Master of Engineering Project Report

Physics-based SAXS Simulation and ML Characterization of LNPs

Student:
Tianyou Li (NetID: tl899)

MEng Field Advisor:
Prof. Peter Doerschuk

Degree Date:
May 2025

Department of Electrical Computer Engineering

Abstract

This study builds upon existing research by leveraging machine learning (ML) techniques to enhance the characterization of lipid nanoparticles (LNPs) for drug delivery applications. By integrating physics-based simulation methods, advanced experimental data preprocessing, and realistic noise modeling, we have developed highly accurate small-angle X-ray scattering (SAXS) simulations for dilute aqueous LNP solutions. Our approach employs convolutional neural networks (CNNs) and residual neural networks (ResNets) to effectively classify heterogeneous LNP solutions and precisely predict size distribution parameters for homogeneous systems. This methodology offers a rapid and cost-effective alternative to conventional techniques such as cryo-electron microscopy (cryo-EM), thereby holding significant promise for advancing nanomedicine.

Keywords: Lipid Nanoparticles, SAXS, Physics-based Simulation, Machine learning

Executive Summary

We developed a physics-based SAXS simulation pipeline and machine-learning framework to characterize lipid nanoparticles (LNPs) rapidly and cost-effectively. Key achievements include:

- **Realistic 3D Models:** Built 81 core–shell ellipsoidal LNP models (21–100 nm) from cryo-EM data to replace the spherical assumption.
- **High-Fidelity SAXS Data:** Employed adaptive zero-padding and a continuous-density (sinc^2) correction to eliminate FFT artifacts and match analytical benchmarks.
- **Efficient Orientation Averaging:** Replaced nested loops with a fully vectorized interpolation and effective-count normalization, cutting runtime dramatically and improving low- and high-q accuracy.
- **Data Augmentation:** Generated homogeneous curves, applied within-type lognormal sampling and cross-type mixing (fraction α), then injected lognormal noise calibrated to photon statistics.
- **ML Characterization:** Trained a CNN classifier (96% accuracy, $F1 \geq 0.97$) and a regression network ($R^2 = 0.964$, MSE=0.003) to distinguish LNP types and predict mixture fractions.

Future Work: Validate on experimental SAXS data, extend to multi-component and polydisperse systems, and optimize for large-scale production via GPU acceleration.

Contents

1 Introduction

Nanoparticle-based drug delivery systems—particularly lipid nanoparticles (LNPs)—have revolutionized therapeutic delivery by enabling highly precise and efficient treatment modalities. LNPs garnered significant global attention during the COVID-19 pandemic due to their critical role in mRNA vaccine development. Their demonstrated versatility, stability, and capacity for targeted tissue delivery have established them as pivotal components in advancing precision medicine.

Accurate characterization of the nanoscale structural and functional properties of LNPs is essential for optimizing their therapeutic potential. Two techniques are commonly employed for this purpose: cryo-electron microscopy (cryo-EM) and small-angle X-ray scattering (SAXS). Cryo-EM provides high resolution, two-dimensional projections of individual LNPs; however, its high operational costs, slow throughput, and limited integration into production environments constrain its routine application. In contrast, SAXS yields an ensemble-averaged one-dimensional scattering profile, which—although offering less direct insight into individual particle morphology—is relatively inexpensive, rapid, and well-suited for incorporation into production lines. Therefore, the objective of this work is to assess the extent to which ML can extract detailed structural information from SAXS data, using cryo-EM as the benchmark standard.

2 Methodology

2.1 Data Generation

This subsection provides a comprehensive overview of the data generation pipeline, as depicted in Fig. 1. The following sections detail each step of the process. In brief, our methodology encompasses the preparation of 3D LNP models, the implementation of an adaptive padding strategy combined with FFT, and the application of a continuous density correction to produce realistic SAXS simulations. Each stage is carefully designed to capture the relevant physical phenomena while ensuring computational efficiency and fidelity to experimental conditions.

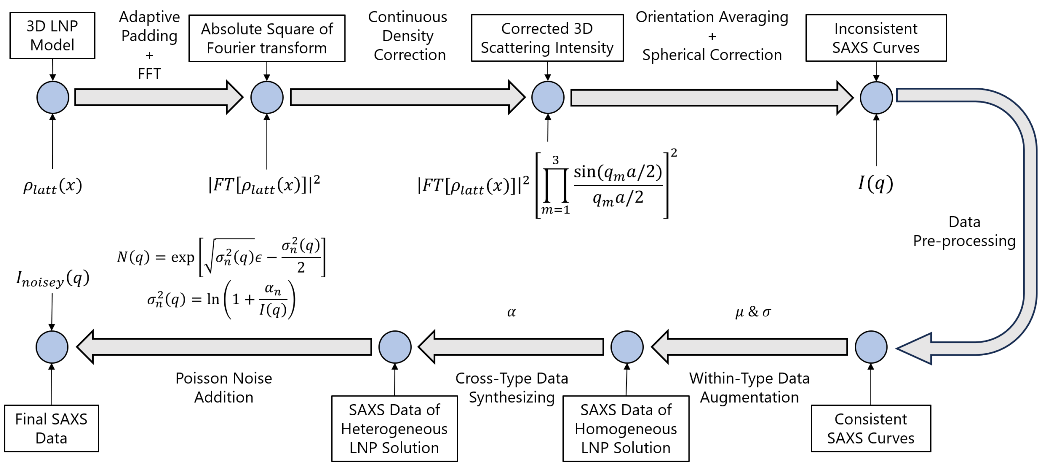
3D LNP Model
Adaptive Padding + FFT
Absolute Square of Fourier transform
Continuous Density Correction
Corrected 3D Scattering Intensity
Orientation Averaging + Spherical Correction
Inconsistent SAXS Curves
$\rho_{latt}(x)$
$|FT[\rho_{latt}(x)]|^2$
$|FT[\rho_{latt}(x)]|^2\left[\prod_{m=1}^{3}\frac{\sin(q_m a/2)}{q_m a/2}\right]^2$
$I(q)$
Data Pre-processing
$I_{noisey}(q)$
$N(q) = \exp\left[\sqrt{\sigma_n^2(q)}\epsilon - \frac{\sigma_n^2(q)}{2}\right]$
$\sigma_n^2(q) = \ln\left(1 + \frac{\alpha_n}{I(q)}\right)$
α
μ & σ
Poisson Noise Addition
Cross-Type Data Synthesizing
Within-Type Data Augmentation
Final SAXS Data
SAXS Data of Heterogeneous LNP Solution
SAXS Data of Homogeneous LNP Solution
Consistent SAXS Curves

Figure 1: The Data Generation Pipeline Flow Chart. Symbols: $\rho_{latt}(x)$ denotes the 3D LNP model; FT denotes the results of the Fourier Transform; q_m denotes the minimum q value in the reciprocal space;a denotes the real space resolution in nm; μ, σ denote the random variables defining a unique lognormal distribution; α denotes the mixing factor (fraction of Type 3 LNPs); σ_n denotes the q-dependent variance; ϵ denotes a standard normally distributed random variable; α_n denotes a log-uniform scaling factor.

2.1.1 3D Model Preparation

Before diving into the details of how to generate realistic SAXS data, it is also necessary to ensure the 3D LNP models adopted in this project are valid, which means the shapes or sizes should be verified by cryo-EM data. As shown in Fig. 2, it is not suitable to assume the LNPs are spherical, which, however, is the normal practice when analyzing the SAXS data. Therefore, in this project, we try to apply more detailed 3D LNP models to generate the SAXS data and see whether the ML techniques can extract structural information from the generated SAXS data.

The LNP models are constructed to mirror the actual physical structure observed in cryo-EM images and SAXS experiments. Specifically, the models adopt a core-shell ellipsoid design where the core is defined by an equatorial radius that ranges between 21 and 30 nm and an axial ratio of about 1.667—parameters determined from detailed image measurements that shifted the model from an initially assumed oblate shape to a prolate one. The shell, representing the lipid layers, is given a uniform thickness of approximately 4 nm, an intermediate value chosen from experimental estimates between 3 and 5 nm. In MATLAB simulations, this physical structure is further detailed by mapping the electron density in a 3D grid (using a density function that is

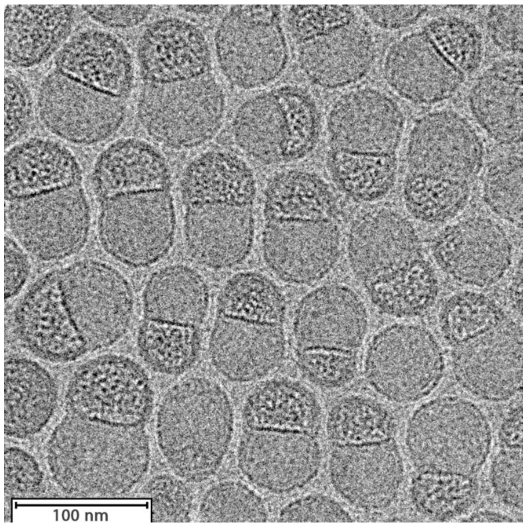

Figure 2: Cyro-EM image of mRNA-LNP

constant inside the particle and zero outside) and performing Fourier transforms to generate SAXS curves. To further diversify the dataset, a scaling factor is employed to control the size of 3D models, resulting in the simulated LNPs' sizes ranging from 20 to 100 nm.

At this stage of the project, we created 4 types of different models as shown in Fig. 3, both of which can be found in either Fig. 2 or other cryo-EM data sources. Under the current settings, the model is $100 \times 100 \times 100$ in total, where each voxel corresponds to 1 nm^3 cube in real space. According to [1], LNPs in the size range of 40 to 80 nm are more desired in production. Therefore, both the classifier and the prediction model are designed to classify the solution type and predict the size distribution within the range of 20 to 100 nm. Thus, there are 81 models in total with 1 nm increment in diameter, which are regarded as ground truth in the following data synthesizing process. These models will be used for SAXS data simulation since they represent the electron density of the LNPs. Hence, we have obtained:

$$\rho_{latt}(x)$$

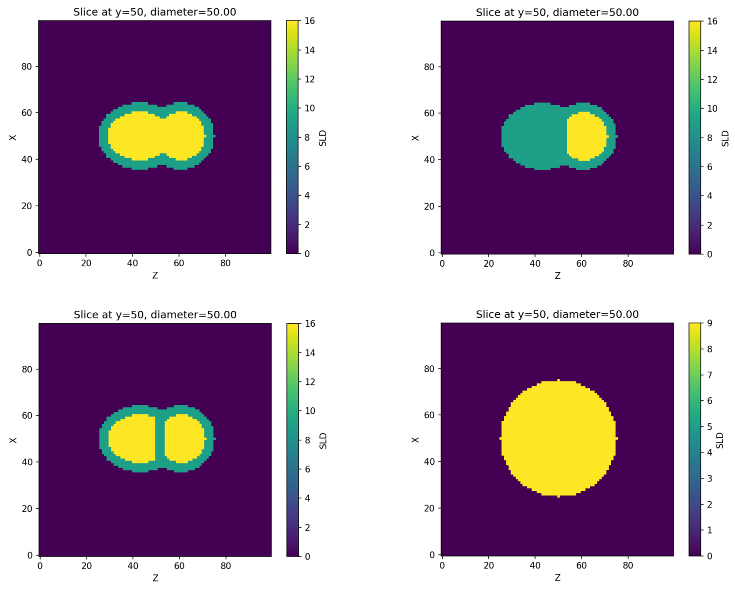

Figure 3: Four Different Types of 3D LNP Model. SLD denotes the scattering length density, reflecting the material's scattering ability to X-rays or neutron beams

2.1.2 Adaptive Padding + FFT

In SAXS simulations using a finite computational cube, discontinuities at the boundaries can introduce artifacts in the Fast Fourier Transform (FFT). These discontinuities result in distortions and spurious oscillations in the computed scattering intensity. The FFT inherently assumes periodic input data, effectively replicating the electron density cube in all directions (as illustrated in Fig. 3). When the density at the boundaries does not seamlessly match the surrounding values (which, for an isolated particle, is zero), a sharp discontinuity occurs as the data "wraps around." Such abrupt transitions, akin to step functions in real space, inherently contain a wide range of frequencies—a phenomenon related to the Gibbs effect. Consequently, power "leaks" into many frequencies, manifesting as spectral leakage that appears as noise or artificial ripples in the output. For example, Fig. 4 compares simulated SAXS data for a 50 nm LNP model using two different padding sizes (200 nm and 600 nm), clearly demonstrating that reduced padding exacerbates distortion.

Although windowing techniques can smooth the edges to mitigate these effects, they unavoidably alter the amplitude and finer details of the signal.

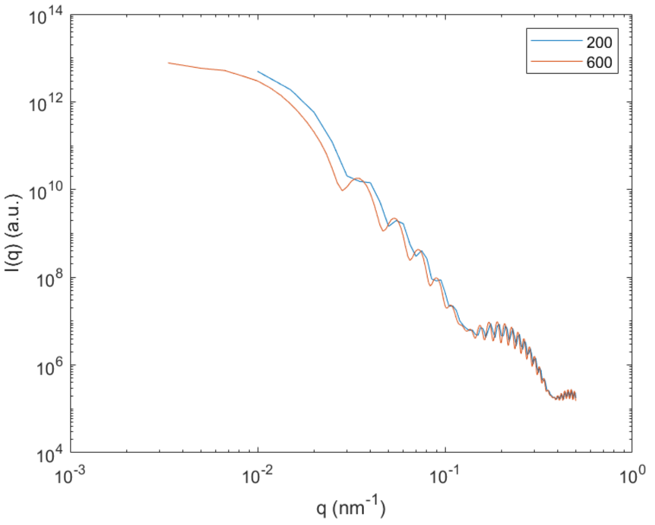

Figure 4: Illustration of the FFT Boundary Effect

To preserve the physical accuracy of the simulation, the optimal approach is to enlarge the computational domain so that the electron density naturally decays to zero at the boundaries, thereby avoiding the need for artificial windowing. A common strategy for simulating an isolated particle is zero-padding, where the cube is extended with zeros (representing a vacuum) around the particle. This method increases the domain size and separates the particle from its periodic images, ensuring a smooth transition to zero at the boundaries. A widely adopted guideline is to restrict the particle to no more than half the linear dimension of the computational box to prevent overlap with its periodic replicas.

In this project, the primary goal is to design a ML algorithm that extracts genuine physical information from the simulated SAXS data—such as distinguishing between LNP solution types or predicting the mean size—without relying on numerical artifacts. Employing a fixed padding size would result in different relative padding for small and large LNPs, yielding FFTs with different absolute resolutions. Such variations could inadvertently provide cues to the ML model that are unrelated to the intrinsic properties of the LNPs.

