Contents lists available at ScienceDirect

Engineering Structures

journal homepage: www.elsevier.com/locate/engstruct

OpenPyStruct: Open-source toolkit for machine learning-driven structural optimization

Danny Smyl [a], *, Bozhou Zhuang [a], Sam Rigby [b], Edvard Bruun [a], Brandon Jones [a], Patrick Kastner [c], Iris Tien [a], Adrien Gallet [d]

[a] School of Civil and Environmental Engineering, Georgia Institute of Technology, 790 Atlantic Drive, Atlanta, 30332-0355, GA, USA
[b] School of Mechanical, Aerospace and Civil Engineering, University of Sheffield, S1 3JD, UK
[c] School of Architecture, Georgia Institute of Technology, 280 Ferst Drive NW, Atlanta, GA, 30332-0355, USA
[d] Unipart Construction Technologies, Advanced Manufacturing Park, Brunel Way, Sheffield, S60 5WG, UK

ARTICLE INFO

Dataset link: https://github.com/dsmyl6/OpenPyStruct.git, https://zenodo.org/records/14742438

Dedicated to Professor Matthew Gilbert, a pioneer in structural optimization.

Keywords:
Structural design
Finite element method
Machine learning
Optimization
Python
Structural engineering

ABSTRACT

OpenPyStruct (Toolkit URL: OpenPyStruct, Repository URL: Data) is an open-source toolkit that provides finite element model based optimization frameworks for generating training data and machine learning models for global structural optimization of indeterminate continuous structures. The key machine learning feature of OpenPyStruct is its ability to optimize single or multiple arbitrary loading and support conditions. The framework utilizes multi-core central processing unit (CPU) and graphics processing unit (GPU)-enhanced implementations integrating OpenSeesPy for structural optimization. PyTorch is used for accelerated computations. Accompanying machine learning scripts enable users to train high-fidelity predictive models such as transformer with diffusion modules, physics-informed neural networks (PINNs), convolutional operations, and contemporary machine learning techniques to analyze and optimize structural designs. By incorporating state-of-the-art optimization tools, robust datasets, and flexible machine learning resources, OpenPyStruct aims to establish a scalable and fully-transparent engine for structural optimization by engaging the structural engineering community in this open-source toolkit.

1. Introduction

Structural design can be fundamentally viewed as an inverse problem [1–3], where the goal is to determine the optimal structural configurations and material properties that satisfy desired performance subject to specified loading and boundary conditions. Unlike forward problems, which involve computing structural responses under known parameters, inverse problems require reasoning backwards from performance targets to identify solutions [4]. For structural design, this process is inherently nonlinear, multi-constrained, and computationally demanding due to the complexity of physical laws, such as equilibrium, compatibility, and material behavior.

Moreover, inverse structural design is ill-posed, meaning that small variations in the desired performance criteria or input conditions can lead to significant changes in the solution, or the solution may not exist or have multiple valid solutions. Ill-posedness arises from the underdetermined and indeterminate nature of many structural design problems, where the available information about the desired performance is insufficient to fully specify the structural configuration [5]. This lack of robustness and determinacy further complicates the optimization process, as the solutions must be regularized or constrained to ensure physical feasibility and stability. Addressing this ill-posedness thus often requires advanced computational methods that can incorporate prior knowledge, enforce physical constraints, and handle uncertainty in the input data, all while maintaining computational efficiency [1].

Traditional approaches to structural optimization, including functionalities such as gradient-based methods and finite element simulations, have provided physically rigorous solutions to these problems [6, 7]. However, their applicability may be limited by high computational costs [8] when considering multiple simultaneous loading and boundary conditions, exploring high-dimensional design spaces or addressing scenarios with significant variability in input conditions. While this is the source of ongoing research [9], many such methods may struggle to incorporate and benefit from the increasing availability of large-scale datasets that capture diverse structural behaviors across a wide range of configurations and conditions.

* Corresponding author.
E-mail address: danny.smyl@ce.gatech.edu (D. Smyl).

https://doi.org/10.1016/j.engstruct.2025.120869

Available online 21 July 2025

Machine learning (ML) provides a powerful alternative solution for addressing these limitations by using data-driven techniques to model complex relationships between structural inputs and performance outcomes [10,11]. ML models, such as neural networks and ensemble methods, excel at capturing nonlinear patterns in high-dimensional datasets and generalizing across diverse conditions. Examples include publications on ML applications of topology optimization [12–16]. With access to large-scale datasets, ML enables rapid predictions, and thus reduces the computational burden of exploring design alternatives. In structural design, ML has been proven to be capable of making mappings between load configurations, boundary conditions, and optimal design parameters at a fast and accurate manner [1].

Incorporating physical laws into ML further enhances its applicability in structural engineering. Physics-informed neural networks (PINNs) embed governing equations from structural design into the learning process [17,18]. Therefore, critical mechanical constraints such as equilibrium and compatibility can be satisfied. The integration of physics with ML mitigates the “black-box” nature of traditional data-driven models and improves the reliability and interpretability in engineering applications.

In our approach, we propose `OpenPyStruct`, an open-source platform that integrates single- and multi-core physics-based optimization with ML workflows. By utilizing `OpenSeesPy` [19] with `PyTorch` [20] for accelerated computation on central processing units (CPUs) and graphics processing units (GPUs), `OpenPyStruct` efficiently generates high-fidelity datasets with diverse structural responses across a wide range of loading conditions, support configurations, and structural geometries. Users can use ML scripts to train predictive models for single and multiple load and support structural optimization with or without physics constraints. This combination of rigorous physics-based simulations, large-scale data generation, and ML tools offers engineers and researchers a scalable, flexible, and high-performance platform for solving complex structural optimization problems. In summary, OpenPyStruct is designed to discover optimized structural designs under complex and ill-posed conditions. It combines physics-based optimization with machine learning, and can be extended to other structural designs such as frame design.

The contributions of this paper are as follows:

- Development of an open-source framework that integrates physics-based structural simulation with ML techniques for structural design and optimization.
- Establishment of a scalable data-generation pipeline that efficiently produces high-fidelity datasets with diverse loading and support conditions.
- Disclosure of datasets for predicting cross-section properties with ML.
- Capacity demonstration of the proposed framework with multiple ML approaches for global structural optimization.

2. The `OpenPyStruct` framework

2.1. Overview

As shown in Fig. 1, the architecture of `OpenPyStruct` is designed to provide a modular framework for continuous structural optimization. It encompasses the following core modules: single load case simulation-based optimization, data generation with a single-core data generator and a multi-core data generator, and ML scripts for feedforward neural networks (FNNs), PINNs, and transformer models with diffusion processes. In addition, `OpenPyStruct` includes advanced tools for data visualization, statistical analysis, and ML optimization to make it a versatile platform for tackling different structural design problems. All optimization tasks, from structural property adjustments to neural network training, are handled by computation graphs in `PyTorch`.

2.2. *`OpenPyStruct` modules*

As shown in Fig. 1(a), the single load simulation-based optimization module uses `OpenSeesPy` to perform physics-based optimization on structural properties such as second moments of area. It should be mentioned that this module can be modified to output multiple design targets. This component is tailored for scenarios where the objective is to optimize a single structural configuration under specified loading and boundary conditions. By directly solving the governing mechanics, the module ensures that results adhere to physical laws for both standalone optimization tasks and dataset generation.

Establishing a scalable structural data-generation pipeline is a core objective of the `OpenPyStruct` toolkit. Relevant structural data includes information on support conditions, loading, geometry and optimized design parameter distributions. To achieve this, three complementary modules are provided, a single-core generator, a GPU generator and a multi-core generator, as shown in Fig. 1(b). The single-core and GPU generator executes simulations sequentially. However, it may be slow for generating large-scale datasets. To address this limitation, the multi-core generator employs parallelized simulations powered by `OpenSeesPy` and `PyTorch` with multi-threaded CPUs. The generated datasets incorporate diverse structural configurations, boundary conditions, spatial information and loading scenarios for training ML models.

The ML scripts in `OpenPyStruct` provide advanced tools for modeling and analysis, as demonstrated in Fig. 1(c). Classical FNNs offer a straightforward and computationally efficient approach for predictive modeling. PINNs incorporate governing equations into the learning process. This allows the physical constraints to be directly embedded into the model to ensure that predictions are consistent with physics principles. Transformer models with diffusion modules have capabilities to handle sequential data, variable-length inputs, and highly nonlinear relationships. All scripts include preprocessing utilities for data standardization, trend visualization, and unscaling operations to ensure that results are interpretable and consistent with physical units, as shown in Fig. 1(d).

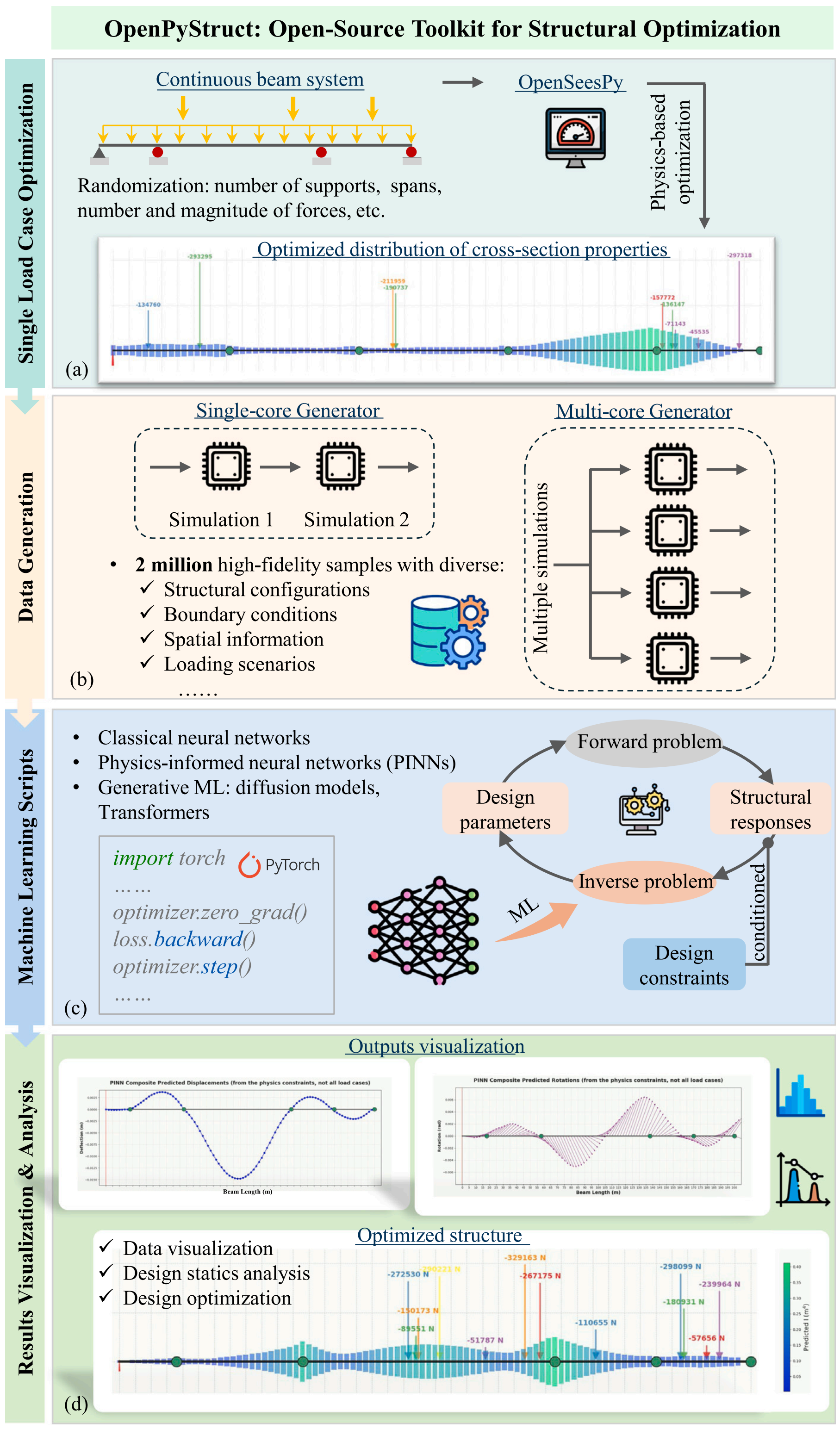
`OpenPyStruct` also integrates advanced ML optimization protocols to enhance the performance and flexibility of its models. Learning rate scheduling ensures adaptive training by dynamically adjusting optimization rates to balance convergence speed and stability. Regularization techniques (e.g., weight decay, noise scheduling, dropout, etc.) aim to mitigate overfitting and improve the generalization of models on unseen data. Additionally, the architecture of the framework allows for flexible customization of neural network configurations to allow users to tailor models to specific problem domains. These features, combined with robust visualization and statistical tools, position `OpenPyStruct` as a flexible framework that bridges physics-based optimization with modern data-driven methodologies for structural design and analysis.

