# Take-Home Assignment Report

## Synthetic Dataset Generation and Error Bar Detection

Date: January 2026

### Submitted By

Nahid Montasir Rifat
Dept. of CSE, RUET
nahidmuntasir2@gmail.com

# Contents

# 1. Introduction

Error bars are commonly used in scientific visualizations to represent uncertainty, variability, or confidence intervals. Automatically extracting error bar information from plot images is a challenging problem due to diverse plotting styles, overlapping elements, and noise.

This project addresses two tasks: (1) synthesizing a large-scale labeled dataset of plot images and (2) detecting upper and lower error bar endpoints given data point locations. A synthetic data pipeline and a deep learning-based regression model were developed and evaluated.

# 2. Synthesized Dataset Generation (Task 1)

## 2.1. Task Description

The original dataset contains only approximately 150 manually annotated plot images, far too few to train robust, generalizable models for error bar detection and related tasks. To overcome this severe data limitation, a scalable synthetic data generation pipeline was designed and implemented to produce 3000 high-fidelity, realistically annotated scientific plots, increasing the training corpus by a factor of 20 while preserving annotation quality.

## 2.2. Synthesis Strategy

A domain-aware pipeline was used to generate the synthetic dataset. First a scientific domain was sampled from a predefined set (Biology/Pharmacology, Physics, Chemistry/Kinetics, Environmental Science, Finance, General Experimental Trends). Only patterns realistic for the chosen domain were then selected from the 37 available functions through weighted sampling. The chart type was decided next (75% probability for line plots, 25% for bar plots). Structural parameters were randomized (1 to 5 series, 4 to 12 points per series, figure size, DPI). Data values were computed in normalized space, scaled to realistic axis ranges, and perturbed with Gaussian noise. Error bars were added with magnitudes scaled to the local data range, 30% probability of asymmetry, and 15% probability of omission. Visual diversity was introduced through randomized colormaps, line and marker styles, Matplotlib themes, fonts, grids, legends, and domain-specific axis labels and titles. Finally the plot was rendered tightly and all points plus error-bar endpoints were transformed from data-space to pixel-space using Matplotlib's `ax.transData.transform()` function so that precise pixel coordinates and distances could be calculated and stored in JSON files matching the original schema. This structured and conditioned approach produced realistic plots with perfect annotation fidelity in an efficient manner.

### 2.2.1 Rationale for the Current Configuration

The presented pipeline setup was deliberately chosen to balance realism, scalability, and annotation precision. Domain-conditioned pattern selection was preferred over fully random sam-

pling because it produces plots that closely resemble real scientific literature, thereby improving model generalization to actual use cases. Probabilistic variations in visual elements (colors, themes, error bar styles) were implemented parametrically rather than through post-rendering image augmentation so that pixel coordinates remain perfectly accurate without requiring complex coordinate recalculations. The use of Matplotlib's native transformation functions for pixel mapping was selected because it guarantees exact correspondence between rendered images and JSON labels even under changing figure sizes and DPI values.

## 2.3. Implementation and Parameters

### 2.3.1 Libraries and Configuration

The pipeline is implemented in a a standard Kaggle P100 GPU using the following core libraries:

Table 1: Core libraries and their primary roles

| Library | Primary Role |
|---|---|
| `matplotlib` | Plot creation, styling, rendering, coordinate transformations |
| `numpy` | Numerical computations, pattern generation, noise addition |
| `json` | Serializing annotations to JSON format |
| `uuid` | Generating unique file identifiers |
| `os, random` | File system operations and controlled randomization |
| `zipfile` | Creating the final dataset archive |
| `PIL (Pillow)` | Image loading and overlay-based verification |
| `tqdm` | Progress bars during generation and zipping |

A fixed seed (42) is used for reproducibility.

### 2.3.2 Domain-Conditioned Pattern Selection

Instead of sampling patterns completely at random, the pipeline first selects a scientific domain and then only considers patterns that are typical and plausible for that field. This targeted approach makes the resulting plots much more believable.

The following table summarizes the scientific domains used and their preferentially sampled pattern families:

Table 2: Domain-guided pattern selection (only realistic patterns for each scientific topic are sampled)

| **Domain** | **Primary Pattern Families** | **Rationale / Typical Use Case** |
|---|---|---|
| Biology / Pharmacology | sigmoid family, michaelis_menten, hill_equation, substrate_inhibition, biphasic_increase/decrease, plateau family | Enzyme kinetics, dose-response curves, cell growth, receptor binding |
| Physics | damped_oscillation, growing_oscillation, oscillating, multi_frequency, exponential_decay, double_exponential | Harmonic motion, wave damping, radioactive decay, RLC circuits |
| Chemistry / Reaction Kinetics | exponential_growth/decay, plateau_high/low, delayed_plateau, step_up/down, multi_step | Reaction progress, concentration profiles, phase transitions |
| Environmental Science | increasing/decreasing (noisy), sigmoid_gradual, u_shape/inverse_u, random_walk | Pollutant accumulation, climate trends, population dynamics |
| Finance / Economics | noisy_increasing/decreasing, step functions, plateau family | Stock price trends, market saturation, economic cycles |
| General Experimental Trends | monotonic (steep/gradual), single_peak/double_peak/asymmetric_peak, random | Generic lab results, sensor data, exploratory plots |

This targeted selection dramatically improves the scientific plausibility of generated plots compared to fully uniform random pattern assignment.

