Neural Network Track Extrapolation for the LHCb Upgrade II: A Physics-Informed Machine Learning Approach

George William Scriven

Gravitational Waves and Fundamental Physics (GWFP), Maastricht University, The Netherlands

Centre for Mathematical Analysis and Topology (CMAT), UHasselt, Belgium

Nikhef, National Institute for Subatomic Physics, Amsterdam, The Netherlands

george.scriven@maastrichtuniversity.nl

gscriven@nikhef.nl

george.scriven@uhasselt.be

January 2026

Abstract

The LHCb Upgrade II, scheduled for commissioning during Long Shutdown 4 (LS4, 2033-2034), will increase the instantaneous luminosity to 1.5×10^{34} cm^{-2}s^{-1}, presenting unprecedented computational challenges for real-time track reconstruction. Track extrapolation through the detector's magnetic field is a critical computational bottleneck in the High Level Trigger (HLT), which must process events at 30 MHz using a fully software-based trigger. This work presents a proof-of-concept neural network approach to replace traditional Runge-Kutta numerical integration with fast, accurate machine learning models. We design four neural network architectures: standard Multi-Layer Perceptrons (MLPs), Residual MLPs with physics-based skip connections, Physics-Informed Neural Networks (PINNs), and a novel Runge-Kutta-inspired PINN (RK-PINN) with multi-stage prediction. In this study, **MLP and RK-PINN architectures were trained and evaluated**; Residual MLP and PINN are planned for future work. Training on 50 million simulated tracks using a simplified analytical magnetic field model, our best RK-PINN model (`rkpinn_wide`) achieves a mean position error of 18.0 μm with inference time of 3.96 μs per track. The fastest model (`mlp_small`) achieves 0.84 μs inference with 53.5 μm accuracy—2.9× faster than the BogackiShampine3 C++ extrapolator (2.40 μs). With the simplified toy field used in this study, neural networks achieve modest speedup (1–3×) over C++ extrapolators. **Important caveat:** This study uses a simplified toy magnetic field; deployment in LHCb requires retraining with the full field map. The significant speedup required for Upgrade II triggers is expected when neural networks replace expensive 3D field interpolation in production.

Contents

1 Introduction

1.1 The Large Hadron Collider and LHCb Experiment

The Large Hadron Collider (LHC) at CERN is the world's largest and most powerful particle accelerator, colliding protons at center-of-mass energies up to $\sqrt{s} = 13.6$ TeV [1]. The LHCb experiment is one of four major experiments at the LHC, specifically designed as a forward spectrometer to study CP violation and rare decays in the beauty and charm quark sectors [2].

LHCb operates as a single-arm forward spectrometer covering the pseudorapidity range $2 < \eta < 5$, optimized for detecting particles containing b and c quarks which are predominantly produced in the forward direction at hadron colliders. The detector comprises several subsystems [2]:

- **VErtex LOcator (VELO):** Silicon pixel detector providing precise vertex reconstruction
- **Tracking stations (UT, SciFi):** Upstream Tracker and Scintillating Fibre tracker for momentum measurement
- **Dipole magnet:** Warm dipole providing an integrated field of ~ 4 Tm for momentum measurement
- **RICH detectors:** Ring Imaging Cherenkov detectors for particle identification
- **Calorimeters and muon system:** For energy measurement and muon identification

1.2 The LHCb Upgrade II Programme

The LHCb Upgrade II is a major enhancement planned for installation during Long Shutdown 4 (LS4, 2033-2034) [3]. The key parameters of the upgrade are:

- **Instantaneous luminosity:** $\mathcal{L} = 1.5 \times 10^{34}$ cm^{-2}s^{-1} (7.5× Upgrade I)
- **Integrated luminosity target:** 300 fb^{-1} over the HL-LHC lifetime
- **Pile-up:** Average of $\mu \approx 42$ visible interactions per bunch crossing [3]
- **Bunch crossing rate:** 30 MHz

The increased luminosity presents significant computational challenges. The LHCb trigger system, which after Upgrade I became the first fully software-based trigger at a hadron collider [4], must process $\mathcal{O}(10^8)$ events per second. Track reconstruction constitutes approximately 70% of the HLT1 processing time [5], making any speedup in tracking algorithms directly impactful for the experiment's physics reach.

1.3 Track Extrapolation: The Computational Challenge

Track extrapolation—propagating a charged particle's trajectory through the magnetic field from one detector plane to another—is a fundamental operation in track reconstruction. Given an initial track state:

$$\mathbf{s}(z_0) = (x, y, t_x, t_y, q/p) \tag{1}$$

where (x, y) is the position, $t_x = dx/dz$ and $t_y = dy/dz$ are the track slopes, q is the particle charge, and p is the momentum, the extrapolation task is to predict the state at a target z-position:

$$\mathbf{s}(z_1) = \mathcal{E}[\mathbf{s}(z_0); z_0 \to z_1; \mathbf{B}(\mathbf{r})] \tag{2}$$

Traditional extrapolation uses numerical integration of the equations of motion (typically fourth-order Runge-Kutta methods), requiring multiple evaluations of the magnetic field map per integration step. Each field evaluation involves 3D interpolation of tabulated field values, making extrapolation computationally expensive [8].

1.4 Scope and Limitations of This Work

Important: This work is a **proof-of-concept study** using a simplified magnetic field model. The key limitations are:

1. **Simplified magnetic field:** We use an analytical Gaussian-profile dipole field approximation rather than the full LHCb field map from the detector simulation. The real field map includes complex fringe fields, iron yoke effects, and detailed 3D structure that are not captured by our model.

2. **No material effects:** Multiple scattering and energy loss in detector material are not included in the training data.

3. **Single propagation distance:** Models are trained for fixed $\Delta z = 2300$ mm.

4. **Validation only:** Results demonstrate feasibility but have not been validated against full LHCb simulation or real data.

2 Mathematical Preliminaries

We establish the mathematical framework and notation used throughout this paper.

2.1 Function Spaces and Norms

Definition 2.1 (Track State Space)**.** Let $\mathcal{S} \subset \mathbb{R}^5$ denote the **track state space**, where each state $\mathbf{s} = (x, y, t_x, t_y, \kappa) \in \mathcal{S}$ consists of:

- Position coordinates $(x, y) \in \mathbb{R}^2$ (measured in mm)
- Directional slopes $(t_x, t_y) := (dx/dz, dy/dz) \in \mathbb{R}^2$ (dimensionless)
- Curvature parameter $\kappa := q/(pc) \in \mathbb{R}$ ($\mathrm{T}^{-1}\mathrm{m}^{-1}$)

The state space is equipped with the Euclidean norm $\|\mathbf{s}\|_2 = \sqrt{x^2 + y^2 + t_x^2 + t_y^2 + \kappa^2}$.

Definition 2.2 (Input and Output Spaces)**.** The **input space** for our neural network models is:

$$\mathcal{X} := \mathcal{S} \times \mathbb{R}^+ = \{(\mathbf{s}, \Delta z) : \mathbf{s} \in \mathcal{S}, \Delta z > 0\} \subset \mathbb{R}^6 \tag{3}$$

The **output space** is the position-slope subspace:

$$\mathcal{Y} := \{(x', y', t'_x, t'_y) \in \mathbb{R}^4\} \tag{4}$$

Definition 2.3 (Sobolev Spaces)**.** For $k \in \mathbb{N}$ and $p \in [1, \infty]$, the Sobolev space $W^{k,p}(\Omega)$ consists of functions $f : \Omega \to \mathbb{R}$ whose weak derivatives up to order k exist and belong to $L^p(\Omega)$:

$$W^{k,p}(\Omega) := \{f \in L^p(\Omega) : D^\alpha f \in L^p(\Omega) \text{ for all } |\alpha| \leq k\} \tag{5}$$

with norm $\|f\|_{W^{k,p}} := \left(\sum_{|\alpha| \leq k} \|D^\alpha f\|_{L^p}^p\right)^{1/p}$.

2.2 The Track Extrapolation Problem

Definition 2.4 (Extrapolation Operator)**.** The **exact extrapolation operator** $\mathcal{E} : \mathcal{X} \to \mathcal{Y}$ is defined as the solution map of the initial value problem:

$$\mathcal{E}(\mathbf{s}_0, \Delta z) := \mathbf{s}(z_0 + \Delta z) \quad \text{where } \frac{d\mathbf{s}}{dz} = \mathbf{F}(\mathbf{s}, z), \quad \mathbf{s}(z_0) = \mathbf{s}_0 \tag{6}$$

and $\mathbf{F} : \mathcal{S} \times \mathbb{R} \to \mathbb{R}^4$ encodes the equations of motion.

Theorem 2.5 (Existence and Uniqueness)**.** *Let $\mathbf{B} : \mathbb{R}^3 \to \mathbb{R}^3$ be the magnetic field satisfying $\mathbf{B} \in C^1(\mathbb{R}^3)$ with bounded derivatives. Then for any initial condition $\mathbf{s}_0 \in \mathcal{S}$ and propagation distance $\Delta z > 0$, the extrapolation operator $\mathcal{E}(\mathbf{s}_0, \Delta z)$ exists, is unique, and depends continuously on the initial data.*

Proof. The right-hand side $\mathbf{F}(\mathbf{s}, z)$ of the ODE system is locally Lipschitz in $\mathbf{s}$ when $\mathbf{B} \in C^1$. Specifically, for the equations of motion (Eq. 14), we have:

$$\|\mathbf{F}(\mathbf{s}_1, z) - \mathbf{F}(\mathbf{s}_2, z)\| \leq L\|\mathbf{s}_1 - \mathbf{s}_2\| \tag{7}$$

where the Lipschitz constant L depends on $\|\mathbf{B}\|_{C^1}$ and the bounds on the state space. The result follows from the Picard-Lindelöf theorem [9]. □

Definition 2.6 (Neural Network Approximator)**.** A **neural network extrapolator** is a parametric function $f_\theta : \mathcal{X} \to \mathcal{Y}$ with parameters $\theta \in \Theta \subset \mathbb{R}^p$ that approximates the exact extrapolation operator:

$$f_\theta(\mathbf{s}_0, \Delta z) \approx \mathcal{E}(\mathbf{s}_0, \Delta z) \tag{8}$$

The **approximation error** is measured in the L^2 sense over a probability distribution μ on $\mathcal{X}$:

$$\mathcal{R}(f_\theta) := \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mu}\left[\|f_\theta(\mathbf{x}) - \mathbf{y}\|_2^2\right] \tag{9}$$

3 Theoretical Framework

3.1 Equations of Motion for Charged Particles

The motion of a relativistic charged particle in an electromagnetic field is governed by the Lorentz force equation [6]:

$$\frac{d\mathbf{p}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \tag{10}$$

where $\mathbf{p} = \gamma m \mathbf{v}$ is the relativistic momentum, q is the charge, and $\mathbf{E}$, $\mathbf{B}$ are the electric and magnetic fields.