2.3. *`OpenPyStruct` scripts and data overview*

Several scripts are provided in `OpenPyStruct` as summarized in Table 1. Specifically, the `OpenPyStruct_PINN_MultiCase.py` script integrates PINNs to simultaneously optimize design parameters, beam deflections and rotations. By combining data-driven learning with physics-based loss functions, this module efficiently captures complex structural behaviors and boundary conditions.

The `OpenPyStruct_FNN_MultiCase.py` script employs FNNs with residual connections for scalability and robust performance across diverse structural configurations. Its architecture is optimized for efficient handling of multi-load cases while incorporating advanced regularization to enhance generalization capabilities.

The `OpenPyStruct_TransformerDiffusionModule_MultiCase.py` script introduces cutting-edge Transformer-based encoders and diffusion models. With multi-head attention mechanisms


OpenPyStruct: Open-Source Toolkit for Structural Optimization
Single Load Case Optimization
Continuous beam system
OpenSeesPy
Physics-based optimization
Randomization: number of supports, spans, number and magnitude of forces, etc.
Optimized distribution of cross-section properties
(a)
Data Generation
Single-core Generator
Simulation 1
Simulation 2
Multi-core Generator
Multiple simulations
2 million high-fidelity samples with diverse:
✓ Structural configurations
✓ Boundary conditions
✓ Spatial information
✓ Loading scenarios
……
(b)
Machine Learning Scripts
Classical neural networks
Physics-informed neural networks (PINNs)
Generative ML: diffusion models, Transformers
PyTorch
Forward problem
Design parameters
Structural responses
Inverse problem
ML
conditioned
Design constraints
(c)
Results Visualization & Analysis
Outputs visualization
PINN Composite Predicted Displacements (from the physics constraints, not all load cases)
PINN Composite Predicted Rotations (from the physics constraints, not all load cases)
Beam Length (m)
Optimized structure
✓ Data visualization
✓ Design statics analysis
✓ Design optimization
(d)

```
import torch
……
optimizer.zero_grad()
loss.backward()
optimizer.step()
……
```

Fig. 1. OpenPyStruct workflow summarizing key functionalities, dependencies and capabilities. (a) Single load case optimization, (b) Data generation, (c) Machine learning scripts, and (d) Results visualization & analysis.

and diffusion-based latent space exploration, this script excels in predicting deflection, rotation, and stress distributions for complex structural systems. Its advanced feature extraction enables improved precision during optimization under intricate load scenarios.

In addition to the core modules, cutting-edge experimental models (labeled ‘Beta’) are also provided in the toolkit for user exploration. These models include Graph Neural Network (GNN), Bayesian Transformer Diffusion, trainable output weighting (meta parameters), Fourier Neural Operation (FNO) and evolutionary approaches. In addition, a discrete (i.e., constant design variables per member) *frame* optimizer is provided. These models are fully functional and offer a nucleation point for future toolkit community development.

Supporting these optimization modules are computational backends including modules `OpenPyStruct_BeamOpt_training_GPU.py`

Table 1
Summary of OpenPyStruct core modules and Datasets as well as experimental ('Beta') modules.

Core Module/Script Shorthand	Functionality	Key Features
`PINN_MultiCase`	PINN-based optimization	Combines data-driven and physics-informed methods
`FNN_MultiCase`	Scalable FNN for multi-load case problems	Residual connections and regularization
`TransformerDiffusionModule_MultiCase`	Transformer and diffusion optimization	Multi-head attention, latent space exploration
`BeamOpt_training_GPU`	GPU-accelerated data generation	Rapid, large-scale dataset creation
`BeamOpt_training_MultiCore`	Paralleli data generation on multiple cores	Efficient handling of large computational loads
`BeamOpt_training_SingleCore`	Single-core data generation	Optimized for environments without parallel processing
`BeamOpt`	Single-load optimization tool	Focused on individual structural load cases
StructDataLite	Compact dataset: 100,000 samples	For initial training and debugging
StructDataHeavy	Comprehensive dataset: 2,000,000 samples	Extensive variability for high-fi training
Beta Module/Script Shorthand		
`Bayesian_TFDModule_MultiCase_Beta`	Bayesian Transformer and diffusion-based optimization	Bayesian representation off MLP, multi-head attention, diffusion module
`Bayesian_TFDModule_MultiCase_Meta_Beta`	Bayesian Transformer and diffusion-based optimization with output weighting	Bayesian representation off MLP, multi-head attention, diffusion module, trainable output weights, output statistical analyzer
`GNN_MultiCase_Beta`	Graph NN representation of the structural system	Graph connectivity, adjacency, chain attention
`FNO_MultiCase_Beta`	FNO-based optimization for multi-load cases	Fourier Neural Operators, spectral convolutions, customizable layer stacking
`BeamOpt_Evolutionary_Beta`	Evolutionary algorithm based single-load optimization tool	Population-based search, tournament selection, uniform crossover, mutation, and elitism
`FrameOpt_Discrete_Beta`	Single-load discrete *frame* optimization tool	Focused on individual load cases for arbitrary bays and stories

and `OpenPyStruct_BeamOpt_training_MultiCore.py`, which leverage GPU acceleration and parallel processing, respectively. These backends enable rapid and large-scale data generation, which is crucial for creating extensive datasets required for ML training and evaluation.

Two primary datasets, **StructDataLite** and **StructDataHeavy**, were generated using the aforementioned backends:

- **StructDataLite**: This dataset contains 100,000 samples and serves as a compact resource for initial training, debugging, and exploratory analysis. Support conditions are fixed in this dataset.
- **StructDataHeavy**: A comprehensive dataset comprising 2,000,000 samples for high-fidelity model training. This dataset provides extensive variability across structural configurations and load cases.

Both datasets include variable features such as roller positions, force locations and magnitudes, beam node coordinates, and derived physics outputs containing deflections, rotations, shear force and bending moments for advanced structural analysis and optimization.

3. OpenPyStruct for classical model based optimization

3.1. Mathematical model based optimization approach

In OpenPyStruct, structural optimization is formulated to refine design parameters, such as I, to achieve an optimal balance between performance, material distribution, and structural feasibility. The framework employs a composite loss function $\mathcal{L}$ that integrates multiple components representing different structural behaviors. The optimization process is driven by the minimization of this loss function to make sure physical constraints such as equilibrium, compatibility, and material behavior are satisfied. This formulation is flexible to allow users to adjust weights for bending and shear penalties, and to extend or replace the design parameters I with others, such as cross-sectional areas A or user-specified material properties.

The structural optimization problem in OpenPyStruct is defined as:

$$\min_I \mathcal{L}(I) = \mathcal{L}_{\text{primary}}(I) + \alpha_{\text{moment}}\mathcal{L}_{\text{bending}}(M, E, I) + \alpha_{\text{shear}}\mathcal{L}_{\text{shear}}(G, k, V, I) \tag{1}$$

where the primary loss term, $\mathcal{L}_{\text{primary}}(I)$, is defined as the sum of the second moments of area I_e for each beam element e:

$$\mathcal{L}_{\text{primary}}(I) = \sum_e I_e. \tag{2}$$

This term minimizes the design parameter I_e, aiming to indirectly optimize material quantity for each element. The bending loss term, $\mathcal{L}_{\text{bending}}(M, E, I)$, penalizes elastic bending energy via:

$$\mathcal{L}_{\text{bending}}(M, E, I) = \sum_e \frac{M_e^2}{2EI_e}. \tag{3}$$

Similarly, the shear energy loss term, $\mathcal{L}_{\text{shear}}(G, k, V, I)$, penalizes shear energy via:

$$\mathcal{L}_{\text{shear}}(G, k, V, I) = \sum_e \frac{V_e^2}{GA_e}. \tag{4}$$

Here, $A_e \approx kI_e^{\frac{1}{2}}$ is the approximate cross-sectional area of the eth element, which is approximated through a user-defined transformation coefficient k,[1] and is tied to the second moment of area I_e for optimization. Moreover, E is the Young's modulus, G is the shear modulus, M is the bending moment and V is the shear force. Meanwhile, the parameters α_{moment} and α_{shear} are user-defined coefficients that control the relative importance of the bending and shear penalties in the total loss function. These coefficients can also be treated as learnable parameters during the optimization process to allow for more flexible and adaptive control over the loss components.

3.2. Optimization implementation and loss computation

The optimization process in OpenPyStruct is implemented iteratively using PyTorch's automatic differentiation to compute gradients

[1] Alternatively, the user may define A and I explicitly or omit the shear energy loss as they see fit.

of the loss function $\mathcal{L}(I)$. The integration of PyTorch ensures that gradients with respect to the design parameters I are computed efficiently. Below is a detailed description of the algorithm for setting up the structural model and computing the loss components.

Structural model setup: The structural model is configured using OpenSeesPy, where nodes, supports, elements, and loads are defined based on the input design parameters and configurations.

Loss function computation: The loss function comprises multiple components, including the primary design parameter, bending energy, and shear energy, each weighted to balance their contributions. The bending energy penalizes large bending moments, while the shear energy accounts for the impact of shear forces. The primary loss minimizes the total second moments of area. This workflow ensures that gradients of the combined loss function with respect to the second moments of area I are efficiently computed.