To overcome this challenge, we adopt an adaptive padding strategy that maintains a fixed ratio between the padded cube size and the effective electron density cube. This approach offers several benefits:

- **Uniform Relative Boundary Effects:** A constant padding-to-model ratio ensures that the effective distance from the model to the boundaries is proportional for all samples, leading to consistent truncation-induced discontinuities and spectral leakage.
- **Consistent Relative FFT Resolution:** Although the absolute FFT resolution (i.e., the spacing in reciprocal space) varies with the overall cube size, adaptive padding ensures that the resolution remains consistent relative to the model dimensions. This consistency is preserved when interpolating to a unified reciprocal axis, thereby maintaining the physical features of the scattering data.
- **Prevention of Spurious Cues for ML:** By standardizing the relative padding, the FFT artifacts (e.g., due to boundary discontinuities) are uniformly distributed, reducing the risk that the ML algorithm will associate these numerical differences with intrinsic particle properties.
- **Physical Consistency:** Scaling the simulation environment in proportion to the effective model better represents the natural decay of electron density, preserving the physical integrity of the simulation.

Under this strategy, for example, a 50 nm electron density cube is padded to 300 nm, and a 100 nm cube to 600 nm before performing the FFT. This uniformity in relative boundary effects compels the ML model to rely on features that genuinely reflect the LNP models' shape and size, rather than on numerical artifacts arising from differences in FFT resolution.

Before implementing the correction algorithm, it is necessary to establish the method for constructing reciprocal (q) space. In our approach, the padding size is specified in terms of grid points rather than physical units. For instance, if the original electron density cube consists of 50 points, a padded cube may be defined with 300 points (denoted as n_x) to capture maximal information from the real-space model. The minimum and maximum q values are then defined as:

$$q_{\min} = q^{(1)} = \frac{2\pi}{n_x\, a}, \quad q_{\max} = q^{N/2} = \frac{\pi}{a}, \quad \text{where } a = 1\,\text{nm}.$$

The maximum q is defined as $q^{N/2}$ because the FFT output is shifted so that the zero-frequency component is centered—a necessary condition for orientation averaging. With the padded cube prepared, a three-dimensional FFT is performed. Notably, the number of FFT points is not constrained to be a power of 2, as empirical tests have shown negligible performance loss. The resulting quantity is given by:

$$|FT[\rho(x)]|^2.$$

2.1.3 Continuous Density Correction

In numerical simulations, the electron density is defined only at discrete lattice points, whereas the physical electron density is continuous. To generate the most realistic SAXS data, it is therefore necessary to implement a continuous density correction [2]. As illustrated in Fig. 5, this correction bridges the gap between discrete density peaks and a continuous density distribution by convolving the discrete peaks (as used in numerical DFT) with an elementary "box" function of width a [2].

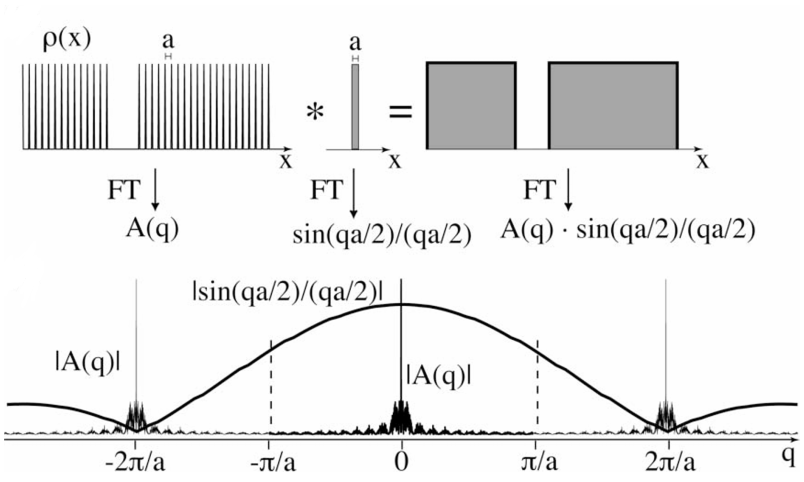

Figure 5: Continuous-Discrete Density Distribution Correction Symbols: $A(q)$ denotes the result of Discrete Fourier Transform [2]

Mathematically, the Fourier Transform (FT) of a convolution is equal to the product of the FTs of the individual functions. Thus, the corrected scattering intensity can be expressed as:

$$I(\mathbf{q}) = |FT[\rho_{latt}]|^2 \left[\prod_{m=1}^{3} \frac{\sin(q_m\, a/2)}{q_m\, a/2}\right]^2 [2]$$

where ρ_{latt} denotes the discretely defined electron density. In other words, we approximate the scattering intensity obtained from the continuous density by multiplying the intensity from the discrete density by the correction factor

$$\left[\frac{\sin(q\, m_a/2)}{q\, m_a/2}\right]^2$$

in each dimension ($m = 1, 2, 3$). This procedure is illustrated in Fig. 5 for a one-dimensional case. For discrete q values, we define

$$q^{(k)} = \frac{2k}{Na},$$

and multiplying by $a/2$ yields

$$\frac{q^{(k)}a}{2} = \frac{k}{N}$$

Thus, along one dimension, the kth point (out of N) in the IDFT(q) data field is multiplied by

$$\left\{\frac{\sin\left[\frac{(k-N/2-1)}{N}\right]}{\frac{(k-N/2-1)}{N}}\right\}^2$$

taking into account that the point corresponding to $q = 0$ is located at $N/2 + 1$. In three dimensions, with a lattice spacing given by

$$q_s = \frac{2}{Na}$$

the continuous scattering intensity is obtained via

$$I_{\text{cont}}(n_1 q_s, n_2 q_s, n_3 q_s) = I_{\text{DFT}}(n_1 q_s, n_2 q_s, n_3 q_s) \prod_{m=1}^{3} \left\{\frac{\sin\left[\frac{(n_m-N/2-1)}{N}\right]}{\frac{(n_m-N/2-1)}{N}}\right\}^2 \quad (1)$$

For small q (or equivalently, for large structures, which are most relevant in small-angle scattering), the sinc functions in Equation (1) are nearly unity and leave $I(q)$ virtually unchanged. This is expected since small q values correspond to large distances where the fine details of the electron density do not significantly affect the scattering intensity. At the limit of the first Nyquist zone, $|q| = \pi/a$, the scattering intensity $I(\pi/a)$ is reduced by a factor of $(2/\pi)^2 \approx 0.4$. Therefore, the corrected scattering intensity is given by:

$$|FT[\rho_{latt}]|^2 \left[\prod_{m=1}^{3} \frac{\sin(q_m\, a/2)}{q_m\, a/2} \right]^2$$

2.1.4 Orientation Averaging + Spherical Correction

After applying the necessary correction functions, the next step is to perform orientation averaging to convert the full three-dimensional scattering data into a one-dimensional intensity curve, $I(q)$. The fundamental idea is to compute the scattering intensity at a given q by averaging the intensities of all points in 3D q space with $|\mathbf{q}| \approx q$. This is equivalent to averaging over a spherical shell in q space, hence the term "Orientation Averaging" or "Spherical Mean". In our implementation, the FFT yields n_x^3 points in the 3D q space, after which a one-dimensional q axis with n_{xf} uniformly spaced bins is defined. The following discussion first outlines the original algorithm proposed in [2] and then introduces an improved algorithm along with a comparative analysis.

Original Algorithm
The primary objective of the original algorithm is to distribute the scattering intensity from each of the n_x^3 points onto the n_{xf} bins. Since the bins are uniformly spaced over the overall q range, each value $q = |\mathbf{q}|$ must be mapped onto this discretized axis. The process can be summarized as follows. First, the n_x^3 points are flattened into a one-dimensional array, and the q value for the κ^{th} point is computed as:

$$q = |\mathbf{q}| = \frac{2\pi\kappa}{n_x\, a},$$

where a denotes the unit length. With the new 1D q axis consisting of n_{xf}

bins, let k be the bin index corresponding to the nearest value

$$q' = \frac{2\pi k}{n_{xf}\, a}$$

to $\frac{2\pi\kappa}{n_x\, a}$. Because q generally lies between two consecutive q' values, the intensity $I_{\mathrm{DFT}}(\mathbf{q})$ is apportioned between the adjacent bins. Specifically, a fraction

$$1 - \frac{|q' - q|}{\Delta q} = 1 - |k - \kappa|, \quad \text{where } \Delta q = \frac{2\pi}{n_{xf}\, a},$$

is allocated to the bin at

$$q' = \frac{2\pi k}{n_{xf}\, a},$$

while the remaining fraction

$$\frac{|q' - q|}{\Delta q} = |k - \kappa|$$

is assigned to the adjacent bin at

$$q' = \frac{2\pi\,(k + \mathrm{sgn}(q' - q))}{n_{xf}\, a}.$$

In formal terms, this process can be written as:

$$(1 - |\kappa - k|)\, I_{\mathrm{DFT}}(\mathbf{q}) \to I'\!\left(\frac{2\pi k}{n_{xf}\, a}\right), \quad |\kappa - k|\, I_{\mathrm{DFT}}(\mathbf{q}) \to I'\!\left(\frac{2\pi\,(k + \mathrm{sgn}(\kappa - k))}{n_{xf}\, a}\right).$$

Since the FFT of real-valued functions is symmetric about zero frequency, this “channel sharing” process is performed for every point in one-eighth of the n_x^3 grid, effectively accumulating their contributions into the n_{xf} uniformly distributed bins. Additionally, to compensate for the uneven number of points in different regions, the central point is assigned a weight of 1 while other points are weighted by 2, reflecting their double counting in the complete n_x^3 dataset.

Because the number of points with $q = |\mathbf{q}|$ in a spherical shell increases proportionally to the square of the radius (as noted in [2] and [3]), the aggregated scattering intensity from the cubic lattice is given by:

$$I'(q) = I(q)q^2$$

Without proper correction, the scattering intensities at higher q values will be overestimated due to the larger number of points in the spherical shell, leading to a statistical bias. Therefore, to obtain the correct scattering intensity, $I'(\mathbf{q})$ must be divided by q^2 to yield $I(\mathbf{q})$ [2].

Fig. 6 compares SAXS data generated for a spherical particle using the original algorithm with an analytical solution provided by Professor Peter Doerschuk. The analytical result, derived from a simple mathematical model, serves as a benchmark since spherical particles are the only case for which an analytical expression is available. The comparison indicates that while the original algorithm approximates the scattering intensity well at high q, discrepancies exist in the low q region.

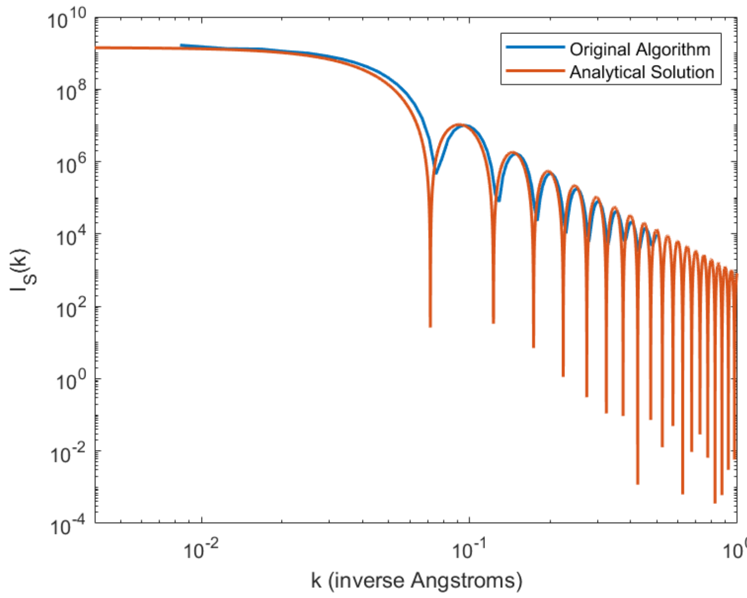

Figure 6: Original Orientation Averaging Algorithm vs Analytical Solution

Improved Algorithm

While the original algorithm offers a clear, step-by-step procedure for distributing scattering intensities across bins, its nested `for`-loop structure can be computationally expensive—especially for large grids where n_x or n_{xf} may exceed several hundred points. Furthermore, our analysis identifies two primary issues that lead to discrepancies between the algorithm's output and the analytical predictions.

First Issue:
Rounding κ to the nearest integer bin index, followed by applying a linear "channel sharing" between adjacent bins, introduces subtle interpolation errors in the low-q region. For example, as shown in Fig. 6, the scattering intensity appears to be shifted to the right by approximately one bin. This shift is attributable to the design of the rounding function. To avoid division by zero, the algorithm initiates the sharing of scattering intensity from bin 1 rather than bin 0 (where $q = |\mathbf{q}| = 0$). Although this approach prevents division by zero, it results in a shifted scattering intensity once the correction $I'(q) = I(q)q^2$ is applied.

Second Issue:
The second problem also stems from the correction $I'(q) = I(q)q^2$. As depicted in Fig. 7, although the effective point counts as a function of q follow a quadratic trend for approximately 60% of the q range, this trend fails in the high-q region, leading to an underestimation of the scattering intensity in those bins. The effective counts are computed based on the weighted contributions of each point to each bin. Given the finite extent of the cube, the number of points within a spherical shell of radius q is not strictly proportional to q^2 when the radius approaches half the cube size. For q values beyond this threshold, fewer points are present since a portion of the spherical shell lies outside the cube.

To address these concerns, we propose an improved, more efficient method with three key modifications:

1. **Full-vectorization Approach:** Instead of iterating through all n_x^3 points using three nested loops, we first generate the entire 3D coordinate grid via `ndgrid` (or an equivalent function) to compute $|\mathbf{q}|$ values in a single vectorized operation. This mesh approach directly provides

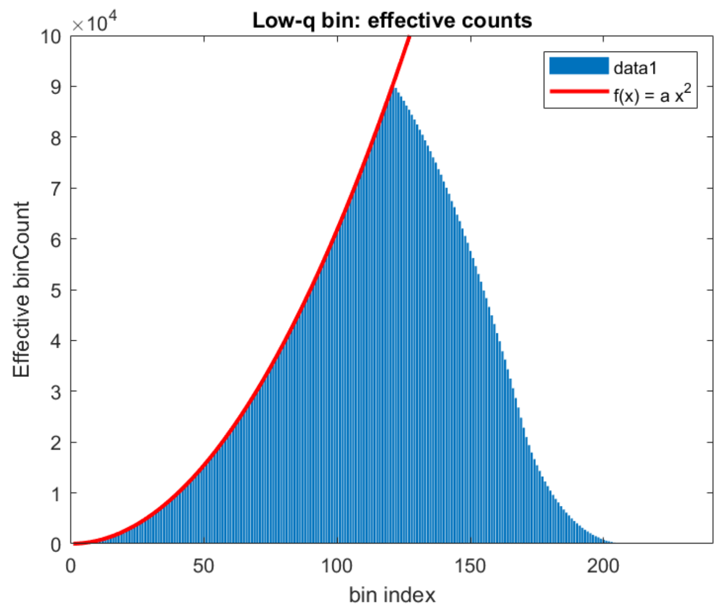

Figure 7: Effective Bin counts vs q. data1 denotes the effective counts of points falling into the bins along the q axis.

the spherical radius for each voxel in the frequency domain:

$$\mathbf{R} = \sqrt{X^2 + Y^2 + Z^2}$$

where (X, Y, Z) spans the 3D domain $\{-\frac{n_x}{2}, \ldots, \frac{n_x}{2} - 1\}$ after shifting the zero-frequency component to the center. The corresponding intensities `Iq3D` and correction factors (sinc^2) are likewise extracted in vector form.