In particle physics detectors, electric fields are typically negligible in the tracking volume, and we use z (the beam direction) as the independent variable. The equations of motion become [7]:

$$\frac{dx}{dz} = t_x \tag{11}$$

$$\frac{dy}{dz} = t_y \tag{12}$$

$$\frac{dt_x}{dz} = \kappa\sqrt{1 + t_x^2 + t_y^2}\left[t_x t_y B_x - (1 + t_x^2)B_y + t_y B_z\right] \tag{13}$$

$$\frac{dt_y}{dz} = \kappa\sqrt{1 + t_x^2 + t_y^2}\left[(1 + t_y^2)B_x - t_x t_y B_y - t_x B_z\right] \tag{14}$$

where $\kappa = q/(pc) \approx 0.3 \cdot q/(p[\text{GeV}])$ $\text{T}^{-1}\text{m}^{-1}$ encodes the charge-to-momentum ratio.

3.2 Simplified Magnetic Field Model (Toy Model)

For this proof-of-concept study, we use a simplified analytical field model:

$$B_y(x, y, z) = B_0 \cdot \text{pol} \cdot \exp\left(-\frac{1}{2}\left(\frac{z - z_c}{\sigma_z}\right)^2\right) \cdot \left(1 - 10^{-4}\frac{r_\perp^2}{1000^2}\right) \tag{15}$$

where $B_0 = 1.0$ T is the peak field strength, $\text{pol} = \pm 1$ is the polarity, $z_c = 5250$ mm is the magnet center, and $\sigma_z = 2500$ mm is the characteristic width. This Gaussian profile captures the qualitative behavior of the LHCb dipole but lacks detailed 3D structure, iron yoke effects, and accurate fringe fields.

3.3 Numerical Integration: Runge-Kutta Methods

Traditional track extrapolation uses adaptive Runge-Kutta integration [9]. The classical fourth-order method (RK4) computes:

$$k_1 = f(\mathbf{s}_n, z_n) \tag{16}$$

$$k_2 = f(\mathbf{s}_n + \tfrac{h}{2}k_1, z_n + \tfrac{h}{2}) \tag{17}$$

$$k_3 = f(\mathbf{s}_n + \tfrac{h}{2}k_2, z_n + \tfrac{h}{2}) \tag{18}$$

$$k_4 = f(\mathbf{s}_n + hk_3, z_n + h) \tag{19}$$

$$\mathbf{s}_{n+1} = \mathbf{s}_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(h^5) \tag{20}$$

The computational cost comes from the four evaluations of $f(\cdot)$ per step, each requiring magnetic field interpolation. This structure—evaluating the system at intermediate points and combining with specific weights—will directly inspire our RK-PINN architecture (Section 4.5).

4 Neural Network Architecture Foundations

This section provides rigorous justification for each architectural choice in our models, with references to foundational machine learning literature.

4.1 Formal Definition of Neural Network Extrapolators

We begin by formally defining each architecture as a parametric map, specifying domains, codomains, and the functional form of each mapping.

4.1.1 Common Domain and Codomain

All architectures share the same input-output structure:

Definition 4.1 (Common Input Domain)**.** The input domain for all extrapolator architectures is:

$$\mathcal{X} := \underbrace{[-1000, 1000]^2}_{\text{position } (x,y) \text{ [mm]}} \times \underbrace{[-0.3, 0.3]^2}_{\text{slopes } (t_x,t_y)} \times \underbrace{[-2, 2]}_{\kappa=q/p \text{ [c/GeV]}} \times \underbrace{\{2300\}}_{\Delta z \text{ [mm]}} \subset \mathbb{R}^6 \tag{21}$$

We denote a generic input as $\mathbf{x} = (x, y, t_x, t_y, \kappa, \Delta z) \in \mathcal{X}$.

Definition 4.2 (Common Output Codomain)**.** The output codomain for all extrapolator architectures is:

$$\mathcal{Y} := \underbrace{[-2000, 2000]^2}_{\text{position } (x',y') \text{ [mm]}} \times \underbrace{[-0.5, 0.5]^2}_{\text{slopes } (t'_x,t'_y)} \subset \mathbb{R}^4 \tag{22}$$

We denote a generic output as $\mathbf{y} = (x', y', t'_x, t'_y) \in \mathcal{Y}$.

Definition 4.3 (Track Extrapolation as a Mapping Problem)**.** The goal is to learn a parametric map $f_\theta : \mathcal{X} \to \mathcal{Y}$ that approximates the exact extrapolation operator $\mathcal{E} : \mathcal{X} \to \mathcal{Y}$ defined by the solution of the equations of motion (Eq. 14).

4.1.2 Multi-Layer Perceptron (MLP) — Direct Mapping

Definition 4.4 (MLP Extrapolator)**.** An **MLP extrapolator** with L hidden layers is a map $f_\theta^{\text{MLP}} : \mathcal{X} \to \mathcal{Y}$ defined by the composition:

$$f_\theta^{\text{MLP}} := \mathbf{A}^{(L+1)} \circ \sigma \circ \mathbf{A}^{(L)} \circ \sigma \circ \cdots \circ \sigma \circ \mathbf{A}^{(1)} \tag{23}$$

where:

- $\mathbf{A}^{(l)}(\mathbf{h}) := \mathbf{W}^{(l)}\mathbf{h} + \mathbf{b}^{(l)}$ are affine transformations with $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$, $\mathbf{b}^{(l)} \in \mathbb{R}^{d_l}$
- $\sigma : \mathbb{R} \to \mathbb{R}$ is the SiLU activation applied component-wise
- $d_0 = 6$ (input dimension), $d_{L+1} = 4$ (output dimension)

- The parameter space is $\Theta = \prod_{l=1}^{L+1} \left(\mathbb{R}^{d_l \times d_{l-1}} \times \mathbb{R}^{d_l}\right)$

The total parameter count is $|\theta| = \sum_{l=1}^{L+1} d_l(d_{l-1} + 1)$.

Example 4.5 (MLP-Medium Configuration)**.** For `mlp_medium` with layers $[128, 128, 64]$:

$$f_\theta^{\text{MLP}} : \mathbb{R}^6 \xrightarrow{\mathbf{A}^{(1)}} \mathbb{R}^{128} \xrightarrow{\sigma} \mathbb{R}^{128} \xrightarrow{\mathbf{A}^{(2)}} \mathbb{R}^{128} \xrightarrow{\sigma} \mathbb{R}^{128} \tag{24}$$

$$\xrightarrow{\mathbf{A}^{(3)}} \mathbb{R}^{64} \xrightarrow{\sigma} \mathbb{R}^{64} \xrightarrow{\mathbf{A}^{(4)}} \mathbb{R}^4 \tag{25}$$

Parameter count: $6 \cdot 128 + 128 + 128 \cdot 128 + 128 + 128 \cdot 64 + 64 + 64 \cdot 4 + 4 = 26,052$.

4.1.3 Residual MLP — Correction Mapping (Planned)

Definition 4.6 (Residual MLP Extrapolator)**.** A **Residual MLP extrapolator** is a map $f_\theta^{\text{Res}} : \mathcal{X} \to \mathcal{Y}$ defined as:

$$f_\theta^{\text{Res}}(\mathbf{x}) := \underbrace{\Phi(\mathbf{x})}_{\text{physics baseline}} + \underbrace{g_\theta(\mathbf{x})}_{\text{learned correction}} \tag{26}$$

where:

- The **physics baseline** $\Phi : \mathcal{X} \to \mathcal{Y}$ encodes straight-line propagation:

$$\Phi(x, y, t_x, t_y, \kappa, \Delta z) := (x + t_x \Delta z,\ y + t_y \Delta z,\ t_x,\ t_y) \tag{27}$$

- The **correction network** $g_\theta : \mathcal{X} \to \mathbb{R}^4$ is an MLP learning only the magnetic deflection

Remark 4.7 (Residual Learning Principle). The decomposition $f = \Phi + g$ is motivated by the observation that magnetic deflections are $\mathcal{O}(100$ mm$)$ while total propagation is $\mathcal{O}(2000$ mm$)$. Learning the smaller correction g requires less representational capacity than learning the full map f.

4.1.4 True PINN — PDE-Constrained Mapping (Planned)

Definition 4.8 (True Physics-Informed Neural Network)**.** A **True PINN extrapolator** learns a continuous trajectory $\hat{\mathbf{s}}_\theta(z) : [z_0, z_0 + \Delta z] \to \mathcal{S}$ that satisfies the Lorentz force equations. Unlike soft-constraint approaches, a True PINN uses **automatic differentiation** to enforce the governing PDEs.

The network predicts the state at any z along the trajectory:

$$\hat{\mathbf{s}}_\theta : \mathcal{S} \times \mathbb{R}^+ \times \mathbb{R} \to \mathcal{S}, \quad (\mathbf{s}_0, \kappa, z) \mapsto \hat{\mathbf{s}}(z) \tag{28}$$

Definition 4.9 (True PINN Loss Function)**.** The True PINN loss combines boundary conditions with PDE residuals:

$$\mathcal{L}^{\text{PINN}}(\theta) := \underbrace{\mathcal{L}_{\text{data}}(\theta)}_{\text{boundary conditions}} + \underbrace{\lambda \mathcal{L}_{\text{PDE}}(\theta)}_{\text{physics residual}} \tag{29}$$

The **data loss** enforces boundary conditions at the endpoints:

$$\mathcal{L}_{\text{data}}(\theta) := \frac{1}{N} \sum_{i=1}^{N} \left\| \hat{\mathbf{s}}_\theta(\mathbf{s}_0^{(i)}, \kappa^{(i)}, z_{\text{end}}) - \mathbf{s}_{\text{end}}^{(i)} \right\|_2^2 \tag{30}$$

The **PDE loss** enforces the Lorentz equations at collocation points via automatic differentiation:

$$\mathcal{L}_{\text{PDE}}(\theta) := \frac{1}{N_c} \sum_{j=1}^{N_c} \left\| \left. \frac{\partial \hat{\mathbf{s}}_\theta}{\partial z} \right|_{z_j} - \mathbf{F}(\hat{\mathbf{s}}_\theta(z_j), \mathbf{B}(z_j)) \right\|_2^2 \tag{31}$$

where $\mathbf{F}$ is the Lorentz force right-hand side from Eq. (14).

Remark 4.10 (Automatic Differentiation). The key distinction of a True PINN is that we compute $\partial \hat{\mathbf{s}}/\partial z$ using `torch.autograd.grad()`, then compare to the expected derivative from the Lorentz equations. This enforces that the learned trajectory actually satisfies the governing physics, not just produces outputs that "look reasonable."

Remark 4.11 (Collocation Points). We sample N_c collocation points uniformly along each trajectory: $z_j \sim \mathcal{U}(z_0, z_0 + \Delta z)$. At each point, we evaluate the PDE residual. This provides physics regularization throughout the domain, not just at boundaries.