### 2.3.3 Plot Generation and Variations

To make the synthetic plots look realistic and varied (like real scientific figures from different papers), many parts of each plot were deliberately randomized in a controlled way. First the basic structure (chart type, number of lines/bars, points, size) was decided. Then visual elements (grid, legend, title, error bar style, etc.) were turned on or off with specific probabilities. Finally extra styling choices (colors, line types, themes, fonts, labels) were randomly picked from large sets of options.

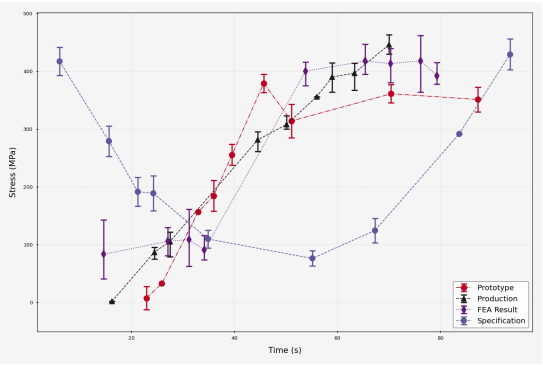
The main structural and probabilistic decisions are listed in the table below:

Table 3: Probabilistic and structural variations in plot generation

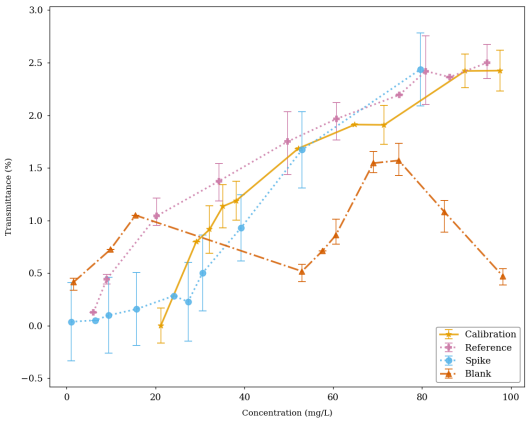
| Aspect | Range / Options | Probability / Frequency |
|---|---|---|
| Chart type | Line plot / Bar plot | 75% line, 25% bar |
| Number of series | 1–5 lines or 3–8 bars | Uniform random |
| Points per series | 4–12 | Uniform random |
| Figure sizes | (9,6), (10,7), (11,7), (12,8), (10,8) | Uniform random selection |
| DPI | 90–120 | Uniform random |
| Grid display | On / Off | 40% on |
| Legend | Present / Absent | 95% present |
| Title | Present / Absent | 60% present |
| Asymmetric error bars | Symmetric / Asymmetric | 30% asymmetric |
| Spine removal (clean style) | Keep all / Remove some | 20% removal |
| Data annotations | Present / Absent | 15% present |
| Sample size labels (n=...) | Present / Absent | 10% present |
| No error bars on some points | All have bars / Some missing | 15% some missing |
| Boundary points inclusion | Always include min/max x points | Always enabled |

#### 2.3.4 Coordinate Transformation and Manual Validation

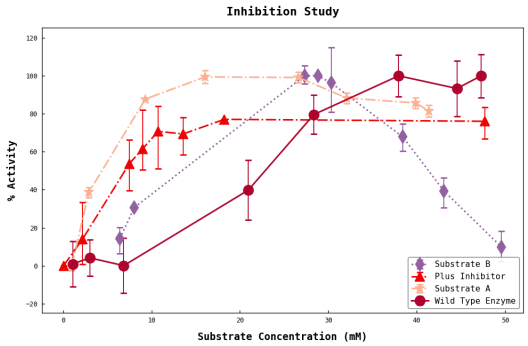
After rendering and saving each plot as a tight PNG, data-space coordinates of points and error bar endpoints were transformed to pixel coordinates using Matplotlib's `ax.transData.transform()` function. Y-values were then flipped to match the top-left origin required by the JSON format. Pixel distances (`topBarPixelDistance`, `bottomBarPixelDistance`, `deviationPixelDistance`) were calculated directly from these transformed positions and written into the corresponding JSON files along with point locations and series names. For quality assurance, 100 randomly selected images were manually reviewed. Overlay visualizations were created in which red lines marked the error bars, all drawn from the JSON values onto the original PNG. Alignment was confirmed within ±1 pixel, and overall plot appearance (clarity, labels, styling) was checked for publication quality.



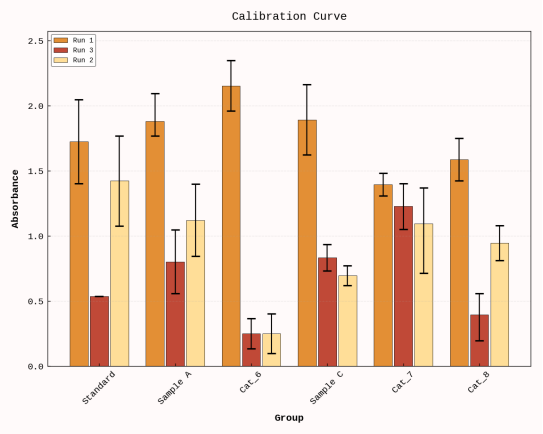

(a) Plot 1




(b) Plot 2




(c) Plot 3




(d) Plot 4

Figure 1: Examples of generated synthetic plots showing diverse scientific styles and error bars.