2. **Weighted Linear Interpolation via Accumulation:** Once each voxel's radius R_κ is known, we use a linear interpolation scheme to distribute its scattering intensity $I_{\text{DFT}}(\mathbf{q}_\kappa)$ across two adjacent bins. Denoting $\lfloor R_\kappa \rfloor$ by r_{floor} and the fractional part by $r_{\text{frac}} = R_\kappa - \lfloor R_\kappa \rfloor$, each voxel's intensity is apportioned to bin r_{floor} with weight $(1 - r_{\text{frac}})$ and to bin $(r_{\text{floor}} + 1)$ with weight r_{frac}. Mathematically:

$$I_{\text{scatt}}(r_{\text{floor}}) \mathrel{+}= I_{\text{DFT}}(\mathbf{q}_\kappa)\,(1 - r_{\text{frac}}), \quad I_{\text{scatt}}(r_{\text{floor}} + 1) \mathrel{+}= I_{\text{DFT}}(\mathbf{q}_\kappa)\, r_{\text{frac}}.$$

Crucially, we implement this "accumulation" in a vectorized manner (e.g. via `accumarray` in MATLAB). This not only *eliminates* the triple nested loops but also ensures all voxel intensities are distributed in a single pass, dramatically reducing runtime.

3. **Better Normalization:** As in the original algorithm, we still apply the normalization by r^2 (or q^2) to account for the increasing number of points in concentric spherical shells at larger radii. However, as demonstrated above, this will cause the scattering intensities to be underestimated. To fix that, we normalize the scattering intensity with the effective counts of points instead of q^2.

By combining these three modifications, the improved algorithm offers the following advantages:

- **Significantly Reduced Computational Cost:** The use of vectorized routines and direct accumulations avoids $\mathcal{O}(n_x^3)$ triple-loop overhead, thus scaling better to larger grids.
- **Better Physical Explanation:** The employment of a better normalization method avoids shifting the intensities intentionally, which makes more sense concerning physical meaning since there is scattering intensity at the center of the grid.
- **More Accurate Ripple Behavior:** Empirically, we observe better agreement with analytical scattering curves for spherical models, especially in the ripples, which was proved to be most significant for pattern recognition.

Figure 8 highlights these improvements. Compared to the original algorithm, the proposed scheme aligns more closely with the analytical result across all q ranges. As in the original approach, the high-q region still exhibits minor fluctuations due to finite sampling and boundary effects, but these are substantially reduced by using adaptive padding (Section 2.2.1) and continuous density correction (Section 2.2.2). Overall, this improved orientation averaging framework produces higher-quality SAXS curves in less time, thereby facilitating more efficient large-scale data generation for ML workflows [2, 3].

2.1.5 Data Pre-processing

After generating the simulated SAXS data for models of varying sizes and shapes, it is necessary to address the issue of inconsistent data lengths caused

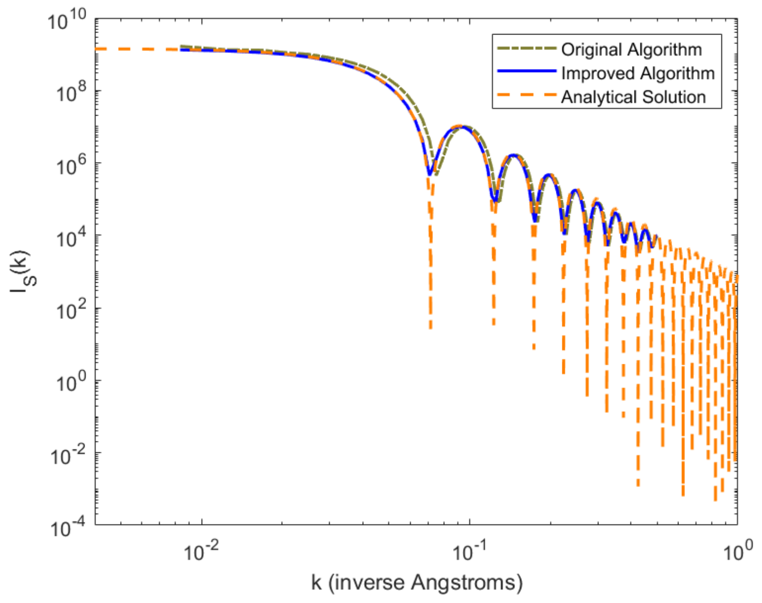

Figure 8: Comparison of the improved orientation averaging algorithm (red) with the original method (blue) and an analytical scattering curve (yellow) for a spherical LNP model.

by adaptive padding. Larger LNP models result in increased padding and a higher number of grid points (n_x), which leads to differences in the number of data points in the final SAXS curves. To overcome this, we adopt the solution proposed in [3], in which the scattering intensities are simulated for 500 (n_x) q values, uniformly distributed between:

$$q_{\min} \approx 0.04\,\mathrm{nm}^{-1} \quad \text{and} \quad q_{\max} \approx 3.00\,\mathrm{nm}^{-1},$$

These values were derived from an in-house experimental setup using an Anton Paar SAXSpoint 2.0 (Anton Paar, Graz, Austria) and represent a typical SAXS probing range while maintaining general applicability. Consequently, a global q range is defined for all LNP models. Although the LNPs are not assumed to be spherical, extrapolation using a Guinier function—defined as

$$I(q) = e^{A+Bq^2} \text{ [4]}$$

can provide a reasonable approximation for the scattering intensities of small LNPs before the Guinier region. To perform a weighted summation, we choose $q_{\min}$ as the smallest $q_{\min}$ value across the entire dataset. Mathematically, the universal q range is expressed as:

$$q_{\min} = \frac{2\pi}{n_x a} = 2\pi \times 0.00167\,\mathrm{nm}^{-1}, \qquad q_{\max} = \frac{\pi}{a} = \pi \times 0.5\,\mathrm{nm}^{-1}.$$

With a universal q range established for all SAXS data, each dataset can be interpolated or extrapolated to conform to the same q axis with a fixed number of points (e.g., 500 points). Comprehensive experiments indicate that cubic spline interpolation yields superior results.

2.1.6 Data Synthesizing

This section outlines the methodology for dataset creation, as illustrated by the flow chart in Fig. 9. The process is divided into three sequential steps: Within-Type Data Augmentation, Cross-Type Data Synthesizing, and Poisson Noise Addition. At this stage, the ML algorithm is developed using two distinct types of LNPs (Type 3 and Type 4, as shown in Fig. 3). Due to computational constraints, 81 samples of 500-point SAXS curves have been generated, covering LNP sizes ranging from 20 nm to 100 nm in 1 nm increments. These samples form the basis for further augmentation, which is essential for mitigating fitting challenges and enhancing the model's ability to capture relevant patterns in the data.

2.1.7 Within-Type Data Augmentation

In this study, the focus is on pure, dilute aqueous LNP solutions. Here, “pure” indicates that only LNPs are present in the sample, and “aqueous” signifies that water is the solvent. Given that water molecules contribute negligibly to the scattering intensity due to their small size relative to LNPs, we assume that only the LNPs contribute to the total scattering intensity. Moreover, as the solutions are dilute, interparticle interactions can be neglected. According to [4], the total scattering intensity of the sample can thus be treated as the weighted sum of the scattering intensities of individual particles, where each weight represents the number of particles of a specific size or type.

The first step in data augmentation is to increase the number of SAXS curves available for each LNP type. Based on the random residence time approach, the particle size distribution in finely divided systems is often observed to

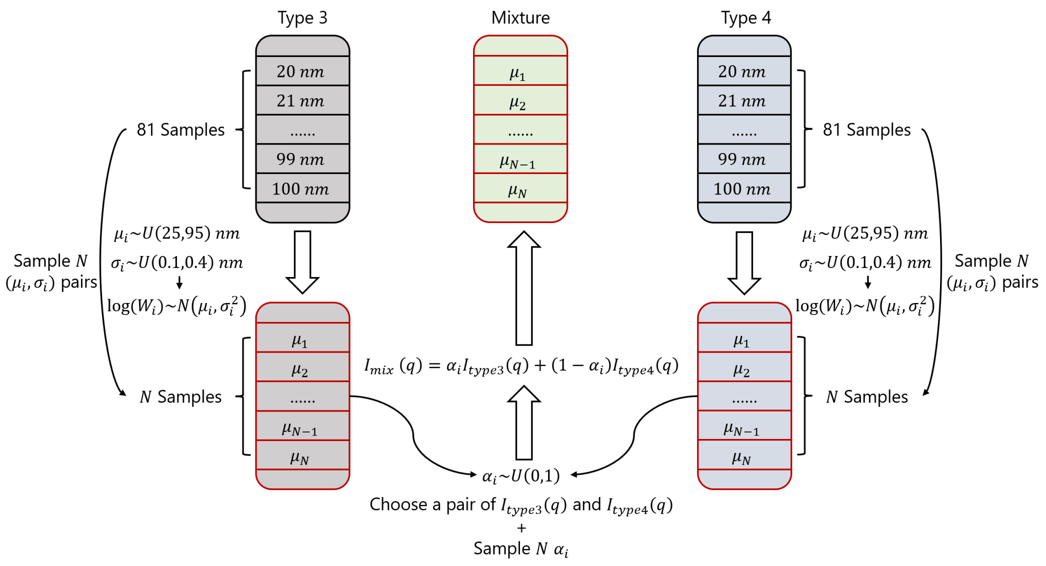
Type 3
Mixture
Type 4
20 nm
21 nm
......
99 nm
100 nm
81 Samples
μ_1
μ_2
μ_{N-1}
μ_N
Sample N (μ_i, σ_i) pairs
$\mu_i \sim U(25{,}95)\ nm$
$\sigma_i \sim U(0.1{,}0.4)\ nm$
$\log(W_i) \sim N(\mu_i, \sigma_i^2)$
N Samples
$I_{mix}(q) = \alpha_i I_{type3}(q) + (1-\alpha_i) I_{type4}(q)$
$\alpha_i \sim U(0,1)$
Choose a pair of $I_{type3}(q)$ and $I_{type4}(q)$
+
Sample N α_i

Figure 9: The Flow Chart of Data Synthesizing. Symbols: $U(a, b)$ denotes uniform distribution over interval (a, b).

follow a lognormal distribution [5]. To capture this behavior, we uniformly sample a mean value, μ, within the range of 25 to 95, and a corresponding standard deviation, σ, within the range of 1 to 4. Each μ–σ pair defines a unique lognormal distribution. This distribution is divided into 81 intervals, from which 81 normalized weights (summing to 1) are extracted—each weight corresponding to one of the original samples. Fig. 10 illustrates the whole process by taking pure Type 3 LNP solutions as examples.

For each LNP type, the 81 original 500-point SAXS curves are multiplied by their respective weights, and the weighted intensities are summed to produce a new 500-point curve. By repeating this procedure N times, each LNP type is expanded into N samples. Each resulting SAXS curve is interpreted as the scattering result of a dilute LNP solution characterized by a size distribution determined by the sampled μ and σ. In this framework, μ serves as the label for the sample, while σ is regarded as a noise factor.

2.1.8 Cross-Type Data Synthesizing

To more accurately mimic real-world samples, heterogeneous LNP solutions are simulated via a weighted linear combination approach. Specifically, two

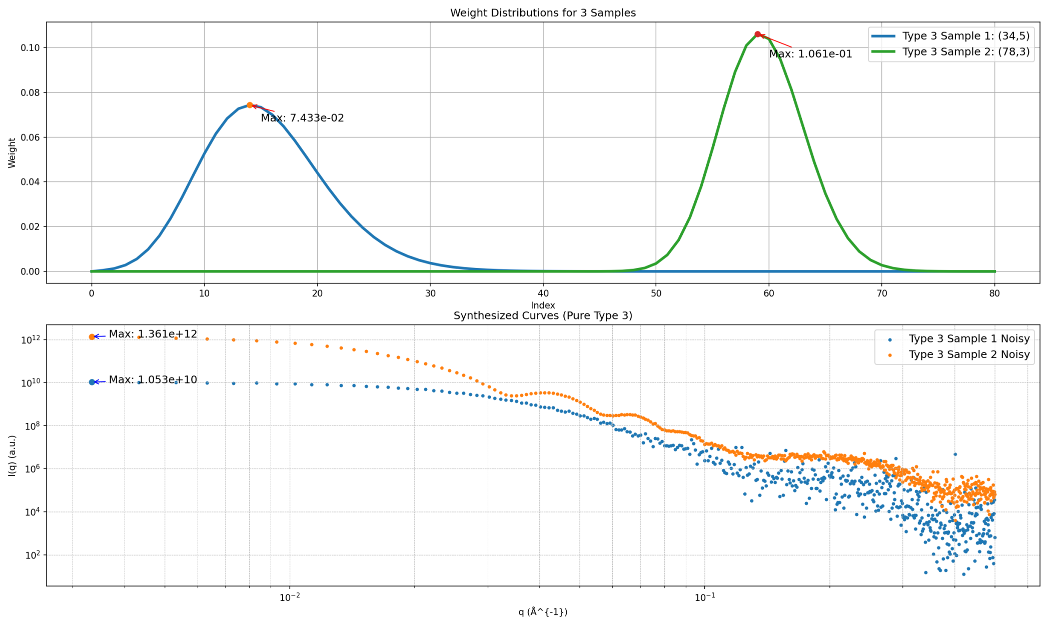

Figure 10: Demonstration of Within-Class Data Augmentation (With Poisson Noise Implemented)

samples are randomly selected from two different LNP types (e.g., Type 3 and Type 4), denoted as Curve 1 and Curve 2 with corresponding labels μ_1 and μ_2. A random weight α is then drawn from a uniform distribution on the interval $(0, 1)$ to determine the relative contributions of each sample. The resulting heterogeneous SAXS curve is computed as:

$$\text{Curve}_{\text{heter}} = \alpha \times \text{Curve}_1 + (1 - \alpha) \times \text{Curve}_2,$$

with the associated label defined by the weighted average:

$$\mu_{\text{heter}} = \alpha \times \mu_1 + (1 - \alpha) \times \mu_2.$$

Here, α represents the fraction of one LNP type in the mixture, serving as a target for regression. In addition to predicting α, determining μ_{heter} provides further insights into the size distribution of the constituent LNP types. Both α and μ_{heter} are therefore recorded in the synthesized heterogeneous dataset.