3.3 Optimization workflow

The general workflow for the optimization workflow is as follows (detailed description provided in Appendix B):

1. **Initialization**: Design parameters I_e are initialized with a constant guess I_0:

   $$I_e^{(0)} = I_0, \quad \forall e.$$

2. **Simulation Setup**: Structural properties, node positions, and boundary conditions are defined in OpenSeesPy.
3. **Gradient-Based Update**: Gradients of $\mathcal{L}(I)$ are computed using PyTorch's computational graph, and design parameters are updated.
4. **Stopping Criteria**: Optimization continues until the following is satisfied at iterate s with patience p:

   $$|\mathcal{L}(I^{(s)}) - \mathcal{L}(I^{(s-1)})| < \text{tolerance}.$$

5. **Final Solution**: The optimized design parameters I_e^{opt} are obtained and can be further analyzed/visualized.

3.4 Optimizer and optimizer-acceleration framework

The optimizer framework offers three execution modes to suit different hardware and debugging needs, all built on a shared core routine (Adam optimizer with learning-rate scheduling and loss minimization):

- **GPU-Accelerated Mode:** Leverages PyTorch to dispatch tensor operations to available GPU(s) and streaming real-time progress via tqdm.
- **Single-Core CPU Mode:** Runs the same training-sample generation and optimization pipeline sequentially – ideal for debugging or small-scale experiments – with user-configurable parameters (e.g., node count, roller supports, applied forces) and robust error handling during analysis.
- **Multi-Core CPU-Accelerated Mode:** Uses joblib's loky backend to parallelize sample generation across multiple CPU cores, supporting user-defined worker counts and batch sizes (e.g., 500 samples/batch), intermediate progress updates, and synchronized shared-data management.

3.5 Comparison of program variants

Each program variant is tailored to specific computational environments. The GPU-accelerated optimizer is fast yet only for multiple-GPU computing utilizing TensorCores. The single-core optimizer is robust and simple, and thus is suitable for small-scale or debugging tasks. The multi-core optimizer strikes a balance between speed and accessibility. It is well-suited for multi-processor systems without GPUs. For example, 128 cores were used in generating the training data herein. It is worth noting that OpenSeesPy is required when running on a remote high-performance computing (HPC) system.

3.6 Visualization features

Visualization plays a convenient role for understanding the structural behavior and the outcomes of the optimization process in OpenPyStruct. The framework provides detailed graphical representations of the structural configurations, internal forces, and design parameter distributions. These visualizations allow users to analyze the effects of design choices, evaluate optimization performance, and gain insights into the structural response under varying loads and boundary conditions. OpenPyStruct supports customizable plotting for beam geometry, support conditions, applied loads, second moments of area, shear force diagrams, and bending moment diagrams.

A key feature of the visualization module is the ability to scale beam thickness based on the optimized second moments of area. This ability provides an intuitive understanding of how structural properties are distributed along the span. Support conditions, such as pinned and roller supports, are highlighted using distinct markers, and applied loads are depicted with arrows indicating their magnitude and direction. In addition to geometric visualization, OpenPyStruct generates shear force and bending moment diagrams for each structural configuration. Shear forces are visualized using stepped plots, while bending moments are plotted as continuous curves.

3.7 Flexibility in design parameters

The second moment of area I is chosen as the primary design variable in this work because it directly governs bending stiffness and enables efficient material distribution. The modularity of OpenPyStruct aims to provide flexibility in defining and optimizing structural design parameters. While the second moments of area I are the primary design parameters in the current implementation, the framework is designed to accommodate alternative or supplementary parameters, such as the cross-sectional area A, elastic modulus E, or even load distributions. This adaptability allows users to tailor the optimization problem to the specific needs of their application, ranging from single-variable optimization to complex multi-variable formulations.

For example, replacing I with A enables the optimization of cross-sectional areas directly, which is particularly useful in scenarios where material usage is a critical factor. Similarly, incorporating E as a variable allows for material property optimization, such as selecting the optimal modulus of elasticity for composite or mixed-material structures. In cases where multiple design parameters need to be optimized simultaneously, OpenPyStruct supports multi-objective formulations, enabling trade-offs between competing objectives like cost, weight, and performance.

This flexibility is achieved through the modular structure of the framework, where the loss function, simulation setup, and optimization workflow are parameterized to allow seamless substitution or addition of design variables. Below is an example of modifying the loss function to include cross-sectional area A as a design parameter. Moreover, OpenPyStruct's flexible architecture makes it easy to incorporate additional constraints or physical relationships. For example, if A is proportional to I, the proportionality constant can be explicitly enforced during the optimization process to maintain physical consistency. This adaptability is particularly beneficial in complex structural optimization problems, where multiple design variables interact or where additional constraints must be imposed. By providing a robust and modular framework, OpenPyStruct allows researchers and engineers to explore a wide variety of structural optimization tasks without being constrained by a rigid implementation.

3.8 Flexibility in loss functions

OpenPyStruct supports the integration of a diverse range of loss functions, which can be readily incorporated into the optimization

Table 2
Potential loss functions applicable to `OpenPyStruct`: higher-order derivatives are also available via AutoGrad.

Loss function	Description	Formula
L_2	Penalizes large deviations from target design	$\mathcal{L}_{L_2} = \sum_e (x_e - x_{\text{target},e})^2$
L_1	Minimizes absolute differences	$\mathcal{L}_{L_1} = \sum_e \lvert x_e - x_{\text{target},e} \rvert$
Total Variation	Reduces abrupt changes in design parameters	$\mathcal{L}_{\text{TV}} = \sum_e \lvert x_e - x_{e+1} \rvert$
Elasticity	Penalizes deviations from optimal elasticity	$\mathcal{L}_{\text{elasticity}} = \sum_e \left(\frac{1}{E_e} - \frac{1}{E_{\text{target},e}} \right)^2$
Shear Strength	Minimizes shear deformations using shear strength	$\mathcal{L}_{\text{shear}} = \sum_e \left(\frac{V_e^2}{\zeta_e} \right)$
Stress Constraint	Penalty for exceeding stress limits	$\mathcal{L}_{\text{stress}} = \sum_e \max(0, \bar{\sigma}_e - \bar{\sigma}_{\text{max}})^2$
Geometric	Penalizes deviations from nonlinear behavior	$\mathcal{L}_{\text{geometric}} = \sum_e \left(\frac{g_e}{g_{\text{linear}}} - 1 \right)^2$
Fatigue	Penalizes high-cycle stresses and strains	$\mathcal{L}_{\text{fatigue}} = \sum_e \left(\frac{\bar{\sigma}_e}{\bar{\sigma}_{\text{fatigue}}} \right)^2$
Buckling	Prevents buckling by penalizing high risk	$\mathcal{L}_{\text{buckling}} = \sum_e \left(1 - \frac{\bar{P}_{\text{critical},e}}{\bar{P}_{\text{applied},e}} \right)^2$
Energy Dissipation	Optimizes energy dissipation and stability	$\mathcal{L}_{\text{energy}} = \sum_e \frac{\Xi_{\text{dissipated},e}}{\Xi_{\text{input},e}}$
Displacement	Penalizes large displacements, ensuring stability	$\mathcal{L}_{\text{displacement}} = \sum_e \left(\delta_e - \delta_{\text{target},e} \right)^2$
Rotation	Minimizes rotational displacement within limits	$\mathcal{L}_{\text{rotation}} = \sum_e \left(\theta_e - \theta_{\text{target},e} \right)^2$
Curvature	Ensures curvature stays within desired bounds	$\mathcal{L}_{\text{curvature}} = \sum_e \left(\kappa_e - \kappa_{\text{target},e} \right)^2$
Deflection	Minimizes deflection at specific points	$\mathcal{L}_{\text{deflection}} = \sum_e \left(\delta_e - \delta_{\text{max},e} \right)^2$

and ML workflows. Below, in Table 2 are some potential loss functions that could be integrated within the framework. Each offers flexibility to address different aspects of structural optimization problems. These loss functions can be applied to various stages of the optimization process, from design refinement to advanced ML tasks, enhancing the model's adaptability and performance.

OpenPyStruct supports multi-objective structural optimization via a modular loss framework that allows users to combine and weight multiple objective functions, such as material usage, stiffness, and stress. The trade-offs between multiple objectives can be flexibly adjusted during training. This flexibility allows users to prioritize specific structural performance criteria, such as reducing stress, enhancing material usage efficiency, or minimizing fatigue damage, all while ensuring that the solution adheres to physical laws and constraints.

Moreover, `PyTorch`'s computation graph enables efficient calculation of gradients for these diverse loss functions, ensuring that complex multi-objective optimization problems can be tackled in a straightforward manner. By adjusting the weights or combining different loss functions, users can customize their optimization objectives to match the requirements of specific structural design tasks, facilitating the creation of robust, efficient, and cost-effective structures.

3.9 Data generation workflow summary

Lastly, we provide pseudo code in Algorithm 1 summarizing the general workflow for the data generators and the JSON data structures the generators save. In the present configuration, the design variables distributions being saved are I, and the additional variables being saved are used as ML model inputs: [support locations, point force locations, point force magnitudes, node locations], physics constraints or supplementary information used in visualization.

4 `OpenPyStruct` for machine learning optimization

In this section, we discuss three ML models integrated into the `OpenPyStruct` framework: the FNN, PINNs, and Transformer-Diffusion model. These models are used to predict optimal structural design parameters based on various input features, such as nodal coordinates, support conditions, and load scenarios. All models are set up for handling multiple or single load cases and support conditions. Specifically, the FNN and Transformer-Diffusion model rely purely on data-driven learning in this formulation whose physics can be added to the Transformer-Diffusion model. The PINN incorporates physical constraints and complex data relationships. Key functionalities of the ML package, including data handling, loss functions, unique features, architecture, and an overview of the workflow, are illustrated in the flow chart below (see Table 3).

Algorithm 1: Beam Data Generation and Storage

```
Input  : Beam parameters, load settings, optimization settings
Output : JSON file containing beam analysis data
Initialize beam parameters, loads, optimization settings, and an empty
  data structure
Function SetupModel():
    Define beam nodes and elements
    Apply supports and loads
    Run static analysis
Function GenerateSample():
    Randomize beam properties (if enabled)
    Assign roller supports and force locations
    Initialize moment of inertia values
    for each optimization step do
        SetupModel()
        Compute shear forces and bending moments
        Update inertia values using optimization
        Check stopping criteria
    Store computed sample data
Function Main():
    for each batch of samples do
        Generate multiple samples in parallel by calling
          GenerateSample()
        Filter and store valid results
        Print progress
    Save all data as a JSON file
Main()
```

4.1 Loss function approach

The loss function is a critical component of any ML framework as it controls the degree to which a network responds to outliers. While there are many loss choices, we select a joint loss for all models employing a combination of L_1 and L_2 norms weighted dynamically by a trainable parameter, α. This allows the model to adaptively emphasize absolute or squared differences during optimization. Moreover, the loss function includes constraints that penalize predictions exceeding predefined physical boundaries, thereby ensuring the outputs remain realistic and interpretable. In our approach, we write our common loss as follows:

$$\mathcal{L} = \alpha \cdot L_1(y_i, \hat{y}_i) + (1-\alpha) \cdot L_2(y_i, \hat{y}_i) + \beta \cdot P(\hat{y}_i) + \lambda \cdot L_2(W), \tag{5}$$

where y_i is the true value for the ith sample, $\hat{y}_i$ is the predicted value, λ is a regularization parameter, W represent the network's parameters to be regularized in an attempt to prevent overfitting and β controls the

Table 3
General workflow for the ML optimization approaches used in `OpenPyStruct`.