### 2.4. Scalability and Output

The pipeline successfully generated 3000 synthetic plot images together with their corresponding JSON label files in approximately 15–20 minutes on a standard Kaggle P100 GPU environment. To ensure balanced representation across scientific domains, the theme sampling was configured to produce roughly even distribution of images per topic. The final dataset was archived into a single ZIP file ( 160 MB) containing structured `images/` and `labels/` folders, exactly matching the format of the original dataset for seamless integration into training workflows. Overall, the pipeline is efficient, reproducible, and readily extensible for future dataset expansions.

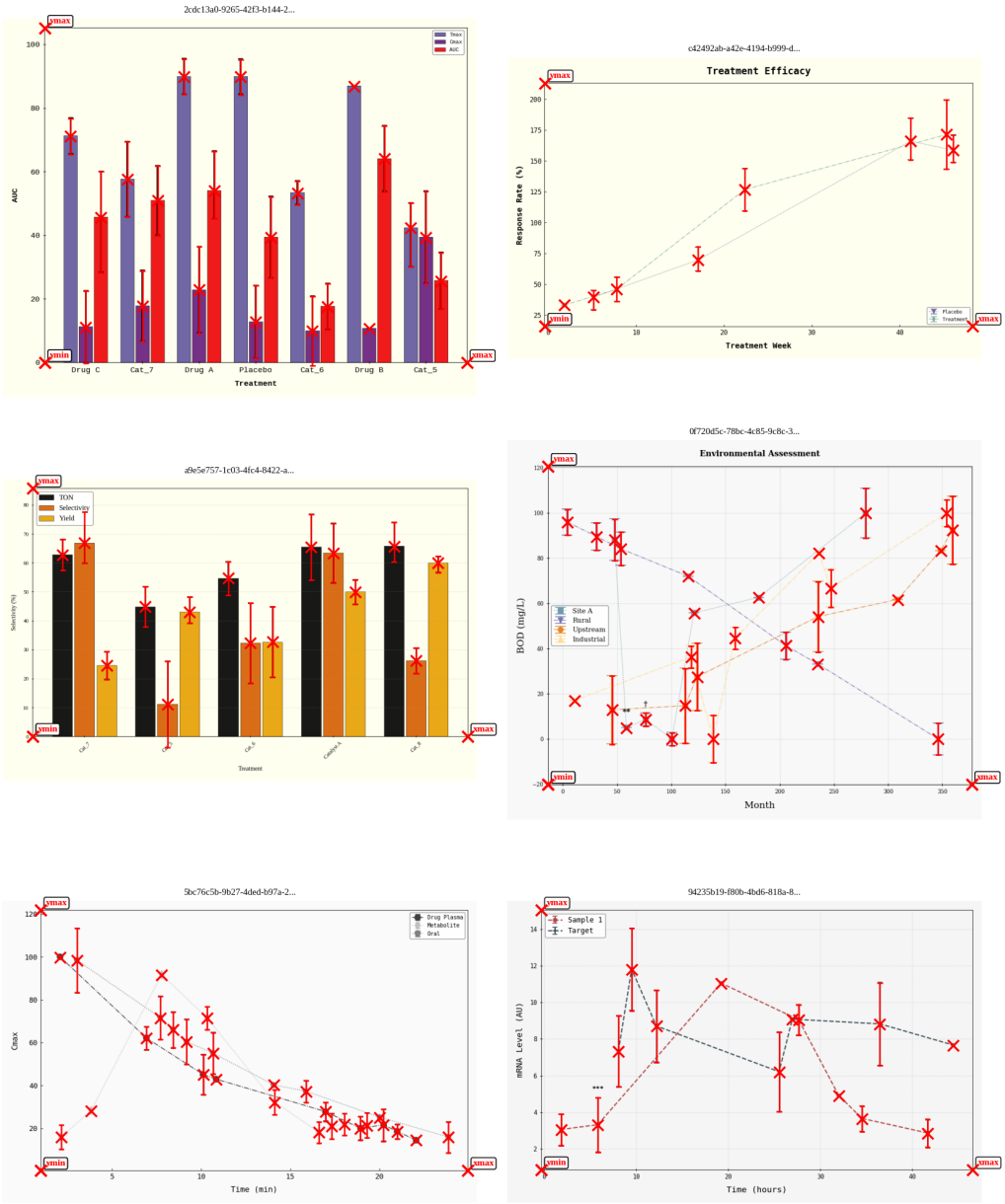



Figure 2: Example overlay visualization confirming pixel-accurate correspondence between JSON annotations and rendered plot.

# 3. Error Bar Detection Pipeline (Task 2)

## 3.1. Task Description

The error bar detection task requires predicting the precise pixel locations of upper and lower error bar endpoints for given data points in scientific plots. The input consists of a plot image and data point coordinates grouped by line name. The expected output specifies the coordinates of error bar endpoints in the format $(x_{\text{data}}, y_{\text{data}})$, $(x_{\text{upper}}, y_{\text{upper}})$, $(x_{\text{lower}}, y_{\text{lower}})$ for each data point. The challenge lies in achieving sub-pixel accuracy across diverse plot styles, varying error bar magnitudes, and visually complex scenes with overlapping elements.

