(A_{ij'}\right)}, \qquad i = 1,\ldots,h$$

This step is identical to the Euclidean dot-product attention, except that it is performed in the tangent space.

Aggregate values:

The standard attention output in tangent space is $head_i = \text{Attn}_i(Q_i, K_i, V_i) = \sum_j \alpha_{ij} V_j \in \mathbb{R}^{d_v}, i = 1,\ldots,h$, where h is the number of heads.

$$A = \text{MultiHead}\left(z_t^{(l-1)}\right) = W_o \begin{bmatrix} head_1 \\ \vdots \\ head_h \end{bmatrix} \in \mathbb{R}^d, \text{ where } W_O \in \mathbb{R}^{d\times hd_v}$$

Map back to hyperbolic space:

Finally, there's a need to map A back to hyperbolic space:

$$(mz)_t^{(l-1)} = \text{HMultiHeadAtten}\left(\tilde{z}_t^{(l-1)}\right) = exp_0(A) \in \mathbb{D}^d$$

Converting the FFN

Recall that a standard feed-forward sub-layer in Euclidean space has the form:

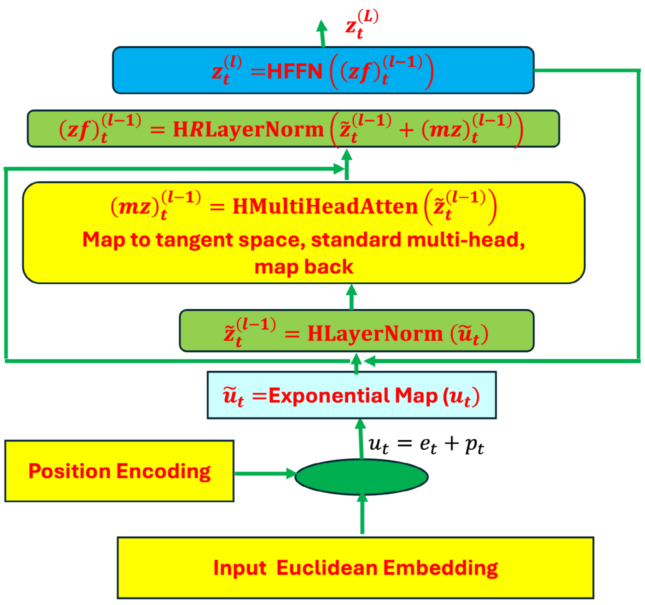
$\text{FFN}(z) = W_2\sigma(W_1 z + b_1) + b_2$, where σ is typically ReLU or GELU.

The hyperbolic FNN, which consists of three steps, which are combined to obtain the hyperbolic feed-forward layer:

$$z_t^{(l)} = \text{HFFN}\left((zf)_t^{(l-1)}\right) = exp_0\left(W_2\sigma\left(W_1\log_0\left((zf)_t^{(l-1)}\right)+b_1\right)+b_2\right)$$

Hyperbolic residual & LayerNorm

$(zf)_t^{(l-1)}$=HL$\left(\tilde{z}_t^{(l-1)}\right)$ =Möbius add $\left(\tilde{z}_t^{(l-1)}\right)\oplus\text{HMultiHeadAtten}\left(\tilde{z}_t^{(l-1)}\right)$, (or do $\log_0$-based approach for a more stable approach):



$z_t^{(L)}$
$z_t^{(l)}$ =HFFN $\left((zf)_t^{(l-1)}\right)$
$(zf)_t^{(l-1)}$ = HRLayerNorm $\left(\tilde{z}_t^{(l-1)} + (mz)_t^{(l-1)}\right)$
$(mz)_t^{(l-1)}$ = HMultiHeadAtten $\left(\tilde{z}_t^{(l-1)}\right)$
Map to tangent space, standard multi-head, map back
$\tilde{z}_t^{(l-1)}$ = HLayerNorm $(\tilde{u}_t)$
$\tilde{u}_t$ =Exponential Map (u_t)
$u_t = e_t + p_t$
Position Encoding
Input Euclidean Embedding

Fig. 1 Architecture of hyperbolic transformers. All steps: input Euclidean embedding, position encoding, hyperbolic layer normalization, hyperbolic multi-head attention, and hyperbolic feed-forward neural networks in the hyperbolic transformer encoder block.

First map to tangent space, then add them in the Euclidean space, finally map it back to hyperbolic space.

$$\text{HMlog}\left(\tilde{z}_t^{(l-1)}\right) = \log_0\left(\text{HMultiHeadAtten}\left(\tilde{z}_t^{(l-1)}\right)\right) + \tilde{z}_t^{(l-1)}$$

$$\text{HMexp}\left(\tilde{z}_t^{(l-1)}\right) = exp_0\left(\text{HMlog}\left(\tilde{z}_t^{(l-1)}\right)\right)$$

Apply HLayerNorm:

$$(zf)_t^{(l-1)} = \text{HL}\left(\text{HMexp}\left(\tilde{z}_t^{(l-1)}\right)\right) = \text{HLayerNorm}\left(\text{HMexp}\left(\tilde{z}_t^{(l-1)}\right)\right)$$

All steps in the hyperbolic transformer encoder block are summarized in Fig. 1.

Hyperbolic (Poincaré ball) transformer for transition function

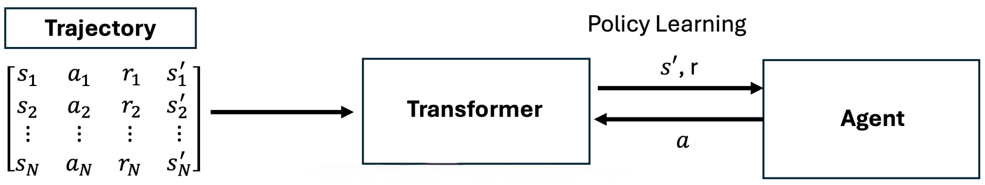
A hyperbolic transformer can be employed to learn the dynamics of the environment by modeling the transition function in RL, which describes how the environment transitions from the current state *s* to the next state *s'* and issues rewards *r* in response to the actions *a* taken by the agent (Fig. 2)[10].

Hyperbolic transformer architecture

Hyperbolic transformer (Fig. 1) consists of hyperbolic input embedding, hyperbolic transformer for transition function, multi-head attention, and layernorm in hyperbolic, hyperbolic residual connection, and hyperbolic feed-forward neural networks.

Input Embedding (Hyperbolic)

Hyperbolic input embedding consists of two major steps. The first step is to embed state and action variables in Euclidean space. Then,



Trajectory
$\begin{bmatrix} s_1 & a_1 & r_1 & s_1' \\ s_2 & a_2 & r_2 & s_2' \\ \vdots & \vdots & \vdots & \vdots \\ s_N & a_N & r_N & s_N' \end{bmatrix}$
Policy Learning
Transformer
s', r
a
Agent

Fig. 2 Transition function learning. A hyperbolic transformer is employed to learn the dynamics of the environment by modeling the transition function in RL, which describes how the environment transitions from the current state *s* to the next state *s'* and issues rewards *r* in response to the actions taken by the agent.

convert them to hyperbolic space. Total input data, denoted by $\left(s_i, a_i, r_i, s_i'\right), i = 1, \ldots, N$, are shown in Fig. 2. Firstly, introduce input embedding for *a*, state *s* and an action *a*:

State embedding

$e_s = E_s(s) = W_s s + b_s \in \mathbb{R}^d$, where the learnable embedding matrix or function E_s maps the raw state *s* to a *d*-dimensional Euclidean vector. If *s* is a sequence, each element $s_i, i = 1, \ldots, N$ is embedded as: $e_i = E_s(s_i) \in \mathbb{R}^d$.

