PolyChristoffel Networks: Interpretable Dynamics via Learnable Curvatures

Mihir Talati
University of Maryland - College Park
CMSC 498Z/838B Final Report

Abstract

We present PolyChristoffel Networks, a dynamics model that treats a torsion-free connection as the primary learnable object. Instead of learning an opaque vector field, the model parameterizes Christoffel symbols as low-degree polynomials in the state, yielding an interpretable, structured acceleration law. Trajectories are produced by integrating the geodesic equation in a discrete ResNet-style rollout. To encourage a (pseudo-)Riemannian interpretation, we reconstruct a metric by integrating the metric-compatibility equation along paths and penalize path dependence via a loop loss. We describe the mathematical formulation, the loss terms, and the implementation details that map directly onto the codebase (e.g., polynomial features in `poly_features.py`, Christoffel coefficients in `christoffel.py`, metric reconstruction in `metric_recon.py`, and training orchestration in `train.py`). A polar-coordinate inertial benchmark with known Christoffels provides diagnostics for both learned connection components and reconstructed metrics. This report expands the project presentation with additional mathematical detail and explicit implementation references.

1 Introduction

Many neural dynamics models fit trajectories but obscure the underlying structure of the equations of motion. Recent physics-inspired approaches, such as Hamiltonian neural networks [3] and Neural ODEs [2], aim to impose inductive biases aligned with known physical laws. We take a complementary view: model the geometry of the state space itself and treat the connection coefficients as the learnable parameters. This yields interpretable, symbolic dynamics where acceleration is a quadratic form in velocity with coefficients that are explicit functions of state.

The PolyChristoffel Network learns a torsion-free connection parameterized by a polynomial basis. Trajectories are geodesics of this learned connection. To ensure compatibility with a (pseudo-)Riemannian metric, we reconstruct the metric via the compatibility equation and enforce path-independence through a loop loss. The approach is lightweight, differentiable, and directly grounded in differential geometry.

2 Geometric background

Let $x(t) \in \mathbb{R}^d$ denote a trajectory in a coordinate chart, with velocity $v(t) = \dot{x}(t)$. A (torsion-free) affine connection is specified by Christoffel symbols $\Gamma^i{}_{jk}(x)$, symmetric in the lower indices: $\Gamma^i{}_{jk} = \Gamma^i{}_{kj}$. Geodesics satisfy

$$\ddot{x}^i + \Gamma^i{}_{jk}(x)\,\dot{x}^j\dot{x}^k = 0. \tag{1}$$

Define the acceleration field

$$a^i(x,v) = -\Gamma^i{}_{jk}(x)\,v^j v^k. \tag{2}$$

This structure makes dynamics interpretable: each term is a coupling of velocities scaled by state-dependent coefficients.

If a metric $g(x)$ exists such that the connection is Levi-Civita, then Γ is torsion-free and metric-compatible ($\nabla g = 0$). In coordinates, metric compatibility yields

$$\partial_i g_{jk} = \Gamma^\ell{}_{ij} g_{\ell k} + \Gamma^\ell{}_{ik} g_{j\ell}. \tag{3}$$

We exploit this to reconstruct a metric implied by the learned connection.

3 Model: Polynomial Christoffel network

We parameterize $\Gamma^i{}_{jk}(x)$ using a monomial basis $\phi(x)$ up to degree p. For d dimensions and degree $p = 2$, the feature dimension is

$$M = 1 + d + \frac{d(d+1)}{2}. \tag{4}$$

The feature map is

$$\phi(x) = [1,\, x_1, \ldots, x_d,\, x_1^2,\, x_1x_2, \ldots, x_d^2]^\top \in \mathbb{R}^M. \tag{5}$$

We then define

$$\Gamma^i_{jk}(x) = \sum_{m=1}^{M} C^i_{jk,m}\,\phi_m(x), \tag{6}$$

with learnable coefficients $C^i_{jk,m}$. In the implementation, polynomial features are in `poly_features.py`, and the coefficient tensor is in `christoffel.py` (parameter `self.C`). Torsion-free symmetry is enforced by construction using a symmetric average in the lower indices.