## 3.2. Problem Formulation and Approach

Error bar detection in scientific plots presents a challenging computer vision task due to varying plot styles, overlapping visual elements, and diverse error bar magnitudes. We formulate this as a regression problem rather than object detection by exploiting the local spatial relationship between data points and their associated error bars. Given a plot image $\mathscr{I} \in R^{H\times W\times 3}$ and data point coordinates $\{(x_i, y_i)\}_{i=1}^{N}$, our goal is to predict the vertical pixel distances to the upper and lower error bar endpoints for each point.

The core methodology transforms this global detection problem into local regression by extracting fixed-size image patches centered on each data point. For a point at location $(x, y)$, we extract a $225 \times 225$ pixel patch $\mathscr{I}_p$ and learn a mapping $f_\theta : R^{225\times 225\times 3} \to R^2$ that predicts $(d_{\text{upper}}, d_{\text{lower}})$, where $d_{\text{upper}}$ is the pixel distance to the upper error bar and $d_{\text{lower}}$ is the distance to the lower error bar. The error bar endpoints are then computed as $(x, y - d_{\text{upper}})$ and $(x, y + d_{\text{lower}})$. This patch-based formulation provides several advantages: it naturally handles varying plot sizes, isolates local context to reduce interference from overlapping elements, and directly optimizes the quantities of interest without intermediate detection stages.

## 3.3. Model Architecture

### 3.3.1 Architecture Selection and Comparison

Three state-of-the-art convolutional neural network architectures pretrained on ImageNet-1K were evaluated to leverage learned visual representations. The selected candidates represent different architectural paradigms: EfficientNet employs compound scaling to balance depth, width, and resolution, whereas ConvNeXt represents a modernized residual architecture incorporating transformer-inspired design elements.

Table 4: Architecture comparison on validation set (8,135 samples) over 30 training epochs

| Architecture | Parameters | Best Val. Loss (Smooth L1) | Training Time |
|---|---|---|---|
| EfficientNet-B0 | 5.3M | 10.0926 | ∼1.8 hrs |
| EfficientNet-B2 | 9.1M | 2.2158 | ∼2 hrs |
| **ConvNeXt-Tiny** | **28.0M** | **1.3405** | **∼8 hrs** |

ConvNeXt-Tiny was selected based on superior validation performance and an appropriate capacity-to-data ratio. With 37,258 training samples, smaller architectures such as EfficientNet-B0 exhibited limited representational capacity, resulting in high prediction error, while larger models risk overfitting given the available dataset size. ConvNeXt-Tiny, with 28M parameters, yields approximately 1,300 samples per million parameters, which provides a balanced trade-off between underfitting and overfitting. Training from scratch was found to be less stable due to the limited size and domain-specific nature of the dataset. Vision Transformer architectures such as ViT were not selected despite their recent successes in computer vision because transformers typically require substantially larger training datasets to learn effective self-attention patterns and positional encodings. With our 37,258 samples, convolutional architectures provide better inductive biases for local spatial patterns inherent in error bar detection, including translation equivariance and hierarchical feature learning. Additionally, transformers demand significantly higher computational resources and longer training times with marginal performance gains on datasets of this scale.

ConvNeXt-Tiny achieved a total mean absolute error of 3.52 pixels, compared to 5.32 pixels for EfficientNet-B2, corresponding to a 34% improvement. The median error of 1.62 pixels indicates near-pixel-level precision, while 93.2% of predictions fall within a five-pixel tolerance. Although ConvNeXt-Tiny has higher computational cost, it demonstrated stable training behavior and no evidence of severe overfitting when trained on Kaggle's NVIDIA P100 GPU. The architectural design of ConvNeXt, including large receptive fields, layer normalization, and inverted bottleneck blocks, provides strong inductive bias for spatial regression tasks and contributes to improved performance on error bar localization.

### 3.3.2 Network Design for ConvNeXt-Tiny

The complete architecture consists of the ConvNeXt-Tiny feature extraction backbone followed by a custom regression head. The backbone processes $225 \times 225$ RGB input patches through four hierarchical stages with channel dimensions of 96, 192, 384, and 768 respectively. Each stage contains multiple ConvNeXt blocks employing depthwise separable convolutions, layer normalization, and GELU activations. Global average pooling following the final stage produces a 768-dimensional feature vector encoding the semantic content of each patch.

The regression head transforms these features into distance predictions through four fully connected layers: $768 \to 512 \to 256 \to 128 \to 2$. Each intermediate layer incorporates batch normalization for training stability, ReLU activation for nonlinearity, and dropout regularization with progressively decreasing rates of 0.3, 0.21, and 0.15. This progressive dropout strategy applies stronger regularization to early layers preventing feature co-adaptation while allowing later layers more flexibility to learn precise distance mappings. Proper weight initialization is critical for stable training of the randomly initialized regression head. Linear layer weights are initialized using Kaiming normal initialization with fan-out mode and ReLU nonlinearity, which sets the standard deviation to $\sqrt{2/n_{\text{out}}}$ where $n_{\text{out}}$ is the number of output units. This initialization scheme maintains variance of activations and gradients across layers,