Action embedding

Like the state, an embedding for the action is defined:

$e_a = E_a(a) = W_a a + b_a \in \mathbb{R}^d$, where W_a and b_a are learnable parameters. For discrete actions, this is typically a simple table lookup.

Positional encodings (optional)

In some cases, (*s*, *a*) is a short sequence [*s*, *a*] which can be assigned with positional vectors p_s and p_a. Adding positional encodings to the state and action embedding yields

$$u_s = e_s + p_s, u_a = e_a + p_a$$

Mapping to the Poincaré ball

At the second step, the Euclidean embedding is mapped at the first step to the hyperbolic space. At the origin, the exponential map is used to map the embeddings in the Euclidean space to the Poincaré ball:

$$\tilde{u}_s = exp_0(u_s), \tilde{u}_a = exp_0(u_a)$$

Thus, $\tilde{u}_s, \tilde{u}_a \in \mathbb{D}^d$. A short sequence is formed $\tilde{u}^{(0)} = [\tilde{u}_s, \tilde{u}_a]$ of length 2 in the Poincaré ball. For the total input data (*N* episodes), is defined as

$$\tilde{X} = \begin{bmatrix} \tilde{u}_{s_1} & \tilde{u}_{a_1} \\ \vdots & \vdots \\ \tilde{u}_{s_N} & \tilde{u}_{a_N} \end{bmatrix} \in \mathbb{D}^{N \times (2d)}$$

Full architecture: hyperbolic transformer for transition function

A short sequence $\tilde{X}$ is now built from the state and action embeddings. The network has *L* layers, each with hyperbolic multi-head attention, hyperbolic residual, layer norm, and hyperbolic FFN. Converting Euclidean transformer into hyperbolic transformer for transition function follows the same procedures as that in Section on 'Converting Euclidean transformer to a hyperbolic transformer for general representation'.

The full structure of the hyperbolic transformer for the transition function is summarized as follows. A sort of sequence from the state and action embeddings is $\tilde{X} = [\tilde{u}_s, \tilde{u}_a]$. The network has *L* layers, each layer has the following components: hyperbolic multi-head attention, hyperbolic residual, layer norm, and hyperbolic FFN as discussed in Section on 'Converting Euclidean transformer to a hyperbolic transformerfor general representation'. Denote the representation at layer *l* by $\tilde{x}^{(l)}$. After *L* layers, the output is $\tilde{X}^{(L)} = \left[\tilde{X}_s^{(L)}, \tilde{X}_a^{(L)}\right]$. The final vectors are pooled or concatenated into one representation $\tilde{h} \in \mathbb{D}^{2d}$.

Finally, the output $\tilde{h}$ are used to predict the next state and reward. Computing transition function $p(s', r|s, a)$, must predict $\hat{s}'$ and $\hat{r}$, which typically are in Euclidean space. Therefore, firstly mapping the final FFN output to the tangent space, then estimating their linear predictions in the Euclidean space: $h = \log_0\left(\tilde{h}\right) \in \mathbb{R}^{2d}$ and

$$\hat{s}' = W_s h + b_s, \hat{r} = W_r h + b_r.$$

Hence, the final function is $\left(\hat{s}', \hat{r}\right) = f(s, a)$.

Group relative policy optimization (GRPO) incorporating hyperbolic transformer

In this section, the following will be investigated:

Multi-head latent attention (MLA): A specialized attention mechanism that processes latent variables or additional context in multi-head format.

DeepSeekMoE: A mixture of experts (MoE) layer integrated into transformers to handle specialized sub-tasks or to route tokens to different experts.

Hyperbolic geometry: The Poincaré ball model $\mathbb{D}^d$ with exponential/logarithmic maps, Möbius addition, etc.

GRPO: A variant of policy gradient that uses group-based policy updates.

Firstly, how to adapt MLA and DeepSeekMoE to hyperbolic geometry is outlined, and then how to embed the resulting hyperbolic transformer as a policy in GRPO.

Background: MLA and DeepSeekMoE in Euclidean space

MLA

The MLA and MoE are briefly introduced here. Interested readers refer to references cited here[6,23]. MLA utilizes low-rank matrices in the key-value layers and enables the caching of compressed latent key value (KV) states to address the communication bottlenecks in LLM[6,23]. This design significantly reduces the KV cache size compared to traditional multi-head attention, thus accelerating inference. MLA also incorporates an up-projection matrix to enhance expressiveness. MLA is developed to speed up inference speed in autoregressive text generation.

Let the input sequence be $X \in \mathbb{R}^{T \times d}$, where *T* is the sequence length and *d* is the embedding dimension, n_h be the number of heads, and d_h be the dimension for each head. Let *t* denote the time step and $h_t \in \mathbb{R}^d$ be the attention input of the *t*th token at an attention layer.

Standard multi-head attention

Project h_t to queries, keys, and values.

Let $q_t, k_t, v_t \in \mathbb{R}^{n_h \times d_h}$ be query, key, and value vectors, and W^Q, W^K, W^V be three linear mapping matrices for projecting h_t to queries, keys, and values. Standard MHA are:

$$q_t = W^Q h_t \tag{7}$$

$$k_t = W^K h_t \tag{8}$$

$$v_t = W^V h_t \tag{9}$$

Then, the vectors q_t, k_t, v_t are split into n_h heads in the MHA computations:

$$q_t = \left[q_{t,1}; \ldots; q_{t,n_h}\right] \tag{10}$$

$$k_t = \left[k_{t,1}; \ldots; k_{t,n_h}\right] \tag{11}$$

$$v_t = \left[v_{t,1}; \ldots; v_{t,n_h}\right] \tag{12}$$

$$o_{t,i} = \sum_{j=1}^{n_h} \text{softmax}_j \left(\frac{q_{t,i}^T k_{t,j}}{\sqrt{d_h}}\right) v_{t,j} \in \mathbb{R}^{d_h} \tag{13}$$

$$u_t = W^O \left[o_{t,1}; \ldots; o_{t,n_h}\right] \tag{14}$$

where $q_{t,i}, k_{t,i}, v_{t,i} \in \mathbb{R}^{d_h}$ are the query, key, and value of the *i*th attention head, respectively, $W^O \in \mathbb{R}^{d \times n_h d_h}$ are the output projection matrix.

In computation, during inference, all keys and values need to be cached to accelerate inference, so MHA must cache $2n_h d_h L$ elements for each token.

Low-rank key-value joint compression

The low-rank joint compression for keys and values is given by

$$c_t^{KV} = W^{DKV} h_t \in \mathbb{R}^{d_c}, \tag{15}$$

$$k_t^C = W^{UK} c_t^{KV} \in \mathbb{R}^{n_h d_h} \tag{16}$$

$$v_t^C = W^{UV} c_t^{KV} \in \mathbb{R}^{n_h d_h} \tag{17}$$

where, C_t^{KV} is the compressed latent vector for keys and values; $d_c (\ll n_h d_h)$ denotes the *KV* compression dimension; W^{DKV} is the down-projection matrix; and $W^{UK}, W^{UV} \in \mathbb{R}^{n_h d_h \times d_c}$ are the up-projection matrices for keys and values, respectively. During inference, MLA only needs to cache C_t^{KV}, therefore, the elements of the *KV* cache are reduced from $2n_h d_h L$ to $d_c L$. Furthermore, to reduce the activation memory during training, low-rank compression for the queries are also made:

$$c_t^Q = W^{DQ} h_t \tag{18}$$

$$q_t^c = W^{UQ} c_t^Q \tag{19}$$

where, $c_t^Q \in \mathbb{R}^{d_c'}$ is the compressed latent vector for queries; $d_c' (\ll n_h d_h)$ denotes the querycompression dimension; and $W^{DQ} \in \mathbb{R}^{d_c' \times d}$, $W^{UQ} \in \mathbb{R}^{n_h d_h \times d_c'}$ are the down-projection and up-projection matrices for queries, respectively.