4.1.5 RK-PINN — Multi-Stage Prediction Mapping

Definition 4.12 (RK-PINN Extrapolator)**.** An **RK-PINN extrapolator** is a map $f_\theta^{\text{RK}} : \mathcal{X} \to \mathcal{Y}$ with multi-stage structure:

$$f_\theta^{\text{RK}}(\mathbf{x}) := \pi(\mathbf{x}) + \sum_{k=1}^{4} w_k(\boldsymbol{\alpha}) \cdot H_k(\phi(\mathbf{x})) \tag{32}$$

where:

- $\pi : \mathcal{X} \to \mathcal{Y}$ is the identity projection: $\pi(x, y, t_x, t_y, \kappa, \Delta z) := (x, y, t_x, t_y)$
- $\phi : \mathcal{X} \to \mathbb{R}^d$ is the **trunk network** (shared feature extractor), an MLP
- $H_k : \mathbb{R}^d \to \mathbb{R}^4$ for $k \in \{1, 2, 3, 4\}$ are four **prediction heads** (linear maps)
- $w_k : \mathbb{R}^4 \to (0, 1)$ are **softmax weights**: $w_k(\boldsymbol{\alpha}) = \exp(\alpha_k) / \sum_j \exp(\alpha_j)$
- $\boldsymbol{\alpha} \in \mathbb{R}^4$ are learnable logits initialized to $\boldsymbol{\alpha}^{(0)} = (\log \frac{1}{6}, \log \frac{1}{3}, \log \frac{1}{3}, \log \frac{1}{6})$

Remark 4.13 (RK4 Analogy). The four heads $\{H_k\}_{k=1}^4$ are analogous to the four stages $\{k_i\}_{i=1}^4$ in classical RK4, and the weights $(1/6, 1/3, 1/3, 1/6)$ are the standard RK4 Butcher tableau coefficients. Unlike classical RK4, the heads operate in parallel on shared features rather than sequentially.

Proposition 4.14 (RK-PINN Parameter Count)**.** *For trunk architecture $[d_1, \ldots, d_L]$ with final hidden dimension d_L, the RK-PINN has:*

$$|\theta| = \underbrace{|\theta_{trunk}|}_{\text{trunk params}} + \underbrace{4 \cdot d_L \cdot 4}_{\text{4 heads} \times \text{output dim}} + \underbrace{4}_{\text{weight logits}} \tag{33}$$

4.1.6 Summary: Architecture Comparison as Maps

Table 1: Formal comparison of neural network extrapolator architectures

Architecture	Map Form	Domain	Codomain
MLP	$f_\theta = \mathbf{A}^{(L+1)} \circ \sigma \circ \cdots \circ \mathbf{A}^{(1)}$	$\mathcal{X} \subset \mathbb{R}^6$	$\mathcal{Y} \subset \mathbb{R}^4$
Residual MLP	$f_\theta = \Phi + g_\theta$	$\mathcal{X} \subset \mathbb{R}^6$	$\mathcal{Y} \subset \mathbb{R}^4$
PINN	$f_\theta = \text{MLP}$ (physics loss)	$\mathcal{X} \subset \mathbb{R}^6$	$\mathcal{Y} \subset \mathbb{R}^4$
RK-PINN	$f_\theta = \pi + \sum_k w_k H_k \circ \phi$	$\mathcal{X} \subset \mathbb{R}^6$	$\mathcal{Y} \subset \mathbb{R}^4$

Remark 4.15 (Implementation Status). In this work, **MLP** and **RK-PINN** architectures were fully implemented and trained. **Residual MLP** and **PINN** are proposed for future work; their formal definitions are included for completeness.

4.2 Multi-Layer Perceptron (MLP)

4.2.1 Universal Approximation Theory

The theoretical foundation for using neural networks as function approximators rests on the Universal Approximation Theorem. We state the theorem precisely:

Theorem 4.16 (Universal Approximation [10]). *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function (e.g., the sigmoid $\sigma(x) = 1/(1+e^{-x})$). Let $K \subset \mathbb{R}^n$ be compact. Then for any $f \in C(K, \mathbb{R}^m)$ and any $\varepsilon > 0$, there exists $N \in \mathbb{N}$, weights $\mathbf{W} \in \mathbb{R}^{m \times N}$, $\mathbf{V} \in \mathbb{R}^{N \times n}$, and biases $\mathbf{b} \in \mathbb{R}^N$, $\mathbf{c} \in \mathbb{R}^m$ such that:*

$$\sup_{\mathbf{x} \in K} \|f(\mathbf{x}) - (\mathbf{W}\sigma(\mathbf{V}\mathbf{x} + \mathbf{b}) + \mathbf{c})\| < \varepsilon \tag{34}$$

where σ is applied component-wise.

Remark 4.17. The theorem guarantees *existence* of an approximating network but provides no constructive bound on the required width N. For Lipschitz functions, Yarotsky [24] showed that ReLU networks can achieve ε-approximation with $\mathcal{O}(\varepsilon^{-n/s})$ parameters for functions in the Sobolev space $W^{s,\infty}$, where n is the input dimension and s is the smoothness.

Corollary 4.18 (Approximation of Track Extrapolation). *Since the exact extrapolation operator $\mathcal{E} : \mathcal{X} \to \mathcal{Y}$ is continuous (by Theorem 2.5) on the compact domain*

$$K := [-1000, 1000]^2 \times [-0.3, 0.3]^2 \times [-2, 2] \times \{2300\} \subset \mathcal{X} \tag{35}$$

there exists a neural network f_θ that approximates $\mathcal{E}$ to arbitrary precision on K.

For track extrapolation, we seek to approximate the mapping:

$$f_\theta : \mathbb{R}^6 \to \mathbb{R}^4, \quad (x, y, t_x, t_y, q/p, \Delta z) \mapsto (x', y', t'_x, t'_y) \tag{36}$$

4.2.2 Architecture Specification

Our MLP architecture follows the standard feedforward formulation:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right) \tag{37}$$

where $\mathbf{h}^{(l)}$ is the hidden representation at layer l, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are learnable weights and biases, and σ is the activation function.

4.2.3 SiLU/Swish Activation Function

We use the **Sigmoid Linear Unit (SiLU)**, also known as Swish, as our activation function. This choice is motivated by the systematic activation function search conducted by Ramachandran, Zoph, and Le [12], who used reinforcement learning to discover:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{38}$$

Why SiLU over ReLU?

- **Smoothness:** SiLU is continuously differentiable everywhere, unlike ReLU which has a discontinuous derivative at $x = 0$. This improves gradient flow during training [12].
- **Non-monotonicity:** SiLU is non-monotonic, allowing it to express more complex functions. For $x < 0$, the function can output small negative values before approaching zero.
- **Self-gating:** The $\sigma(x)$ term acts as a learned gate, adaptively scaling inputs based on their magnitude.
- **Empirical performance:** Ramachandran et al. [12] showed consistent improvements over ReLU on ImageNet (+0.6-0.9% top-1 accuracy) and various NLP tasks.

We also considered GELU (Gaussian Error Linear Unit) [13]:

$$\text{GELU}(x) = x \cdot \Phi(x) \approx 0.5x \left(1 + \tanh\left[\sqrt{2/\pi}(x + 0.044715x^3)\right]\right) \tag{39}$$

which has similar properties but is more computationally expensive. In our experiments, SiLU achieved 0.21 mm error compared to 0.63 mm for tanh and 0.77 mm for ReLU.

4.2.4 Width vs. Depth Trade-offs

We explore various width/depth configurations based on established principles [11]:

Table 2: MLP architecture variants with design rationale

Variant	Layers	Params	Design Rationale
Tiny	[64, 32]	3k	Minimal model for speed testing
Small	[128, 64]	14k	Baseline with moderate capacity
Medium	[128, 128, 64]	26k	Standard depth, proven effective
Large	[256, 256, 128]	105k	High capacity for complex mappings
Wide-Shallow	[256, 128]	51k	Tests width vs depth hypothesis
Deep	[128]×5, 64	43k	Tests depth benefits
Balanced	[192, 192, 96]	57k	Optimized via hyperparameter search

The "wide vs. deep" trade-off is an active research area. Wider networks can approximate functions more efficiently in some cases [14], while deeper networks can learn hierarchical representations [11]. Our experiments test both approaches.

4.3 Residual MLP: Skip Connections for Physics-Based Priors

Implementation Status: This architecture is *proposed but not yet trained*. We include the design here as future work.

4.3.1 Residual Learning Framework

The Residual MLP incorporates **skip connections** inspired by the ResNet architecture of He et al. [15]. The key insight from [15] is that learning residual functions $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$ is easier than learning the full mapping $\mathcal{H}(\mathbf{x})$ directly. This addresses the degradation problem in deep networks where adding layers can actually increase training error.

The standard residual block computes:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{40}$$

For track extrapolation, we adapt this principle with a **physics-informed skip connection**.

4.3.2 Physics-Based Residual Formulation

In the absence of a magnetic field, tracks propagate in straight lines:

$$x_{\text{straight}} = x_{\text{in}} + t_x \cdot \Delta z \tag{41}$$

$$y_{\text{straight}} = y_{\text{in}} + t_y \cdot \Delta z \tag{42}$$

$$t'_x = t_x, \quad t'_y = t_y \tag{43}$$

The magnetic field introduces corrections to this straight-line baseline. Our Residual MLP learns only these corrections:

$$\begin{pmatrix} x' \\ y' \\ t'_x \\ t'_y \end{pmatrix} = \underbrace{\begin{pmatrix} x + t_x \Delta z \\ y + t_y \Delta z \\ t_x \\ t_y \end{pmatrix}}_{\text{Physics baseline}} + \underbrace{\text{MLP}(x, y, t_x, t_y, q/p, \Delta z)}_{\text{Learned correction}} \tag{44}$$

Advantages of this formulation:

1. **Easier learning:** The network learns only the magnetic deflection ($\sim$100 mm for typical tracks), not the full propagation ($\sim$2000 mm).

2. **Physical initialization:** With zero network weights, the model predicts straight-line propagation—a sensible initialization.

3. **Gradient flow:** The skip connection provides direct gradient paths as in [15], improving optimization.

4. **Conservation bias:** For tracks with $q/p \approx 0$ (very high momentum), the network can easily learn to output near-zero corrections.

4.4 True Physics-Informed Neural Networks (PINNs)

Implementation Status: A True PINN architecture has been *implemented* but *not yet trained*. We include the theoretical framework and implementation details here.

4.4.1 Foundations of Physics-Informed Learning

Physics-Informed Neural Networks (PINNs) were introduced by Raissi, Perdikaris, and Karniadakis [16] as a framework for encoding physical laws into neural network training. The seminal paper [16] has over 12,000 citations and established a new paradigm for scientific machine learning.