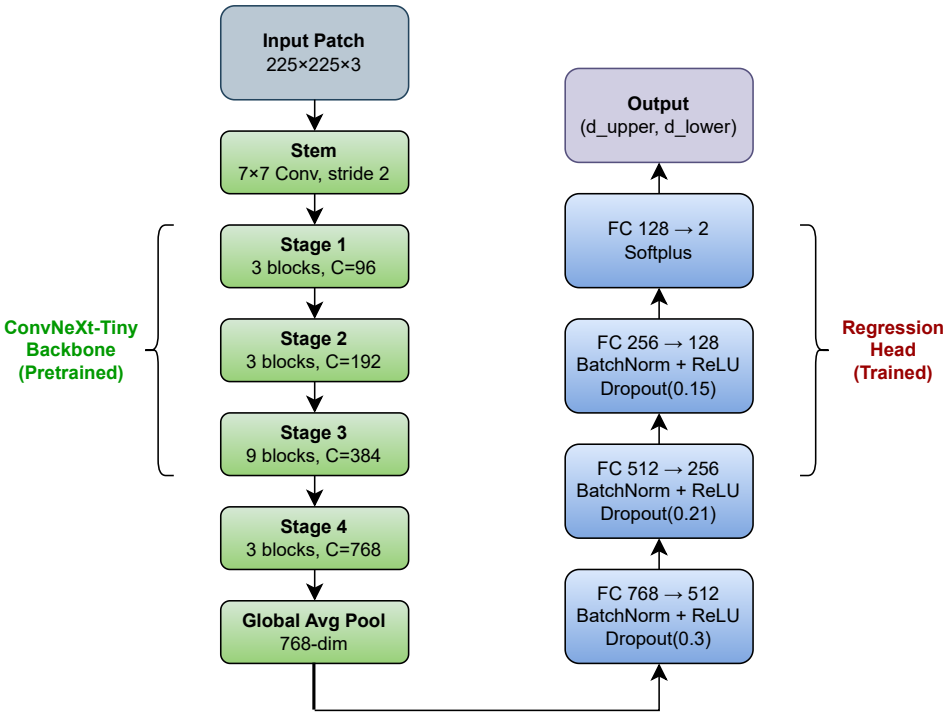



Figure 3: Overall network architecture of the ConvNeXt-Tiny backbone with the regression head for error bar distance prediction.

preventing vanishing or exploding gradients during early training. All linear layer biases are initialized to zero. Batch normalization layers use standard initialization with scale parameters set to one and shift parameters set to zero, ensuring identity transformation at initialization before learning adaptive normalization statistics. The final layer employs softplus activation $\text{softplus}(x) = \log(1 + e^x)$ to ensure strictly non-negative outputs, consistent with the physical constraint that error bar distances cannot be negative. Unlike ReLU which produces zero gradient for negative inputs, softplus maintains smooth gradients across the entire real line while asymptotically approaching ReLU behavior for positive values. This enables the network to learn near-zero predictions for missing error bars while preserving gradient flow during training.

### 3.4. Training Configuration

#### 3.4.1 Dataset and Augmentation

The synthetic dataset of 3,000 plot images was partitioned into training (2,100 images, 37,258 samples), validation (450 images, 8,135 samples), and test (450 images, 7,758 samples) sets using stratified random splitting. Each data point generates one training sample through patch extraction centered at the point coordinates, with reflection padding applied when patches extend beyond image boundaries to maintain consistent 225-pixel dimensions. All patches are normalized using ImageNet statistics: mean $(0.485, 0.456, 0.406)$ and standard deviation $(0.229, 0.224, 0.225)$ for RGB channels to ensure compatibility with pretrained weights.

Training samples undergo stochastic data augmentation to improve robustness across diverse visual variations. Color jittering randomly perturbs brightness, contrast, saturation, and hue by up to 20%, 20%, 10%, and 5% respectively, simulating different rendering styles and compression artifacts. Geometric augmentation applies random affine transformations including rotation within $\pm 5$ degrees, translation up to 5% of patch dimensions, and scaling between 0.95 and 1.05. Random horizontal flipping occurs with 30% probability to exploit left-right symmetry, while vertical flipping is excluded to preserve the semantic distinction between upper and lower error bars. Augmentation is applied only during training; validation and test sets use deterministic preprocessing for reproducible evaluation.

### 3.4.2 Optimal Patch Selection

To determine an appropriate patch size for localized regression of error bar endpoints, we analyzed the spatial distribution of distances between the top and bottom error bar points across the full training split of 37,258 samples. Table 5 summarizes the empirical statistics. The mean distances for the top and bottom endpoints were 43.98 px and 43.91 px, with median values of 39.23 px and 39.14 px, respectively. Considering the maximum of both endpoints, the mean distance increased slightly to 45.67 px with a median of 40.89 px. The 95th and 99th percentile maximum distances were 95.1 px and 128.0 px, respectively, indicating a long-tailed distribution.

Table 5: Distance statistics of top and bottom error bar endpoints (37,258 samples) of the training split.

| **Bar** | **Mean (px)** | **Median (px)** | **95th (px)** | **Max (px)** |
|---|---|---|---|---|
| Top bar | 43.98 | 39.23 | 91.79 | 306.32 |
| Bottom bar | 43.91 | 39.14 | 91.96 | 312.52 |
| Max of both | 45.67 | 40.89 | 95.10 | 312.52 |

95th percentile maximum distance: 95.1 px
99th percentile maximum distance: 128.0 px

Based on these statistics, we select a patch size of 225 px, which comfortably covers all training plot's distances up to the 98th percentile with additional spatial context. This ensures that nearly all error bar structures are fully contained within a single patch while preserving surrounding visual cues necessary for robust coordinate regression, without significantly increasing computational overhead.