Step	Description
Start: Input Data	The initial data provided by the user or dataset, containing structural configurations and load scenarios.
Data Preprocessing	Cleaning, normalizing, and preparing the input data for further analysis. Includes padding sequences and feature scaling.
Feature Extraction	Extracting meaningful features from the input data to ensure relevant information is passed to the model.
Model Processing	The main computational engine processes the extracted features to understand patterns and relationships.
Prediction Generation	Producing predictions based on the processed data. Predictions include key output metrics for structural evaluation.
Output Refinement	Post-processing the predictions to ensure they align with physical constraints or domain-specific requirements.
Evaluation	Comparing predictions against validation data to assess accuracy and model performance (e.g., using metrics like R^2).
Finish: Deployment	Preparing the model for real-world applications, allowing users to input new data for live inference and predictions.

penalty magnitude. The penalty terms P act as a regularizer, applying additional loss for predictions outside an allowable range for the design parameters (i.e. a box constraint). This box penalty mechanism aims to improve convergence and enforce adherence to domain-specific constraints, for example that physical parameters such as I cannot be negative.

4.2 Data preprocessing and label aggregation approach

The following steps are adopted using data from the generator or using the provided StructDataHeavy/StructDataLite datasets:

Data preprocessing The raw input data consists of diverse structural configurations, including roller locations, force positions, force values, and node positions. These inputs vary in size and dimension, thus preprocessing to standardize and align the data is required. The steps include:

1. *Padding Sequences:* Input arrays are padded to a consistent length based on the maximum observed size within the dataset. This ensures that all samples conform to the expected dimensionality.
2. *Feature Scaling:* Each input feature is normalized using a standard scaler to zero mean and unit variance. This prevents numerical instabilities and accelerates convergence during training.
3. *Dimensional Padding:* The feature dimensions are padded to be a multiple of the number of Transformer attention heads to ensure compatibility with the multi-head attention mechanism.

Label aggregation The design variables for each structural configuration are aggregated across multiple design cases, as shown schematically in Fig. 2. The aggregation process computes a unified label that captures the variability across cases while maintaining physical interpretability. The approach involves:

1. *Statistical Aggregation:* For each element, the mean μ and standard deviation σ of design variables are computed over each subset of output data containing arrays of dimension [number of design cases, number of elements] and used for label unification. This is a key facet in handling optimization over multiple simultaneous structural configurations. The unified label we adopt is written:

$$y = \mu + c \cdot \sigma \tag{6}$$

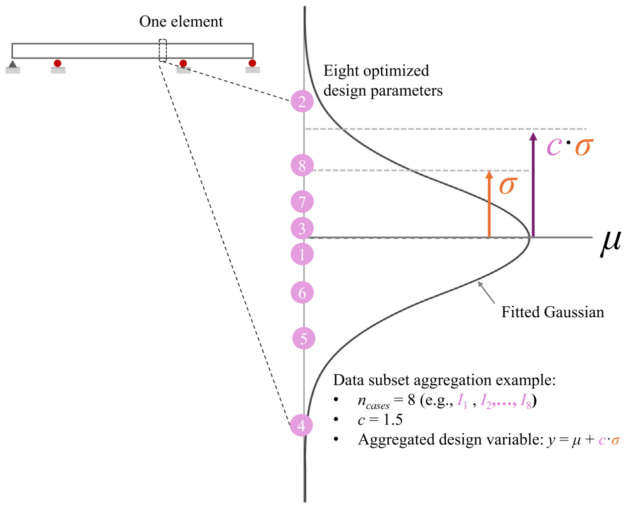

One element
Eight optimized design parameters
2
8
7
3
1
6
5
4
$c \cdot \sigma$
σ
μ
Fitted Gaussian
Data subset aggregation example:
• $n_{cases} = 8$ (e.g., $I_1, I_2, \ldots, I_8$)
• $c = 1.5$
• Aggregated design variable: $y = \mu + c \cdot \sigma$

Fig. 2. Schematic aggregation of design parameters for a single data subset across multiple load cases on single element.

where c is a scalar coefficient controlling design conservativeness. This value is user-selectable, where $c \geq 1.0$ emphasizes a higher threshold of design parameters, for example, c standard deviations above the mean (representing a more conservative choice for I in our case). Conversely, $0 < c < 1.0$ could be interpreted as a minimal design.

2. *Handling Outliers:* Outliers in the design parameter space are thus mitigated through the unification approach, thus promoting stability in training.

This preprocessing pipeline ensures that the input data and labels are consistent, normalized, and aligned. In addition, functions including mode and median statistics are available in the script which can be used depending on the user's desired statistical approach.

4.3 Learning dynamics approach

The neural networks learn by iteratively minimizing the loss function $\mathcal{L}$ over the training dataset. During forward propagation, the diffusion module processes noisy inputs, while the Transformer encoder extracts sequence-level features. Gradients of $\mathcal{L}$ with respect to model parameters W are computed via backpropagation:

$$\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial W}. \tag{7}$$

These gradients are handled using the `PyTorch` AutoGrad functionality and are used to update parameters using the Adam optimizer via:

$$W_{s+1} = W_s - l_s\nabla\mathcal{L} \tag{8}$$

where l_s is the learning rate at step s which is gradually adjusted with a scheduler. In all cases noise injection is used during training to sequentially regularize the model, improving its ability to generalize while optimized via the Adam optimizer. In our approach, we utilize the Gradscaler functionality in `PyTorch` with Autocast for computational speedup via mixed-precision utilization.

4.4 Evaluation and inference approach

The evaluation and inference are designed to assess model performance and enable predictions. These processes are implemented for the user to ensure robustness, interpretability, and efficiency of their ML optimization model.

Evaluation procedure The evaluation phase involves:

- **Data Loading:** Validation data is processed into batches using the same preprocessing pipeline as the training data. This ensures consistency in scaling and dimensionality.
- **Batch-wise Predictions:** The model generates predictions for each batch of validation inputs.
- **Metric Computation:** The predictions are compared against ground truth values using the R^2 score after unstandardizing. This metric provides a measure of how well the model's predictions align with the true values.

Inference pipeline (user input phase) Inference is structured to handle user-provided input data and generate predictions with the trained model:

- **Input Preparation:** User inputs, such as force positions, values, and node positions, are scaled and padded to align with the model's expected input dimensions. Features are concatenated to form a unified input tensor.
- **Model Forward Pass:** The prepared input tensor is passed through the model to obtain predictions.
- **Post-Processing:** Predicted values are unscaled and optionally clipped to predefined physical ranges.
- **Visualization:** Results are visualized through custom plots, including representations of forces applied to beams and corresponding second moments of area. Semi-transparent blocks and arrows illustrate predicted and input forces for enhanced interpretability.

4.5 Feedforward neural network optimizer

The FNN optimizer serves as the foundational model within the `OpenPyStruct` framework. As the most basic neural architecture, the FNN processes data in a unidirectional flow, from input to output, through a sequence of hidden layers. Each layer applies a weighted linear transformation followed by a nonlinear activation, enabling the network to model relationships between structural input features and optimized design parameters.

This model is designed to handle structural optimization tasks with relatively low computational overhead, such as the optimized structure with one load case (more loads cases are possible with the FNN, as described in the Case Studies) shown in Fig. 3. The aim is that the FNN provides the groundwork for introducing more sophisticated models within the framework, such as PINNs and Transformer-Diffusion models. In the context of this package, the FNN Optimizer represents the most accessible and computationally lightweight approach, and it bridges traditional data-driven methodologies with the advanced physics-informed and transformer-based models detailed used in the PINN model. Its straightforward implementation and training pipeline make it an ideal starting point for users new to machine learning in structural optimization. Furthermore, the FNN establishes a benchmark for comparison with more advanced architectures, highlighting the progressive capabilities of `OpenPyStruct` to address increasingly complex structural design challenges.

4.6 Physics-informed neural network optimizer

The PINN module in `OpenPyStruct` incorporates structural mechanics into the structural optimization problem to directly constrain and regularize the model during the learning process. Our aim is to enable PINNs to bridge the gap between data-driven machine learning models and physics-based methods by enforcing governing physics, which include equilibrium and compatibility conditions as part of the optimization framework.

Loss function and constraints The loss function in the PINN module is designed to minimize discrepancies in structural parameters while ensuring adherence to physical constraints. The total loss function $\mathcal{L}_{\text{total}}$ combines contributions from the primary data-driven loss $\mathcal{L}$ and physics-informed penalties $\mathcal{L}_{\text{physics}}$. This formulation is expressed as:

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda_{\text{physics}}\mathcal{L}_{\text{physics}} \tag{9}$$

where λ_{physics} is a scaling parameter that balances the influence of the physics-informed penalty. Meanwhile, the data-driven loss is the same as described in Eq. (5) while the physics loss $\mathcal{L}_{\text{physics}}$ incorporates penalties for violations of the governing equations. For example, the deflection and rotation residuals are computed as:

$$\mathcal{L}_{\text{physics}} = \mathcal{L}_{\text{deflection}} + \mathcal{L}_{\text{rotation}}, \tag{10}$$

where the residual physics penalties are defined by:

$$\mathcal{L}_{\text{deflection}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|\delta_i - \delta_i^{\text{true}}\right|}{\left|\delta_i^{\text{true}}\right| + \epsilon}, \tag{11}$$

$$\mathcal{L}_{\text{rotation}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|\theta_i - \theta_i^{\text{true}}\right|}{\left|\theta_i^{\text{true}}\right| + \epsilon}. \tag{12}$$

where δ_i and θ_i represent the predicted deflections and rotations, while δ_i^{true} and θ_i^{true} are the corresponding ground truth values. The small constant ϵ prevents division by zero. It should be chosen that residual physics penalties were selected from empirical testing, generally providing better performance than the (valid) L_1 and L_2 loss choices. For bespoke PINN losses for structural design, we refer the reader to [18].

4.7 Transformer-diffusion optimizer

This algorithm represents the state-of-the art in ML based structural optimization - the approach utilizes a transformer-based architecture enhanced with diffusion mechanisms to predict structural responses across various load cases. The framework integrates attention mechanisms and diffusion processes to provide adaptive learning capabilities. Key features include:

- Integration of positional encodings and transformer layers for handling sequence-based input data efficiently;
- Diffusion modules that simulate noise and denoising processes to enhance model generalization and robustness;
- A custom loss function that balances L_1 and L_2 losses with learnable weights;
- Scalability and adaptability to different structural configurations and load scenarios.

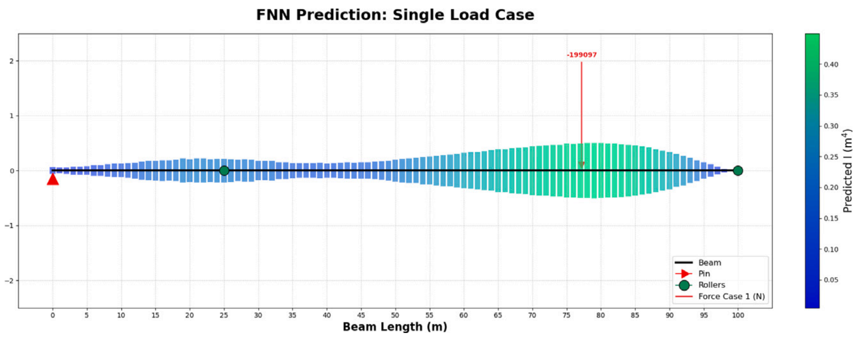


Fig. 3. Proof of concept: FNN optimized structure subject to one point load and simple support conditions.