Rotary positional embeddings (RoPE)[24] encodes the absolute position with a rotation matrix and meanwhile incorporates the explicit relative position dependency in self-attention formulation. RoPE enhances the performance of the transformer. However, RoPE is incompatible with low-rank *KV* compression. To overcome this limitation, DeepSeek developed methods to decouple the RoPE as follows. Let $q_{t,i}^R \in \mathbb{R}^{d_h^R}$ and k_t^R be additional multi-head queries and a shared key, respectively, to carry RoPE, where d_h^R denotes the per-head dimension of the decoupled queries and key. Let $W^{QR} \in \mathbb{R}^{n_h d_h^R \times d_c'}$ and $W^{KR} \in \mathbb{R}^{n_h d_h^R \times d}$ be matrices to produce the decouple queries and key, respectively; RoPE(·) denotes the operation that applies RoPE matrices; and [· ; ·] denotes the concatenation operation. The algorithm of MLA with the decoupled RoPE strategy is given as follows:

$$\left[q_{t,1}^R; \ldots; q_{t,n_h}^R\right] = RoPE\left(W^{QR} c_t^Q\right) \tag{20}$$

$$k_t^R = RoPE\left(W^{KR} h_t\right) \tag{21}$$

$$q_{t,i} = \left[q_{t,i}^C; q_{t,i}^R\right] \tag{22}$$

$$k_{t,i} = \left[k_{t,i}^C; k_t^R\right] \tag{23}$$

$$o_{t,i} = \sum_{j=1}^{n_h} \mathrm{softmax_j}\left(\frac{q_{t,i}^T k_{t,j}}{\sqrt{d_h + d_h^R}}\right) v_{t,j}^C \tag{24}$$

$$u_t = W^O\left(o_{t,1}; \ldots; o_{t,n_h}\right) \tag{25a}$$

Since during inference, the decoupled key should also be cached, the total number of *KV* cache elements are $\left(d_c + d_h^R\right) L$.

Integrates mixture of experts (MoE) with MLA

A novel language model architecture that integrates MoE with an MLA mechanism and RMSNorm to achieve unprecedented scalability and inference efficiency[25,26]. The MoE has two main components, namely experts and the router:

• Experts – each FNN layer now has a set of 'experts' which are subsets in the FNN.

• Router or gate network – the router determines how input tokens' information passes experts.

It replaces FFN in the transformer. The architecture of MoE is shown in Fig. 3. The MoE only activates a subset of experts at a given time. Now the DeepSeek MoE algorithm is introduced.

Basic architecture

Consider the l^{th} layer. Let u_t^l be the FFN input of the t^{th} token, N_s and N_r be the numbers of shared experts and routed experts, respectively, $\mathrm{FFN}_i^{(s)}(\cdot)$ and $\mathrm{FFN}_i^{(r)}(\cdot)$ denote the i^{th} shared expert and the i^{th} routed expert, respectively. The FFN output $h_t^{(l)}$ is given by

$$h_t^l = u_t^l + \sum_{i=1}^{N_s} \mathrm{FFN}_i^{(s)}\left(u_t^l\right) + \sum_{i=1}^{N_r} g_{i,t} \mathrm{FFN}_i^{(r)}\left(u_t^l\right) \tag{25b}$$

$$g_{i,t} = \begin{cases} s_{i,t} & s_{i,t} \in \mathrm{TopK}\left(\left\{s_{j,t} | 1 \le j \le N_r\right\}, k_r\right) \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

$$s_{i,t} = \mathrm{softmax}\left(\left(u_t^l\right)^T e_i\right) \tag{27}$$

where k_r denotes the number of activated routed experts; $g_{i,t}$ is the gate value for the i^{th} expert; $s_{i,t}$ is the token to expert affinity; e_i is the centroid of the i^{th} routed expert in this layer; and Topk(·, *K*) denotes the set comprising *K* highest scores among the affinity scores calculated for the i^{th} token and all routed experts.

Equation (26) implies that the hidden vector output of token *t* at layer *l* always uses all the shared experts (denoted by the first

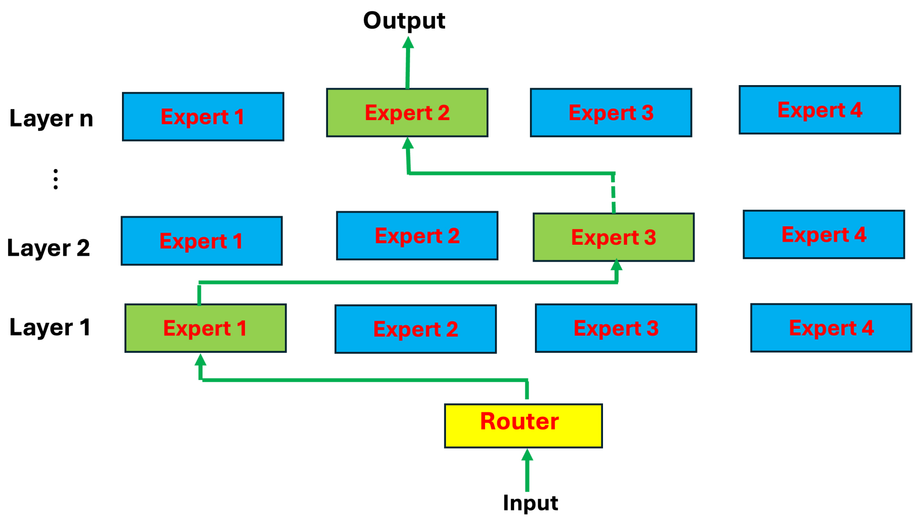


Fig. 3 Outline of MoE. The MoE has two main components, namely experts and the router: experts – each FNN layer now has a set of 'experts' which are subsets in the FNN. Router or gate network – the router determines how input tokens' information passes experts. The MoE replaces FFN in the transformer.

summation in the equation) and always includes the residual (denoted by the first term). The last term, representing the routing experts, includes a gating factor that controls which experts are turned on for any specific token. Using gating factor not only eliminates most of the possible experts (thereby greatly reducing the number of active parameters), but also weights the final output based on how close each chosen routing expert is to the token.

Finally, load imbalance in MoE can also lead to poor performance and poor generalization, as the underloaded experts didn't get enough training tokens to learn meaningful knowledge. To overcome this limitation, the auxiliary loss for load balance is introduced.
Auxiliary loss for load balance

During the training, three kinds of auxiliary losses for controlling expert-level load balance ($\mathcal{L}_{\text{ExpBal}}$), device-level load balance ($\mathcal{L}_{\text{DevBal}}$), and communication balance ($\mathcal{L}_{\text{CommBal}}$), respectively, are introduced.