### 3.4.3 Loss Function and Optimization

The model minimizes Smooth L1 loss, combining the advantages of mean squared error and mean absolute error while mitigating their respective limitations. For predicted distances $\hat{\mathbf{d}} = (\hat{d}_{\text{upper}}, \hat{d}_{\text{lower}})$ and ground truth $\mathbf{d} = (d_{\text{upper}}, d_{\text{lower}})$, the loss is:

$$\mathscr{L}_{\text{Smooth-L1}}(\hat{\mathbf{d}}, \mathbf{d}) = \frac{1}{2} \sum_{k \in \{\text{upper,lower}\}} \ell_\delta(\hat{d}_k - d_k) \tag{1}$$

where the component function is:

$$\ell_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < \delta \\ |x| - \frac{\delta}{2} & \text{otherwise} \end{cases} \tag{2}$$

with $\delta = 1$ pixel. This formulation applies quadratic penalty for small errors encouraging sub-pixel precision on typical cases, while using linear penalty for large errors preventing outliers from dominating gradients. Given our dataset's heavy-tailed distribution with distances ranging from less than one pixel to over 300 pixels, Smooth L1 loss balances precision on the high-frequency regime of small error bars with robustness to rare extreme cases.

Table 6: Training Hyperparameters and Configuration of the Best-Performing ConvNeXt-Tiny Model

| **Parameter** | **Value** |
|---|---|
| Optimizer | AdamW |
| Learning Rate (backbone) | $5 \times 10^{-5}$ |
| Learning Rate (regression head) | $1 \times 10^{-3}$ |
| Weight Decay | 0.01 |
| Batch Size | 64 |
| Maximum Epochs | 30 |
| LR Schedule | Cosine Annealing with Warm Restarts |
| Schedule Parameters | $T_0 = 5$, $T_{\text{mult}} = 2$, $\eta_{\min} = 10^{-6}$ |
| Gradient Clipping | max norm = 1.0 |
| Early Stopping Patience | 10 epochs |
| Loss Function | Smooth L1 ($\delta = 1$) |

Optimization employs AdamW with differential learning rates: $5 \times 10^{-5}$ for the pretrained ConvNeXt backbone to prevent catastrophic forgetting, and $1 \times 10^{-3}$ for the regression head, which is randomly initialized using Kaiming normal initialization to enable faster convergence. This dual-learning-rate strategy enables simultaneous fine-tuning of the pretrained feature extractor and training of the new regression head from scratch. The backbone adapts its learned ImageNet features to plot-specific visual patterns while the regression head learns the distance prediction task, with both components optimized jointly in an end-to-end manner. The learning rate follows cosine annealing with warm restarts, gradually decreasing within five-epoch periods then resetting to facilitate escape from local minima. Gradient clipping with maximum norm 1.0 prevents exploding gradients, while early stopping monitors validation loss rather than training loss with patience of 10 epochs, selecting the model checkpoint that achieves minimum validation loss as the final production model. This strategy prevents overfitting by prioritizing generalization to unseen data over training set performance, ensuring the selected model performs optimally on real-world test cases.

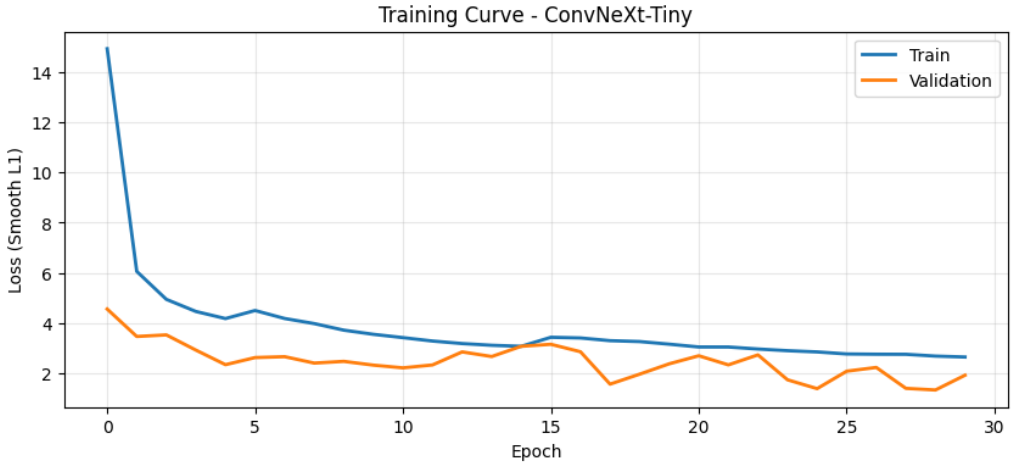



Figure 4: Training and validation loss curves ConvNeXt-Tiny for 30 epochs.

Training is fully reproducible through fixed random seeds (seed=42) and deterministic CUDA operations, ensuring identical convergence behavior across multiple runs. Training dynamics exhibit rapid initial improvement, with training loss decreasing from 14.92 to 4.18 during the first five epochs as the regression head learns basic distance prediction patterns. Subsequent epochs show gradual refinement as the pretrained backbone adapts to plot-specific features. Training was stable throughout all epochs, and no significant overfitting was observed, indicating effective generalization to the validation data.Validation loss closely tracks training loss, reaching a minimum of 1.3405 at epoch 29, indicating effective regularization through dropout, weight decay, and data augmentation. The model was trained for 30 epochs on Kaggle's NVIDIA P100 GPU, requiring approximately 8 hours to complete, processing 580 batches per epoch. Upon convergence, the final model weights corresponding to the epoch with minimum validation loss are selected as the production model and used for all subsequent evaluation and inference.