4 Discrete geodesic rollout

We integrate Eq. 1 by converting it to a first-order system:

$$\dot{x} = v, \qquad \dot{v}^i = a^i(x,v). \tag{7}$$

A simple semi-implicit Euler update is used:

$$v_{t+1} = v_t + \Delta t\, a(x_t, v_t), \qquad x_{t+1} = x_t + \Delta t\, v_{t+1}. \tag{8}$$

This defines a ResNet-style rollout, implemented in `integrators.py` (function `rollout_geodesic_resnet`). Optional projection onto a manifold constraint and velocity tangent projection are applied per step in `manifold.py`.

5 Metric reconstruction and geometric regularizers

Equation 3 is a linear first-order PDE. Along a path $x(s)$ from a base point x_0 to x, it becomes an ODE:

$$\frac{d}{ds} g_{jk}(s) = \left(\Gamma^\ell_{ij}(x(s))g_{\ell k}(s) + \Gamma^\ell_{ik}(x(s))g_{j\ell}(s)\right)\frac{dx^i}{ds}. \tag{9}$$

We discretize this ODE with Euler steps in `metric_recon.py` (`metric_ode_step` and `reconstruct_metric_straight_path`).

Because a general connection need not be metric-compatible, the reconstructed metric can be path-dependent: $g^{(A)}(x) \neq g^{(B)}(x)$ for different paths. We measure this with a loop loss by comparing a direct path $x_0 \to x$ to a two-segment path $x_0 \to x_m \to x$:

$$\mathcal{L}_{\text{loop}} = \|g^{(A)}(x) - g^{(B)}(x)\|_F^2. \tag{10}$$

The two-segment reconstruction is implemented in `reconstruct_metric_two_segment`.

To encourage a valid (pseudo-)Riemannian metric, we include symmetry, non-degeneracy, and signature penalties in `losses.py`:

$$\mathcal{L}_{\text{sym}} = \|g - g^\top\|_F^2, \tag{11}$$

$$\mathcal{L}_{\text{det}} = \frac{1}{B}\sum_b \text{softplus}(\alpha - \log|\det g_b|), \tag{12}$$

$$\mathcal{L}_{\text{sig}} = \frac{1}{B}\sum_b \sum_{i=1}^{d} \text{softplus}(-\beta s_i \lambda_i(g_b)), \tag{13}$$

where g_b is the reconstructed metric for sample b, $\lambda_i(g_b)$ are eigenvalues in ascending order, and $s_i \in \{+1,-1\}$ encodes the target signature (p,q). The symmetry loss enforces the metric tensor properties, the determinant barrier avoids degeneracy, and the signature loss penalizes eigenvalues that violate the prescribed signs. The base metric $g(x_0)$ is parameterized as a signed Cholesky factor with fixed signature in `model.py` (class `BaseMetric`),[1] which keeps the optimization in a valid metric family. In the implementation, α corresponds to the log-determinant floor (default -2.0) and β controls how sharply signature violations are penalized.

6 Training objective

Given a trajectory $x_{0:T}$, the model either predicts in state space (no autoencoder) or predicts in latent space and decodes. The primary loss is

$$\mathcal{L}_{\text{traj}} = \frac{1}{BT}\sum_{b,t}\|\hat{x}_{b,t} - x_{b,t}\|^2, \tag{14}$$

or a reconstruction loss in observation space when the autoencoder is enabled. The full objective is

$$\mathcal{L} = \mathcal{L}_{\text{traj/recon}} + \lambda_{\text{loop}}\mathcal{L}_{\text{loop}} + \lambda_{\text{sym}}\mathcal{L}_{\text{sym}} + \lambda_{\text{det}}\mathcal{L}_{\text{det}} + \lambda_{\text{sig}}\mathcal{L}_{\text{sig}} + \lambda_{\text{wd}}\|C\|_2^2, \tag{15}$$

where C is the Christoffel coefficient tensor. In AE mode, the reconstruction term is

$$\mathcal{L}_{\text{recon}} = \frac{1}{BT}\sum_{b,t}\|D(\hat{z}_{b,t}) - x_{b,t}\|^2. \tag{16}$$