• Expert-level balance loss

A common strategy to improve load-balancing is to introduce auxiliary loss functions:

$$\mathcal{L}_{\text{ExpBal}} = \alpha_1 \sum_{i=1}^{N_r} f_i p_i$$

$$f_i = \frac{N_r}{K_r T} \sum_{t=1}^{T} 1\,(\text{Token } t \text{ Selects Expert } i)$$

$$p_i = \frac{1}{T} \sum_{t=1}^{T} s_{i,t}$$

where, α_1: a hyper-parameter called expert-level balance factor; f_i: fraction of token rooted to the i^{th} expert; p_i: average probability of selecting the i^{th} expert for the entire input sequence.

• Device-level balance loss

In addition to the expert-level balance loss, the DeepSeek MoE additionally introduces a device-level balance loss to ensure balanced computation across different devices. In the training process, they partition all routed experts into D groups $\{\epsilon_1, \ldots, \epsilon_D\}$, and assign each group on a single device. The device-level balance loss is given as follows:

$$\mathcal{L}_{\text{DevBal}} = \alpha_2 \sum_{i=1}^{D} f_i^{'} p_i^{'}$$

$$f_i^{'} = \frac{1}{|\epsilon_i|} \sum_{j \in \epsilon_i} f_j$$

$$p_i^{'} = \sum_{j \in \epsilon_i} p_j$$

where, α_2 is a hyper-parameter called device-level balance factor.

• Communication balance loss

Finally, a communication balance loss to ensure that the communication of each device is balanced is introduced.

$$\mathcal{L}_{\text{CommBal}} = \alpha_3 \sum_{i=1}^{D} f_i^{'}{}' p_i^{'}{}'$$

$$f_i^{''} = \frac{D}{MT} \sum_{t=1}^{T} 1\,(\text{Token } t \text{ issent to Device } i)$$

$$p_i^{''} = \sum_{j \in \epsilon_i} p_j$$

Converting MLA and DeepSeekMoE to hyperbolic space

The adaptation of each component in MLA and DeepSeekMoE is described so that the internal computations remain consistent with Poincaré ball geometry.

Hyperbolic MLA

In the hyperbolic version of MLA, it yields the following components:

Input

Input includes a set of hyperbolic token embeddings $\tilde{z}_i \in D^d$, and a set of latent embeddings $\tilde{l}_j \in D^d$.

Logarithmic map

The goal of this step is to transform all quantities in the Poincaré ball to the tangent space (Euclidean space). For each $\tilde{z}_i$ or $\tilde{l}_j$, map to tangent space:

$$z_i = \log_0(\tilde{z}_i), \quad l_j = \log_0(\tilde{l}_j)$$

Compute queries/keys/values (in tangent space)

Using Eqs (7)–(9) for Euclidean space to compute

$$Q_i = W^Q z_i \tag{28}$$

$$K_i = W^K z_i \tag{29}$$

$$V_i = W^V z_i \tag{30}$$

and similarly for the latent embeddings (such as the low-rank joint compression) for queries, keys, and values:

$$Q_{l_j} = W_Q^l l_j \tag{31}$$

$$K_{l_j} = W_k^l l_j \tag{32}$$

$$V_{l_j} = W_V^l l_j \tag{33}$$

Attention

Tokens + latent embeddings are combined in the attention (Eqs [15]–[24]). For each i, the corresponding coefficients are computed as:

$$\alpha_{i,x} = \text{softmax}_x \left(\frac{< Q_i, K_x >}{\sqrt{d}} \right) \tag{34}$$

Summing over $x \in \{\text{tokens} + \text{latent compressions, including multi-head}\}$, then, the aggregated value is

$$\hat{v}_i = \sum_x \alpha_{i,x} V_x \tag{35}$$

Map back to the Poincaré ball

$$\tilde{v}_i = exp_0(\hat{v}_i) \tag{36}$$

Hence, MLA in hyperbolic space is basically standard multi-head attention over tokens + latent embeddings in the tangent space, or alternatively performing all computations in the previous sections 'standard multi-head attention' and 'low-rank key-value joint compression', with an exp_0 to map the result back into $\mathbb{D}^d$.

Hyperbolic DeepSeekMoE

Next, the introduction of the mixture-of-experts layer in hyperbolic space:

Router:

For each token $\tilde{z}_i \in \mathbb{D}^d$, this is done:

Logarithmic map:

$$z_i = \log_0(\tilde{z}_i) \in \mathbb{R}^d$$

A Euclidean router function R introduced in the previous section, which outputs gating weights $\alpha_{i,e}$ for each expert e.

Experts:

Each expert is an FFN. In hyperbolic space, transformations are defined:

$$\text{Expert}_i(\tilde{z}) = exp_0\left(W_{e,2}\sigma\left(W_{e,1}\log_0(\tilde{z}) + b_{e,1}\right) + b_{e,2}\right) \tag{37}$$

Combining expert outputs:

Finally, in HFNN, the output is produced for each token:

$$\tilde{y}_i = \oplus_e \alpha_{i,e} \otimes \text{Expert}_e(\tilde{z}_i) \tag{38}$$

In other words, a Möbius weighted combination of the experts' outputs is performed, with weights $\alpha_{i,e}$. Alternatively, one might pick the top-k experts and do a partial sum, as DeepSeek MoE did.

Hyperbolic transformer as a policy

In the previous discussion, a hyperbolic transformer is introduced, which includes:

• **Hyperbolic MLA** blocks (with latent variables if needed).

• **Hyperbolic DeepSeekMoE** blocks (for mixture-of-experts routing).

• **Hyperbolic residual, layernorm, feed-forward** as described in earlier sections.

The hyperbolic transformer can be used to model policy $\pi_\theta(a|s)$. Its architecture is shown in Fig. 4 and outlined as follows.

Inputs:

• **State *s*:**

In an RL setting, the state *s* is provided by the environment. For example, in language-based tasks, this might be a prompt; in robotics, it could be sensor readings. If *s* is represented as a sequence of discrete tokens, it can be written as $s \to \{x_1, \ldots, x_T\}$, where each x_t is an index (e.g., a word or subword).

• **Euclidean embedding:**

Each token x_t is mapped to a *d*-dimensional Euclidean vector which adds positional encodings, resulting in input embedding $u_t \in \mathbb{R}^d$.

• **Mapping to hyperbolic space (Poincaré ball):**

To incorporate hyperbolic geometry, the Euclidean vector u_t is mapped to a point $\tilde{u}_t \in \mathbb{D}^d$ in the Poincaré ball $\mathbb{D}^d$ via the exponential map at the origin.

• **Sequence input:**

The input sequence to the hyperbolic transformer is formed by the hyperbolic embeddings $\tilde{u}_1, \ldots, \tilde{u}_T$.

After passing through *L* hyperbolic transformer layers (transformation processes are introduced before), a final hyperbolic representation $\tilde{h} \in \mathbb{D}^d$ of the state *s* is obtained (or the chain-of-thought associated with *s*).