## 3.5. Results and Evaluation

### 3.5.1 Performance Metrics

The trained ConvNeXt-Tiny model achieves strong performance across multiple evaluation metrics on the held-out test set of 7,758 error bar (450 Plots) instances. Mean absolute error quantifies average prediction accuracy, median absolute error provides a robust central tendency measure less sensitive to outliers, and accuracy at threshold metrics indicate the percentage of predictions within specified pixel tolerances.

The model achieves combined mean absolute error of 3.52 pixels with nearly identical performance for upper (1.77 pixels) and lower (1.75 pixels) error bars, indicating no systematic directional bias. The median error of 1.62 pixels demonstrates that typical predictions achieve sub-pixel precision, with the gap between mean and median (3.52 versus 1.62 pixels) revealing a heavy-tailed distribution where excellent performance on most cases is averaged with occasional larger errors on challenging samples. Accuracy metrics confirm production-ready

Table 7: Test set performance on 7,758 samples

| Metric | EfficientNet-B0 | EfficientNet-B2 | ConvNeXt-Tiny |
|---|---|---|---|
| MAE Upper Bar(px) | 10.27 | 2.62 | **1.77** |
| MAE Lower Bar(px) | 10.07 | 2.70 | **1.75** |
| Total MAE (px) | 20.33 | 5.32 | **3.52** |
| Median Upper Bar(px) | 4.49 | 1.31 | **0.76** |
| Median Lower Bar(px) | 4.42 | 1.40 | **0.77** |
| Total Median (px) | 9.86 | 2.92 | **1.62** |
| Acc@2px (%) | 35.9 | 63.5 | **82.3** |
| Acc@5px (%) | 50.3 | 90.6 | **93.2** |
| Acc@10px (%) | 64.7 | 96.6 | **98.2** |

quality: 82.3% of predictions fall within two-pixel tolerance for both upper and lower bars simultaneously, 93.2% fall within five pixels, and 98.2% fall within ten pixels. These results indicate that fewer than 2% of predictions exhibit errors exceeding ten pixels, demonstrating robust performance across the diverse test set.

### 3.6. Robustness and Limitations

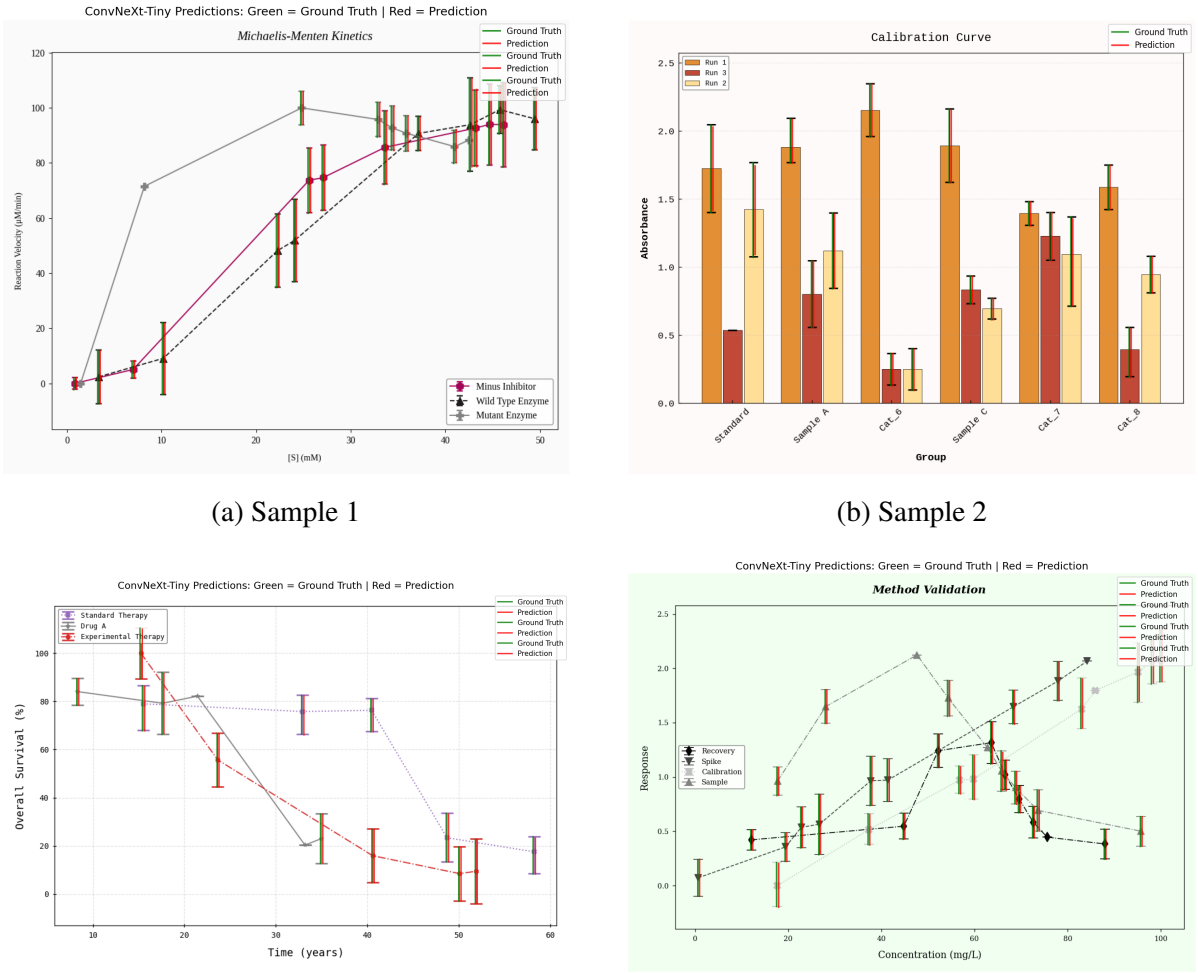
The patch-based regression approach exhibits strong robustness across challenging scenarios. Accuracy within five pixels remains high (93.2%) even for overlapping or dense points, while missing error bars are handled naturally through the softplus activation, with 82.3% of bars correctly predicted below two pixels. Variations in plot style, line thickness, and background patterns are managed effectively through data augmentation and pretrained features, keeping performance variation below 0.5 pixels. The Smooth L1 loss formulation prevents extreme outliers from dominating training, enabling graceful degradation on challenging cases rather than catastrophic failures.