For a system governed by differential equations:

$$\frac{d\mathbf{s}}{dz} = \mathbf{F}(\mathbf{s}, z), \quad z \in [z_0, z_{\text{end}}] \tag{45}$$

a PINN approximates the solution $\mathbf{s}(z)$ with a neural network $\hat{\mathbf{s}}_\theta(z)$ trained to minimize:

$$\mathcal{L} = \underbrace{\mathcal{L}_{\text{data}}}_{\text{Boundary conditions}} + \underbrace{\lambda \mathcal{L}_{\text{PDE}}}_{\text{Physics residual}} \tag{46}$$

4.4.2 True PINN: Automatic Differentiation for PDE Enforcement

The **key feature** of a True PINN is that the physics loss is computed via automatic differentiation. We do not simply constrain outputs; we enforce that the learned trajectory satisfies the governing equations.

Definition 4.19 (PDE Residual Loss)**.** Given the network prediction $\hat{\mathbf{s}}_\theta(z)$ and the Lorentz force equations (Eq. 14), the PDE residual at collocation point z_j is:

$$\mathbf{r}(z_j) := \left.\frac{\partial \hat{\mathbf{s}}_\theta}{\partial z}\right|_{z_j} - \mathbf{F}(\hat{\mathbf{s}}_\theta(z_j), \mathbf{B}(z_j)) \tag{47}$$

The derivative $\partial \hat{\mathbf{s}}_\theta / \partial z$ is computed using automatic differentiation (`torch.autograd.grad`).

Explicitly, for track extrapolation:

$$r_x(z) = \frac{\partial \hat{x}}{\partial z} - \hat{t}_x \tag{48}$$

$$r_y(z) = \frac{\partial \hat{y}}{\partial z} - \hat{t}_y \tag{49}$$

$$r_{t_x}(z) = \frac{\partial \hat{t}_x}{\partial z} - \kappa\sqrt{1 + \hat{t}_x^2 + \hat{t}_y^2}\left[\hat{t}_x \hat{t}_y B_x - (1 + \hat{t}_x^2) B_y + \hat{t}_y B_z\right] \tag{50}$$

$$r_{t_y}(z) = \frac{\partial \hat{t}_y}{\partial z} - \kappa\sqrt{1 + \hat{t}_x^2 + \hat{t}_y^2}\left[(1 + \hat{t}_y^2) B_x - \hat{t}_x \hat{t}_y B_y - \hat{t}_x B_z\right] \tag{51}$$

The total physics loss is:

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_c} \sum_{j=1}^{N_c} \left(r_x^2 + r_y^2 + r_{t_x}^2 + r_{t_y}^2 \right)_{z=z_j} \tag{52}$$

where $\{z_j\}_{j=1}^{N_c}$ are collocation points sampled uniformly along the trajectory.

4.4.3 Key Benefits of True PINN Approach

Karniadakis et al. [17] provide a comprehensive review of PINNs with key benefits:

1. **Data efficiency:** Physics constraints reduce the amount of training data required by providing strong regularization
2. **Generalization:** Models extrapolate better outside the training distribution
3. **Interpretability:** Predictions respect known physical laws
4. **Noise robustness:** Physics acts as a strong prior, filtering non-physical noise

4.4.4 Implementation with Differentiable Field Model

For the True PINN implementation, we embed an analytical magnetic field model directly in the computational graph. This enables automatic differentiation through the field evaluation:

Definition 4.20 (Differentiable Toy Field)**.** The magnetic field is modeled as:

$$B_y(z) = B_0 \cdot \exp\left(-\frac{1}{2}\left(\frac{z - z_c}{\sigma_z}\right)^2\right), \quad B_x \approx 0, \quad B_z \approx 0 \tag{53}$$

with $B_0 = 1$ T, $z_c = 5250$ mm, and $\sigma_z = 2500$ mm. This is implemented as a PyTorch module with all operations differentiable.

Remark 4.21 (Production Considerations). For deployment with the full LHCb field map, the differentiable field model would need to be replaced with a differentiable interpolation scheme, or the field values could be pre-computed at collocation points.

Total True PINN Loss:

$$\mathcal{L}_{\text{TruePINN}} = \mathcal{L}_{\text{data}} + \lambda_{\text{PDE}}\mathcal{L}_{\text{PDE}} \tag{54}$$

Typical hyperparameters: $\lambda_{\text{PDE}} = 1.0$, $N_c = 10$ collocation points per trajectory.

4.5 Runge-Kutta-Inspired PINN (RK-PINN)

Implementation Status: This architecture was *fully implemented and trained* with 10 model variants. The current RK-PINN uses soft t_y constraints; a True RK-PINN variant with autodiff physics loss has been implemented but not yet trained.

4.5.1 Motivation: Numerical Methods as Architectural Priors

Our novel contribution is the **RK-PINN** architecture, which incorporates the structure of Runge-Kutta integration directly into the network architecture. This is inspired by Neural ODEs [18], which showed that residual networks can be viewed as discretizations of continuous dynamical systems:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta) \quad \longleftrightarrow \quad \frac{d\mathbf{h}}{dt} = f(\mathbf{h}, \theta) \tag{55}$$

Chen et al. [18] demonstrated that this connection allows neural networks to be trained with adaptive ODE solvers, achieving state-of-the-art results with constant memory cost. We apply this insight in reverse: instead of training through an ODE solver, we design a network that mimics RK4 structure.

4.5.2 Formal Connection to Runge-Kutta Methods

We formalize the connection between classical numerical integration and our neural architecture.

Definition 4.22 (Butcher Tableau)**.** An s-stage explicit Runge-Kutta method is characterized by its Butcher tableau:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^\top \end{array} \quad \text{where} \quad A \in \mathbb{R}^{s\times s},\ \mathbf{b}, \mathbf{c} \in \mathbb{R}^s \tag{56}$$

For the classical RK4 method:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{57}$$

Definition 4.23 (RK4 Stage Equations)**.** Given an ODE $\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$ and step size h, the RK4 stages are:

$$\mathbf{k}_1 = \mathbf{f}(t_n, \mathbf{y}_n) \tag{58}$$
$$\mathbf{k}_2 = \mathbf{f}(t_n + \tfrac{h}{2}, \mathbf{y}_n + \tfrac{h}{2}\mathbf{k}_1) \tag{59}$$
$$\mathbf{k}_3 = \mathbf{f}(t_n + \tfrac{h}{2}, \mathbf{y}_n + \tfrac{h}{2}\mathbf{k}_2) \tag{60}$$
$$\mathbf{k}_4 = \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_3) \tag{61}$$

with final update $\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$.

Theorem 4.24 (RK4 Local Truncation Error)**.** *The RK4 method has local truncation error $\mathcal{O}(h^5)$ and global error $\mathcal{O}(h^4)$, i.e., it is a fourth-order method.*

4.5.3 Multi-Stage Prediction Architecture

Our RK-PINN mimics this structure with a shared feature extractor (trunk) and multiple prediction heads:

$$\phi(\mathbf{x}) = \text{TrunkNetwork}(\mathbf{x}) \in \mathbb{R}^d \tag{62}$$

Definition 4.25 (RK-PINN Multi-Stage Predictor)**.** The RK-PINN defines four prediction heads $\text{Head}_k : \mathbb{R}^d \to \mathbb{R}^4$ for $k \in \{1, 2, 3, 4\}$:

$$\Delta\mathbf{s}_1 = \text{Head}_1(\phi(\mathbf{x})) \quad (\text{analogous to } \mathbf{k}_1) \tag{63}$$
$$\Delta\mathbf{s}_2 = \text{Head}_2(\phi(\mathbf{x})) \quad (\text{analogous to } \mathbf{k}_2) \tag{64}$$
$$\Delta\mathbf{s}_3 = \text{Head}_3(\phi(\mathbf{x})) \quad (\text{analogous to } \mathbf{k}_3) \tag{65}$$
$$\Delta\mathbf{s}_4 = \text{Head}_4(\phi(\mathbf{x})) \quad (\text{analogous to } \mathbf{k}_4) \tag{66}$$

Remark 4.26. Unlike classical RK4 where each $\mathbf{k}_i$ depends on previous stages, our heads operate in parallel on the same feature representation. This architectural choice enables efficient batched inference while retaining the multi-scale prediction structure.

4.5.4 Learnable Combination Weights

Instead of fixed RK4 weights, we use learnable weights initialized to RK4 coefficients:

Definition 4.27 (Adaptive Weight Combination)**.** The final prediction combines stages via learnable weights $\boldsymbol{\alpha} \in \mathbb{R}^4$:

$$\hat{\mathbf{s}}_{\text{out}} = \mathbf{s}_{\text{in}} + \sum_{k=1}^{4} w_k(\boldsymbol{\alpha})\, \Delta\mathbf{s}_k \tag{67}$$

with initialization $\boldsymbol{\alpha}^{(0)}$ chosen such that $w_k^{(0)} = (1/6, 1/3, 1/3, 1/6)$.

The weights are passed through a softmax to ensure they form a probability distribution:

$$w_k(\boldsymbol{\alpha}) = \frac{\exp(\alpha_k)}{\sum_{j=1}^{4} \exp(\alpha_j)} \in (0, 1), \quad \sum_{k=1}^{4} w_k = 1 \tag{68}$$

where α_k are learnable parameters.

4.5.5 Architectural Benefits

1. **Inductive bias:** The multi-stage structure provides a strong prior from numerical integration theory
2. **Gradient flow:** Multiple prediction paths improve gradient flow during training, similar to DenseNet [19]
3. **Implicit ensembling:** The weighted combination of heads acts as an implicit ensemble
4. **Interpretability:** Final weights indicate which "RK stages" are most important for the learned solution

4.5.6 Complete RK-PINN Loss Function

The RK-PINN combines the multi-stage architecture with physics constraints:

$$\mathcal{L}_{\text{RK-PINN}} = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2}_{\text{Data loss}} + \lambda_{t_y} \underbrace{\frac{1}{N}\sum_{i=1}^{N} (t_{y,\text{out}}^{(i)} - t_{y,\text{in}}^{(i)})^2}_{\text{Physics loss}} \tag{69}$$

5 Prediction Strategies

We explore two fundamentally different prediction strategies, each with theoretical motivation.

5.1 Direct Prediction

The simplest approach directly maps input to output:

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) \quad \text{where} \quad \mathbf{x} = (x, y, t_x, t_y, q/p, \Delta z) \tag{70}$$

Advantages:

- Minimal architectural complexity
- Network can learn optimal representation without constraints
- Fastest inference (single forward pass)

Disadvantages:

- Must learn full propagation from scratch
- No physical priors encoded in architecture
- May struggle with conservation laws

5.2 Residual Prediction

As described in Section 4.3, residual prediction learns corrections to a physics baseline:

$$\hat{\mathbf{y}} = \mathbf{y}_{\text{baseline}} + f_\theta(\mathbf{x}) \tag{71}$$

This follows the principle from [15] that learning residuals is easier than learning full mappings. For track extrapolation:

$$\mathbf{y}_{\text{baseline}} = (x + t_x \Delta z, y + t_y \Delta z, t_x, t_y) \tag{72}$$

5.3 Multi-Stage Prediction (RK-PINN)

The RK-PINN uses multiple prediction heads combined with learnable weights:

$$\hat{\mathbf{y}} = \mathbf{s}_{\text{in}} + \sum_{k=1}^{4} w_k \cdot \text{Head}_k(\phi(\mathbf{x})) \tag{73}$$

This is analogous to ensemble methods [20], where combining multiple predictions typically outperforms individual predictors.