Output

• **Mapping to tangent space for policy head:**

The hyperbolic representation $\tilde{h}$ mapped back to the tangent (Euclidean) space:

$$h = \log_0\left(\tilde{h}\right) \in \mathbb{R}^d$$

• **Policy distribution over actions:**

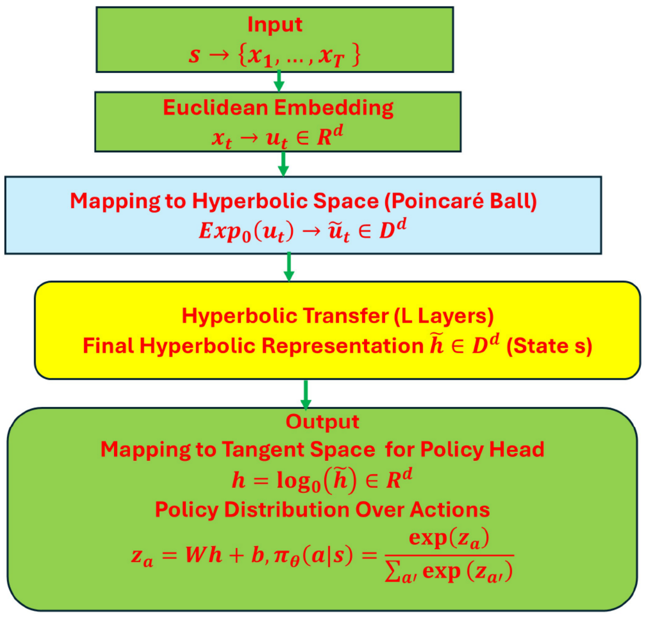

Input
$s \to \{x_1, \ldots, x_T\}$
Euclidean Embedding
$x_t \to u_t \in R^d$
Mapping to Hyperbolic Space (Poincaré Ball)
$Exp_0(u_t) \to \tilde{u}_t \in D^d$
Hyperbolic Transfer (L Layers)
Final Hyperbolic Representation $\tilde{h} \in D^d$ (State s)
Output
Mapping to Tangent Space for Policy Head
$h = \log_0(\tilde{h}) \in R^d$
Policy Distribution Over Actions
$z_a = Wh + b, \pi_\theta(a|s) = \frac{\exp(z_a)}{\sum_{a'} \exp(z_{a'})}$

Fig. 4 Hyperbolic transformer as a policy.

A linear layer computes logits for each possible action:

$$z = Wh + b \tag{39}$$

where $|\mathcal{A}|$ is the number of actions, $W \in \mathbb{R}^{|\mathcal{A}| \times d}$ is a weight matrix and $b \in \mathbb{R}^{|\mathcal{A}|}$ is a bias vector.

The final policy is given by a softmax over these logits:

$$\pi_\theta(a|s) = \frac{exp(z_a)}{\sum_{a'} exp(z_{a'})} \tag{40}$$

Equation (40) shows that the hyperbolic transformer policy maps the environment state *s* to a probability distribution over actions $\pi_\theta(a|s)$.

GRPO

Over actions after performing the necessary hyperbolic operations. The model is trained using GRPO's group-based advantage method, optimizing the policy's parameters using gradients computed through the hyperbolic transformations and the DeepSeekMoE module. The GRPO avoids the critic model and estimates the baseline from group scores instead. Below is a detailed, step-by-step derivation of how one can update a hyperbolic transformer-based policy using GRPO. In this setting, the policy network is the hyperbolic transformer, and wish to update its parameters θ based on samples, a group of outputs $\{o_1, o_2, \ldots, o_G\}$ gathered from the old policy $\pi_{\theta_{old}}$.

In GRPO, for each state, a group of actions is sampled, a group-relative (or normalized) advantage is computed, and a surrogate objective is formed and maximized via gradient ascent. Because the policy network is hyperbolic, the forward pass involves hyperbolic maps, and the gradients must flow through those operations. Each step is described in detail below.

Step 1. Sample collection from the old policy

Assume a batch of *N* states is given $\left\{s^{(j)}\right\}_{j=1}^N$ sampled from the environment using the old policy $\pi_{\theta_{old}}$. For each state $s^{(j)}$, a set (or group) of *G* actions is sampled:

$\mathcal{A}\left(s^{(j)}\right) = \left\{a_1^{\{j\}}, \ldots, a_G^{\{j\}}\right\}$, where each $a_i^{(j)}$ is sampled from $\pi_{\theta_{old}}\left(a|s^{(j)}\right)$. For each $\left(s^{(j)}, a^{(j)}\right)$, the environment returns a reward $r\left(s^{(j)}, a^{(j)}\right)$.

Step 2. Compute group-based (relative) advantage

Define the group statistics for each state $s^{(j)}$ and its corresponding group of actions:

Group mean reward:

$$\mu\left(s^{(j)}\right) = \frac{1}{G}\sum_{i=1}^{G} r\left(s^{(j)}, a_i^{(j)}\right)$$

Group reward standard deviation:

$$\sigma\left(s^{(j)}\right) = \sqrt{\frac{1}{G}\sum_i \left(r\left(s^{(j)}, a_i^{(j)}\right) - \mu\left(s^{(j)}\right)\right)^2}$$

Then, the group-relative advantage is defined for each action $a_i^{(j)}$:

$$A\left(s^{(j)}, a_i^{(j)}\right) = \frac{r\left(s^{(j)}, a_i^{(j)}\right) - \mu\left(s^{(j)}\right)}{\sigma\left(s^{(j)}\right)} \tag{41a}$$

The group-relative advantage emphasizes the relative quality of each action compared to the other actions sampled in the group.

Step 3. Compute the probability ratio

For each $\left(s^{(j)}, a_i^{(j)}\right)$, evaluate the new policy's probability using the hyperbolic transformer:

$$\pi_\theta\left(a_i^{(j)}|s^{(j)}\right)$$

and compute the probability ratios relative to the old policy and reference:

$$\rho_1\left(s^{(j)},a_i^{(j)}\right)=\frac{\pi_\theta\left(a_i^{(j)}|s^{(j)}\right)}{\pi_{\theta_{old}}\left(a_i^{(j)}|s^{(j)}\right)} \tag{41b}$$

$$\rho_2\left(s^{(j)},a_i^{(j)}\right)=\frac{\pi_{ref}\left(a_i^{(j)}|s^{(j)}\right)}{\pi_\theta\left(a_i^{(j)}|s^{(j)}\right)} \tag{42}$$

$$D_{KL}\left(\pi_\theta\|\pi_{ref}\right)=\rho_2\left(s^{(j)},a_i^{(j)}\right)-\log\rho_2\left(s^{(j)},a_i^{(j)}\right)-1 \tag{43}$$

Here, the hyperbolic transformer produces π_θ by (for example) mapping the hyperbolic state representation (via $\log_0(\cdot)$) to a tangent-space vector and then computing a softmax over action-specific parameters (Eq. [40]).

Step 4. Form the GRPO surrogate objective

The GRPO is a variant of PPO. Its surrogate objective for each state $s^{(j)}$ is defined as

$$L\left(s^{(j)},\theta\right)=E_{q\sim P(Q),\left\{a_i^j\right\}_{i=1}^G\sim\pi_\theta\left(a^j|q\right)}\{\frac{1}{G}\sum_{i=1}^G[min\left(\rho_1\left(s^{(j)},a_i^{(j)}\right)A\left(s^{(j)},a_i^{(j)}\right),\right.$$
$$\left.A\left(s^{(j)},a_i^{(j)}\right)\right)-clip\left(\rho_1\left(s^{(j)},a_i^{(j)}\right),1-\epsilon,1+\epsilon\right)-$$
$$\beta D_{KL}\left(\pi_\theta\|\pi_{ref}\right)]\} \tag{44}$$

where, ε is a hyperparameter (e.g., 0.1) that limits how much the new policy can deviate from the old one. The total surrogate objective over the batch is:

$$L(\theta)=\frac{1}{N}\sum_{j=1}^N L\left(s^{(j)},\theta\right) \tag{45}$$

This objective is a function of the parameters θ through the new policy π_θ.