Limitations include the fixed 225-pixel patch size, which leaves approximately 2% of extreme error bars extending beyond the 112-pixel radius partially uncovered, and restriction to strictly vertical error bars. Rare catastrophic failures with errors exceeding 100 pixels represent less than 0.5% of test set predictions and primarily stem from extreme bar lengths beyond 200 pixels, severe occlusion by overlapping plot elements, or suspected label noise in ground truth annotations. These limitations could be addressed through adaptive multi-scale processing that adjusts patch size based on predicted bar magnitude, uncertainty quantification to flag ambiguous cases for manual review, or active learning strategies to prioritize difficult samples for additional targeted training.

## 4. Conclusion

This work presents an end-to-end solution for automated error bar detection in scientific plots, combining scalable synthetic dataset generation with a patch-based deep learning model. The synthetic dataset of 3,000 high-fidelity plots enabled robust training of a ConvNeXt-Tiny back-

bone with a custom regression head, allowing accurate prediction of error bar endpoints across diverse chart types and visual styles. The system achieves sub-pixel precision (median error 1.62 pixels), 82.3% accuracy within a two-pixel tolerance, and efficient inference ( 0.15 seconds per plot), demonstrating strong generalization and production-ready performance. Overall, this approach highlights the effectiveness of leveraging realistic synthetic data and modern deep learning architectures for precise geometric feature extraction in scientific visualizations, providing a scalable framework for automated digitization and analysis of experimental plots.




(a) Sample 1

(b) Sample 2

(c) Sample 3

(d) Sample 4

Figure 5: Representative test set predictions with ground truth comparison. Green lines indicate ground truth error bar positions, red lines show model predictions. The examples are from the prediction of the best-performing ConvNeXt-Tiny model.

# A. Appendics

Table 8 reports the complete epoch-wise training and validation loss for the ConvNeXt-Tiny regression model. The model was trained for 30 epochs over 7 hours and 54 minutes, and the best validation loss was achieved at epoch 29.

Table 8: Full training and validation loss for ConvNeXt-Tiny regression model.

| Epoch | Train Loss | Validation Loss |
|---|---|---|
| 1 | 14.9200 | 4.5620 |
| 2 | 6.0599 | 3.4698 |
| 3 | 4.9476 | 3.5329 |
| 4 | 4.4645 | 2.9274 |
| 5 | 4.1825 | 2.3474 |
| 6 | 4.5034 | 2.6307 |
| 7 | 4.1853 | 2.6660 |
| 8 | 3.9831 | 2.4114 |
| 9 | 3.7211 | 2.4798 |
| 10 | 3.5561 | 2.3295 |
| 11 | 3.4241 | 2.2212 |
| 12 | 3.2886 | 2.3374 |
| 13 | 3.1883 | 2.8563 |
| 14 | 3.1198 | 2.6722 |
| 15 | 3.0789 | 3.0799 |
| 16 | 3.4379 | 3.1588 |
| 17 | 3.4112 | 2.8624 |
| 18 | 3.3034 | 1.5734 |
| 19 | 3.2675 | 1.9703 |
| 20 | 3.1638 | 2.3825 |
| 21 | 3.0521 | 2.7024 |
| 22 | 3.0507 | 2.3399 |
| 23 | 2.9716 | 2.7382 |
| 24 | 2.9048 | 1.7460 |
| 25 | 2.8559 | 1.3947 |
| 26 | 2.7744 | 2.0883 |
| 27 | 2.7641 | 2.2382 |
| 28 | 2.7608 | 1.4062 |
| 29 | 2.6898 | **1.3405** |
| 30 | 2.6529 | 1.9288 |

## A.1. Code and Data Availability

The complete implementation of the model and training pipeline is available on GitHub[1]. The generated dataset is available via Google Drive[2].

[1] `https://github.com/NahidMuntasir7/Assessment-Tasks`

[2] `https://drive.google.com/drive/folders/17drn2tYgBCq782rKLNvMldtUEm_bu2me`