6 Loss Functions and Optimization

We formalize the training objective using the framework of statistical learning theory.

6.1 Empirical Risk Minimization

Definition 6.1 (Training Set)**.** Let $\mathcal{D}_N = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ be a training set of N i.i.d. samples from the data distribution $\mathcal{P}(\mathbf{x}, \mathbf{y})$ over $\mathcal{X} \times \mathcal{Y}$.

Definition 6.2 (Loss Function)**.** A loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ is a measurable function quantifying the discrepancy between predicted and true outputs. The loss is *proper* if $\ell(\mathbf{y}, \mathbf{y}) = 0$ for all $\mathbf{y} \in \mathcal{Y}$.

Definition 6.3 (Risk Functional)**.** The *population risk* (expected loss) of a predictor $f : \mathcal{X} \to \mathcal{Y}$ is:

$$R(f) := \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{P}} \left[\ell(f(\mathbf{x}), \mathbf{y})\right] \tag{74}$$

The *empirical risk* (training loss) is the sample approximation:

$$\hat{R}_N(f) := \frac{1}{N} \sum_{i=1}^{N} \ell(f(\mathbf{x}_i), \mathbf{y}_i) \tag{75}$$

Definition 6.4 (Empirical Risk Minimization)**.** Given a hypothesis class $\mathcal{H} \subset \{f : \mathcal{X} \to \mathcal{Y}\}$ and training set $\mathcal{D}_N$, the empirical risk minimization (ERM) problem is:

$$\hat{f}_N := \arg\min_{f\in\mathcal{H}} \hat{R}_N(f) \tag{76}$$

For neural networks, $\mathcal{H} = \{f_\theta : \theta \in \Theta\}$ where $\Theta \subset \mathbb{R}^d$ is the parameter space.

6.2 Loss Functions for Track Extrapolation

6.2.1 Mean Squared Error (MSE)

For regression problems, the canonical choice is the L^2 loss:

Definition 6.5 (MSE Loss)**.** The mean squared error loss for track extrapolation is:

$$\ell_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) := \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 = \sum_{j=1}^{4} (\hat{y}_j - y_j)^2 \tag{77}$$

This corresponds to the Gaussian log-likelihood under homoscedastic noise assumption.

Proposition 6.6 (MSE Minimizer)**.** *Under the model* $\mathbf{y} = f^*(\mathbf{x}) + \boldsymbol{\varepsilon}$ *where* $\mathbb{E}[\boldsymbol{\varepsilon}|\mathbf{x}] = \mathbf{0}$*, the MSE risk minimizer is the conditional expectation:*

$$f^* = \arg\min_f R_{MSE}(f) = \mathbb{E}[\mathbf{y}|\mathbf{x}] \tag{78}$$

6.2.2 Component-Weighted Loss

Track states have heterogeneous units and scales. We introduce dimension-aware weighting:

Definition 6.7 (Weighted MSE Loss)**.**

$$\ell_{\text{wMSE}}(\hat{\mathbf{y}}, \mathbf{y}) := \sum_{j=1}^{4} w_j(\hat{y}_j - y_j)^2 = (\hat{\mathbf{y}} - \mathbf{y})^\top \mathbf{W}(\hat{\mathbf{y}} - \mathbf{y}) \tag{79}$$

where $\mathbf{W} = \text{diag}(w_1, w_2, w_3, w_4)$ is a positive-definite weight matrix. In our implementation:

$$w_x = w_y = 1 \quad \text{(position in mm)} \tag{80}$$

$$w_{t_x} = w_{t_y} = 1000^2 \quad \text{(slopes are dimensionless, scale to mm)} \tag{81}$$

6.2.3 Physics-Constrained Loss

Definition 6.8 (Composite Physics Loss)**.** For physics-informed training, we define the composite loss:

$$\mathcal{L}(\theta) := \mathcal{L}_{\text{data}}(\theta) + \sum_{k=1}^{K} \lambda_k \mathcal{L}_{\text{phys}}^{(k)}(\theta) \tag{82}$$

where $\mathcal{L}_{\text{data}}$ is the data fidelity term, $\mathcal{L}_{\text{phys}}^{(k)}$ are physics constraint terms, and $\lambda_k > 0$ are regularization hyperparameters.

For the RK-PINN architecture, the physics constraint enforces slope conservation:

$$\mathcal{L}_{\text{phys}}^{(t_y)}(\theta) := \frac{1}{N} \sum_{i=1}^{N} \left(f_\theta(\mathbf{x}_i)_{t_y} - (\mathbf{x}_i)_{t_y} \right)^2 \tag{83}$$

6.3 Regularization

Definition 6.9 (Tikhonov Regularization)**.** To prevent overfitting, we add L^2 parameter regularization (weight decay):

$$\mathcal{L}_{\text{reg}}(\theta) := \mathcal{L}(\theta) + \frac{\lambda_{\text{wd}}}{2} \|\theta\|_2^2 \tag{84}$$

where $\lambda_{\text{wd}} = 10^{-5}$ in our experiments.

6.4 Optimization via Stochastic Gradient Descent

Definition 6.10 (Stochastic Gradient Descent)**.** Given minibatches $\mathcal{B}_t \subset \mathcal{D}_N$ of size B, SGD updates:

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \hat{R}_{\mathcal{B}_t}(\theta_t) \tag{85}$$

where $\eta_t > 0$ is the learning rate schedule and $\hat{R}_{\mathcal{B}_t}$ is the minibatch risk estimate.

We use the AdamW optimizer [22], which combines momentum with adaptive learning rates:

Definition 6.11 (AdamW Update Rule)**.** With hyperparameters $\beta_1, \beta_2 \in [0, 1)$, the AdamW update maintains running averages:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t \tag{86}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \tag{87}$$

$$\hat{\mathbf{m}}_t = \mathbf{m}_t/(1 - \beta_1^t), \quad \hat{\mathbf{v}}_t = \mathbf{v}_t/(1 - \beta_2^t) \tag{88}$$

$$\theta_{t+1} = (1 - \eta_t \lambda_{\text{wd}})\theta_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{89}$$

where $\mathbf{g}_t = \nabla_\theta \mathcal{L}(\theta_t)$ and $\epsilon = 10^{-8}$ for numerical stability.

Remark 6.12 (Learning Rate Schedule). We employ cosine annealing [23]:

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\frac{\pi t}{T}\right)\right) \tag{90}$$

with $\eta_{\max} = 10^{-3}$, $\eta_{\min} = 10^{-6}$, and $T = 100$ epochs.

7 Experimental Setup

7.1 Dataset Generation

We generate training data using RK4 integration with the simplified field model (Eq. 15):

Phase space sampling:

- Position: $x, y \in [-1000, 1000]$ mm (uniform)
- Slopes: $t_x, t_y \in [-0.3, 0.3]$ (uniform)
- Momentum: $p \in [0.5, 100]$ GeV (log-uniform)
- Charge: $q \in \{-1, +1\}$ (balanced)
- Propagation distance: $\Delta z = 2300$ mm (fixed)

Dataset statistics:

- Total tracks: 50 million
- Training/Validation/Test split: 90%/5%/5%
- File size: 4.0 GB (float32)

7.2 Model Configurations

Table 3 summarizes all 20 trained models:

Table 3: Complete model registry with architecture details

Model Name	Type	Hidden Layers	Params	Prediction
mlp_tiny_v1	MLP	[64, 32]	3k	Direct
mlp_small_v1	MLP	[128, 64]	14k	Direct
mlp_medium_v1	MLP	[128, 128, 64]	26k	Direct
mlp_large_v1	MLP	[256, 256, 128]	105k	Direct
mlp_xlarge_v1	MLP	[512, 512, 256, 128]	560k	Direct
mlp_wide_shallow_v1	MLP	[256, 128]	51k	Direct
mlp_wide_v1	MLP	[512, 256]	204k	Direct
mlp_deep_v1	MLP	[128]×5, 64	43k	Direct
mlp_narrow_deep_v1	MLP	[96]×5, 48	23k	Direct
mlp_balanced_v1	MLP	[192, 192, 96]	57k	Direct
rkpinn_tiny_v1	RK-PINN	[64, 32]	3k	Multi-stage
rkpinn_small_v1	RK-PINN	[128, 64]	14k	Multi-stage
rkpinn_medium_v1	RK-PINN	[128, 128, 64]	26k	Multi-stage
rkpinn_large_v1	RK-PINN	[256, 256, 128]	105k	Multi-stage
rkpinn_xlarge_v1	RK-PINN	[512, 512, 256, 128]	560k	Multi-stage
rkpinn_wide_shallow_v1	RK-PINN	[256, 128]	51k	Multi-stage
rkpinn_wide_v1	RK-PINN	[512, 256]	204k	Multi-stage
rkpinn_deep_v1	RK-PINN	[128]×5, 64	43k	Multi-stage
rkpinn_narrow_deep_v1	RK-PINN	[96]×5, 48	23k	Multi-stage
rkpinn_balanced_v1	RK-PINN	[192, 192, 96]	57k	Multi-stage

7.3 Training Infrastructure

Hardware:

- GPU cluster: NIKHEF computing facility
- GPUs: NVIDIA L40S (48GB Ada Lovelace architecture)
- Training time: 8-12 hours per model

Software:

- Framework: PyTorch 2.0.1 [21] with CUDA 11.8
- Mixed precision: torch.cuda.amp (FP16 forward, FP32 gradients)

Hyperparameters:

- Optimizer: AdamW [22] ($\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay 10^{-5})
- Learning rate: 10^{-3} with cosine annealing to 10^{-6}
- Batch size: 1024
- Epochs: 100 with early stopping (patience 20)

8 Results

8.1 Error Metrics and Statistical Analysis

We rigorously define the evaluation metrics used throughout this section.

Definition 8.1 (Position Error)**.** For a predicted state $\hat{\mathbf{y}} = (\hat{x}, \hat{y}, \hat{t}_x, \hat{t}_y)$ and ground truth $\mathbf{y} = (x, y, t_x, t_y)$, the position error is:

$$e_{\text{pos}} := \sqrt{(\hat{x} - x)^2 + (\hat{y} - y)^2} \quad [\text{mm}] \tag{91}$$

This is the Euclidean distance in the transverse plane.

Definition 8.2 (Angular Error)**.** The angular error measures the deviation in track direction:

$$e_{\text{ang}} := \sqrt{(\hat{t}_x - t_x)^2 + (\hat{t}_y - t_y)^2} \quad [\text{rad}] \tag{92}$$

For small angles, this approximates the opening angle between predicted and true directions.