Functional Overview

Architecture The core architecture comprises three main components: a diffusion module, a Transformer encoder, and a fully connected prediction network. The overall structure is as follows:

1. **Diffusion Module:** This module applies Gaussian noise to input features and learns to remove it using a standard perception (a NN). The noise application follows a predefined schedule to enhance model robustness by training it to recover clean signals from noisy data.
2. **Positional Encoding and Transformer Encoder:** Input sequences are encoded with positional information to retain spatial correlations. The Transformer encoder, consisting of multi-head attention and feedforward layers, processes these sequences to capture interdependencies between different load cases.
3. **Prediction Network:** The final network layers aggregate the learned features and output predictions for structural properties, such as second moments of area, across multiple elements.

Transformer-based processing The Transformer encoder employs multi-head self-attention to model interactions between input features across load cases. Positional encodings are added to retain spatial relationships, and the sequence is passed through stacked encoder layers to extract hierarchical representations. This architecture enables efficient handling of sequence-based inputs, making it ideal for multi-load case analysis.

Diffusion mechanism The diffusion mechanism is characterized by a controlled noise injection and removal process designed to enhance model generalization. Let $\mathbf{x} \in \mathbb{R}^d$ denote the original clean input vector, where $d \in \mathbb{N}$ is its dimension, $\mathbf{I}_d \in \mathbb{R}^{d\times d}$ denote the $d \times d$ identity matrix, $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$ be Gaussian noise drawn from a multivariate normal distribution with mean zero and covariance $\mathbf{I}_d$, $T \in \mathbb{N}$ denote the total number of diffusion steps, and $\tau \in \{1, 2, \dots, T\}$ index each discrete time step. At time step τ, noise is injected into $\mathbf{x}$ to produce the noisy latent vector $\mathbf{x}_\tau \in \mathbb{R}^d$:

$$\mathbf{x}_\tau = \sqrt{\alpha_\tau}\,\mathbf{x} + \sqrt{1-\alpha_\tau}\,\epsilon,$$

where $\alpha_\tau \in (0, 1]$ is the predefined noise schedule parameter at step τ.

A multi-layer perceptron (MLP) with parameters θ, denoted $\mathrm{MLP}_\theta : \mathbb{R}^d \times \{1, \dots, T\} \to \mathbb{R}^d$, is trained to predict the added noise:

$$\hat{\epsilon} = \mathrm{MLP}_\theta(\mathbf{x}_\tau, \tau),$$

where $\hat{\epsilon} \in \mathbb{R}^d$ is the MLP's noise estimate. Training minimizes the mean-squared error

$$\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x}),\,\epsilon\sim\mathcal{N}(\mathbf{0},\mathbf{I}_d),\,\tau\sim\mathrm{Uniform}(\{1,\dots,T\})}\left\|\epsilon - \hat{\epsilon}\right\|^2.$$

Once $\hat{\epsilon}$ is obtained, the denoised estimate $\hat{\mathbf{x}} \in \mathbb{R}^d$ is recovered as:

$$\hat{\mathbf{x}} = \frac{\mathbf{x}_\tau - \sqrt{1-\alpha_\tau}\,\hat{\epsilon}}{\sqrt{\alpha_\tau}}.$$

The iterative denoising over $\tau = 1, \dots, T$ ensures robustness against perturbations, stabilizing the learning process and reducing overfitting. The noise schedule $\{\alpha_\tau\}_{\tau=1}^{T}$ can follow any user-defined sequence, such as a geometric progression as is prescribed in this work.

Transformer-based processing The Transformer-based processing employs multi-head self-attention to model dependencies within input sequences. Given an input sequence $\mathbf{X} \in \mathbb{R}^{n\times d}$, where n is the sequence length and d is the feature dimension, the self-attention mechanism computes attention scores $\mathbf{A}$ as:

$$\mathbf{A} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right), \tag{13}$$

where $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are query, key, and value matrices derived from $\mathbf{X}$, and d_k is the dimensionality of the queries and keys. The output of the attention mechanism is:

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{A}\mathbf{V}. \tag{14}$$

Positional encodings $\mathbf{P}$ are added to retain spatial relationships:

$$\mathbf{X}_{\text{encoded}} = \mathbf{X} + \mathbf{P}, \tag{15}$$

where $\mathbf{P}$ is a sinusoidal positional encoding matrix. The encoded sequence is passed through L stacked encoder layers, each comprising:

$$\mathbf{X}^{(l+1)} = \mathrm{LayerNorm}\left(\mathbf{X}^{(l)} + \mathrm{MultiHeadAttention}\left(\mathbf{X}^{(l)}\right)\right), \tag{16}$$

$$\mathbf{X}^{(l+1)} = \mathrm{LayerNorm}\left(\mathbf{X}^{(l+1)} + \mathrm{FeedForward}\left(\mathbf{X}^{(l+1)}\right)\right), \tag{17}$$

where l denotes the layer index, and FeedForward refers to a two-layer fully connected network with appropriate activation. This architecture attempts to efficiently capture hierarchical features across sequences. Uncertainty is addressed through noise scheduling in the transformer-diffusion model, where structured noise is added during training and predicted through denoising. The forward and inverse diffusion enhance robustness and regularization in design optimization.

5 Examples and discussion

5.1 Case study 1: PINN and FNN simultaneous optimization for 6 load cases with fixed roller locations

In this subsection, a basic case study is conducted on a beam under 6 simultaneous point load cases. The total length of the beam is a 200 m and roller positions are fixed. The data StructLite and the FNN optimizer are utilized, and results are compared with the PINN optimizer.

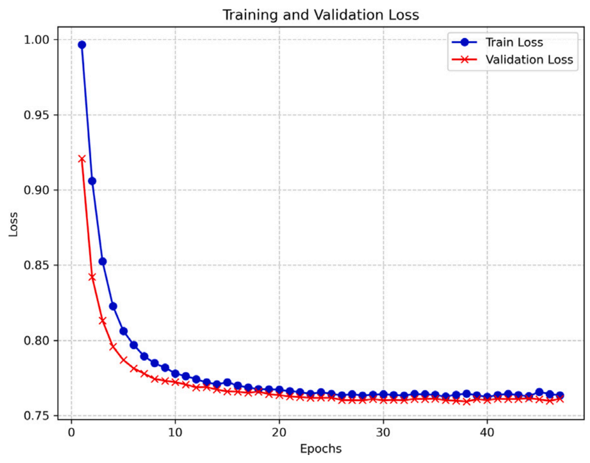


Fig. 4. Training and validation losses plotted at each epoch for the case study 1.

The total number of load cases is defined as 6 in the first line. Thereafter, the number of elements in the discretization (and length of the I array), constraints, hyperparameters for the NN, and optimization parameters are input. At present, the defaults will provide a good starting point for optimizing this particular problem. However, user parameters such as the label aggregation level c may be throttled to modulate the degree of conservativeness one would like to consider in their final optimized structure. Thereafter, the code scales automatically with respect to data sizes and the optimizer executes seamlessly with respect to the predefined parameters.

During training, the training and validation loss curves are provided at each epoch as shown in Fig. 4. It is important to note that hyperparameters including the dropout rate, weight decay, initial added noise level σ_0 and the size of the network all influence the susceptibility for the network to underfitting or overfitting. As a rule of thumb it is good practice to adjust these such that the model reaches a balance between the training and validation set.

Training will then complete once the maximum number of epochs has been attained or terminate early via the provided stopping criteria. The best model is automatically saved at each epoch so that the most suitable model is used in prediction. Here, the default patience is 10, meaning that the training will stop if the validation loss has not improved in 10 epochs. This can be reduced if a particular design case has a consistently low slope after the initial learning is complete (i.e. after the initial knee point). Following, the R^2 values are computed from all validation data, comparing true validation data against all predictions from validation data. In general, as the number of input data (e.g. load cases) increases, so does the difficulty in solving the optimization problems, so users should expect successively decreased R^2 in direct proportion to the number of cases considered. For simple problems, for example cases < 5, a good goal (based on our empirical trailing) is to exceed an R^2 value greater than 0.8 with the FNN.[2] For this case study, we report an $R^2 = 0.71$ which is deemed satisfactory given the large output space and aggregation of 6 I distributions.

Lastly, the user is able to make predictions of optimized design parameters at the inference and plotting phase. While the default setting predefines the roller locations and randomizes force magnitudes and locations consistent with the training data, these can be set by the user. For example, if all roller locations, force locations, force magnitudes and beam lengths are randomized, the inputs can be arbitrarily set. It is emphasized again that these values should fit within the space of the training data (current entries are valid for StructLite and StructHeavy).

[2] The transformer-diffusion model is expected to outperform both of these models with R^2 in excess of 0.9 using the StructLite dataset.

After this point, the user may execute the rest of the code to visualize the output, as seen in Fig. 5 for the FNN. In this figure, the distribution of optimized design parameters I is shown with associated color mapping and sizing, along with the location and magnitude of the test forces represented by plotted arrows and the support locations. It is noted that the I is larger near the areas of expected maximum moment (above the interior supports and midspans) and smaller at the end spans, which is an anticipated result.

To conclude this section, we will compare the FNN results with the PINN optimizer which provided additional mechanics information. In the PINN evaluation, we report an $R^2 = 0.77$, which is a noteworthy improvement over the FNN and is owed to the NN learning the physics of the underlying design problem and resulting regularization. Next, we plot the optimized parameter space for the same randomized testing scenario as shown in Fig. 6. Here, we additionally plot the predicted displacements and rotations in the bottom two subplots respectively. We observe a general expected consistency with the problem mechanics, for example 0 displacements at the rollers and consistent rotations at the rollers. We emphasize that these predicted displacements and rotations are predicted from the aggregation of all output training data and are *not* a summation or combination of all load cases in an additive sense. However, we do observe some displacement artifacts near the rollers and pin at the left hand side. These are owed to their relatively lower contribution to the overall loss function, with respect to the larger displacements at midspan. In future iterations of this code, weighted loss could be incorporated to mitigate these artifacts.

5.2 Case study 2: Transformer-diffusion simultaneous optimization for arbitrary load cases, spans and support conditions

In this case study we aim to predict optimized I using maximally-randomized data – i.e. by randomizing roller number, locations, span widths and force locations/magnitudes in the training data with $c = 1.0$. For this, we set a maximum allowable structural length of 100 m and randomize the number of rollers and number of applied point loads between 1 and 8 and sample 500k data points. Given the ill-posed nature and high dimensionality of this inverse problem, we select the transformer-diffusion model as our optimizer and increase the number of neurons in the feedforward layers to 512, attention heads to 24 and transformer blocks to 4.