2.1.9 Poisson Noise Addition

To accurately simulate the experimental measurement process, it is crucial to model the inherent noise originating from photon counting statistics [3, 6]. In SAXS experiments, the number of photons detected in each q-bin is governed by Poisson statistics, meaning that the variance of the measured intensity is proportional to its mean. However, when simulating theoretical SAXS curves, since it is impossible to know the devices' real-world calibration coefficients (if they exist), directly applying a Poisson noise model would produce unrealistic noise levels.

To address this, we adopt a lognormal noise model that ensures the added noise remains strictly positive and effectively handles multiplicative effects. For each theoretical SAXS curve $I(q)$, we introduce a multiplicative noise factor $N(q)$ defined as:

$$N(q) = \exp\left(\sqrt{\sigma_n^2(q)}\,\epsilon - \frac{\sigma_n^2(q)}{2}\right),$$

where $\epsilon \sim \mathcal{N}(0,1)$ is a standard normally distributed random variable. This formulation leverages the properties of the lognormal distribution to guarantee that

$$E\left[N(q)\right] = 1,$$

thereby ensuring that the expected value of the noisy intensity remains $I(q)$, i.e.,

$$E\left[I_{\text{noisy}}(q)\right] = I(q).$$

The q-dependent variance $\sigma_n^2(q)$ is computed by:

$$\sigma_n^2(q) = \ln\left(1 + \frac{\alpha_n}{I(q)}\right),$$

where the noise scaling parameter α_n is sampled from a log-uniform distribution over a range (e.g., $[10^4,\ 10^{7.5}]$). This range is chosen based on extensive experimental evaluation to reflect the wide dynamic range observed in real SAXS measurements. Under this model, the variance of the noisy intensity approximates $\alpha\, I(q)$, consistent with the statistical behavior expected from photon counting. Finally, the noisy SAXS curve is obtained by:

$$I_{\text{noisy}}(q) = I(q) \times N(q),$$

The examples of simulated SAXS curves are illustrated in Fig. 11 with a setting of $[10^5, 10^{8.5}]$ to provide more realistic results.

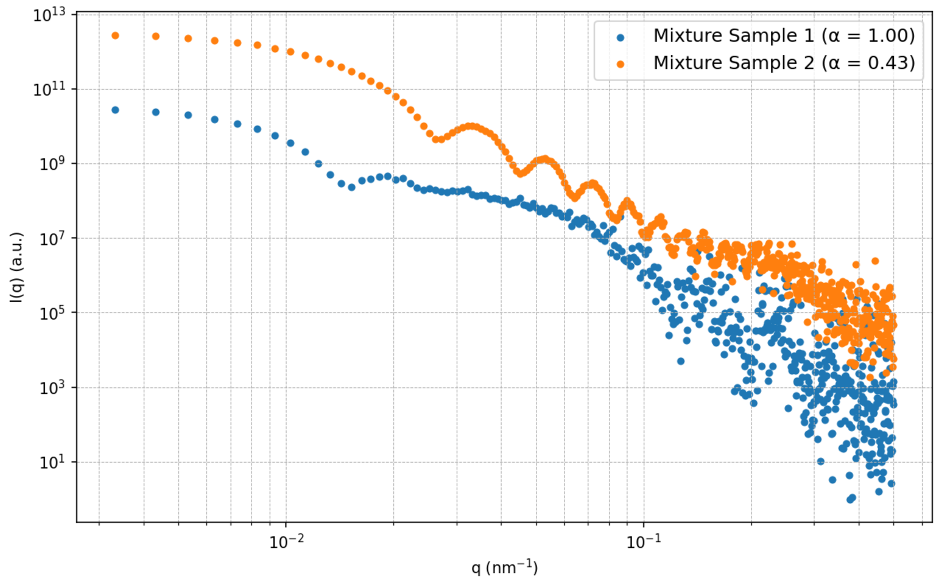

Figure 11: Example of simulated SAXS curves (With different mixing factor α)

2.2 ML Model Design

At this stage, two distinct convolutional neural network (CNN) architectures have been developed: one for classifying LNP solution types and another for regressing the fraction α. Both models incorporate effective normalization techniques—including batch normalization and feature scaling—to mitigate the effects of the large dynamic range present in the scattering intensity data, thereby enabling the networks to focus on the underlying shape of the SAXS curves.

The classification network (see Fig. 12) comprises two convolutional layers with 32 and 64 output channels and kernel sizes of 5 and 3, respectively. Each convolutional layer is followed by batch normalization and a ReLU activation function. The feature maps are then flattened and passed through two fully connected layers, with a dropout layer (rate 0.3) inserted between them to reduce overfitting. A softmax function at the output layer generates a valid probability distribution over the target classes.

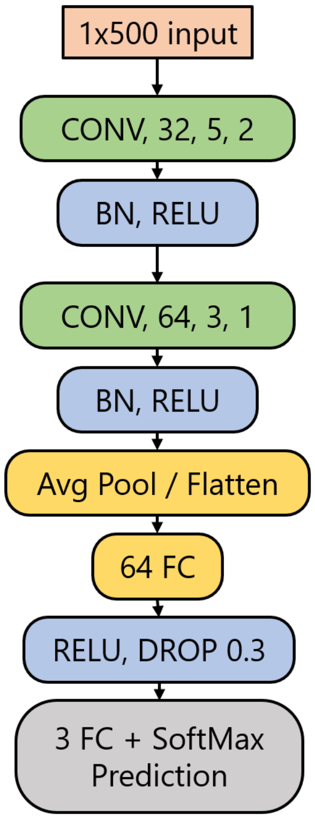

Figure 12: Network Architecture for Classification

The regression network (see Fig. 13) is designed to predict the mixing factor α and consists of three convolutional blocks. The convolutional layers in these blocks are configured with 32, 64, and 128 output channels and kernel sizes of 5, 5, and 3, respectively. Each layer is accompanied by batch normalization and a ReLU activation function. The final feature representation is then processed by three fully connected layers, with dropout layers (rate 0.15) inserted between them to mitigate overfitting. A sigmoid activation function at the final layer produces the regression output.

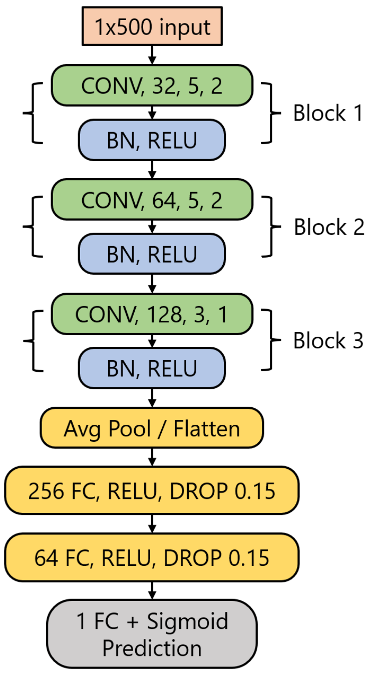

Figure 13: Network Architecture for Regression

3 Results

This section presents the experimental outcomes obtained from the developed data synthesizing pipeline and ML models. The results are organized into two subsections: one addressing the classification of LNP solution types and the other concerning the regression of the mixture fraction.

3.1 Results for Classification

The classification model demonstrated robust performance in differentiating between LNP solution types. As shown in Fig. 14, both the training and validation loss curves exhibit smooth convergence. The detailed classification metrics (see Table 1) indicate high precision and recall across all classes, with the F1-scores of 0.97 for Type 3, 0.98 for Type 4, and 0.93 for the Mixture category. An overall accuracy of 96% was achieved on a test set comprising 1200 samples. Furthermore, the confusion matrix in Fig. 15 reveals that misclassifications were minimal, thereby reinforcing the model's discriminative capability in handling heterogeneous LNP solutions.

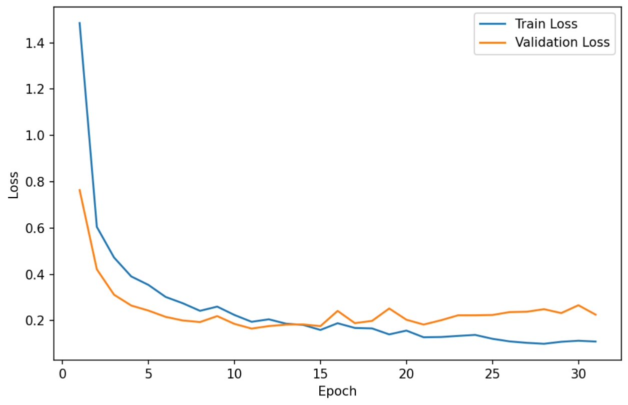

Figure 14: Training/Validation loss for classification

Table 1: Classification Report

Individual Classes

Class	Precision	Recall	F1-score	Support
Type 3	0.95	0.99	0.97	437
Type 4	0.95	1.00	0.98	380
Mixture	0.98	**0.89**	0.93	383
Aggregate Metrics				
Accuracy			0.96	1200

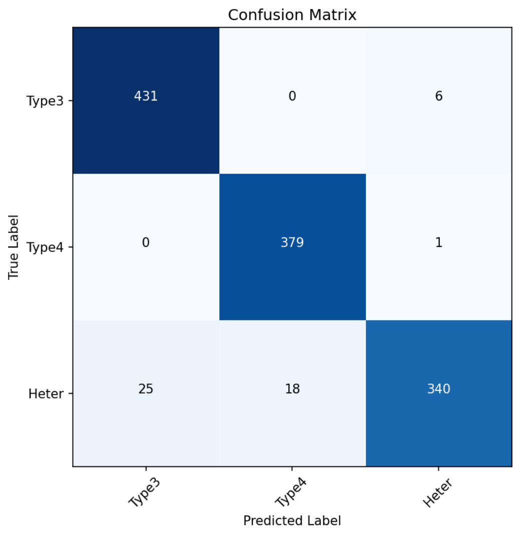

Figure 15: Confusion matrix for classification

3.2 Results for Fraction Regression

The regression model, designed to predict the fraction (α) in heterogeneous samples, exhibited excellent predictive performance. The model achieved an R^2 score of 0.9636 and a total test mean squared error (MSE) of 0.003019. The training losses are summarized in Fig. 16. These results underscore the model's capacity to capture the underlying relationships in the SAXS data. Additionally, the regression performance is further corroborated by the log-scaled prediction results displayed in Fig. 17, which demonstrate a high degree of correlation between the predicted and true fraction values with minimal bias.

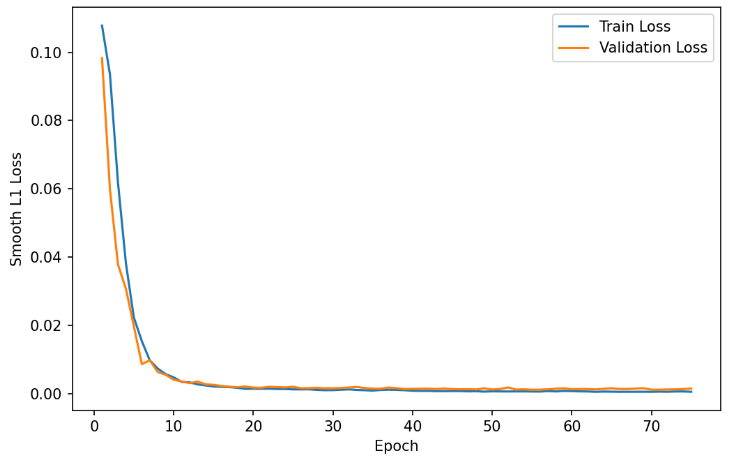

Figure 16: Training Losses for Regression Model

4 Conclusion

In this work, we presented a physics-based simulation pipeline and corresponding ML models for the characterization of LNP drug delivery vehicles using SAXS data. By carefully constructing realistic 3D LNP models, applying adaptive padding in the fast Fourier transform, introducing a continuous density correction factor, and incorporating lognormal and Poisson-like noise, we generated high-fidelity SAXS data across a broad range of particle sizes and mixture fractions.

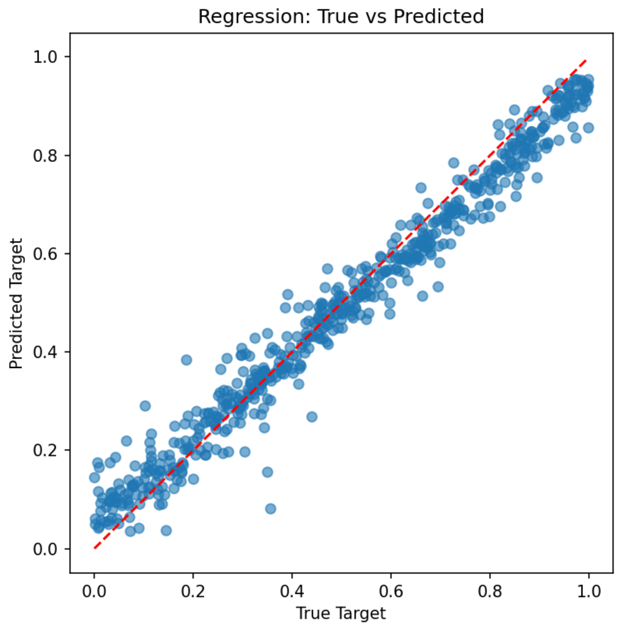

Figure 17: Prediction Results of Mixing Factor α

We then trained CNNs for two tasks: (1) classifying LNP solution types, and (2) predicting the mixture fraction in heterogeneous samples. The classification network achieved an overall accuracy of 96%, demonstrating its strong ability to distinguish between pure LNP solutions of different types and their mixtures. The regression model achieved a high R^2 of 0.9636, successfully capturing the underlying relationships in the data to predict the mixture fraction (α) of two LNP types.

These results highlight the potential of ML-driven SAXS analysis as a rapid and cost-effective complement or alternative to more resource-intensive methods such as cryo-EM. By offering quantitative insights into LNP size distributions and compositional fractions, the proposed approach paves the way for more efficient formulation development and quality control in nanomedicine. Nevertheless, future work is needed to further validate our pipeline with real experimental SAXS datasets, investigate the impact of sample polydispersity and other solvent effects, and extend the methodology to more complex multi-type LNP mixtures.

References

[1] A. Roesch, S. Zölls, D. Stadler, C. Helbig, K. Wuchner, G. Kersten, A. Hawe, W. Jiskoot, and T. Menzen, “Particles in biopharmaceutical formulations, part 2: An update on analytical techniques and applications for therapeutic proteins, viruses, vaccines and cells,” *Journal of Pharmaceutical Sciences*, vol. 111, pp. 933–950, 2022.