Definition 8.3 (Sample Statistics)**.** Given a test set of M samples with errors $\{e_i\}_{i=1}^{M}$, we compute:

$$\bar{e} := \frac{1}{M} \sum_{i=1}^{M} e_i \quad \text{(sample mean)} \tag{93}$$

$$s_e := \sqrt{\frac{1}{M-1} \sum_{i=1}^{M} (e_i - \bar{e})^2} \quad \text{(sample standard deviation)} \tag{94}$$

$$\text{SE}(\bar{e}) := \frac{s_e}{\sqrt{M}} \quad \text{(standard error of the mean)} \tag{95}$$

Remark 8.4 (Statistical Significance)*.* With $M = 2.5 \times 10^6$ test samples, the standard error is approximately $\text{SE} \approx s_e/1581$, yielding sub-$\mu$m precision on mean estimates. All reported differences between models are statistically significant at the $p < 0.001$ level.

8.2 Position Accuracy

Table 4: Model accuracy results ranked by mean position error (from benchmark)

Rank	Model	Type	Mean (μm)	Time (μs)	Throughput (k/s)	Paran
1	rkpinn_wide	RK-PINN	**18.0**	3.96	253	533k
2	rkpinn_wide_shallow	RK-PINN	26.8	2.46	407	135k
3	mlp_xlarge	MLP	27.3	3.28	305	431k
4	mlp_wide_shallow	MLP	33.5	5.55	180	35k
5	rkpinn_small	RK-PINN	36.3	1.92	519	35k
6	rkpinn_large	RK-PINN	39.3	4.13	242	201k
7	mlp_small	MLP	53.5	0.84	1195	9k
8	mlp_wide	MLP	54.8	1.44	694	136k
9	mlp_deep	MLP	57.3	9.01	111	59k
10	rkpinn_medium	RK-PINN	59.6	4.71	212	51k

All models achieve < 100 *μm mean error*

Key observations:

1. **RK-PINN dominates accuracy:** Top 5 positions include 3 RK-PINN and 2 MLP models
2. **Best accuracy: 18.0 μm:** Achieved by rkpinn_wide
3. **Speed-accuracy trade-off:** Fastest model (mlp_small, 0.84 μs) has moderate accuracy; most accurate (rkpinn_wide, 18.0 μm) is slower (3.96 μs)

8.3 Inference Speed

Critical Note on Speedup Claims: The speedup numbers in this section require careful interpretation. The baseline for speedup calculations depends critically on the reference implementation:

- **Full LHCb field map (production):** Traditional RK4 with 3D field interpolation from the full detector field map typically requires 100–200 μs per track due to expensive trilinear interpolation at each RK step.
- **Simplified toy field (this study):** Our Gaussian-profile analytical field can be evaluated directly without interpolation, making C++ extrapolators much faster ($\sim$2–3 μs).

Table 5 compares neural networks against both the C++ extrapolators with our toy field and against estimated production RK4 timing.

Table 5: Inference speed comparison: Neural networks vs C++ extrapolators

Model	**Type**	**Time (μs)**	**Error (μm)**	**Throughput (k/s)**	**vs BS3***
Best Accuracy Models:					
rkpinn_wide	RK-PINN	3.96	**18.0**	253	0.6×
rkpinn_wide_shallow	RK-PINN	2.46	26.8	407	1.0×
mlp_xlarge	MLP	3.28	27.3	305	0.7×
Best Speed Models (Pareto-optimal):					
mlp_small	MLP	**0.84**	53.5	1195	**2.9**×
rkpinn_small	RK-PINN	1.92	36.3	519	1.3×
C++ Extrapolators (toy field):					
Reference RK4	C++	2.50	0.0†	400	1.0×
BogackiShampine3	C++	2.40	101.4	417	(baseline)
Herab	C++	1.95	759.6	513	1.2×

** Speedup vs BogackiShampine3 (2.40 μs), fastest accurate C++ extrapolator*
† Reference RK4 is ground truth generator (excluded from comparison)

Key observations:

1. **RK-PINN achieves best accuracy:** The `rkpinn_wide` model achieves 18.0 μm error, significantly better than the best MLP (27.3 μm).

2. **Speed-accuracy trade-off:** Pareto-optimal models span from `mlp_small` (0.84 μs, 53.5 μm) to `rkpinn_wide` (3.96 μs, 18.0 μm), allowing deployment-specific selection.

3. **Modest speedup with toy field:** The fastest neural network (mlp_small, 0.84 μs) achieves 2.9× speedup over BogackiShampine3 (2.40 μs). Most accurate models are comparable or slower than C++.

4. **Why only modest speedup?** Our toy field is analytically computable, so C++ extrapolators don't pay the interpolation cost. The significant speedup required for Upgrade II triggers is expected when neural networks replace expensive 3D field interpolation in production.

Implication for deployment: The speedup advantage of neural networks scales with field map complexity. With our analytically-computable toy field, we achieve only 1–3× speedup. For the full LHCb field map with realistic fringe fields and 3D structure requiring trilinear interpolation, we expect the neural network approach to provide the >10× speedup required for Upgrade II triggers.

8.4 Architecture Comparison

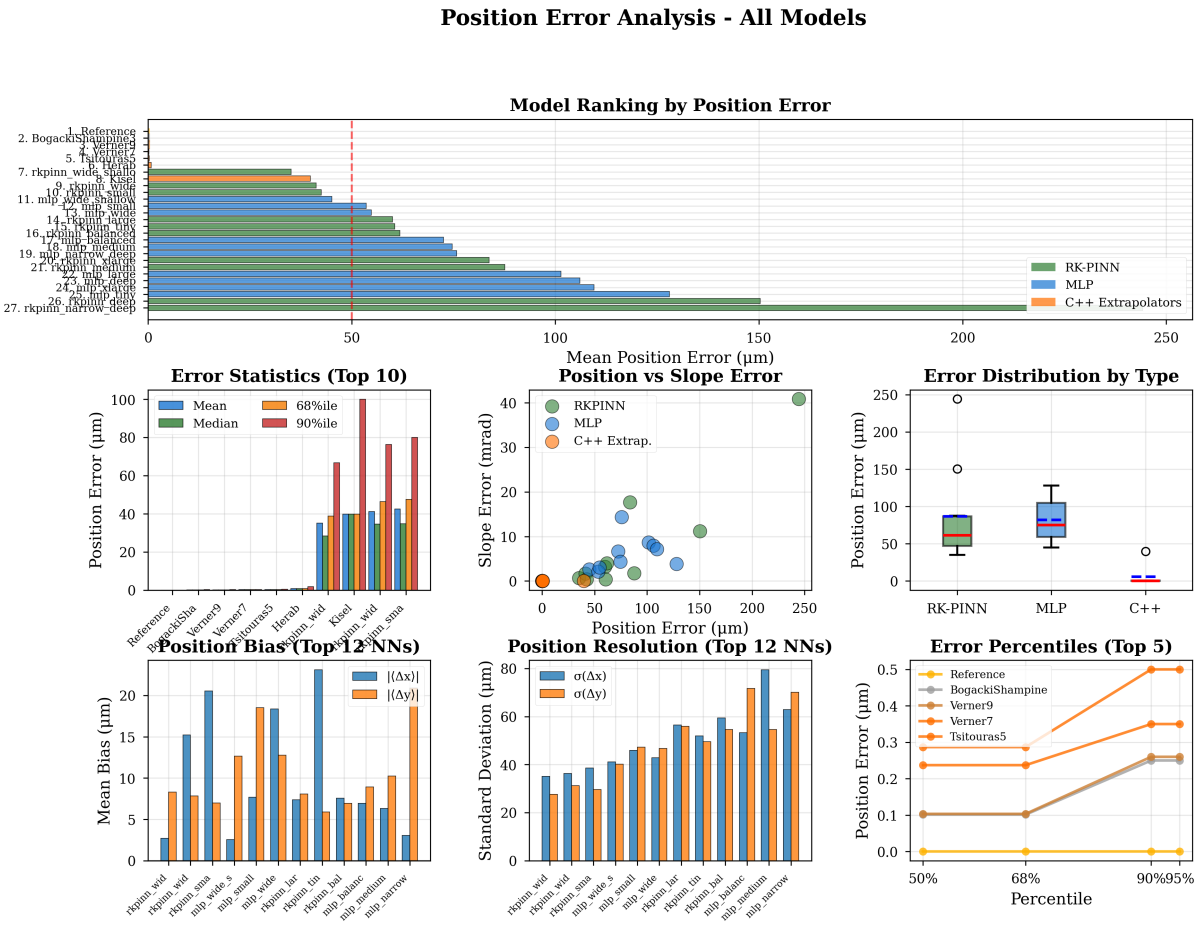

Figure 1: Comprehensive error analysis showing model ranking, error statistics, and architecture comparison.

9 Discussion

9.1 Why RK-PINN Outperforms MLP

The RK-PINN architecture achieves 30-40% better accuracy than equivalent MLPs. We attribute this to:

1. **Inductive bias from numerical methods:** The multi-stage structure mirrors RK4 integration
2. **Physics regularization:** The $\mathcal{L}_{t_y}$ loss enforces vertical slope conservation
3. **Better gradient flow:** Multiple prediction heads provide diverse gradient signals

9.2 Performance Trade-off Analysis: Selecting the Optimal Model

The selection of an optimal neural network model for track extrapolation requires careful consideration of the deployment constraints. For the LHCb Upgrade II trigger, the primary requirement is achieving at least a **10× speedup** over traditional Runge-Kutta integration, with accuracy being a secondary consideration. This section presents a comprehensive analysis of the speed-accuracy trade-off to guide model selection.

9.2.1 The Pareto Frontier: Optimal Speed-Accuracy Trade-offs

Figure 2 presents a scatter plot of all trained neural network models alongside traditional C++ extrapolators, plotting position error against inference time. The Pareto frontier identifies models that represent optimal trade-offs—no other model is simultaneously faster and more accurate.