After training, we test our model using 4 different trial support cases and 6 randomized force/magnitudes cases, as reported in Fig. 7. We consider a structure that is clamped at the origin and has increasing numbers of supports, increasing from the top figure to the bottom. As was discussed for the previous case study, we consistently see higher I magnitudes at areas where the moment is expected to be elevated while the reverse is also observed (albeit, predictions in this case study are smoother due to the highly ill-posed learning problem presented here despite an $R^2 > 0.92$). These observations support the feasibility of the diffusion-transformer model for predicting optimized structures subject to arbitrary support and loading conditions when data is randomized. This is a desired feature as we aim to train a model with as little bias towards a given suite of support or loading conditions as possible. Lastly, this supports the potential for generalization of the transformer-diffusion model for handling more complex structures, such as 2D/3D frames.

5.3 Computational considerations

For the previous cases studies, a local machine with an NVIDIA RTX 4080, Core i9-14900K CPU and 64 Gb DDR6 RAM was utilized. For these problems, this is more than sufficient to quickly (less than 8 s per epoch for a total of 200 or less epochs while employing the Transformer-Diffusion Model at default settings) train the models. Upon further testing, it was found that similar spec laptops with an NVIDIA A2000 or RTX 4060 were also adequate, albeit at approximately half

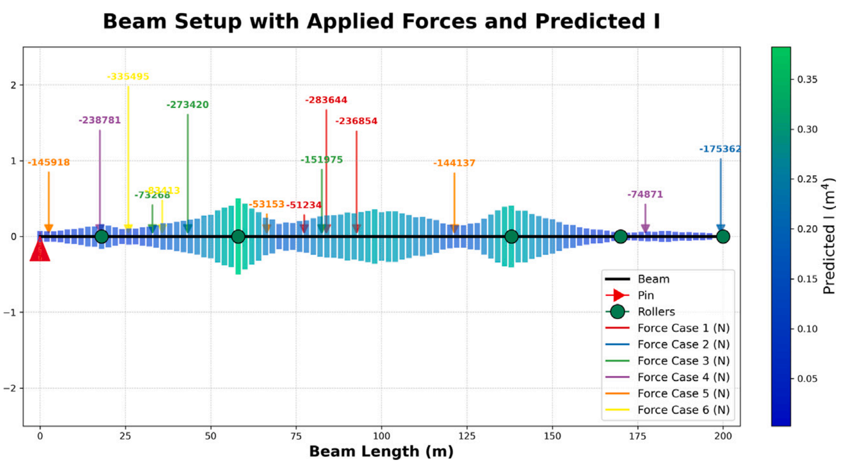


Fig. 5. Optimized section properties predicted by the FNN optimizer including the locations of test forces, force magnitudes and roller locations.

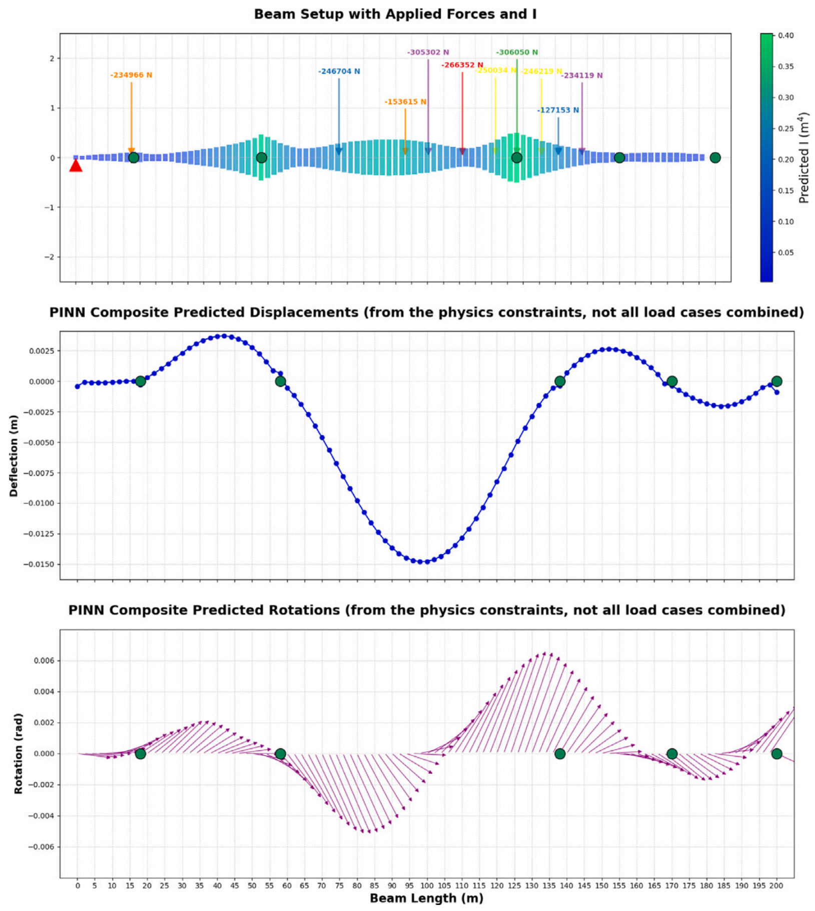


Fig. 6. Top: Optimized section properties predicted by the PINN optimizer including the locations of test forces, force magnitudes and roller locations; Middle: Predicted displacement field; Bottom: Predicted rotations where arrow directions indicate rotation angle and magnitude. Note the beam lines are removed for better visualization of displacements and rotations.

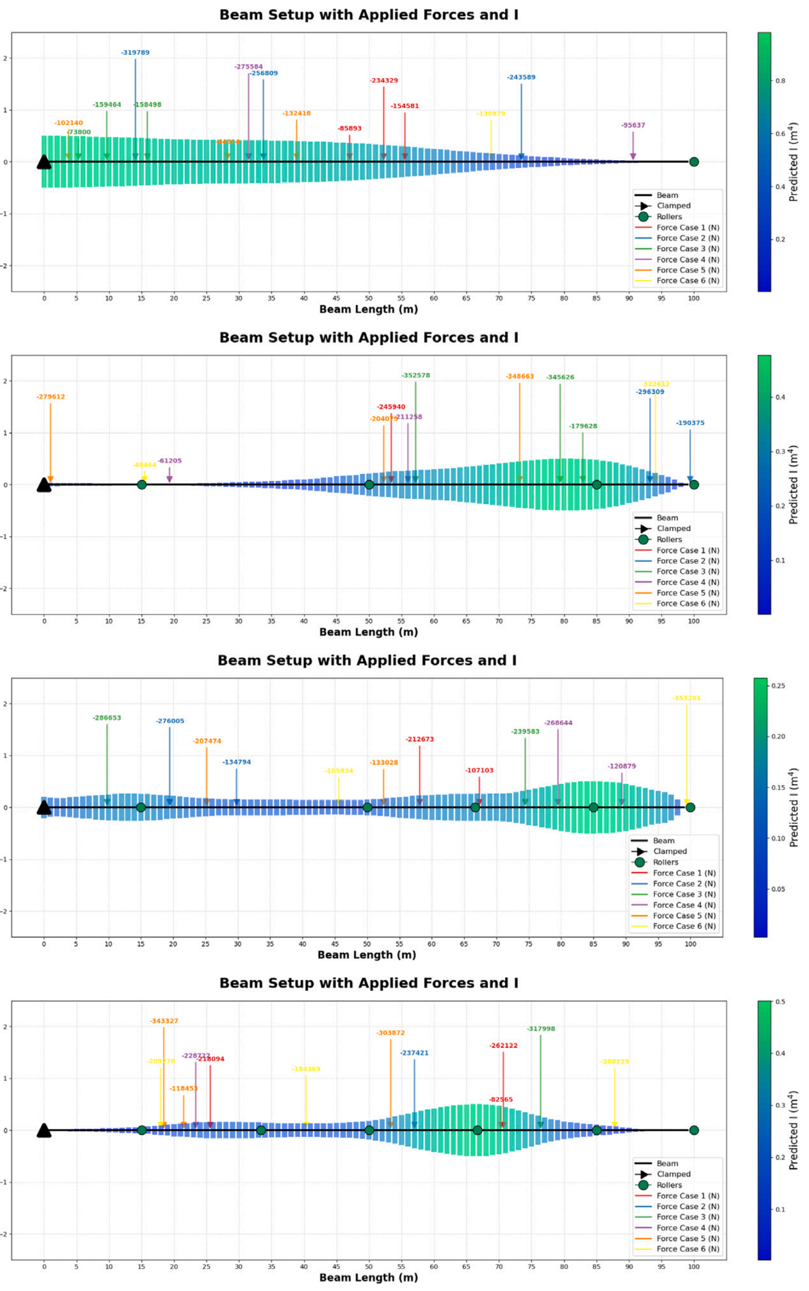


Fig. 7. Transformer-Diffusion model predictions of optimized I for various random load and support conditions. Training was carried out using randomized 500k data (including roller number, locations, span widths and force locations/magnitudes). Test case predictions include a *clamped* left end and various support conditions increasing from the top row to the bottom row.

the speed per epoch. In general, for the case studies and more broadly, it was found that the major computing bottlenecks include (a) the number of CPUs or GPUs available for generating training data and (b) the available VRAM when training models with StructDataHeavy. For example, when trialing the multicore data generator with the local machine previously mentioned, generating 2,000,000 data samples took over 24 h (hence why this was done on a remote HPC). In the case of limited VRAM (RTX 4060 machine), this may be mitigated by using smaller batch sizes (while accepting higher noise in the gradients during training) and overall increased training time. Lastly, for machines with limited RAM, it is advisable to load massive data in chunks rather than the default setting (noting that StructDataHeavy is > 24 Gb). From the model perspective, the data-driven models (i.e., FNNs, Transformer-Diffusion models) are more suitable for rapid prediction and large-scale exploratory optimization, while PINNs are better suited for scenarios where physics consistency and interpretability are critical — such as design verification or training with limited data.

5.4 Discussion

5.4.1 Potential for extended capabilities: Simultaneous load and support optimization

`OpenPyStruct`'s current `v1` framework handles multiple load and support conditions (although the default configuration is meant to have consistent supports and randomized force locations), enabling multi-load case conceptual structural optimization. Building upon this foundation, there exists significant potential to extend the toolkit to perform simultaneous optimization of both loading conditions and support configurations. This dual optimization approach can lead to the discovery of more resilient and efficient structural designs by considering the interplay between applied forces and support placements holistically.

By integrating advanced multi-objective optimization algorithms, OpenPyStruct could balance conflicting objectives such as minimizing material usage while maximizing structural integrity under diverse loading scenarios. Leveraging machine learning models, particularly those capable of capturing complex dependencies like Transformer-Diffusion models, the framework can predict optimized configurations that simultaneously address multiple loads and supports. Additionally, incorporating reinforcement learning techniques could enable the system to iteratively learn and adapt support strategies in response to, for example, dynamic loading conditions. Such an extension could streamline design workflows by reducing the need for sequential optimizations and facilitate the exploration of novel structural configurations that might be overlooked in traditional optimization paradigms.

Future developments could include user-friendly interfaces for defining complex load and support interactions, real-time optimization feedback, and enhanced visualization tools – potentially integrated with CAD – to intuitively present the interplay between optimized loads and supports. Given that OpenPyStruct is developed in CPython, the authors aim to explore its integration into CAD environments that incorporate visual programming interfaces, such as Rhinoceros & Grasshopper, Revit & Dynamo as well as Blender & Sverchok. To facilitate this, OpenPyStruct could be packaged as a core library that is referenced in each of the visual programming interfaces and accessed via standardized API calls. Such an integration would expose OpenPyStruct with both the design community and practitioners, which in turn could contribute to the project with real-world test cases.