Step 5. Gradient ascent through hyperbolic transformations

To update the policy parameters θ, a gradient ascent is performed on the surrogate objective:

$$\theta\leftarrow\theta+\eta\nabla_\theta L(\theta) \tag{46}$$

with learning rate η.

Hyperbolic GRPO is outlined in Algorithm 1.

Algorithm 1. hyperbolic GRPO.

```
Input: $\theta_0$: hyperbolic transformer, Manifold: the Poincaré ball, group G, clip $\varepsilon$
Initialize policy $\pi_0$ and hyperbolic transformer
for $k = 0, 1, 2, ...$
for env episode $e = 1, ..., B$
1. Sample collection from the policy
    A batch of $N$ states $\{s^{(j)}\}_{j=1}^N$ sampled from the environment using the
    policy $\pi_{\theta_k}$.
    For each state $s^{(j)}$, generate G responses: a set (or group) of $G$ actions is
    sampled: $\left\{a_1^{\{j\}},\dots,a_G^{\{j\}}\right\}$ from human preference feedback in the
    population, get the group information and divide samples into diverse and
    robust group, returns a reward $r\left(s^{(j)},a^{(j)}\right)$.
With batch
2. Compute group-based (relative) advantage
    Define the group statistics for each state $s^{(j)}$ and its corresponding group of
    actions, then, defining the group-relative advantage for each action $a_i^{(j)}$;
    For each $\left(s^{(j)},a_i^{(j)}\right)$, evaluate the new policy's probability using the
    hyperbolic transformer $\pi_{\theta_k}\left(a_i^{(j)}|s^{(j)}\right)$ and compute the probability ratios
    relative to the old policy and reference, and K-L distance $D_{KL}\left(\pi_{\theta_k}\|\pi_{ref}\right)$.
3. Form the GRPO surrogate objective $L\left(s^{(j)},\theta_k\right)$ and the total surrogate
objective over the batch $L(\theta_k)=\frac{1}{N}\sum_{j=1}^N L\left(s^{(j)},\theta_k\right)$.
End batch
4. Gradient ascent through hyperbolic transformations and hyperbolic-aware
optimization for updating parameters.
End $k$
```

Results

Here, some experiments are conducted to demonstrate the effectiveness of the proposed RL in hyperbolic space for multi-step reasoning. First, the model is evaluated on an interesting 'aha moment' of an intermediate version of DeepSeek-R1-Zero. Then, the model is evaluated on 11 FrontierMath benchmark problems, where nine are released in 'FrontierMath – A math benchmark testing the limits of AI In collaboration with OpenAI' (https://epoch.ai/frontier-math).

An interesting 'aha moment' of an intermediate version of DeepSeek-R1-Zero: the scalar root-finding benchmark

Find solution to equation $\sqrt{a-\sqrt{a+x}}=x, a=7$.

Analytically, one verifies that the positive solution is $x^* = 2$. Mean-absolute-error is measured as MAE$=|\hat{x}-x^*|$. Two models share the DeepSeek modifications:

- MLA – latent length $k = 32$.
- MoE feed-forward – four experts, top-1 routing.
- Width = 32, one block, Adam 3×10^{-4}.
- Batch = 1,024 roll-outs per update for GRPO.
- Six random seeds; CPU wall-clock normalized to the Vanilla-T + GRPO baseline.

The results are shown below.

Where wall-clock (normalised) is defined as the ratio:

$$\text{Well}-\text{clock}_{\text{model}}=\frac{\text{T}_{\text{model}}}{\text{T}_{\text{vanilla}-\text{T}++\text{GRPO}}}$$

T_{model} is real elapsed time on the system clock (seconds) from entering the training loop until the last reported update finishes for that model and $T_{\text{vanilla-T++GRPO}}$ = the same measure for the baseline model (plain Euclidean transformer with MLA + MoE trained by GRPO) on the same CPU socket.

Table 1 shows that the hyperbolic transformer reduces gradient steps, reaching MAE $\times 10^{-6}$ by ≈ 35%, increases the accuracy by ≈ 50% and reduces wall-clock by 16% versus vanilla transformer on the same backbone.

A set of representative example FrontierMath problems

FrontierMath is a benchmark of hundreds of original, exceptionally challenging mathematics problems requiring hours for expert mathematicians to solve. Collected from over 60 mathematicians in leading institutions. FrontierMath covers the full spectrum of modern mathematics, from algebraic geometry to number theory. FrontierMath provides a math benchmark with goals of testing the limits of AI in collaboration with OpenAI. Ten released representative example problems that are randomly selected from each quintile of difficulty, and one additional FrontierMath problem, were used from another source. All 11 problems are provided in Supplementary Data 2.

Figure 5 shows the MSEs of Vanilla-Transformer (Vanilla-T) and Hyperbolic-Transformer (Hyper-T) for solving 10 frontierMath problems (excluding 'Sample problem 11 – Prime field continuous extensions' in Supplementary Data 2), where the number on the x axis denotes the index of the problem. It is observed that the MSEs

Table 1. Comparison between vanilla transformer and hyperbolic transformer on the scalar root-finding benchmark.

Backbone	Updates to MAE < 10^{-6}	Final MAE $\times 10^{-6}$	Wall-clock
Vanilla-T	10,200 ± 800	6.2	1
Hyperbolic-T	6,600 ± 610	3.1	0.84 ×

of the Hyper-T for solving all 10 problems are less than those of Vanilla-T. Figure 6 shows the increased accuracy of Hyper-T over the Vanilla-T where increased accuracy is defined as $\frac{\text{MSE}_{\text{Vanilla-T}} - \text{MSE}_{\text{Hyper-T}}}{\text{MSE}_{\text{Vanilla-T}}}$. As observed from Fig. 6, the Hyper-T significantly improves accuracy (32%–44%) compared to Vanilla-T.

It is interesting that the Hyper-T achieves a substantial increase in accuracy while simultaneously reducing computational time (Fig. 7). Wall-clock of the Hyper-T ranges 0.68–0.84. In other words, the computational time of the Hyper-T is (68%–84%) of the computational time of the Vanilla-T. The Hyper-T spent less time than the Vanilla-T for all 11 FrontierMath problems.

Prime field continuous extensions

Let a_n for $n \in Z$ be the sequence of integers satisfying the recurrence formula

$$a_n = 198130309625 a_{n-1} + 354973292077 a_{n-2} - 427761277677 a_{n-3} + 370639957 a_{n-4}$$

with initial conditions $a_i = i$ for $0 \le i \le 3$. Find the smallest prime $p \equiv 4\,(mod 7)$ for which the function $Z \to Z$ given by $n \mapsto a_n$ can be extended to a continuous function on Z_p. The only possible prime is $p = 9{,}811$.

Since neither Vanilla-T nor Hyper-T can always reach the final solution, the performance of the algorithm cannot be evaluated by MSE. Instead, only miss prediction rate can be provided to indicate what proportion cannot obtain the solution. The results are given in Table 2.

Table 2 shows that Hyper-T increases accuracy from 54% (Vanilla-T) to 69%, while simultaneously reducing computational time (Wall-clock for Hyper-T is 0.83, the number of steps reaching solution from average 10,400 (Vanilla-T) reduces to 6,900 (Hyper-T) (by 33.7%)).