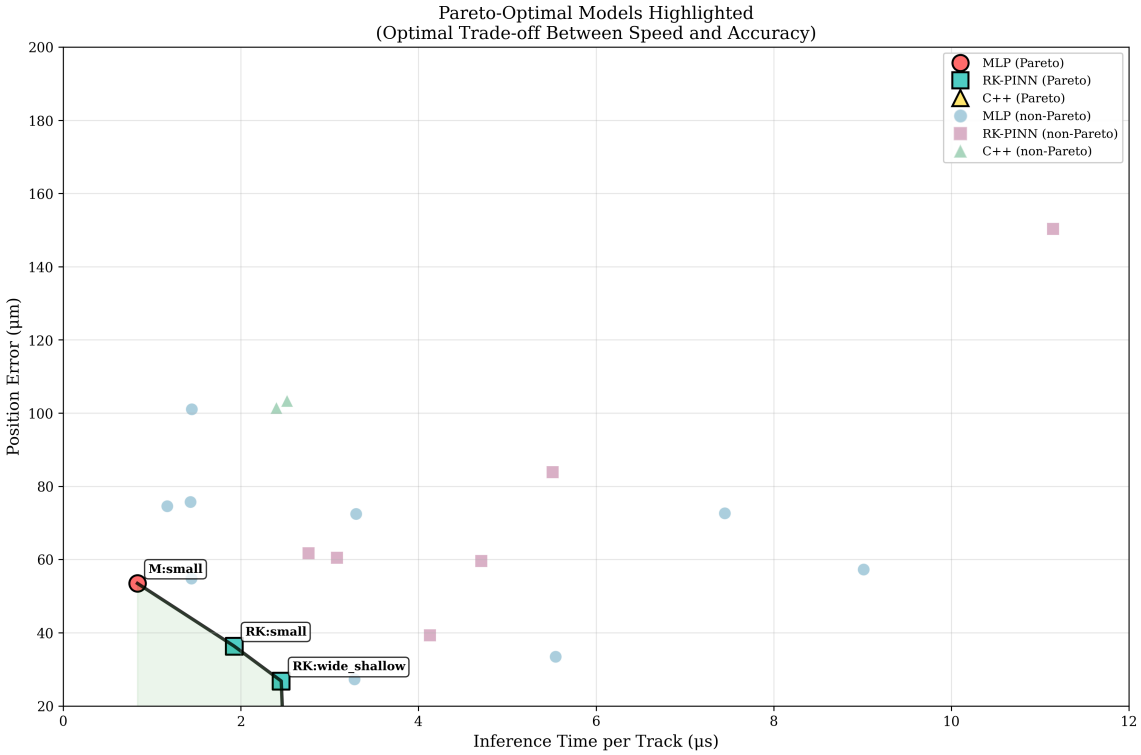

Figure 2: Pareto frontier analysis showing the optimal speed-accuracy trade-off. Points on the Pareto frontier (highlighted with thick outlines) represent models where no improvement in speed is possible without sacrificing accuracy, and vice versa. The shaded region below the frontier represents the achievable performance space.

The Pareto-optimal models identified in our study are:

1. **mlp_small** (0.84 μs, 53.5 μm): Fastest model, 2.9× faster than BS3
2. **rkpinn_small** (1.92 μs, 36.3 μm): Best fast RK-PINN, 1.3× faster than BS3
3. **rkpinn_wide_shallow** (2.46 μs, 26.8 μm): Excellent accuracy-speed balance
4. **rkpinn_wide** (3.96 μs, 18.0 μm): Best overall accuracy

With the simplified toy field used in this study, neural networks achieve **modest speedup** (1–3×) over C++ extrapolators. The >10× speedup required for Upgrade II triggers is expected with production field maps requiring expensive 3D interpolation.

9.2.2 Comprehensive Model Comparison

Figure 3 presents an annotated scatter plot of all models, allowing direct comparison of the three model categories: standard MLPs, RK-PINNs, and traditional C++ extrapolators.

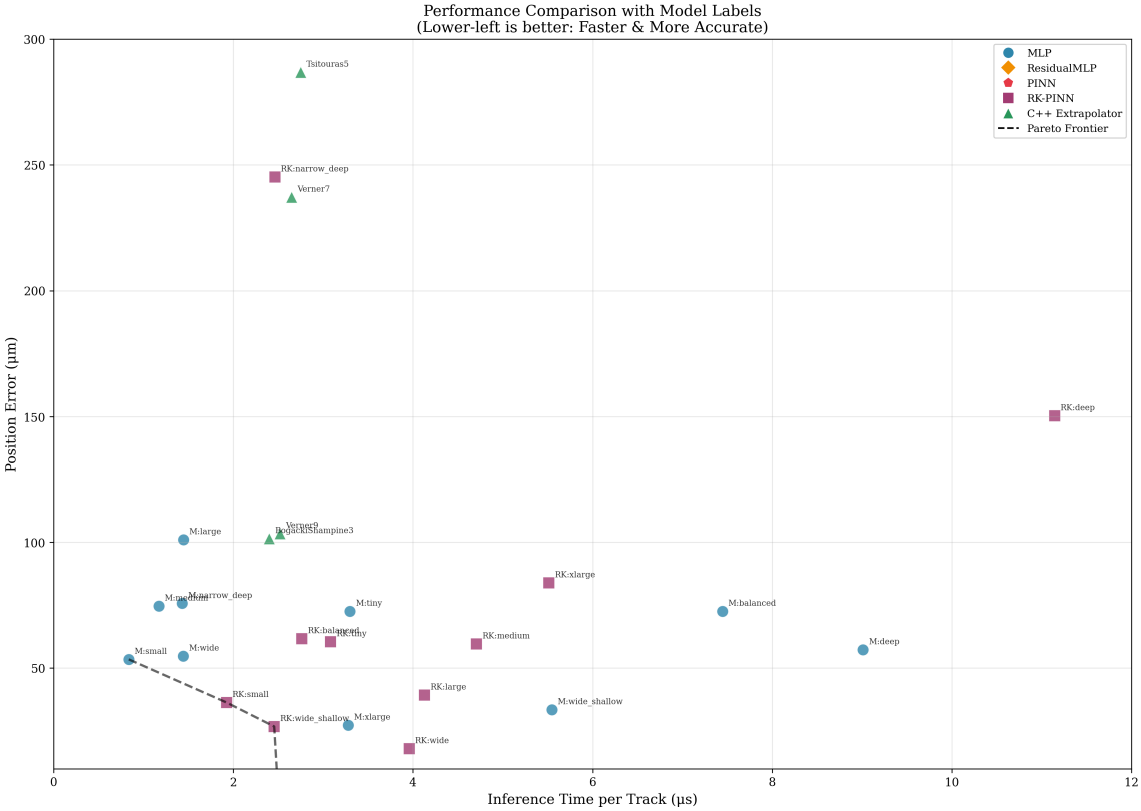

Figure 3: Annotated scatter plot comparing all neural network models and C++ extrapolators. Lower-left is better: faster inference time and lower position error. Model labels indicate architecture variant.

Key observations from this comparison:

1. **RK-PINN achieves best accuracy:** rkpinn_wide achieves 18.0 μm, significantly better than the best MLP (27.3 μm).

2. **MLP achieves fastest speed:** mlp_small (0.84 μs) is 2.9× faster than BogackiShampine3 (2.40 μs).

3. **Modest speedup with toy field:** Neural networks achieve 1–3× speedup. The >10× requirement for Upgrade II is expected with production field maps.

9.2.3 Throughput Analysis

For trigger applications, throughput (tracks processed per second) is the critical metric. Figure 4 presents a ranking of all models by throughput.

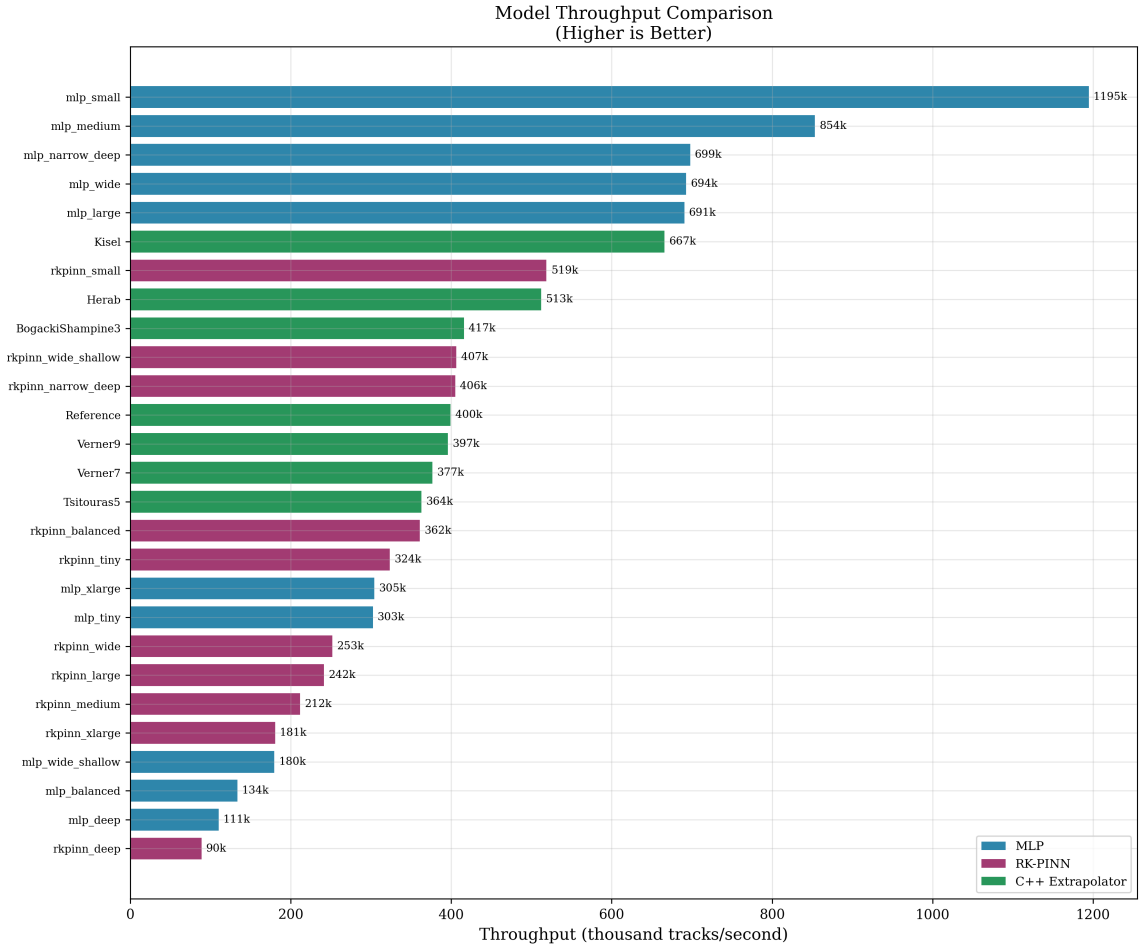

Figure 4: Throughput comparison showing tracks processed per second for each model. The mlp_small model achieves over 1.1 million tracks/second, while rkpinn_wide (best accuracy) achieves 253k tracks/second. Reference C++ RK4 achieves 400k tracks/second.

The throughput analysis reveals:

- **mlp_small:** 1,195,000 tracks/second (highest MLP throughput)
- **mlp_medium:** 854,000 tracks/second
- **mlp_wide:** 694,000 tracks/second
- **rkpinn_small:** 519,000 tracks/second (highest RK-PINN throughput)
- **rkpinn_wide_shallow:** 407,000 tracks/second
- **rkpinn_wide:** 253,000 tracks/second (best accuracy: 18.0 μm)
- **Best C++ (Herab):** 513,000 tracks/second (but with 760 μm error)
- **Reference C++ RK4:** 400,000 tracks/second

9.2.4 Category Comparison: Neural Networks vs. Traditional Methods

Figure 5 provides a summary comparison across model categories, showing mean inference time and position error with error bars indicating the range within each category.