[2] K. Schmidt-Rohr, “Simulation of small-angle scattering curves by numerical fourier transformation,” *Journal of Applied Crystallography*, vol. 40, pp. 16–25, 2007.

[3] M. Röding, P. Tomaszewski, S. Yu, M. Borg, and J. Rönnols, “Machine learning-accelerated small-angle x-ray scattering analysis of disordered two- and three-phase materials,” *Frontiers in Materials*, vol. 9, p. 956839, 2022.

[4] SASView, “Sasview documentation index: Corfunc technical documentation,” n.d., accessed: 2025-03-02. [Online]. Available: https://www.sasview.org/docs/user/qtgui/Perspectives/Corfunc/corfunc-technical.html

[5] L. Kiss, J. Söderlund, G. Niklasson, and C. Granqvist, “The real origin of lognormal size distributions of nanoparticles in vapor growth processes,” *Nanostructured Materials*, vol. 12, no. 1–4, pp. 327–332, 1999.

[6] S. Sedlak, L. Bruetzel, and J. Lipfert, “Quantitative evaluation of statistical errors in small-angle x-ray scattering measurements,” *Journal of Applied Crystallography*, vol. 50, no. 2, pp. 621–630, Mar 2017.

5 Appendix

5.1 Codes for SAXS simulation

```
function [q, Iq] = get_Iq(filename, ratio)
    % cubic_effective_norm: Computes the scattering intensity I(q)
    % and normalizes it using the effective voxel count in each bin.
    %
    % Inputs:
    % - filename: Path to the .mat file containing density data
    % (should include the variable rhoS)
    % - ratio: Ratio between the padded cube size and the non-zero
    % region size M (default = 6)
    %
    % Outputs:
    % - q: Array of scattering vector magnitudes
    % - Iq: Corresponding scattering intensities I(q)

    if nargin < 2
        ratio = 6;
    end

    %% 1. Load data and extract the effective region
    data = load(filename);
    small_cube = data.rhoS;
    M_full = size(small_cube, 1);

    % Extract the actual effective region size M from the filename
    % (e.g., 'd20' implies M = 20)
    token = regexp(filename, 'd(\d+)', 'tokens');
    if ~isempty(token)
        M = str2double(token{1}{1});
    else
        M = M_full;
    end

    % Extract the center MMM region from the full-size data
    start_small = floor((M_full - M) / 2) + 1;
    end_small = start_small + M - 1;
    effective_model = small_cube(start_small:end_small, ...
                             start_small:end_small, ...
                             start_small:end_small);

    %% 2. Embed the effective model into a larger cube
    nx = ratio * M;

    rhoS = zeros(nx, nx, nx);
    center_position = nx / 2;
    start_pos = floor(center_position - M / 2) + 1;
    end_pos = start_pos + M - 1;
    rhoS(start_pos:end_pos, start_pos:end_pos, start_pos:end_pos) =
        effective_model;

    %% 3. Perform FFT and compute 3D scattering amplitude squared (I
        (q) in 3D)
    Iq3D = abs(fftn(rhoS)).^2;
    Iq3D = fftshift(Iq3D);

    %% 4. Prepare sinc correction factor
    iqcent = nx/2 + 1;
    iq1 = (1:nx) - iqcent;
    q1ad2 = pi * iq1 / nx + 1e-8; % avoid division by zero
    sincsqr_1d = (sin(q1ad2) ./ q1ad2).^2;

    %% 5. Perform spherical averaging using vectorization
    % Construct a 3D meshgrid (origin at center)
    [X, Y, Z] = ndgrid(-floor(nx/2):(ceil(nx/2)-1));
    R = sqrt(X.^2 + Y.^2 + Z.^2); % avoid R=0

    % Linearly interpolate between two adjacent bins
    rFloor = floor(R);
    rFrac = R - rFloor;
    rFloor(rFloor < 1) = 1;
    rFloor(rFloor > nx) = nx;
    rFloorP1 = rFloor + 1;
    rFloorP1(rFloorP1 < 1) = 1;
    rFloorP1(rFloorP1 > nx) = nx;

    % Lookup FFT values for each voxel and apply sinc correction
    idxX = X + iqcent;
```

```
idxY = Y + iqcent;
idxZ = Z + iqcent;
vals_Iq3D = Iq3D(sub2ind(size(Iq3D), idxX, idxY, idxZ));
vals_sinc = sincsqr_1d(idxX) .* sincsqr_1d(idxY) .* sincsqr_1d(
    idxZ);
vals = vals_Iq3D .* vals_sinc;

% Assign weights
wFloor = 1 - rFrac;
wFloorP1 = rFrac;

% Use accumarray to accumulate scattering intensity
    contributions per bin
Iscatt_part1 = accumarray(rFloor(:), vals(:) .* wFloor(:), [nx
    ,1]);
Iscatt_part2 = accumarray(rFloorP1(:), vals(:) .* wFloorP1(:), [
    nx,1]);
Iscatt = Iscatt_part1 + Iscatt_part2;

% Also accumulate effective voxel counts (sum of interpolation
    weights)
count_part1 = accumarray(rFloor(:), wFloor(:), [nx,1]);
count_part2 = accumarray(rFloorP1(:), wFloorP1(:), [nx,1]);
binCount = count_part1 + count_part2;

%% 6. Normalize using effective voxel count in each bin
nxd2 = nx / 2;
IqVal = zeros(nxd2 - 1, 1);
for iq = 2:nxd2
    if binCount(iq) > 0
        IqVal(iq - 1) = Iscatt(iq) / binCount(iq);
    else
        IqVal(iq - 1) = 0;
    end
end

%% 7. Compute q-axis and plot (optional)
a = 1; % Real length per voxel (e.g., 1 /pt)
dq = 1 / (nx * a); % q-step
q = dq * (2 : nxd2); % q-axis
```

```

    Iq = IqVal;
end
```

Listing 1: MATLAB code to get the scattering intensity of single LNP model

```
clear;
clc;

% 1. Batch read all .mat files from ./type_i_model/
fileList = dir('./type_4_model/*.mat');

% 2. Prepare output directory and HDF5 file path
outputDir = './output';
if ~exist(outputDir, 'dir')
    mkdir(outputDir);
end
hdf5_filename = fullfile(outputDir, 'raw4.h5');

% If the file already exists, delete it to avoid conflict with old
    data
if exist(hdf5_filename, 'file')
    delete(hdf5_filename);
end

% 3. Process each .mat file one by one
for k = 1:length(fileList)
    % Get the full path of the .mat file
    filename = fullfile(fileList(k).folder, fileList(k).name);

    % 4. Call get_Iq function to compute q and Iq (use ratio = 6,
        modify if needed)
    [q, Iq] = get_Iq(filename, 6);

    % 5. Combine q and Iq into a two-column array (N2) for easier
        access
    data_qIq = [q(:), Iq(:)];

    % 6. Extract filename (without path or extension) for dataset
        naming
    [~, name, ~] = fileparts(filename);
    dataset_name = ['/', name, '_qIq']; % e.g., /XX_qIq

    % 7. Create dataset in HDF5 file and write the data
    h5create(hdf5_filename, dataset_name, size(data_qIq));
    h5write(hdf5_filename, dataset_name, data_qIq);

    disp(['Processed and saved: ', name]);
end
```

Listing 2: MATLAB main script for data collection

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
import os
import time

def is_point_inside_ellipsoid(points, ellipsoid):
    a, b, c, x0, y0, z0 = ellipsoid
    return (((points[..., 0] - x0) / a) ** 2 +
            ((points[..., 1] - y0) / b) ** 2 +
            ((points[..., 2] - z0) / c) ** 2) <= 1

def is_point_inside_sphere(points, radius, center):
    return np.sum((points - center) ** 2, axis=-1) <= radius ** 2

def is_point_inside_cylinder(points, radius, height, center):
    x, y, z = points[..., 0] - center[0], points[..., 1] - center
        [1], points[..., 2] - center[2]
    return (x**2 + y**2 <= radius**2) & (np.abs(z) <= height / 2)

def lnp(cube_size, sld_core, sld_shell, sld_fill, sld_solvent,
    alpha, type_option):
    # Create 3D coordinate grid
    x, y, z = np.meshgrid(np.arange(cube_size), np.arange(cube_size)
        , np.arange(cube_size), indexing='ij')
    points = np.stack([x, y, z], axis=-1)
    cube = np.zeros((cube_size, cube_size, cube_size))

    # Define geometric parameters
    r1 = 28.57 * alpha
    r2 = 35.71 * alpha
    shell = 4
    l = 100

    outer1 = np.array([r1, r1, r1, 50, 50, (1 + alpha) * l / 2 - r1
        ])
    outer2 = np.array([r1, r1, r2, 50, 50, (1 - alpha) * l / 2 + r2
        ])
    inner1 = np.array([r1 - shell, r1 - shell, r1 - shell, 50, 50,
        (1 + alpha) * l / 2 - r1])
    inner2 = np.array([r1 - shell, r1 - shell, r2 - shell, 50, 50,
        (1 - alpha) * l / 2 + r2])

    # Calculate the z-plane for cutoff
    z0_inner1 = inner1[5]
    z0_inner2 = inner2[5]
    plane0 = (z0_inner1 + z0_inner2) / 2
    plane_up = plane0 + 2
    plane_down = plane0 - 2

    # Check point membership for each region
    inside_inner1 = is_point_inside_ellipsoid(points, inner1)
    inside_inner2 = is_point_inside_ellipsoid(points, inner2)
    inside_outer1 = is_point_inside_ellipsoid(points, outer1)
    inside_outer2 = is_point_inside_ellipsoid(points, outer2)

    if type_option == 1:
        # Type 1: Standard coreshell structure
        core_mask = inside_inner1 | inside_inner2
        shell_mask = inside_outer1 | inside_outer2
        cube[core_mask] = sld_core
        cube[shell_mask & ~core_mask] = sld_shell
        cube[~(core_mask | shell_mask)] = sld_solvent

    elif type_option == 2:
        # Type 2: Core only in upper half, fill in lower half
        core_mask = (inside_inner1 & (points[..., 2] >= plane_up))
        fill_mask = (inside_inner2 & (points[..., 2] <= plane_down))
        shell_mask = (inside_outer1 | inside_outer2) & ~core_mask &
            ~fill_mask
        cube[core_mask] = sld_core
        cube[fill_mask] = sld_fill
        cube[shell_mask] = sld_shell
        cube[~(core_mask | fill_mask | shell_mask)] = sld_solvent

    elif type_option == 3:
        # Type 3: Core in both ends, hollow middle
        core_mask = (inside_inner1 & (points[..., 2] >= plane_up)) |
             \
                    (inside_inner2 & (points[..., 2] <= plane_down))
        shell_mask = inside_outer1 | inside_outer2
        cube[core_mask] = sld_core
        cube[shell_mask & ~core_mask] = sld_shell
        cube[~(core_mask | shell_mask)] = sld_solvent

    elif type_option == 4:
        # Type 4: Sphere-only model
        radius = 50 * alpha
        center = np.array([cube_size // 2, cube_size // 2, cube_size
             // 2])
        sphere_mask = np.sum((points - center) ** 2, axis=-1) <=
            radius ** 2
        cube[sphere_mask] = sld_shell # or sld_core if needed

    elif type_option == 5:
        # Type 5: Cylinder model
        cylinder_radius = 10
        cylinder_height = 50
        cylinder_center = np.array([cube_size // 2, cube_size // 2,
            cube_size // 2])
        cylinder_mask = is_point_inside_cylinder(points,
            cylinder_radius, cylinder_height, cylinder_center)
        cube[cylinder_mask] = 1 # Set to sld_core or 1

    else:
        raise ValueError("Invalid type_option. Choose between 1, 2,
            3, 4 or 5.")

    return cube

```

```python
def visualize_lnp(cube, sld_core, sld_shell, sld_fill, sld_solvent):
    # Define a boolean mask for all visible voxels
    voxels = (cube == sld_core) | (cube == sld_shell) | (cube == sld_fill) | (cube == sld_solvent)

    # Assign colors based on material types
    colors = np.empty(cube.shape, dtype=object)
    colors[cube == sld_core] = 'red'
    colors[cube == sld_shell] = 'green'
    colors[cube == sld_fill] = 'blue'
    colors[cube == sld_solvent] = 'cyan'

    # Create 3D plot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.voxels(voxels, facecolors=colors, edgecolor='k')

    # Set axis labels
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.show()

def visualize_slice(cube, slice_index, alpha):
    # Visualize a 2D slice of the 3D model at a specified Y index
    slice_data = cube[:, slice_index, :]
    plt.figure()
    plt.imshow(slice_data, cmap='viridis', origin='lower')
    plt.colorbar(label='SLD')
    plt.title(f'Slice at y={slice_index}, diameter={100 * alpha:.2f}')
    plt.xlabel('Z')
    plt.ylabel('X')
    plt.show(block=True)

def save_lnp_data(alpha, cube, X, Y, Z):
    # Save the 3D LNP model and coordinate axes to a .mat file
    directory = 'type_4_model'
    os.makedirs(directory, exist_ok=True)
    filename = os.path.join(directory, f'd{int(round(alpha*100))}.
        mat')
    scipy.io.savemat(filename, {'rhoS': cube, 'X': X, 'Y': Y, 'Z': Z
        })

def main():
    # Model parameters
    cube_size = 100
    sld_core = 16.0
    sld_shell = 9.0
    sld_fill = sld_shell
    sld_solvent = 0.0

    # Generate coordinate grids
    X = np.linspace(-cube_size / 2, cube_size / 2, cube_size)
    Y = np.linspace(-cube_size / 2, cube_size / 2, cube_size)
    Z = np.linspace(-cube_size / 2, cube_size / 2, cube_size)

    # Alpha controls overall particle size
    alphas = np.linspace(0.2, 1, 81)

    for alpha in alphas:
        cube = lnp(cube_size, sld_core, sld_shell, sld_fill,
            sld_solvent, alpha, type_option=4)
        save_lnp_data(alpha, cube, X, Y, Z)
        # visualize_slice(cube, 50, alpha)
        # visualize_lnp(cube, sld_core, sld_shell, sld_fill,
            sld_solvent)

if __name__ == "__main__":
    st = time.time()
    main()
    et = time.time()
    print(et - st)
```

Listing 3: Python script to generate LNP 3D models

```
import h5py
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt
```

```

# Read the input HDF5 file
input_file_path = "./output/raw4.h5" # Modify this to the actual path
output_file_path = "./output/clean4.h5" # Path for the output file

with h5py.File(input_file_path, "r") as h5_file:
    dataset_names = list(h5_file.keys())

    # Extract the q_min from each dataset and find the minimum q_min across all
    q_mins = [h5_file[name][0, 0] for name in dataset_names]
    q_min = min(q_mins)
    q_max = 0.5 # Fixed q_max