We also include explicit weight decay on C:

$$\mathcal{L}_{\text{wd}} = \sum_{i,j,k,m} C^2_{ijkm}, \tag{17}$$

implemented as the mean of squared parameters in `train.py`. The training loop and metric warmup schedule are implemented in `train.py`, with a warmup fraction `metric_warmup_epochs` that delays metric-based losses until stable rollouts have formed. Metric losses are enabled only when `use_metric_losses` is true. The dataset for the polar benchmark is in `data.py`.

7 Backpropagation and Autodiff

All losses are differentiated with standard backpropagation through the unrolled computation graph. The discrete rollout in Eq. 8 is a sequence of differentiable operations, so gradients flow from the trajectory loss to the Christoffel coefficients via the chain

$$\mathcal{L} \to \{\hat{x}_t, \hat{v}_t\}_{t=0}^T \to \Gamma(x_t) \to C. \tag{18}$$

[1] Specifically, $g_0 = L\,\text{diag}(\text{signs}\cdot e^\ell)L^\top$, with learnable lower-triangular L and log-scales ℓ.

Metric reconstruction is similarly unrolled: the Euler updates in Eq. 9 are executed for each step along a path, and gradients flow through the sequence to the same parameters. This is implemented directly in PyTorch without adjoint methods; each step becomes part of the computation graph.

For numerical stability, the integrator clamps Christoffel values and velocities during rollout (`gamma_max` and `vmax` in `integrators.py`). Training uses PyTorch AMP when enabled: `torch.cuda.amp.autocast` wraps the forward pass, and gradients are scaled with `torch.amp.GradScaler` in `train.py`. When multiple GPUs are available, the model is wrapped in `nn.DataParallel`. Data loading uses pinned memory and non-blocking transfers for throughput, and CUDA kernels can be autotuned via `torch.backends.cudnn.benchmark`.

7.1 Autoencoder latent-space variant (AE=1)

We additionally evaluate a variant where an autoencoder learns a latent coordinate system $z = E(x)$ and the Poly-Christoffel dynamics model is trained in the latent space. In this setting, the learned Christoffels and reconstructed metric live in *latent* coordinates, so direct comparison to the polar-coordinate ground truth tensors is generally not meaningful: a smooth coordinate change can dramatically alter coordinate expressions of both Γ and g. Accordingly, we interpret AE results primarily through (i) trajectory reconstruction quality and (ii) intrinsic geometric consistency diagnostics (path dependence, symmetry, signature, and conditioning).

Trajectory fidelity improves substantially. With AE enabled, rollout MSE drops by an order of magnitude relative to the no-AE baseline: `traj_mse` $= 2.31 \times 10^{-1}$ and `traj_mse_final` $= 4.03 \times 10^{-1}$ (Table **??**). The per-timestep MSE increases gradually over the horizon, from $\approx 9.84 \times 10^{-2}$ at $t = 1$ to $\approx 4.03 \times 10^{-1}$ at $t = T$, indicating reduced but still present compounding error. []

Latent-space geometry is metrizable and well-conditioned. Unlike the no-AE evaluation where rollout-state metric diagnostics can explode, the AE run yields strong geometric consistency: `loop_mse_mean` $\approx 4.22 \times 10^{-3}$, near-zero signature penalty (`signature_loss` $\approx 4.40 \times 10^{-4}$), strictly positive minimum eigenvalue (`metric_min_eig` $\approx 3.50 \times 10^{-1}$), and zero fraction of negative eigenvalues. Together, these indicate that the learned connection in latent space is close to being compatible with a smooth SPD metric (fixed signature $(2,0)$) and that metric reconstruction remains stable on rollout states.