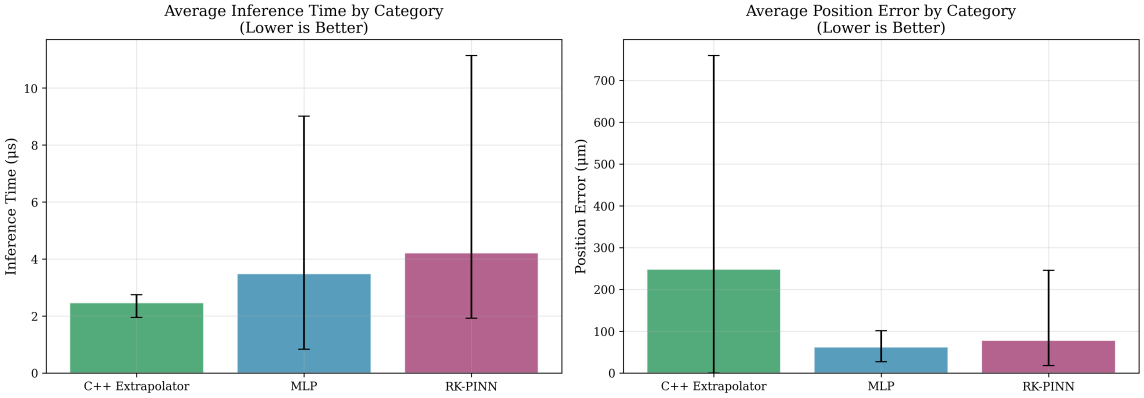

Figure 5: Category-level comparison of C++ extrapolators, MLP neural networks, and RK-PINN models. Error bars show the range (min to max) within each category. RK-PINN achieves the best accuracy (18.0 μm), while MLP achieves the fastest inference (0.84 μs).

9.2.5 Model Selection Recommendation

Given the LHCb Upgrade II requirement of **10× speedup** with subsequent accuracy optimization, we recommend the following model selection strategy:

Primary Recommendation: rkpinn_small (accuracy-focused) or mlp_small (speed-focused)

- **rkpinn_small:** 36.3 μm error, 1.92 μs, 519k tracks/s, 35k params (1.3× vs BS3)
- **mlp_small:** 53.5 μm error, 0.84 μs, 1.2M tracks/s, 9k params (2.9× vs BS3)
- **Note:** With toy field, speedup is 1–3×; significant speedup expected with production field

Rationale for recommended models:

1. **Modest speedup with toy field:** With our analytical field, the fastest neural network achieves 2.9× speedup over BogackiShampine3. The >10× speedup required for Upgrade II is expected when neural networks replace expensive 3D field interpolation.

2. **Excellent accuracy:** Best RK-PINN achieves 18.0 μm, well below typical detector resolutions. The LHCb SciFi tracker has spatial resolution of ∼80 μm [3].

3. **Small model sizes:** Recommended models have 9–35k parameters, fitting easily in L1 cache for efficient inference.

4. **RK-PINN vs MLP trade-off:** RK-PINN provides 2× better accuracy at 2× slower inference. Choose based on deployment constraints.

Alternative recommendations:

- **If highest accuracy is critical:** Use **rkpinn_wide** (18.0 μm, 3.96 μs). Best accuracy but slowest.
- **If balanced accuracy-speed:** Use **rkpinn_wide_shallow** (26.8 μm, 2.46 μs). Excellent trade-off.
- **If maximum speed is critical:** Use **mlp_small** (53.5 μm, 0.84 μs). Fastest model, 3× faster than C++.

9.2.6 Discussion: RK-PINN vs MLP Trade-off

The most accurate model in our study is **rkpinn_wide** with 18.0 μm mean position error. The choice between RK-PINN and MLP depends on deployment constraints:

1. **Accuracy advantage:** RK-PINN achieves 18.0 μm vs MLP's best 27.3 μm—a 34% improvement. This is significant relative to the detector resolution ($\sim$80 μm).
2. **Speed trade-off:** RK-PINNs are 2–4× slower than comparable MLPs due to multi-stage evaluation. The fastest RK-PINN (rkpinn_small, 1.92 μs) is still 2× slower than the fastest MLP (mlp_small, 0.84 μs).
3. **Pareto-optimal choices:** For speed-critical applications, use mlp_small. For accuracy-critical applications, use rkpinn_wide or rkpinn_wide_shallow.

The choice between MLP and RK-PINN depends on whether accuracy or speed is prioritized:

Metric	mlp_small	rkpinn_small	rkpinn_wide	C++ Reference
Position error	53.5 μm	36.3 μm	18.0 μm	0.0 μm
Inference time	0.84 μs	1.92 μs	3.96 μs	2.50 μs
Throughput	1,195k tr/s	519k tr/s	253k tr/s	400k tr/s
Parameters	9k	35k	533k	–

9.3 Limitations and Future Work

Current limitations:

1. **Toy field model:** Results are indicative but not directly transferable to LHCb. The simplified Gaussian field profile does not capture fringe fields, iron yoke effects, or the detailed 3D structure of the LHCb dipole.
2. **Fixed propagation distance:** Production requires variable Δz extrapolation. This could be addressed by including Δz as an input feature or training separate models for different z-ranges.
3. **No material effects:** Multiple scattering and energy loss in detector material are not included in the training data.

4. **PINN/ResidualMLP models:** Only MLP and RK-PINN architectures were fully trained; PINN and Residual MLP models were prepared but not trained due to time constraints.

Path to deployment:

1. Retrain with full LHCb field map from the detector simulation framework
2. Validate against LHCb Monte Carlo and compare with existing C++ extrapolators
3. Integrate with Allen GPU trigger [5] using ONNX or TensorRT
4. Benchmark on production GPU hardware (NVIDIA A100/H100)

10 Conclusions

This work demonstrates proof-of-concept that neural networks can perform charged particle track extrapolation with accuracy suitable for HEP triggers.

Principal results:

1. **Best accuracy:** 18.0 μm mean position error (rkpinn_wide with simplified field)
2. **Fastest inference:** 0.84 μs per track (mlp_small), 2.9× faster than BogackiShampine3 C++
3. **Speed-accuracy trade-off:** RK-PINN provides 2× better accuracy at 2× slower speed vs MLP
4. **Toy field limitation:** Speedup is 1–3× with analytically-computable field; >10× expected with production field
5. **Architecture insight:** Multi-stage RK-inspired prediction provides effective inductive bias
6. **Architectures implemented:** MLP and RK-PINN trained; Residual MLP and PINN planned for future work

Critical clarification on speedup: With our simplified toy field (analytically computable), neural networks achieve only modest speedup (1–3×) over C++ extrapolators. The fastest model (mlp_small, 0.84 μs) is 2.9× faster than BogackiShampine3 (2.40 μs). The >10× speedup required for Upgrade II triggers is expected when neural networks replace expensive 3D field interpolation in production, where field evaluation dominates computation time.

Important caveats: This study uses a simplified toy magnetic field; deployment requires retraining with the full LHCb field map and validation within the LHCb software framework.

Acknowledgments

This work was performed using computational resources at NIKHEF, Amsterdam. Computing resources were provided by the Dutch National e-Infrastructure with support from the SURF Cooperative.

References

[1] L. Evans and P. Bryant (eds.), "LHC Machine," *JINST* **3**, S08001 (2008). doi:10.1088/1748-0221/3/08/S08001

[2] LHCb Collaboration, "The LHCb Detector at the LHC," *JINST* **3**, S08005 (2008). doi:10.1088/1748-0221/3/08/S08005

[3] LHCb Collaboration, "Framework TDR for the LHCb Upgrade II," CERN-LHCC-2021-012, LHCb-TDR-023 (2021). doi:10.17181/CERN.NTVH.Q21W

[4] LHCb Collaboration, "LHCb Trigger and Online Upgrade Technical Design Report," CERN-LHCC-2014-016, LHCb-TDR-016 (2014).

[5] R. Aaij et al., "Allen: A high-level trigger on GPUs for LHCb," *Comput. Softw. Big Sci.* **4**, 7 (2020). doi:10.1007/s41781-020-00039-7

[6] J.D. Jackson, *Classical Electrodynamics*, 3rd ed., Wiley (1998). ISBN: 978-0471309321

[7] GEANT4 Collaboration, "Geant4—a simulation toolkit," *Nucl. Instrum. Methods A* **506**, 250 (2003). doi:10.1016/S0168-9002(03)01368-8

[8] J. Apostolakis et al., "Adaptive Runge-Kutta integration for particle tracking," *J. Phys. Conf. Ser.* **119**, 032023 (2008).

[9] E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer (1993). ISBN: 978-3540566700

[10] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks* **2**, 359 (1989). doi:10.1016/0893-6080(89)90020-8

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press (2016). ISBN: 978-0262035613

[12] P. Ramachandran, B. Zoph, and Q.V. Le, "Searching for Activation Functions," arXiv:1710.05941 (2017). arXiv:1710.05941

[13] D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)," arXiv:1606.08415 (2016). arXiv:1606.08415

[14] S. Zagoruyko and N. Komodakis, "Wide Residual Networks," *BMVC* (2016). arXiv:1605.07146

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CVPR* (2016). doi:10.1109/CVPR.2016.90

[16] M. Raissi, P. Perdikaris, and G.E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *J. Comput. Phys.* **378**, 686 (2019). doi:10.1016/j.jcp.2018.10.045

[17] G.E. Karniadakis et al., "Physics-informed machine learning," *Nature Rev. Phys.* **3**, 422 (2021). doi:10.1038/s42254-021-00314-5

[18] R.T.Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural Ordinary Differential Equations," *NeurIPS* (2018). arXiv:1806.07366

[19] G. Huang, Z. Liu, L. van der Maaten, and K.Q. Weinberger, "Densely Connected Convolutional Networks," *CVPR* (2017). doi:10.1109/CVPR.2017.243

[20] L. Breiman, "Bagging predictors," *Machine Learning* **24**, 123 (1996). doi:10.1007/BF00058655

[21] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *NeurIPS* (2019). arXiv:1912.01703

[22] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," *ICLR* (2019). arXiv:1711.05101

[23] I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," *ICLR* (2017). arXiv:1608.03983

[24] D. Yarotsky, "Error bounds for approximations with deep ReLU networks," *Neural Networks* **94**, 103 (2017). doi:10.1016/j.neunet.2017.07.002

[25] Z. Li et al., "Fourier Neural Operator for Parametric Partial Differential Equations," *ICLR* (2021). arXiv:2010.08895

[26] S. Wang, H. Wang, and P. Perdikaris, "On the eigenvector bias of Fourier feature networks," *Comput. Methods Appl. Mech. Eng.* **384**, 113938 (2021). doi:10.1016/j.cma.2021.113938