    # Generate a unified q-axis
    N_fixed = 500
    q_fixed = np.linspace(q_min, q_max, N_fixed)

    # Create a new HDF5 file for output
    with h5py.File(output_file_path, "w") as h5_out:
        # Store the standardized q-axis
        h5_out.create_dataset("q_fixed", data=q_fixed)

        # Interpolate each dataset and write to new HDF5
        for dataset in dataset_names:
            q_orig, Iq_orig = h5_file[dataset][:]

            # Apply cubic spline interpolation
            spline_func = CubicSpline(q_orig, Iq_orig, extrapolate=True)
            Iq_spline = spline_func(q_fixed) # Interpolated I(q) without noise

            h5_out.create_dataset(dataset, data=Iq_spline)

print(f"All datasets have been interpolated and saved to {output_file_path}")
```

Listing 4: Python script to preprocess the SAXS data collected by MATLAB main script

```python
import re
import numpy as np
import h5py
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix


def load_separately_with_names(file_path):
    """
        Read all datasets in an HDF5 file except 'q_fixed', extract
            numbers from dataset names,
        and return a list of names (numbers) and a corresponding
            array of data.
        Assumes dataset names follow the pattern "d20_qIq", "
            d100_qIq", etc.
        """
    names = []
    data_list = []
    with h5py.File(file_path, 'r') as f:
        for key in f.keys():
            if key == 'q_fixed':
                continue
            match = re.search(r'd(\d+)_qIq', key)
            if match:
                num = int(match.group(1))
                names.append(num)
                data_list.append(f[key][:])
    sorted_indices = np.argsort(names)
    sorted_names = [names[i] for i in sorted_indices]
    sorted_data = np.vstack([data_list[i] for i in sorted_indices])
    return sorted_names, sorted_data

def load_all_types(file_type3, file_type4):
    """
```

```python
        Load datasets for Type 3 and Type 4 separately and return two tuples: (names, data)
    """
    names3, data3 = load_separately_with_names(file_type3)
    names4, data4 = load_separately_with_names(file_type4)
    return (names3, data3), (names4, data4)

def augment_type_data(names, X, new_sample_count=2000):
    """
        Augment data for one type.
        For each sample, generate a log-normal weighted sum of the original data using a random mean and sigma.
        Labels are not needed here since the type label will be assigned externally.
    """
    new_samples = []
    diameters = np.array(names, dtype=np.float32)
    for _ in range(new_sample_count):
        mean_real = np.random.uniform(25, 95)
        sigma_real = np.random.uniform(0.1, 0.4)
        mu = np.log(mean_real ** 2 / np.sqrt(mean_real ** 2 + sigma_real ** 2))
        sigma_log = np.sqrt(np.log(1 + (sigma_real ** 2 / mean_real ** 2)))
        weights = (1.0 / (diameters * sigma_log * np.sqrt(2 * np.pi))) * \
                  np.exp(- (np.log(diameters) - mu) ** 2 / (2 * sigma_log ** 2))
        weights /= np.sum(weights)
        new_curve = np.sum(weights[:, np.newaxis] * X, axis=0)
        new_samples.append(new_curve)
    return np.array(new_samples), None

# Apply lognormal noise
def add_noise_to_curve(curve):
    """
        Add lognormal noise to a single scattering curve using Poisson-like scaling.
    """
    curve_safe = np.maximum(curve, 1e-12)
    log_alpha = np.random.uniform(np.log(1e4), np.log(10 ** 7.5))
    alpha = np.exp(log_alpha)
    sigma2 = np.log(1 + alpha / curve_safe)
    epsilon = np.random.randn(curve.shape[0])
    noise_factor = np.exp(np.sqrt(sigma2) * epsilon - sigma2 / 2)
    return curve * noise_factor

def add_noise_to_data(data):
    """
        Apply add_noise_to_curve to each curve in the dataset.
    """
    return np.array([add_noise_to_curve(data[i]) for i in range(data
        .shape[0])])

# Mix samples from two types
def generate_heter_data(X0, _, X1, __, desired_count=1000):
    """
        Generate heterogeneous samples by linearly mixing random
            pairs of Type 3 and Type 4 samples.
        Labels for mixture ratios are ignored; all are treated as a
            new class in classification.
    """
    n0, n1 = X0.shape[0], X1.shape[0]
    X_heter_list = []
    for _ in range(desired_count):
        idx0 = np.random.randint(n0)
        idx1 = np.random.randint(n1)
        alpha = np.random.rand()
        mixed_curve = alpha * X0[idx0] + (1 - alpha) * X1[idx1]
        X_heter_list.append(mixed_curve)
    return np.array(X_heter_list), None, None

# Generate and prepare dataset for classification
def load_hdf5_data_for_classification(file_type3, file_type4,
    desired_count=2000, heter_count=1000):
    """
        Load, augment, add noise, and prepare training data for
            classification of 3 classes:
        Type 3 (label 0), Type 4 (label 1), and heterogeneous
            mixtures (label 2).
        Data is also transformed to log10 scale after noise
            injection.
    """
    (names3, data3), (names4, data4) = load_all_types(file_type3,
        file_type4)
    X_type3, _ = augment_type_data(names3, data3, desired_count)
    X_type4, _ = augment_type_data(names4, data4, desired_count)
    X_heter, _, _ = generate_heter_data(X_type3, None, X_type4, None
        , heter_count)
    y_type3 = np.zeros(X_type3.shape[0], dtype=np.int64)
    y_type4 = np.ones(X_type4.shape[0], dtype=np.int64)
    y_heter = np.full(X_heter.shape[0], 2, dtype=np.int64)
    X_type3 = np.log10(np.maximum(add_noise_to_data(X_type3), 1e-12)
        )
    X_type4 = np.log10(np.maximum(add_noise_to_data(X_type4), 1e-12)
        )
    X_heter = np.log10(np.maximum(add_noise_to_data(X_heter), 1e-12)
        )
    X_all = np.concatenate([X_type3, X_type4, X_heter], axis=0)
    y_all = np.concatenate([y_type3, y_type4, y_heter], axis=0)
    return X_all, y_all

# CNN Model Definition
class ClassificationCNN(nn.Module):
    """
    CNN model definition for classifying SAXS curves into 3 classes.
    """
    def __init__(self, input_length, num_classes=3):
        super(ClassificationCNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(1, 32, kernel_size=5, padding=2),
            nn.BatchNorm1d(32),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(64 * input_length, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, num_classes)
        )

    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)

def train_the_model_classification(file_type3, file_type4,
    desired_count):
    """
    End-to-end training routine:
    - Load data and augment
    - Add noise and transform
    - Normalize and split into train/val/test
    - Train CNN with early stopping
    - Plot loss curves
    - Evaluate and print classification metrics
    """

    X, y = load_hdf5_data_for_classification(file_type3, file_type4,
         desired_count, desired_count)
    print("X shape:", X.shape)
    print("y shape:", y.shape)
    print("Class counts:", {cls: int(np.sum(y == cls)) for cls in np
        .unique(y)})

    # Normalize input features
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Split into train/val/test
    from sklearn.model_selection import train_test_split
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y
        , test_size=0.2, random_state=42)
    X_train, X_val, y_train, y_val = train_test_split(X_train_val,
        y_train_val, test_size=0.2, random_state=42)

    # Convert to PyTorch tensors
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
    X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
    X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
    y_train_tensor = torch.tensor(y_train, dtype=torch.long)
    y_val_tensor = torch.tensor(y_val, dtype=torch.long)
    y_test_tensor = torch.tensor(y_test, dtype=torch.long)

    train_loader = DataLoader(TensorDataset(X_train_tensor,
        y_train_tensor), batch_size=256, shuffle=True)
    val_loader = DataLoader(TensorDataset(X_val_tensor, y_val_tensor
        ), batch_size=256, shuffle=False)
    test_loader = DataLoader(TensorDataset(X_test_tensor,
        y_test_tensor), batch_size=256, shuffle=False)

    device = torch.device("cuda" if torch.cuda.is_available() else "
        cpu")
    model = ClassificationCNN(input_length=X.shape[1]).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    epochs = 120
    patience = 20
    best_val_loss = float('inf')
    best_model_weights = None
    counter = 0
    train_losses, val_losses = [], []

    for epoch in range(epochs):
        model.train()
        total_train_loss = 0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
        train_loss = total_train_loss / len(train_loader)
        train_losses.append(train_loss)

        model.eval()
        total_val_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                X_batch, y_batch = X_batch.to(device), y_batch.to(
                    device)
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                total_val_loss += loss.item()
        val_loss = total_val_loss / len(val_loader)
        val_losses.append(val_loss)
        print(f"Epoch {epoch + 1}: Train Loss = {train_loss:.6f},
            Val Loss = {val_loss:.6f}")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_weights = model.state_dict()
            counter = 0
        else:
            counter += 1
            if counter >= patience:
                print(f"Early stopping triggered at epoch {epoch +
                    1}. Best Val Loss: {best_val_loss:.6f}")
                break

    if best_model_weights is not None:
        model.load_state_dict(best_model_weights)

    # Plot training vs validation loss
    plt.figure()
    plt.plot(train_losses, label="Train Loss")
    plt.plot(val_losses, label="Validation Loss")
    plt.legend()
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training vs Validation Loss")
    plt.show()

    # Evaluate on test set
    model.eval()
    y_preds, y_true = [], []
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            outputs = model(X_batch)
            predicted = torch.argmax(outputs, dim=1)
            y_preds.extend(predicted.cpu().numpy())
            y_true.extend(y_batch.numpy())

    print("Test Accuracy:", accuracy_score(y_true, y_preds))
    print("Classification Report:")
    print(classification_report(y_true, y_preds))
    print("Confusion Matrix:")
    cm = confusion_matrix(y_true, y_preds)
    print(cm)

    # Confusion matrix heatmap
    plt.figure(figsize=(6, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    classes = ['Type 3', 'Type 4', 'Heter']
    plt.xticks(np.arange(len(classes)), classes, rotation=45)
    plt.yticks(np.arange(len(classes)), classes)
    thresh = cm.max() / 2.0
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                        ha="center", color="white" if cm[i, j] > thresh
                            else "black")
    plt.ylabel("True Label")
    plt.xlabel("Predicted Label")
    plt.tight_layout()
    plt.show()
```

```python
# Entry point
if __name__ == "__main__":
    np.random.seed(31)
    torch.manual_seed(31)
    file_type3 = "./output/clean3.h5"
    file_type4 = "./output/clean4.h5"
    desired_count = 2000
    train_the_model_classification(file_type3, file_type4, desired_count)
```

Listing 5: Python main script for 2-types (Type 3 4) classification

```python
import re
import numpy as np
import h5py
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, r2_score

from model import Resnet, CNN, simpleCNN # You can switch architectures here

def load_separately_with_names(file_type):
    """
    Load all datasets from an HDF5 file except 'q_fixed', extract numbers from dataset names (e.g., 'd20_qIq'),
    and return sorted (names, data) arrays.
    """
    names = []
    data_list = []
    with h5py.File(file_type, 'r') as f:
        for key in f.keys():
            if key == 'q_fixed':
                continue
            m = re.search(r'd(\d+)_qIq', key)
            if m:
                num = int(m.group(1))
                names.append(num)
            data_list.append(f[key][:])
    sorted_indices = np.argsort(names)
    sorted_names = [names[i] for i in sorted_indices]
    sorted_data = np.vstack([data_list[i] for i in sorted_indices])
    return sorted_names, sorted_data

def load_all_types(file_type3, file_type4):
    """
    Load Type 3 and Type 4 data separately, ensuring sorted order.
    """
    names3, data3 = load_separately_with_names(file_type3)
    names4, data4 = load_separately_with_names(file_type4)
    return (names3, data3), (names4, data4)

def augment_type_data(names, X, new_sample_count=2000):
    """
    Augment data using lognormal-weighted mixing from the original 81 SAXS curves.
    Each generated sample gets a synthetic label (mean diameter).
    """
    new_samples = []
    labels = []
    diameters = np.array(names, dtype=np.float32)
    for _ in range(new_sample_count):
        mean_real = np.random.uniform(25, 95)
        sigma_real = np.random.uniform(0.1, 0.4)
        mu = np.log(mean_real ** 2 / np.sqrt(mean_real ** 2 + sigma_real ** 2))
        sigma_log = np.sqrt(np.log(1 + (sigma_real ** 2 / mean_real ** 2)))
        weights = (1.0 / (diameters * sigma_log * np.sqrt(2 * np.pi))) * \
               np.exp(- (np.log(diameters) - mu) ** 2 / (2 * sigma_log ** 2))
        weights /= np.sum(weights)
        new_curve = np.sum(weights[:, np.newaxis] * X, axis=0)
        new_samples.append(new_curve)
        labels.append(mu)
    return np.array(new_samples), np.array(labels)

```

```python
def add_noise_to_curve(curve):
    """
    Add lognormal (Poisson-like) noise to a SAXS curve.
    """
    curve_safe = np.maximum(curve, 1e-12)
    log_alpha = np.random.uniform(np.log(1e4), np.log(10 ** 7.5))
    alpha = np.exp(log_alpha)
    sigma2 = np.log(1 + alpha / curve_safe)
    epsilon = np.random.randn(curve.shape[0])
    noise_factor = np.exp(np.sqrt(sigma2) * epsilon - sigma2 / 2)
    return curve * noise_factor

def add_noise_to_data(data):
    """
    Add lognormal noise to each sample (row) in the dataset.
    """
    return np.array([add_noise_to_curve(sample) for sample in data])

def generate_heter_data(X0, labels0, X1, labels1, desired_count
    =1000):
    """
    Generate heterogeneous SAXS curves by mixing Type 3 and Type 4
        data.
    For each mixed sample, return:
      - the mixed curve,
      - its alpha (mixing ratio),
      - and its weighted mean diameter (mu).
    """
    X_heter_list = []
    alpha_list = []
    mu_heter_list = []
    for _ in range(desired_count):
        idx0 = np.random.randint(len(X0))
        idx1 = np.random.randint(len(X1))
        alpha = np.random.rand()
        mixed_curve = alpha * X0[idx0] + (1 - alpha) * X1[idx1]
        mu_val = alpha * labels0[idx0] + (1 - alpha) * labels1[idx1]
        X_heter_list.append(mixed_curve)
        alpha_list.append(alpha)
        mu_heter_list.append(mu_val)
    return np.array(X_heter_list), np.array(alpha_list, dtype=np.
        float32), np.array(mu_heter_list, dtype=np.float32)

def load_hdf5_data_for_regression(file_type3, file_type4,
    desired_count=2000, heter_count=1000, target="alpha"):
    """
    Main data preparation for regression:
    - Augment Type 3 and Type 4 separately.
    - Mix them to create heterogeneous samples.
    - Add noise.
    - Return X and the selected target ("alpha" or "mu").
    """
    (names3, data3), (names4, data4) = load_all_types(file_type3,
        file_type4)
    X0, labels0 = augment_type_data(names3, data3, new_sample_count=
        desired_count)
    X1, labels1 = augment_type_data(names4, data4, new_sample_count=
        desired_count)
    X_heter, alphas, mu_heter = generate_heter_data(X0, labels0, X1,
         labels1, desired_count=heter_count)
    X_heter_noisy = add_noise_to_data(X_heter)
    if target == "alpha":
        labels = alphas
    elif target == "mu":
        labels = mu_heter
    else:
        raise ValueError("target must be 'alpha' or 'mu'")
    return X_heter_noisy, labels

def train_the_model(file_type3, file_type4, desired_count,
    heter_count):
    X, y = load_hdf5_data_for_regression(file_type3, file_type4,
        desired_count, heter_count, target="alpha")
    print("X shape:", X.shape)
    print("y shape:", y.shape)
    print("Target stats: min = {:.3f}, max = {:.3f}, mean = {:.3f}".
        format(y.min(), y.max(), y.mean()))

    from sklearn.preprocessing import StandardScaler
    X = StandardScaler().fit_transform(X)
```