Interpreting Christoffels/metrics under AE. Grid-based plots (Figures 1–2) should be read differently than in the no-AE case. Because $z = E(x)$ is a learned coordinate transform, the "correct" latent Christoffels are not the polar Christoffels; even if the underlying geometry is Euclidean, a nonlinear encoder generally induces nontrivial $\Gamma(z)$ and $g(z)$. Therefore:

- It is expected that $\Gamma(z)$ does *not* match the polar closed forms.
- The important signal is that reconstructed $\hat{g}(z)$ is symmetric, non-degenerate, and exhibits low path dependence (loop error), which we observe.

For a coordinate-invariant comparison back in observation space, the appropriate diagnostic is a *pullback* metric using the encoder Jacobian:

$$g_x(x) \;=\; J_E(x)^\top \, g_z\big(E(x)\big)\, J_E(x),$$

which can be compared to a target observation-space metric (e.g., Euclidean or the known polar metric). This is implemented in the AE metric inspection script (`inspect_metric_ae.py`) and is the recommended way to assess whether the learned latent geometry corresponds to the intended geometry in x.

Metric (rollout states)	**AE=0**	**AE=1**
Trajectory fidelity		
Trajectory MSE (avg over time)	1.492	2.312×10^{-1}
Trajectory MSE (final step)	2.178	4.027×10^{-1}
Per-step MSE at t=1	1.01×10^{-1}	1.07×10^{-1}
Per-step MSE at t=T	2.178	4.027×10^{-1}
Connection / dynamics diagnostics		
MSE$\left[\Gamma^r_{\theta\theta}\right]$ (vs GT, rollout states)	5.050	–
MSE$\left[\Gamma^\theta_{r\theta}\right]$ (vs GT, rollout states)	2.118	–
Acceleration MSE	1.37×10^{6}	–
Metric reconstruction / geometry consistency		
Loop/path MSE mean $\lVert g^{(A)} - g^{(B)} \rVert^2$	3.79×10^{28}	4.22×10^{-3}
Symmetry loss $\lVert g - g^\top \rVert^2$	0	0
Log$\lvert\det\rvert$ barrier loss	1.73×10^{-1}	1.10×10^{-1}
Signature penalty (aggregate)	1.22×10^{6}	4.40×10^{-4}
Min eigenvalue of $\hat{g}$	-4.92×10^{7}	3.50×10^{-1}
Fraction negative eigenvalues	3.48×10^{-1}	0

Table 1: Rollout-state evaluation metrics from `evaluate.py`. AE=1 substantially improves rollout accuracy and stabilizes metric reconstruction (low path dependence and correct SPD signature). Entries marked "–" are not directly comparable in AE mode because learned latent coordinates change the coordinate expression of Γ and acceleration unless pullback-to-x diagnostics are used.