While OpenPyStruct, in principle, is ready to be used as an aid for teaching, we currently forego communities without at least an intermediate set of Python skills. Integrating with the user interfaces discussed above would allow OpenPyStruct to be used in more architectural communities. Such advancements would help position OpenPyStruct as an integrable tool capable of tackling a suite of practical structural optimization challenges.

5.4.2 Potential for extended capabilities: Integration of novel standardization

Standardization plays a pivotal role in ensuring consistency, interoperability, and scalability within structural optimization frameworks. OpenPyStruct stands to benefit from the integration of novel standardization protocols, which can enhance its robustness and facilitate broader adoption across diverse design applications. One avenue for novel standardization is the adoption of industry-recognized data schemes for structural properties, load conditions, and support configurations. Implementing standardized input and output formats would simplify data exchange between OpenPyStruct and other engineering software, enabling seamless integration into existing design pipelines. Additionally, embracing standardized application programming interfaces (APIs) and modular architectures can promote extensibility, allowing users to incorporate custom modules or integrate with other machine learning frameworks without significant overhead.

Furthermore, integrating standards related to model validation and benchmarking can enhance the credibility and reliability of OpenPyStruct's optimization outcomes. Establishing standardized test cases and performance metrics would facilitate comparative studies, enabling users to assess model performance consistently and identify areas for improvement. Collaborative efforts to define and adhere to such standards may also be used to drive the development of best practices within the open-source structural community.

5.4.3 Potential for extended capabilities: Integration of novel losses

The optimization efficacy of OpenPyStruct is inherently tied to the design and implementation of its loss functions, which guide the model towards desirable structural configurations. Integrating novel loss functions presents a promising avenue to enhance the framework's ability to capture intricate design nuances, enforce multifaceted constraints, and prioritize specific performance criteria. By diversifying the loss landscape, OpenPyStruct can achieve more nuanced and context-aware optimizations, tailored to the unique demands of various engineering scenarios.

One potential innovation is the incorporation of multi-objective loss functions that simultaneously address multiple performance metrics, such as enforcing stability, reducing material usage, and enhancing structural stability. Techniques like Pareto optimization can be employed to balance these competing objectives, enabling the discovery of optimal trade-offs that align with real-world structural engineering priorities. Additionally, incorporating uncertainty-aware loss functions can bolster the framework's resilience to variability in load conditions and material properties, fostering designs that maintain performance under unforeseen circumstances.

Another promising direction is the integration of physics-informed loss components that encapsulate higher-order mechanical principles or emergent phenomena not currently captured by existing loss terms. For instance, losses that penalize excessive vibrations, thermal stresses, or dynamic load responses can enrich the optimization process, leading to more comprehensive and reliable structural designs. Moreover, leveraging adversarial loss functions, inspired by generative adversarial networks (GANs), might enhance the model's ability to generate realistic and robust structural configurations by challenging it to outperform a discriminator network trained to identify suboptimal designs.

Lastly, adaptive loss functions that evolve during the training process based on performance feedback can offer dynamic optimization pathways, possibly enabling OpenPyStruct to refine its focus as the design evolves. Such adaptability can be beneficial in complex optimization landscapes where static loss functions may struggle to navigate local minima or saddle points. By experimenting with and integrating these novel loss paradigms, OpenPyStruct has potential to significantly elevate its optimization capabilities, delivering more sophisticated and contextually appropriate structural solutions.

5.4.4 Potential for extended capabilities: Simplification using standard shapes and inclusion of environmental effects

Simplified structural modeling approaches are often necessary in engineering practice where clarity, speed, and code compliance play critical roles. Rather than optimizing arbitrary or continuous cross-sectional properties, practitioners typically select from a discrete set of standardized sections (e.g., American Wide Flange (W) shapes, British Universal Beams (UB), etc.) that already meet design and manufacturing constraints. In the case of localized ML-based conceptual design of steel members integrating the influence zone concept [5], models used in [1, 2] have demonstrated promise via incorporating discrete members. Indeed, this may simplify the decision space (and ill-posedness of the design problem) and ensure that resulting structures are more readily constructible in some scenarios. Moreover, standardized shapes come with comprehensive tabulated design properties, proven fabrication methods, and guidelines that reduce the risk of design errors. In this regard, hybrid methods incorporated in OpenPyStruct with discrete selection of recognized section shapes have promise for synergizing automated optimization and practical engineering specifications.

In addition to geometric and material properties, continuous structural designs often need to account for environmental effects such as temperature fluctuations and wind loads. Temperature changes, in particular, can lead to deformation of elements that induce significant stress in statically indeterminate systems or systems with constrained supports. Therefore, when training data-driven structural models, introducing features related to temperature ranges or thermal load distributions can improve the fidelity and designability of the resultant design recommendations. It may be beneficial to represent such environmental boundary conditions explicitly in the optimization framework – either in physics constraints or as part of the input features for data-driven models – thereby ensuring that predicted designs include these real-world variations.

Combining standardized shapes and environmental modeling could expand the applicability of computational frameworks proposed here beyond purely idealized scenarios. This integrated approach could allow designers to more reliably and transparently assess design trade-offs involving discrete section selection, temperature-induced effects, and other environmental considerations. Indeed, future versions of `OpenPyStruct` should likely incorporate industry-standard steel or concrete shape libraries alongside data describing average regional temperature swings, enhancing the realism of automated ML-based design.

6 Conclusion and future work

6.1 Summary of key contributions

In our contribution, we introduced `OpenPyStruct` (Toolkit: https://github.com/dsmyl6/OpenPyStruct.git, Repository: https://zenodo.org/records/14742438), an open-source toolkit designed to bridge physics-based structural optimization with conventional and modern machine learning techniques. `OpenPyStruct` integrates `OpenSeesPy` for precise structural simulations and `PyTorch` for efficient gradient-based optimization, facilitating the optimization of structural properties under diverse loading and support conditions. To the best of our knowledge, OpenPyStruct is the first open-source toolkit designed to optimize structural properties under multiple simultaneous load cases using integrated physics-based and machine learning approaches. Key contributions of this study include:

1. **Comprehensive Framework**: Development of a modular architecture that supports single and multi-load case optimizations, enabling versatile applications in structural engineering.
2. **Flexible Data Generation**: Integration of single-core, multi-core, and GPU-based pipelines that enable efficient generation of datasets at varying quantity to support diverse modeling and training requirements.
3. **Modern Machine Learning Integration**: Incorporation of various ML models, including Feedforward Neural Networks, Physics-Informed Neural Networks, and Transformer-Diffusion architectures, to enhance predictive capabilities and optimization performance.
4. **Flexible Optimization Tools**: The suite of customizable loss functions and design parameters, empowering users to tailor the optimization process to specific engineering requirements and constraints.

6.2 Future directions for `OpenPyStruct` and machine learning structural optimization

The current version of OpenPyStruct focuses on continuous beam systems and may face scalability and generalization challenges in high-dimensional or ill-posed settings. To address this, we have outlined several future research directions as follows.

- **Enhanced Multi-Objective Optimization**: Expanding the framework to support simultaneous optimization of multiple (non-physics based) objectives, such as cost, weight, and structural resilience, will allow for more comprehensive and balanced design solutions.
- **Integration of Advanced Standardization Protocols**: Adopting and contributing to emerging industry standards for data representation and model interoperability can improve consistency, facilitate collaboration, and promote broader adoption of `OpenPyStruct` across different open-source engineering applications.
- **Development of Novel Loss Functions**: Exploring and incorporating innovative loss functions that capture higher-order mechanical principles or specific engineering constraints can refine optimization outcomes, ensuring designs are not only efficient but also robust and reliable under various conditions.
- **Incorporation of Uncertainty Quantification**: Integrating techniques to account for uncertainties in material properties, load conditions, and environmental factors can enhance the reliability and safety of optimized structural designs.
- **Expansion to 2D and 3D**: As the availability of powerful ML approaches increases, even beyond the robust transformer-diffusion model, the ability for future `OpenPyStruct` ML design optimizers to handle evermore diverse and massive data sets (and their resultant increasing degree of ill posedness) will also increase. As this field progresses, we aim to expand the toolkit to handle progressively more difficult conceptual design tasks.
- **Simplification Using Standard Shapes and Inclusion of Environmental Effects:** Incorporating standard shapes (e.g., W sections) and environmental effects (e.g., temperature) into `OpenPyStruct` may enhavce practicality, realism, and compliance in many scenarios, paving the way for more robust and constructible designs.
- **User-Friendly Interfaces**: Improving the accessibility of `OpenPyStruct` through intuitive user interfaces will lower the barrier to entry, enabling a wider range of users to engage the toolkit effectively.
- **Collaborative and Community-Driven Development**: Encouraging community contributions and fostering collaborative development efforts can accelerate the evolution of `OpenPyStruct`, ensuring it remains at the cutting edge of structural optimization technology.

By pursuing these directions, `OpenPyStruct` aims to continuously evolve through community contributions, where we mean to meaningfully contribute to addressing some of the complex challenges of structural optimization and design. Thus, we aim to refine the toolkit based on user feedback and emerging research/industry trends to support the structural engineering community with an evermore robust, efficient, and innovative optimization toolkit.

CRediT authorship contribution statement

Danny Smyl: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Bozhou Zhuang:** Writing – review & editing, Writing – original draft, Formal analysis, Conceptualization, Methodology, Investigation. **Sam Rigby:** Writing – review & editing, Validation. **Edvard Bruun:** Writing – review & editing, Validation. **Brandon Jones:** Writing – review & editing, Validation. **Patrick Kastner:** Writing – review & editing, Validation. **Iris Tien:** Writing – review & editing, Validation. **Adrien Gallet:** Validation, Software, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Statements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Appendix A. Summary of variables and functions

This section provides a detailed summary of the key variables and functions employed `OpenPyStruct` as follows: (see Tables 4 and 5).

Table 4
Summary of key non-visualization variables and functions used in the beam optimization algorithm.

Variables and Descriptions	
E	Young's modulus
G	Shear modulus
L	Total length of the beam
N	Number of nodes in the beam model
M	Number of elements in the beam model
I_0	Initial guess for the second moments of area for each beam element
α_{moment}	Weighting coefficient controlling the bending loss term in the total objective
α_{shear}	Weighting coefficient controlling the shear loss term in the total objective
I_e	Second moment of area for the eth beam element, the primary design variable
M_e	Bending moment at the eth beam element (defined at end nodes)
V_e	Shear force at the eth beam element (defined at end nodes)
A_e	Cross-sectional area of the eth beam element
$\mathcal{L}_{\text{primary}}$	Primary loss function minimizing the sum of second moments of area I_e
$\mathcal{L}_{\text{bending}}$	Bending loss function penalizing bending energy
$\mathcal{L}_{\text{shear}}$	Shear loss function penalizing shear energy
$\mathcal{L}_{\text{total}}$	Total loss function combining primary, bending, and shear loss terms
Functions and Descriptions	
`BeamOptimization()`	Main function coordinating the beam optimization procedure, from initialization to output
`ops.analyze(1)`	Executes a static analysis on the beam model for one iteration using `OpenSeesPy`
`total_loss.backward()`	Computes the gradients of the total loss function with respect to the design parameters
`optimizer.step()`	Updates the design parameters (second moments of area I_e) based on the gradients
`scheduler.step()`	Adjusts the learning rate according to the predefined schedule to improve convergence

Table 5
Summary of key non-visualization variables, mathematics, and functions from the modules.