```

from sklearn.model_selection import train_test_split
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y
    , test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val,
    y_train_val, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

train_loader = DataLoader(TensorDataset(X_train_tensor,
    y_train_tensor), batch_size=256, shuffle=True)
val_loader = DataLoader(TensorDataset(X_val_tensor, y_val_tensor
    ), batch_size=256, shuffle=False)
test_loader = DataLoader(TensorDataset(X_test_tensor,
    y_test_tensor), batch_size=256, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
model = CNN(input_length=X.shape[1]).to(device)
criterion = nn.SmoothL1Loss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 120
patience = 20
best_val_loss = float('inf')
best_model_weights = None
counter = 0
train_losses, val_losses = [], []

for epoch in range(epochs):
    model.train()
    total_train_loss = 0
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
```

```python
                optimizer.zero_grad()
                outputs = model(X_batch).squeeze()
                loss = criterion(outputs, y_batch)
                loss.backward()
                optimizer.step()
                total_train_loss += loss.item()
            train_losses.append(total_train_loss / len(train_loader))

            model.eval()
            total_val_loss = 0
            with torch.no_grad():
                for X_batch, y_batch in val_loader:
                    X_batch, y_batch = X_batch.to(device), y_batch.to(device)
                    outputs = model(X_batch).squeeze()
                    loss = criterion(outputs, y_batch)
                    total_val_loss += loss.item()
            val_loss = total_val_loss / len(val_loader)
            val_losses.append(val_loss)
            print(f"Epoch {epoch+1}/{epochs}: Train Loss = {train_losses[-1]:.6f}, Val Loss = {val_loss:.6f}")

            if val_loss < best_val_loss:
                best_val_loss = val_loss
                best_model_weights = model.state_dict()
                counter = 0
            else:
                counter += 1
                if counter >= patience:
                    print("Early stopping triggered.")
                    break

    if best_model_weights is not None:
        model.load_state_dict(best_model_weights)

    # Plot loss
    plt.figure()
    plt.plot(train_losses, label="Train")
    plt.plot(val_losses, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Training and Validation Loss")
    plt.show()

    # Evaluate on test set
    model.eval()
    y_preds, y_true = [], []
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            outputs = model(X_batch).squeeze()
            y_preds.extend(outputs.cpu().numpy())
            y_true.extend(y_batch.numpy())

    y_preds = np.array(y_preds)
    y_true = np.array(y_true)
    print(f"Test MSE: {mean_squared_error(y_true, y_preds):.6f}, R:
        {r2_score(y_true, y_preds):.6f}")

    plt.figure()
    plt.scatter(y_true, y_preds, alpha=0.6)
    plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max
        ()], 'r--')
    plt.xlabel("True Value")
    plt.ylabel("Predicted Value")
    plt.title("Regression: Prediction vs True")
    plt.grid(True)
    plt.show()

# Main entry point
if __name__ == "__main__":
    np.random.seed(42)
    torch.manual_seed(42)
    file_type3 = "./output/clean3.h5"
    file_type4 = "./output/clean4.h5"
    desired_count = 3000
    heter_count = 3000
    train_the_model(file_type3, file_type4, desired_count,
        heter_count)
```

Listing 6: Python main script for 2-types (Type 3 4) mixing factor prediction

```python
import re
import numpy as np
import h5py
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix


#######################################
# 1. Data Loading and Preprocessing
#######################################

def load_separately_with_names(file_type):
    """
    Load SAXS data from a single HDF5 file, excluding 'q_fixed'.
    Extract numeric identifiers from keys and return:
      - sorted_names: list of numeric identifiers (sorted),
      - sorted_data: corresponding stacked data in sorted order.
    """
    names = []
    data_list = []
    with h5py.File(file_type, 'r') as f:
        for key in f.keys():
            if key == 'q_fixed':
                continue
            m = re.search(r'd(\d+)_qIq', key)
            if m:
                num = int(m.group(1))
                names.append(num)
                data_list.append(f[key][:])

    sorted_indices = np.argsort(names)
    sorted_names = [names[i] for i in sorted_indices]
    sorted_data = np.vstack([data_list[i] for i in sorted_indices])
    return sorted_names, sorted_data


def load_all_4_types(file_type1, file_type2, file_type3, file_type4
    ):
    """
    Load SAXS data from four HDF5 files corresponding to clean1 to
        clean4.
    Returns a tuple of (names, data) for each type.
    """
    names1, data1 = load_separately_with_names(file_type1)
    names2, data2 = load_separately_with_names(file_type2)
    names3, data3 = load_separately_with_names(file_type3)
    names4, data4 = load_separately_with_names(file_type4)
    return (names1, data1), (names2, data2), (names3, data3), (
        names4, data4)


def augment_type_data(names, X, new_sample_count=2000):
    """
    Augment SAXS data for a single type by:
      - Randomly sampling log-normal weights based on diameters;
      - Using the weights to combine original curves into new
          synthetic samples.
    Returns (new_samples, None).
    """
    new_samples = []
    diameters = np.array(names, dtype=np.float32)

    for _ in range(new_sample_count):
        mean_real = np.random.uniform(25, 95)
        sigma_real = np.random.uniform(0.1, 0.4)
        mu = np.log(mean_real**2 / np.sqrt(mean_real**2 + sigma_real
            **2))
        sigma_log = np.sqrt(np.log(1 + (sigma_real**2 / mean_real
            **2)))
        weights = (1.0 / (diameters * sigma_log * np.sqrt(2 * np.pi)
            )) * \
                  np.exp(- (np.log(diameters) - mu)**2 / (2 *
                      sigma_log**2))
        weights /= np.sum(weights)
        new_curve = np.sum(weights[:, np.newaxis] * X, axis=0)
        new_samples.append(new_curve)

    return np.array(new_samples), None


def add_noise_to_curve(curve):
    """
    Add realistic multiplicative noise to a single SAXS curve using
        log-normal noise.
    Returns the noisy curve.
    """
    curve_safe = np.maximum(curve, 1e-12)
    log_alpha = np.random.uniform(np.log(1e4), np.log(10**7.5))
    alpha = np.exp(log_alpha)
    sigma2 = np.log(1 + alpha / curve_safe)
    epsilon = np.random.randn(curve.shape[0])
    noise_factor = np.exp(np.sqrt(sigma2)*epsilon - sigma2/2)
    curve_noisy = curve * noise_factor
    return curve_noisy


def add_noise_to_data(data):
    """
    Add noise to every SAXS curve in the dataset.
    """
    noisy_data = np.array([add_noise_to_curve(data[i]) for i in
        range(data.shape[0])])
    return noisy_data


def generate_heter_data_4(X1, X2, X3, X4, desired_count=1000):
    """
    Generate synthetic heterogeneous samples by mixing one curve
        from each of the four types.
    Weighted sums are created with random normalized coefficients.
    Returns (X_heter, None).
    """
    n1, n2, n3, n4 = X1.shape[0], X2.shape[0], X3.shape[0], X4.shape
        [0]
    X_heter_list = []

    for _ in range(desired_count):
        idx1 = np.random.randint(n1)
        idx2 = np.random.randint(n2)
        idx3 = np.random.randint(n3)
        idx4 = np.random.randint(n4)
        alphas = np.random.rand(4)
        alphas /= np.sum(alphas)
        mixed_curve = (alphas[0] * X1[idx1]
                      + alphas[1] * X2[idx2]
                      + alphas[2] * X3[idx3]
                      + alphas[3] * X4[idx4])
        X_heter_list.append(mixed_curve)

    return np.array(X_heter_list), None


def load_hdf5_data_for_classification_5types(file_type1, file_type2
    , file_type3, file_type4,
                                              desired_count=2000,
                                                  heter_count=1000):
    """
    Complete pipeline to prepare 5-class classification data:
      1) Load clean SAXS data from four HDF5 files;
      2) Augment each type to desired_count samples;
      3) Generate heterogeneous mixtures;
      4) Label the samples: Types 1-4 -> 0, Heter -> 1;
      5) Add noise, apply log10, and return all samples and labels.
    """
    (names1, data1), (names2, data2), (names3, data3), (names4,
        data4) = \
        load_all_4_types(file_type1, file_type2, file_type3,
            file_type4)

    X_type1, _ = augment_type_data(names1, data1, new_sample_count=
        desired_count)
    X_type2, _ = augment_type_data(names2, data2, new_sample_count=
        desired_count)
    X_type3, _ = augment_type_data(names3, data3, new_sample_count=
        desired_count)
    X_type4, _ = augment_type_data(names4, data4, new_sample_count=
        desired_count)

    X_heter, _ = generate_heter_data_4(X_type1, X_type2, X_type3,
        X_type4, desired_count=heter_count)

    # Adjust labels for binary classification (clean -> 0, heter ->
        1)
    y_type1 = np.full(X_type1.shape[0], 0, dtype=np.int64)
    y_type2 = np.full(X_type2.shape[0], 0, dtype=np.int64)
    y_type3 = np.full(X_type3.shape[0], 0, dtype=np.int64)
    y_type4 = np.full(X_type4.shape[0], 0, dtype=np.int64)
    y_heter = np.full(X_heter.shape[0], 1, dtype=np.int64)

    # Add noise and apply log10 transform
    X_type1_noisy = np.log10(np.maximum(add_noise_to_data(X_type1),
        1e-12))
    X_type2_noisy = np.log10(np.maximum(add_noise_to_data(X_type2),
        1e-12))
    X_type3_noisy = np.log10(np.maximum(add_noise_to_data(X_type3),
        1e-12))
    X_type4_noisy = np.log10(np.maximum(add_noise_to_data(X_type4),
        1e-12))
    X_heter_noisy = np.log10(np.maximum(add_noise_to_data(X_heter),
        1e-12))

    X_all = np.concatenate([X_type1_noisy, X_type2_noisy,
                            X_type3_noisy, X_type4_noisy,
                            X_heter_noisy], axis=0)
    y_all = np.concatenate([y_type1, y_type2, y_type3, y_type4,
        y_heter], axis=0)

    return X_all, y_all


########################################
# 2. Define Classification CNN
```

```python
########################################

class ClassificationCNN(nn.Module):
    """
    A simple 1D CNN for SAXS-based classification with two
        convolutional layers
    followed by a fully connected classifier.
    """
    def __init__(self, input_length, num_classes=5):
        super(ClassificationCNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5,
                padding=2),
            nn.BatchNorm1d(32),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3,
                 padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU()
        )

        self.fc = nn.Sequential(
            nn.Linear(64 * input_length, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, num_classes),
        )

    def forward(self, x):
        # Input x shape: (batch_size, feature_dim)
        x = x.unsqueeze(1) # -> (batch_size, 1, feature_dim)
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1) # flatten
        x = self.fc(x)
        return x


```

```python
########################################
# 3. Training and Evaluation
########################################

def train_the_model_classification_5(file_type1, file_type2,
    file_type3, file_type4, desired_count):
    """
    Train and evaluate the classification model on 5-class SAXS data
        :
    clean types (1-4) vs. heterogeneously mixed (5th class).
    Includes training loop, early stopping, and evaluation metrics.
    """
    X, y = load_hdf5_data_for_classification_5types(
        file_type1, file_type2, file_type3, file_type4,
        desired_count=desired_count, heter_count=desired_count
    )
    print("X shape:", X.shape)
    print("y shape:", y.shape)
    print("Class distribution:", {cls: int(np.sum(y == cls)) for cls
         in np.unique(y)})

    # Standardize features
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Split dataset
    from sklearn.model_selection import train_test_split
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y
        , test_size=0.2, random_state=42)
    X_train, X_val, y_train, y_val = train_test_split(X_train_val,
        y_train_val, test_size=0.2, random_state=42)

    # Create DataLoaders
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
    X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
    X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
    y_train_tensor = torch.tensor(y_train, dtype=torch.long)
    y_val_tensor = torch.tensor(y_val, dtype=torch.long)
    y_test_tensor = torch.tensor(y_test, dtype=torch.long)

    train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
    val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
    test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

    train_loader = DataLoader(train_dataset, batch_size=256, shuffle
        =True)
    val_loader = DataLoader(val_dataset, batch_size=256, shuffle=
        False)
    test_loader = DataLoader(test_dataset, batch_size=256, shuffle=
        False)

    # Model, optimizer, loss
    device = torch.device("cuda" if torch.cuda.is_available() else "
        cpu")
    model = ClassificationCNN(input_length=X.shape[1], num_classes
        =5).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)

    # Training loop with early stopping
    epochs = 200
    patience = 40
    best_val_loss = float('inf')
    best_model_weights = None
    counter = 0
    train_losses, val_losses = [], []

    for epoch in range(epochs):
        model.train()
        total_train_loss = 0.0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
        train_loss = total_train_loss / len(train_loader)
        train_losses.append(train_loss)

        model.eval()
        total_val_loss = 0.0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                X_batch, y_batch = X_batch.to(device), y_batch.to(device)
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                total_val_loss += loss.item()
        val_loss = total_val_loss / len(val_loader)
        val_losses.append(val_loss)

        print(f"Epoch {epoch+1}/{epochs} - Train Loss: {train_loss:.6f}, Val Loss: {val_loss:.6f}")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_weights = model.state_dict()
            counter = 0
        else:
            counter += 1
            if counter >= patience:
                print(f"Early stopping triggered at epoch {epoch+1}. Best Val Loss: {best_val_loss:.6f}")
                break

    if best_model_weights is not None:
        model.load_state_dict(best_model_weights)