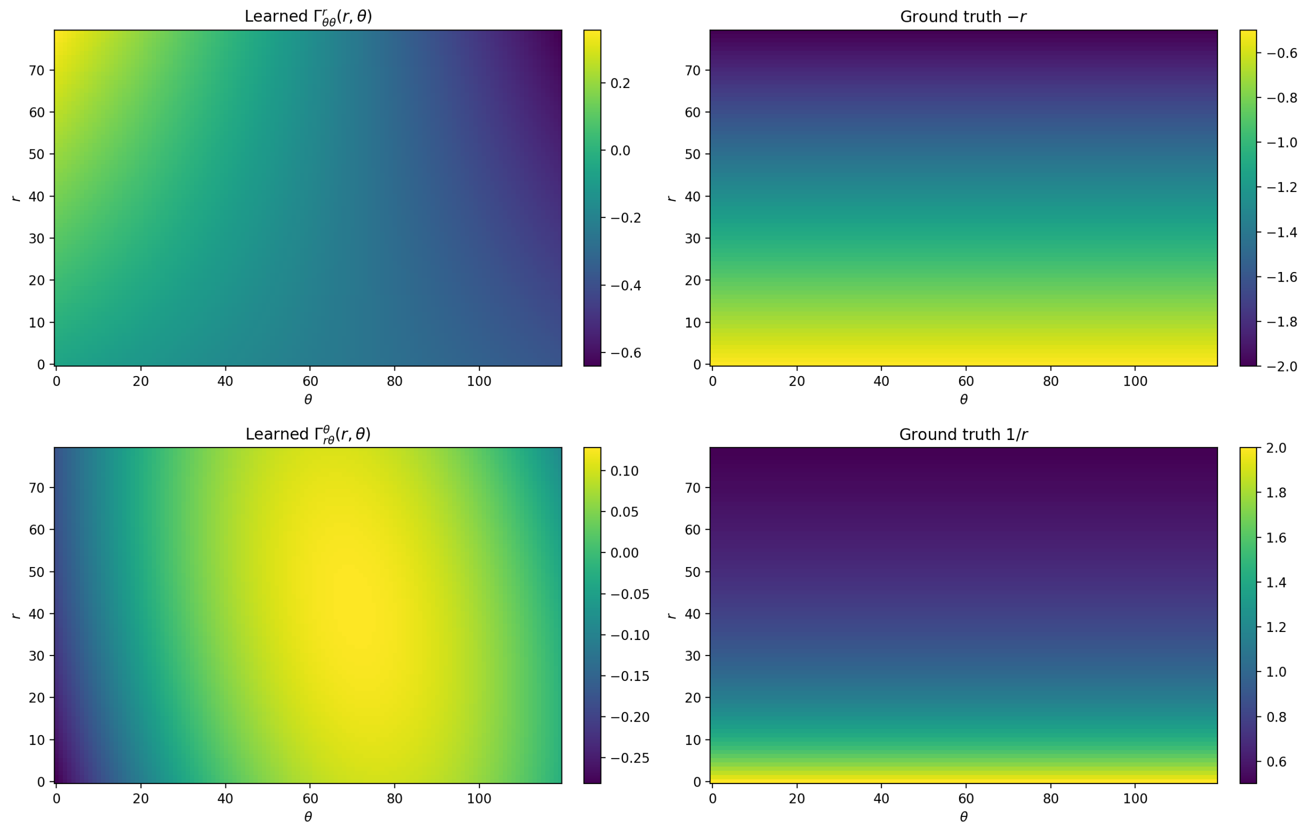

Figure 1: AE=1 Christoffel diagnostics (latent coordinates). Note that direct comparison to polar coordinate ground truth is not coordinate-invariant and is shown primarily for visualization. The key empirical improvement under AE is not exact component matching, but improved rollout accuracy and stable metric reconstruction with low path dependence.