Variables and Descriptions	
`n_cases`	Number of sub-cases per sample
`nelem` (d_{elem})	Number of elements in the model
`hidden_units`	Number of hidden units per perceptron layer
`dropout_rate`	Dropout rate for regularization
`num_epochs`	Maximum number of training epochs
`batch_size` (B)	Batch size for training data
`learning_rate` l_r	Learning rate for optimization
`weight_decay` (λ)	L_2 regularization coefficient for the optimizer
`initial_alpha` (α_0)	Initial weighting coefficient for loss terms
`box_constraint_coeff` (β)	Coefficient for penalty constraints on predictions
`device`	Hardware device for computation (CPU or GPU)
`diffusion_T` (T)	Number of steps in the diffusion process
`dim_feedforward`	Dimension of feedforward layers in Transformer
`num_heads`	Number of attention heads in Transformer layers
`sigma_0` (σ_0)	Initial noise level for inputs
`gamma_noise` (γ_{noise})	Decay rate for noise during training
`gamma` (γ)	Learning rate decay factor for the scheduler
`c`	Scaling factor for unifying across the output space
`deflection_dim` (d_δ)	Dimension of deflections in the output
`rotation_dim` (d_θ)	Dimension of rotations in the output
`output_dim` (d_{output})	Total output dimension: $d_{\text{output}} = d_{\text{elem}} + d_\delta + d_\theta$
Functions and Descriptions	
`pad_sequences()`	Pads sequences to a uniform length for batch processing
`unify_label_with_c()`	Aggregates labels via $\mu + c \cdot \sigma$, where μ is the mean and σ is the std
`fit_transform_3d()`	Applies scaling to 3D arrays using a StandardScaler
`merge_sub_features()`	Concatenates multiple feature arrays into a single array
`scale_user_inputs()`	Prepares and scales user-provided input features for modeling
`ResidualBlock`	Implements residual connections with normalization and convolutions
`FNNWithResidual`	Feedforward neural network with residual blocks and normalization
`TrainableL1L2Loss`	Custom loss combining L_1 and L_2 terms weighted by α
`CompositeLoss`	Combines loss terms for I, deflections, and rotations
`DiffusionModule`	Implements noise addition and removal using a perception
`PositionalEncoding`	Adds positional encodings
`ModelOnePassTransformerWithDiffusion`	Transformer-based model incorporating a diffusion module.

Appendix B. Classical optimization pseudocode

See Algorithm 2.

Algorithm 2: Beam Optimization Algorithm using `OpenSeesPy` and `PyTorch`

```
Data: Material properties $E$, $G$; cross-sectional area $A$; beam length $L$;
      number of nodes $N$; number of elements $M$; initial guess for
      $I_0$; optimization parameters (learning rate, momentum, weight
      decay) Result: Optimized second moments of area $I_{\text{opt}}$
1  Initialization
2      Set initial guess $I_0$ for each element; Initialize learning rate,
       momentum, and regularization parameters
3  Beam Structure Setup
4      Compute node positions based on $N$ and $L$
5      Define roller supports and point forces
6      Specify boundary conditions (pin and roller supports)
7      Apply uniformly distributed load (UDL) and point forces
8  OpenSees Model Setup
9      for $e = 1$ to $M$ do
10         Define beam nodes and apply boundary conditions
11         Define element properties with updated second moments of
           area $I_e$
12         Configure static analysis in OpenSeesPy
13 Loss Function Definition
14     for $e = 1$ to $end$ do
15         Compute bending energy loss:
           $\mathcal{L}_{\text{bending}} = \sum_{e=1}^{M} \frac{M_e^2}{2\,E\,I_e}$
16         Compute shear energy loss:
           $\mathcal{L}_{\text{shear}} = \sum_{e=1}^{M} \frac{V_e^2}{G\,A_e}$
17         Compute primary loss:
           $\mathcal{L}_{\text{primary}} = \sum_{e=1}^{M} I_e$
18     Compute total loss:
       $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{primary}} + \alpha_{\text{moment}}\,\mathcal{L}_{\text{bending}} + \alpha_{\text{shear}}\,\mathcal{L}_{\text{shear}}$
19 Optimization Loop
20     for $epoch = 1$ to $max\_epochs$ do
21         optimizer.zero_grad()
22         Rebuild OpenSees model with current $I_{\text{tensor}}$
23         ops.analyze(1)
24         total_loss.backward()
25         optimizer.step()
26         scheduler.step()
27         if $total\_loss \le best\_loss - tolerance$ then
28             best_loss = total_loss
29             no_improvement_epochs = 0
30         else
31             if $no\_improvement\_epochs \ge patience$ then
32                 break
33             no_improvement_epochs = no_improvement_epochs + 1
34 return optimized second moments of area $I_{\text{opt}}$
```

Appendix C. ML optimization pseudocode (Transformer-diffusion example)

See Algorithm 3.

Algorithm 3: Machine Learned Multi Load Case Optimizer

```
Data: Input features: Roller locations, Force positions, Force values,
      Node positions; Ground truth: second moments of area
Result: Predicted second moments of area for structural optimization
1  Initialize
2      Define hyperparameters: $n_{\text{cases}}, n_{\text{elem}}$, dropout rate, learning rate;
3      Configure diffusion parameters: $T, \alpha_\tau$;
4      Initialize device (CPU or GPU);
5  Input Preprocessing
6      Load and normalize input data;
7      Pad sequences to align dimensions;
8      Aggregate labels using mean and standard deviation with
         coefficient $c$;
9      Split data into training and validation sets;
10     Standardize input features and targets;
11 Model Definition
12     Define Diffusion Module for noise injection and removal:
         $\mathbf{x}_t = \sqrt{\alpha_\tau}\,\mathbf{x} + \sqrt{1-\alpha_\tau}\,\epsilon$
         Predict noise $\epsilon$ and recover signal:
         $\hat{\mathbf{x}} = \frac{\mathbf{x}_t - \sqrt{1-\alpha_\tau}\,\hat{\epsilon}}{\sqrt{\alpha_\tau}}$
         Implement Transformer Encoder with positional encoding:
         $\mathbf{A} = \text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right), \quad \mathbf{X}_{\text{encoded}} = \mathbf{X} + \mathbf{P}$
         Define fully connected network for final prediction;
13 Training
14     Use Adam optimizer with weight decay and learning rate
         scheduling:
         $l_s = l_0 \cdot \gamma^s$
         Train for $n_{\text{epochs}}$ using a custom loss function combining $L_1$ and
         $L_2$ losses, and penalties:
         $\mathcal{L} = \alpha \cdot L_1(y_i, \hat{y}_i) + (1-\alpha) \cdot L_2(y_i, \hat{y}_i) + \beta \cdot P(\hat{y}_i) + \lambda \cdot L_2(W)$
         Apply early stopping based on validation loss;
15 Evaluation
16     Load the best-performing model;
17     Evaluate predictions on the validation set;
18     Compute performance metrics (e.g., $R^2$ score);
19     Visualize predictions alongside ground truth second moments of
         area;
```

Data availability

Code and data are openly available at the respective locations: `OpenPyStruct` (Toolkit URL: https://github.com/dsmyl6/OpenPyStruct.git and Repository URL: https://zenodo.org/records/14742438.

References

[1] Gallet A. Machine learning for structural design models from the inverse problem perspective. (Ph.D. thesis), University of Sheffield; 2024.

[2] Gallet A, Rigby S, Tallman T, Kong X, Hajirasouliha I, Liew A, Liu D, Chen L, Hauptmann A, Smyl D. Structural engineering from an inverse problems perspective. Proc R Soc A 2022;478(2257):20210526.

[3] Turco E. Tools for the numerical solution of inverse problems in structural mechanics: review and research perspectives. Eur J Environ Civ Eng 2017;21(5):509–54.

[4] Mueller JL, Siltanen S. Linear and Nonlinear Inverse Problems with Practical Applications. USA: Society for Industrial and Applied Mathematics; 2012.

[5] Gallet A, Liew A, Hajirasouliha I, Smyl D. Influence zones of continuous beam systems. In: Structures, vol. 68, Elsevier; 2024, 107069.

[6] Almeida F, Awruch A. Design optimization of composite laminated structures using genetic algorithms and finite element analysis. Compos Struct 2009;88(3):443–54.

[7] Toropov VV, Filatov A, Polynkin A. Multiparameter structural optimization using FEM and multipoint explicit approximations. Struct Optim 1993;6:7–14.

[8] Kudela J, Matousek R. Recent advances and applications of surrogate models for finite element method computations: a review. Soft Comput 2022;26(24):13709–33.

[9] Fairclough HE, Gilbert M. Layout optimization of long-span structures subject to self-weight and multiple load-cases. Struct Multidiscip Optim 2022;65(7):197.

[10] Thai H-T. Machine learning for structural engineering: A state-of-the-art review. In: Structures, vol. 38, Elsevier; 2022, p. 448–91.

[11] Wang X, Zhuang B, Smyl D, Zhou H, Naser M. Machine learning for design, optimization and assessment of steel-concrete composite structures: A review. Eng Struct 2025.

[12] James KA, Hansen JS, Martins JR. Structural topology optimization for multiple load cases using a dynamic aggregation technique. Eng Optim 2009;41(12):1103–18.

[13] Zhou Y, Zhan H, Zhang W, Zhu J, Bai J, Wang Q, Gu Y. A new data-driven topology optimization framework for structural optimization. Comput Struct 2020;239:106310.

[14] Zheng L, Kumar S, Kochmann DM. Data-driven topology optimization of spinodoid metamaterials with seamlessly tunable anisotropy. Comput Methods Appl Mech Engrg 2021;383:113894.

[15] Hoang V-N, Nguyen N-L, Tran DQ, Vu Q-V, Nguyen-Xuan H. Data-driven geometry-based topology optimization. Struct Multidiscip Optim 2022;65(2):69.

[16] Jeong H, Bai J, Batuwatta-Gamage CP, Rathnayaka C, Zhou Y, Gu Y. A physics-informed neural network-based topology optimization (PINNTO) framework for structural optimization. Eng Struct 2023;278:115484.

[17] Karniadakis GE, Kevrekidis IG, Lu L, Perdikaris P, Wang S, Yang L. Physics-informed machine learning. Nat Rev Phys 2021;3(6):422–40.

[18] Gallet A, Smyl D. Design of continuous structures using physics informed neural networks. EngrXiv 2023. http://dx.doi.org/10.31224/4242, Preprint. URL https://engrxiv.org/preprint/view/4242.

[19] Zhu M, McKenna F, Scott MH. OpenSeesPy: Python library for the OpenSees finite element framework. SoftwareX 2018;7:6–11.

[20] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. Pytorch: An imperative style, high-performance deep learning library. Adv Neural Inf Process Syst 2019;32.