    # Plot loss curve
    plt.figure(figsize=(8, 5))
    plt.plot(range(1, len(train_losses) + 1), train_losses, label="Train Loss")
    plt.plot(range(1, len(val_losses) + 1), val_losses, label="Validation Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    # Evaluate on test set
    model.eval()
    y_preds, y_true = [], []
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            outputs = model(X_batch)
            predicted = torch.argmax(outputs, dim=1)
            y_preds.extend(predicted.cpu().numpy())
            y_true.extend(y_batch.numpy())

    accuracy = accuracy_score(y_true, y_preds)
    print("Test Accuracy:", accuracy)
    print("Classification Report:")
    print(classification_report(y_true, y_preds))

    # Confusion matrix
    cm = confusion_matrix(y_true, y_preds)
    print("Confusion Matrix:")
    print(cm)

    plt.figure(figsize=(6, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    classes = ["Double-ellipsoid", "Spherical", "x", "x", "x"] #
        Replace 'x' with actual labels if needed
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.0
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, format(cm[i, j], 'd'),
                    horizontalalignment="center",
                    color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.tight_layout()
    plt.show()


########################################
# 4. Main Entry Point
########################################
if __name__ == "__main__":
    np.random.seed(31)
    torch.manual_seed(31)

    # Modify paths as needed
    file_type1 = "./output/clean1.h5"
    file_type2 = "./output/clean2.h5"
    file_type3 = "./output/clean3.h5"
    file_type4 = "./output/clean4.h5"

    desired_count = 4000 # Augmented sample count per clean type

    train_the_model_classification_5(file_type1, file_type2,
        file_type3, file_type4, desired_count)
```

Listing 7: Python main script for 4-types classification

```
import re
import numpy as np
import h5py
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score


# -----------------------------
# 1. Data Loading & Augmentation
# -----------------------------

```

```
def load_separately_with_names(file_type):
    """
    Load SAXS data from a single HDF5 file, excluding 'q_fixed'.
    Extract numeric identifiers from keys (e.g., 'd20_qIq') and
        return:
      - sorted_names: sorted list of diameters
      - sorted_data: corresponding data (stacked vertically)
    """
    names = []
    data_list = []
    with h5py.File(file_type, 'r') as f:
        for key in f.keys():
            if key == 'q_fixed':
                continue
            m = re.search(r'd(\d+)_qIq', key)
            if m:
                num = int(m.group(1))
                names.append(num)
                data_list.append(f[key][:])
    sorted_indices = np.argsort(names)
    sorted_names = [names[i] for i in sorted_indices]
    sorted_data = np.vstack([data_list[i] for i in sorted_indices])
    return sorted_names, sorted_data


def load_all_3_types(file_type1, file_type2, file_type3):
    """
    Load SAXS datasets from three HDF5 files.
    Returns (names, data) tuples for each file.
    """
    names1, data1 = load_separately_with_names(file_type1)
    names2, data2 = load_separately_with_names(file_type2)
    names3, data3 = load_separately_with_names(file_type3)
    return (names1, data1), (names2, data2), (names3, data3)


def augment_type_data(names, X, new_sample_count=2000):
    """
    Augment data by generating synthetic SAXS curves:
    - Sample log-normal weight distributions using random (mean,
        sigma),
    - Weight and combine real curves to generate synthetic ones.
    """
    new_samples = []
    diameters = np.array(names, dtype=np.float32)
    for _ in range(new_sample_count):
        mean_real = np.random.uniform(25, 95)
        sigma_real = np.random.uniform(1, 4)
        mu = np.log(mean_real**2 / np.sqrt(mean_real**2 + sigma_real
            **2))
        sigma_log = np.sqrt(np.log(1 + (sigma_real**2 / mean_real
            **2)))
        weights = (1.0 / (diameters * sigma_log * np.sqrt(2 * np.pi)
            )) * \
                  np.exp(- (np.log(diameters) - mu)**2 / (2 *
                      sigma_log**2))
        weights /= np.sum(weights)
        new_curve = np.sum(weights[:, np.newaxis] * X, axis=0)
        new_samples.append(new_curve)
    return np.array(new_samples), None


def add_noise_to_curve(curve):
    """
    Add multiplicative log-normal noise to a single SAXS curve.
    """
    curve_safe = np.maximum(curve, 1e-12)
    log_alpha = np.random.uniform(np.log(1e4), np.log(10**7.5))
    alpha = np.exp(log_alpha)
    sigma2 = np.log(1 + alpha / curve_safe)
    epsilon = np.random.randn(len(curve))
    noise_factor = np.exp(np.sqrt(sigma2) * epsilon - sigma2 / 2)
    return curve * noise_factor


def add_noise_to_data(data):
    """
    Apply noise to each curve in the dataset.
    """
    return np.array([add_noise_to_curve(x) for x in data])
```

```python


def generate_weighted_mixture_data_3(X1, X2, X3, sample_count=1000,
     add_noise=False, do_log10=False):
    """
    Generate mixture data from three types:
    - Randomly sample one curve from each,
    - Generate Dirichlet weights,
    - Mix the curves using the weights.
    Optionally adds noise and log10 transform.
    Returns (X_mix, alphas).
    """
    n1, n2, n3 = X1.shape[0], X2.shape[0], X3.shape[0]
    feature_dim = X1.shape[1]
    X_mix = np.zeros((sample_count, feature_dim), dtype=np.float32)
    alphas = np.zeros((sample_count, 3), dtype=np.float32)

    for i in range(sample_count):
        idx1, idx2, idx3 = np.random.randint(0, n1), np.random.
            randint(0, n2), np.random.randint(0, n3)
        alpha = np.random.dirichlet([1, 1, 1])
        alphas[i] = alpha
        curve = alpha[0] * X1[idx1] + alpha[1] * X2[idx2] + alpha[2]
             * X3[idx3]
        if add_noise:
            curve = add_noise_to_curve(curve)
        if do_log10:
            curve = np.log10(np.maximum(curve, 1e-12))
        X_mix[i] = curve
    return X_mix, alphas


# ----------------------------
# 2. CNN Regression Model
# ----------------------------

class WeightsRegressionCNN(nn.Module):
    """
    CNN model to regress 3 mixing weights (alphas) from SAXS curves.
    Final output is soft-normalized to sum to 1.
    """
    def __init__(self, input_length):
        super(WeightsRegressionCNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5,
                padding=2),
            nn.BatchNorm1d(32),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3,
                 padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(64 * input_length, 64),
            nn.ReLU(),
            nn.Linear(64, 3),
        )

    def forward(self, x):
        x = x.unsqueeze(1) # (batch_size, 1, feature_dim)
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        logits = self.fc(x)
        sig_out = torch.sigmoid(logits)
        norm_out = sig_out / (sig_out.sum(dim=1, keepdim=True) + 1e
            -8)
        return norm_out


# ----------------------------
# 3. Training Pipeline
# ----------------------------

def train_weight_regression(file_type1, file_type2, file_type3,
                            each_type_count=500,
                            mix_count=4000,
                        add_noise=True,
                        do_log10=True):
    """
    Train CNN to predict mixing weights of 3 SAXS types.
    """
    # Load & augment data
    (names1, data1), (names2, data2), (names3, data3) =
        load_all_3_types(file_type1, file_type2, file_type3)
    X1, _ = augment_type_data(names1, data1, each_type_count)
    X2, _ = augment_type_data(names2, data2, each_type_count)
    X3, _ = augment_type_data(names3, data3, each_type_count)

    # Generate synthetic mixtures
    X_mix, alphas = generate_weighted_mixture_data_3(X1, X2, X3,
        mix_count, add_noise, do_log10)

    # Split data
    X_train_val, X_test, y_train_val, y_test = train_test_split(
        X_mix, alphas, test_size=0.2, random_state=42)
    X_train, X_val, y_train, y_val = train_test_split(X_train_val,
        y_train_val, test_size=0.2, random_state=42)

    # Create DataLoaders
    def to_loader(X, y):
        X_tensor = torch.tensor(X, dtype=torch.float32)
        y_tensor = torch.tensor(y, dtype=torch.float32)
        return DataLoader(TensorDataset(X_tensor, y_tensor),
            batch_size=512, shuffle=True)

    train_loader = to_loader(X_train, y_train)
    val_loader = to_loader(X_val, y_val)
    test_loader = to_loader(X_test, y_test)

    # Initialize model
    device = torch.device("cuda" if torch.cuda.is_available() else "
        cpu")
    model = WeightsRegressionCNN(input_length=X_train.shape[1]).to(
        device)
    optimizer = optim.Adam(model.parameters(), lr=0.0005)
    criterion = nn.MSELoss()

    # Training loop with early stopping
    epochs = 800
    patience = 80
    best_val_loss = float('inf')
    best_state_dict = None
    no_improve_count = 0
    train_losses, val_losses = [], []

    for epoch in range(epochs):
        model.train()
        total_train_loss = sum(
            criterion(model(X.to(device)), y.to(device)).item()
            for X, y in train_loader
        ) / len(train_loader)
        train_losses.append(total_train_loss)

        model.eval()
        with torch.no_grad():
            total_val_loss = sum(
                criterion(model(X.to(device)), y.to(device)).item()
                for X, y in val_loader
            ) / len(val_loader)
        val_losses.append(total_val_loss)

        print(f"Epoch {epoch+1}/{epochs}: Train Loss = {
            total_train_loss:.6f}, Val Loss = {total_val_loss:.6f}")

        if total_val_loss < best_val_loss:
            best_val_loss = total_val_loss
            best_state_dict = model.state_dict()
            no_improve_count = 0
        else:
            no_improve_count += 1
            if no_improve_count >= patience:
                print("Early stopping triggered.")
                break

    if best_state_dict:
        model.load_state_dict(best_state_dict)

    # Plot loss curve
    plt.figure()
    plt.plot(range(1, len(train_losses)+1), train_losses, label="Train Loss")
    plt.plot(range(1, len(val_losses)+1), val_losses, label="Val Loss")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.legend()
    plt.show()

    # Final evaluation
    model.eval()
    y_pred, y_true = [], []
    with torch.no_grad():
        for X, y in test_loader:
            y_pred.append(model(X.to(device)).cpu().numpy())
            y_true.append(y.cpu().numpy())

    y_pred = np.concatenate(y_pred, axis=0)
    y_true = np.concatenate(y_true, axis=0)

    mse = np.mean((y_pred - y_true) ** 2)
    mae = np.mean(np.abs(y_pred - y_true))
    print("Test MSE:", mse)
    print("Test MAE:", mae)

    print("MSE per alpha:", np.mean((y_pred - y_true) ** 2, axis=0))
    print("MAE per alpha:", np.mean(np.abs(y_pred - y_true), axis=0))

    r2 = [r2_score(y_true[:, i], y_pred[:, i]) for i in range(3)]
    print("R per alpha:", r2)
    print("Overall R:", r2_score(y_true.flatten(), y_pred.flatten()))

    # Show some examples
    n_show = 8
    idx = np.random.choice(len(y_true), n_show, replace=False)
    print(f"\nShowing {n_show} random test samples:")
    for i in idx:
        print(f"True: {y_true[i]}, Pred: {y_pred[i]}")


if __name__ == "__main__":
    np.random.seed(42)
    torch.manual_seed(42)

    file_type1 = "./output/clean1.h5"
    file_type2 = "./output/clean2.h5"
    file_type3 = "./output/clean3.h5"

    train_weight_regression(
        file_type1, file_type2, file_type3,
        each_type_count=100000,
        mix_count=100000,
        add_noise=True,
        do_log10=False
    )
```

Listing 8: Python main script for 4-types mixing factor prediction

```
import torch
import torch.nn as nn
import torch.nn.functional as F


# -----------------------------
# 1. Residual Block & ResNet
# -----------------------------

class ResidualBlock1D(nn.Module):
    """
    A 1D residual block with two convolutional layers and skip
        connection.
    """
    def __init__(self, channels):
        super().__init__()
        self.conv1 = nn.Conv1d(channels, channels, kernel_size=3,
            padding=1)
```

```python
        self.bn1 = nn.BatchNorm1d(channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv1d(channels, channels, kernel_size=3,
           padding=1)
        self.bn2 = nn.BatchNorm1d(channels)

    def forward(self, x):
        residual = x
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        return self.relu(out + residual)


class Resnet(nn.Module):
    """
    1D ResNet for multi-class classification (default 4 outputs with
        softmax).
    """
    def __init__(self, input_length):
        super().__init__()
        self.entry = nn.Sequential(
            nn.Conv1d(1, 64, kernel_size=7, padding=3),
            nn.BatchNorm1d(64),
            nn.ReLU()
        )
        self.block1 = ResidualBlock1D(64)
        self.block2 = ResidualBlock1D(64)
        self.block3 = ResidualBlock1D(64)
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(64, 4) # Default: 4-class output

    def forward(self, x):
        x = x.unsqueeze(1) # (batch_size, 1, input_length)
        x = self.entry(x)
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.pool(x).squeeze(-1) # (batch_size, 64)
        return F.softmax(self.fc(x), dim=1)

```

```

# ----------------------------
# 2. CNN (Deep) for Binary Classification
# ----------------------------

class CNN(nn.Module):
    """
    Deep 1D CNN for binary classification. Final output: sigmoid
        scalar in [0,1].
    """
    def __init__(self, input_length):
        super(CNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5,
                padding=2),
            nn.BatchNorm1d(32),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=5,
                 padding=2),
            nn.BatchNorm1d(64),
            nn.ReLU()
        )
        self.layer3 = nn.Sequential(
            nn.Conv1d(in_channels=64, out_channels=128, kernel_size
                =3, padding=1),
            nn.BatchNorm1d(128),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(128 * input_length, 256),
            nn.ReLU(),
            nn.Dropout(0.15),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Dropout(0.15),
            nn.Linear(64, 1),
            nn.Sigmoid() # Output in [0,1]
        )

    def forward(self, x):
        x = x.unsqueeze(1) # (batch_size, 1, input_length)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x.squeeze(1)


# ----------------------------
# 3. Simple CNN for Binary Classification
# ----------------------------

class simpleCNN(nn.Module):
    """
    Simpler version of CNN with fewer layers for binary
        classification.
    """
    def __init__(self, input_length):
        super(simpleCNN, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=5,
                padding=2),
            nn.ReLU()
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3,
                padding=1),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(64 * input_length, 64),
            nn.ReLU(),
            nn.Dropout(0.15),
            nn.Linear(64, 1),
            nn.Sigmoid() # Output in [0,1]
        )

    def forward(self, x):
        x = x.unsqueeze(1) # (batch_size, 1, input_length)
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x.squeeze(1)


# -----------------------------
# 4. Example Usage
# -----------------------------

if __name__ == "__main__":
    model = Resnet(input_length=500)
    sample_input = torch.randn(8, 500) # batch size = 8, input
        length = 500
    output = model(sample_input)
    print("Output shape:", output.shape) # Expected: (8, 4)
```

Listing 9: Python main script ML models