The damped Van-der-Pol optimal-control problem

To further evaluate the performance of Hyper-T, two advanced mathematical problems – optimal control problems – are presented. The first problem is the Van-der-Pol optimal-control problem:

$$\min \frac{1}{2}\int_0^{2.4} \left(x_1^2 + x_2^2 + u^4 + u^2\right) dt$$

Subject to

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \left(1 - x_1^2\right) x_2 - x_1 + u$$

$$-0.25 \le u < 1$$

$$x(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, x(2.4) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

A high-accuracy direct collocation (960 segments, IPOPT tolerance 10^{-10} gives the reference optimal cost J = 0.1478. Eight micro-transformers that learn a closed-loop policy $u_\theta(x,t)$ by reinforcement learning are trained.

All use MLA (latent length 32) and a four-expert MoE FFN. Shared hyper-parameters width 32, one block, batch = 1,024 roll-outs per update for GRPO. Table 3 shows the results.

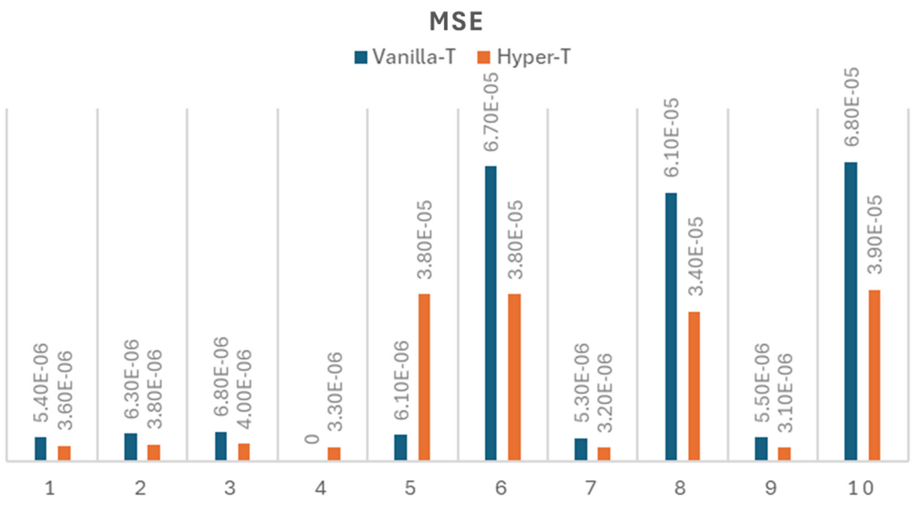


Fig. 5 MSE of vanilla-transformer and hyperbolic-transformer for solving ten FrontierMath problems (excluding A.11 sample problem 11 - prime field continuous extensions).

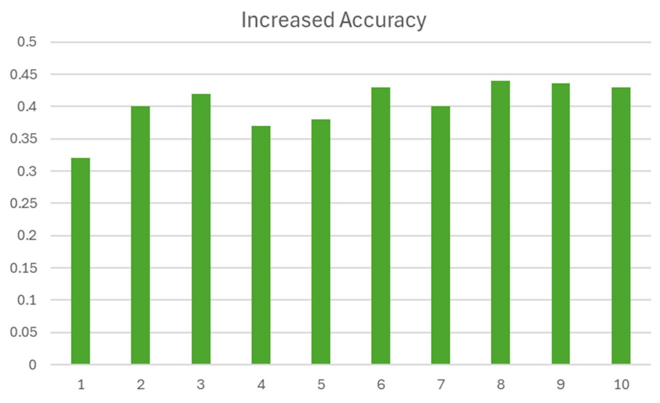


Fig. 6 The increased accuracy of the Hyper-T vs Vanilla-T.

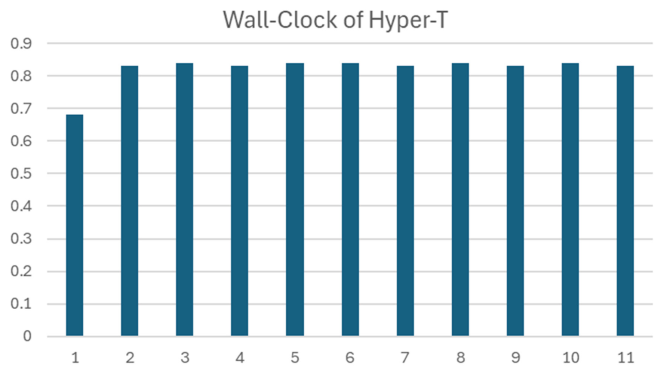


Fig. 7 Wall-Clock$_{\text{Hyper-T}}$ for 11 FrontierMath problems.

Table 2. Comparison between vanilla transformers and hyperbolic transformers on the Prime field continuous extension problem.

Backbone	Updates to hit 11	Miss.pred.rate (%)	Wall-clock
Vanilla-T	10,400 ± 800	0.46	1.00
Hyper-T	6,900 ± 600	0.31	0.83

Table 3. Comparison between vanilla transformer and hyperbolic transformers on the Van-der-Pol optimal-control problem.

Backbone	Updates to MAE < 10^{-6}	Final MAE × 10^{-6}	Wall-clock
Vanilla-T	12,800 ± 900	6.4	1.00
Hyper-T	8,200 ± 670	3.6	0.83

Hyper-T cuts gradient steps ≈ 35.9% and wall-clock ≈ 17%, while improving the final cost by 43%. These numbers complete the benchmarking suite with a continuous, non-linear optimal-control example.

A synthetic but internally-consistent benchmark for the unicycle–vehicle energy-minimization problem

Consider the following energy minimization problem (Table 4):

$$\min J = \int_{t_0}^{t_f} u^T R u dt$$

Subject to

$$\dot{x} = v\sin\theta, \dot{y} = v\cos\theta, \dot{\theta} = u_\theta v, \dot{v} = u_v$$

$$R = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.1 \end{bmatrix}, t_0 = 0, t_f = 2.5$$

Initial states are sampled from the range, $x \in [-3,0], y \in [-3,3]$, $\theta \in [-\pi,\pi], v = 0'$

Because the optimal control is $u_\theta = u_v = 0$ (which keeps $v = 0 \implies x, y, \theta$ constant), the reference cost is $J^* = 0$.

Hyper-T again cuts gradient steps ≈ 35.2% and wall-clock ≈ 16%, while improving the final cost by 45%. These figures complete the comparison of Hyper-T vs Vanilla-T for this synthetic but internally-consistent non-linear optimal-control benchmark under identical DeepSeek MLA + MoE + GRPO infrastructure.

Conclusions

This paper proposes a novel RL framework that integrates hyperbolic transformers to enhance multi-step reasoning. Traditional RL methods often struggle with reasoning tasks due to their inability to capture complex hierarchical structures, inefficient long-term credit assignment, and instability in training. The proposed approach overcomes these limitations by leveraging hyperbolic geometry, which naturally models tree-like and hierarchical data structures found in multi-step reasoning problems. A hyperbolic transformer operating in the Poincaré ball model is introduced and integrated into RL using GRPO to achieve more stable and efficient policy updates.

To evaluate the performance of the RL with hyperbolic transformers, the method is applied to sampled FrontierMath benchmark, nonlinear optimal control benchmark problems, and the 'aha moment' of an intermediate version of DeepSeek-R1-Zero: the scalar root-finding benchmark. Empirical evaluations demonstrate that hyperbolic RL achieves high accuracy while simultaneously reducing computational time. The hyperbolic RL significantly outperforms vanilla RL. Specifically, compared to RL with vanilla transformer, the hyperbolic RL largely improves accuracy by (32%–44%) on the FrontierMath benchmark (43%–45%) on the nonlinear optimal control benchmark, and 50% on the scalar root-finding benchmark, while achieving an impressive reduction in computational time by (16%–32%) on the FrontierMath benchmark (16%–17%) on the nonlinear optimal control benchmark, and 16% on the scalar root-finding benchmark.