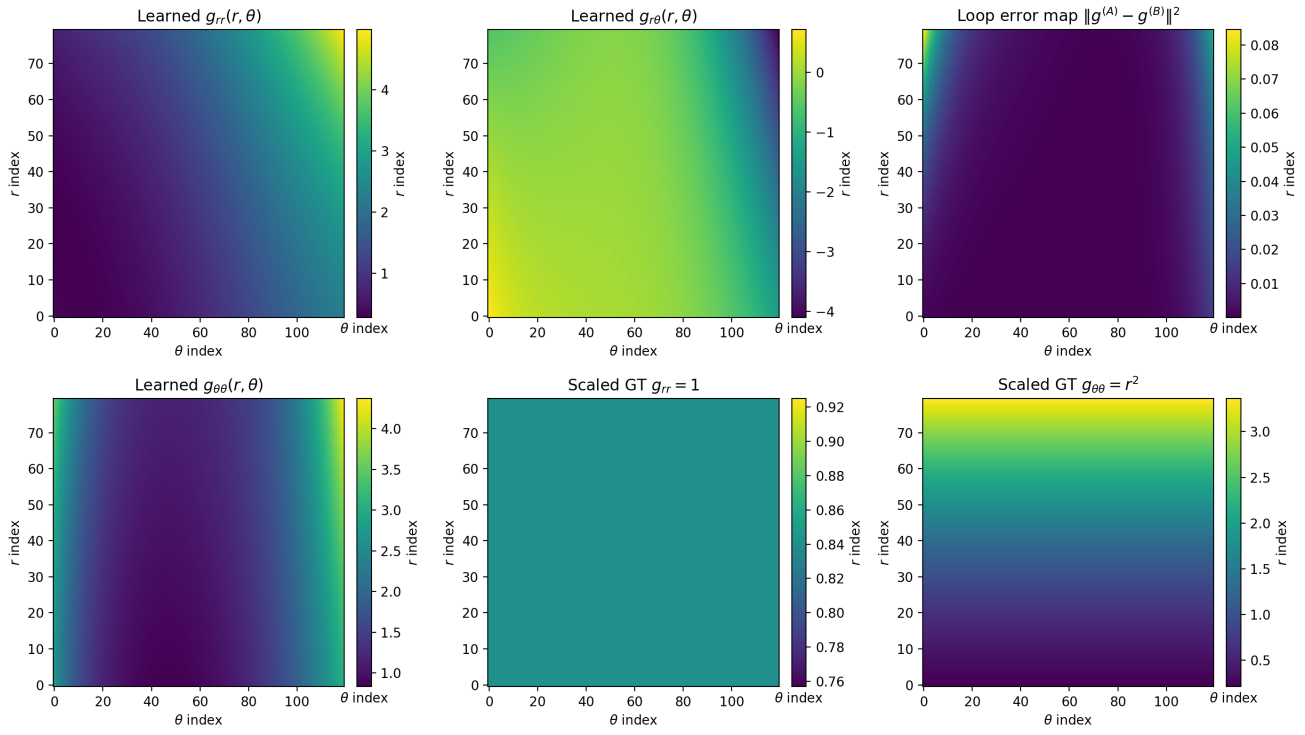

Figure 2: AE=1 metric reconstruction diagnostics in latent space. The loop/path-dependence map is uniformly small, and eigenvalue checks confirm an SPD metric across the evaluated grid.

8 Limitations and future work

The approach enforces metrizability only softly via loop and validity losses. Some connections may fit trajectories yet remain far from Levi-Civita of any metric. Integration uses a simple Euler scheme, which may introduce numerical drift for long rollouts. More accurate integrators (e.g., RK4 or ODE solvers) and explicit co-training of a metric network could strengthen geometric consistency. Extending to higher-dimensional observations (e.g., images) will require robust encoders and careful Jacobian estimation, as studied in atlas-based methods [4] and related work on geodesic flows [1].

Availability

The full implementation and source code is available at (https://github.com/Mihir-Null/PolyChristoffelNet) under `poly_christoffel/` and includes training, diagnostics, and visualization scripts referenced throughout this report.

References

[1] J. Burge. Neural geodesic flows. Master's thesis, ETH Zurich, 2025.

[2] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018. https://arxiv.org/abs/1806.07366.

[3] Sam Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. *arXiv preprint arXiv:1906.01563*, 2019. https://arxiv.org/abs/1906.01563.

[4] E. O. Korman. Atlas based representation and metric learning on manifolds. *arXiv preprint arXiv:2106.07062*, 2021. https://arxiv.org/abs/2106.07062.

A Implementation details

- **Polynomial features:** Eq. 6 is implemented in `poly_features.py` (monomials up to degree two).
- **Connection tensor:** The coefficient tensor $C^i{}_{jk,m}$ and torsion-free symmetrization are in `christoffel.py`.
- **Geodesic acceleration:** Eq. 2 is computed in `integrators.py` (function `geodesic_accel`).
- **Integrator:** The discrete update in Eq. 8 is implemented in `rollout_geodesic_resnet` with optional manifold projections.
- **Metric reconstruction:** Eq. 9 is discretized in `metric_recon.py`.
- **Losses:** Symmetry, determinant barrier, and signature penalties are in `losses.py`.
- **Training:** Orchestrated in `train.py`, including data loading, velocity initialization, rollout, and loss aggregation.