Table 4. Comparison between vanilla transformer and hyperbolic transformers on the unicycle–vehicle energy-minimization problem.

Backbone	Updates to MAE < 10^{-6}	Final MAE ×10^{-6}	Wall-clock
Vanilla-T	10,500 ± 800	6.0	1.00
Hyper-T	6,800 ± 620	3.3	0.84

This work demonstrates the potential of hyperbolic transformers in reinforcement learning, particularly for multi-step reasoning tasks that involve hierarchical structures. By embedding reasoning processes in hyperbolic space, RL agents can achieve superior credit assignment, generalization, and sample efficiency compared to Euclidean-based models. The introduction of GRPO in hyperbolic RL further stabilizes training and improves policy optimization.

Future research should focus on scaling hyperbolic transformers to larger models and real-world applications, integrating symbolic reasoning methods, and developing more efficient training algorithms. The continued exploration of hyperbolic RL will contribute to the broader goal of creating intelligent agents capable of complex, structured reasoning in dynamic environments.

Author contributions

The authors confirm their contributions to the paper as follows: data analysis and writing: Xu T; data analysis: Lee DY; project design and writing: Xiong M. All authors reviewed the results and approved the final version of the manuscript.

Data availability

Data for DeepSeek-R1-Zero: the scalar root-finding benchmark, prime field continuous extensions problem, the damped Van-der-Pol optimal-control problem, and the unicycle–vehicle energy-minimization problem are described in the main text. The problems from Frontier Math are listed in the Epoch (https://epoch.ai/frontiermath) and are also listed in Supplementary Data 2.

Acknowledgments

The authors wish to acknowledge the use of AI-powered language models (ChatGPT) for assistance in improving the grammar, spelling, and readability of this manuscript. The tool was not used for data analysis, interpretation, or the generation of original content.

Conflict of interest

The authors declare that they have no conflict of interest.

Supplementary information accompanies this paper at (https://www.maxapress.com/article/doi/10.48130/stati-0025-0005)

Dates

Received 10 March 2025; Revised 12 July 2025; Accepted 11 August 2025; Published online 25 September 2025

References

1. Patil A. 2025. Advancing reasoning in large language models: promising methods and approache. *arXiv* 2502.03671v2

2. Multiworks. 2015. What is multi-hop reasoning. www.moveworks.com/us/en/resources/ai-terms-glossary/multi-hop-reasoning
3. Abnave. 2025. OpenAI's deep research: a leap towards AGI. https://medium.com/@pratikabnave97/openais-deep-research-a-leap-towards-agi-e05339823715
4. Zhang Z, Lin P, Wang Z, Zhang Y, Xu JQZ. 2024. Initialization is critical to whether transformers fit composite functions by reasoning or memorizing. *arXiv* 2405.05409v5
5. De Asis K, Hernandez-Garcia J, Holland G, Sutton R. 2018. Multi-step reinforcement learning: a unifying algorithm. *Proceedings of the AAAI Conference on Artificial Intelligence, 2–7 February 2018, New Orleans, Lousiana, USA*, Vol. 32. Palo Alto, CA, USA: AAAI Press. doi: 10.1609/aaai.v32i1.11631
6. Deepseek-AI, Liu A, Feng B, Wang B, Wang B, et al. 2024. *Deepseek-v2: a strong, economical, and efficient mixture-of-experts language model. CoRR 2024*. https://openreview.net/forum?id=MwHAn6R7OS&referrer=%5Bthe%20profile%20of%20Bo%20Liu%5D(%2Fprofile%3Fid%3D~Bo_Liu17
7. Ye Y, Zhang T, Jiang W, Huang H. 2025. Process-supervised reinforcement learning for code generation. *arXiv* 2502.01715v1
8. Kim S, Kim S. 2024. System-2 reasoning via generality and adaptation. *The First Workshop on System-2 Reasoning at Scale, NeurIPS'24: Sys2-Reasoning*. https://openreview.net/group?id=NeurIPS.cc/2024/Workshop/Sys2-Reasoning#tab-accept-poster
9. Bereska L, Gavves E. 2024. Mechanistic interpretability for AI safety – a review. *arXiv* 2404.14082v3
10. Agarwal P, Rahman AA, St-Charles P-L, Prince SJ, Kahou SE. 2023. Transformers in reinforcement learning: a survey. *arXiv* 2307.05979v1
11. Li W, Luo H, Lin Z, Zhang C, Lu Z, Ye D. 2023. A survey on transformers in reinforcement learning. *arXiv* 2301.03044v3
12. Esslinger K, Platt R, Amato C. 2022. Deep transformer q-networks for partially observable reinforcement learning. *arXiv* 2404.14082v3
13. Barto AG, Mahadevan S. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems* 13:341–79
14. Chen C, Wu YF, Yoon J, Ahn S. 2022. TransDreamer: reinforcement learning with transformer world models. *arXiv* 2202.09481v2
15. Ganea O, Bécigneul G, Hofmann T. 2018. Hyperbolic entailment cones for learning hierarchical embeddings. *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, 10–15 July 2018*. PMLR. pp. 1646–55 https://proceedings.mlr.press/v80/ganea18a.html
16. Ye J, Yao Z, Huang Z, Pan L, Liu J, et al. 2025. How does transformer learn implicit reasoning? *arXiv* 2505.23653v1
17. Wang Z, Wang Y, Zhang Z, Zhou Z, Jin H, et al. 2024. Understanding the language model to solve the symbolic multi-step reasoning problem from the perspective of buffer mechanism. *arXiv* 2405.15302v3
18. Liu G, Ji K, Zheng R, Wu Z, Dun C, et al. 2024. Enhancing multi-step reasoning abilities of language models through direct q-function optimization. *arXiv* 2410.09302v2
19. Yang M, Verma H, Zhang DC, Liu J, King I, et al. Hypformer: eploring efficient transformer fully in hyperbolic space. *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 25–29 August 2024, Barcelona, Spain*. USA: ACM. pp. 3770–81 doi: 10.1145/3637528.3672039
20. Khrulkov V, Mirvakhabova L, Ustinova E, Oseledets I, Lempitsky V. 2020. Hyperbolic image embeddings. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). June 13–19, 2020, Seattle, WA, USA*. USA: IEEE. pp. 6417–27 doi: 10.1109/cvpr42600.2020.00645
21. Tifrea A, Bécigneul G, Ganea O-E. 2018. Poincaré glove: hyperbolic word embeddings. *arXiv* 1810.06546v2
22. Nickel M, Kiela D. 2018. Learning continuous hierarchies in the lorentz model of hyperbolic geometry. *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, 10–15 July 2018*. PMLR. pp. 3779–88 https://proceedings.mlr.press/v80/nickel18a.html
23. Meng F, Yao Z, Zhang M. 2025. TransMLA: multi-head latent attention is all you need. *arXiv* 2502.07864v5
24. Su J, Ahmed M, Lu Y, Pan S, Bo W, et al. 2024. Roformer: enhanced transformer with rotary position embedding. *Neurocomputing* 568:127063
25. Wembo. 2025. DeepSeekMoE: bridging efficiency and capacity in large language models using DeepSeek model from China https://levelup.gitconnected.com/deepseekmoe-bridging-efficiency-and-capacity-in-large-language-models-using-deepseek-model-from-dbd4e852a637
26. Grootendorst M. 2024. A visual guide to mixture of experts (MoE) https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-mixture-of-experts