Learning to Simulate and Design for Structural Engineering

Kai-Hung Chang [1]   **Chin-Yi Cheng** [1]

Abstract

The structural design process for buildings is time-consuming and laborious. To automate this process, structural engineers combine optimization methods with simulation tools to find an optimal design with minimal building mass subject to building regulations. However, structural engineers in practice often avoid optimization and compromise on a suboptimal design for the majority of buildings, due to the large size of the design space, the iterative nature of the optimization methods, and the slow simulation tools. In this work, we formulate the building structures as graphs and create an end-to-end pipeline that can learn to propose the optimal cross-sections of columns and beams by training together with a pre-trained differentiable structural simulator. The performance of the proposed structural designs is comparable to the ones optimized by genetic algorithm (GA), with all the constraints satisfied. The optimal structural design with the reduced the building mass can not only lower the material cost, but also decrease the carbon footprint.

1. Introduction

Structural design of buildings is to design the optimal structures subject to a design objective, such as minimizing material usage for cost and sustainability reasons. The design also has to satisfy a set of rules from established standards known as building codes, for example, a limited displacement under loading and seismic force. However, most structural engineers do not employ optimization in real-world cases for several reasons. First, the design space is large. A classic five-story building has typically over 500 columns and beams as design variables. Moreover, optimization algorithms usually require many iterations, and the evaluation
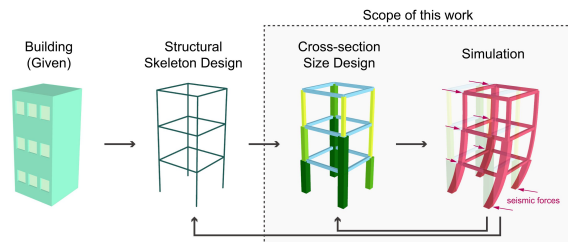

Figure 1. The iterative industrial structural design workflow.

for each optimization iteration takes 2 to 15 minutes to run structural simulation. As a result, a single optimization process can take days to converge, which does not fit the need for frequent design changes in a construction project. In practice, the majority of structural engineers complete structural designs with trial-and-error. After getting simulation results from a proposed design, structural engineers have to revise the design iteratively until all the building codes are satisfied. Since finding a valid design is already challenging and laborious, the outcome is usually over-designed, which means it satisfies the code and constraints but has poor performance in the design objective.

The main contribution of this paper is proposing an end-to-end solution to automate the structural design process. As visualized in Figure 1, a typical structural design process starts from a given building design, and then the structural engineer will propose a skeleton design, where the locations and connectivities of columns and beams are defined. After proposing the skeleton, the engineer will decide the cross-section size for each bar (column and beam). The engineer will then evaluate the design by running structural simulation and update the skeleton and cross-section sizes iteratively. As a starting point, the scope of this work only focuses on automating the size design as well as the structural simulation process.

We discover that a building skeleton can be naturally represented as a graph and therefore propose two graph neural networks, NeuralSim and NeuralSizer. We train NeuralSizer to assign the optimal cross-section sizes to the given columns and beams and evaluate the size design using the pre-trained neural approximator, NeuralSim, instead of using real structural simulation tool. Taking advantage of the differentiable nature of NeuralSim, the gradient of the
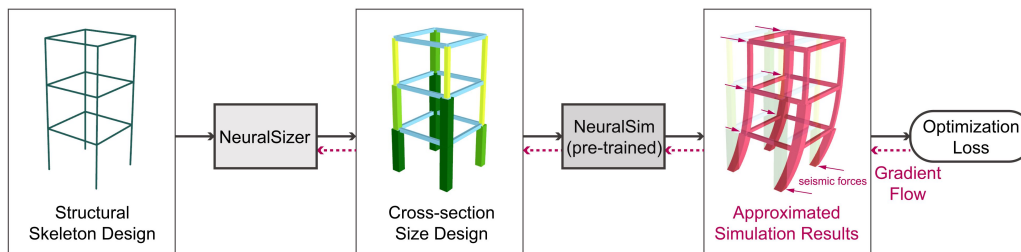
[1]Autodesk Research, San Francisco, California, United States. Correspondence to: Kai-Hung Chang <kai-hung.chang@autodesk.com>.

Figure 2. Our proposed end-to-end learning pipeline for solving the size design optimization problem.

optimization loss can flow through it and update the learning parameters in NeuralSizer. The learning pipeline is illustrated in Figure 2. Our results show that NeuralSizer can produce design comparable to the design generated by genetic algorithm running for 1000 iterations. We also perform experiments including ablation, extrapolation, and user study.

2. Related Work

2.1. Structural Optimization

Most structural engineering research solves building design optimization problems with evolutionary algorithms, such as genetic algorithms (Rajeev & Krishnamoorthy, 1997; Balogh & Vigh, 2012; Imran et al., 2019). These methods evaluate candidate solutions using a fitness function and iteratively evolve the solution population. The computational complexity is high due to the evaluation, especially with structural simulation tools. Latest studies that apply deep learning approaches use vector inputs to encode information of a structural component (Greco, 2018; Torky & Aburawwash, 2018; Cheng et al., 2017), a single structure(Hasançebi & Dumlupınar, 2013), or 2D images to coarsely describe the structural geometry (Tamura et al., 2018). These representations suffer from scalability and expressiveness and thus can hardly be applied to real-world cases.

2.2. Graph Neural Network

Graph neural networks (GNNs) have shown great successes in many learning tasks that require graph data and are applied to many new scientific domains including physics systems (Kipf et al., 2018; Sanchez-Gonzalez et al., 2018; Watters et al., 2017), fluid dynamics(Sanchez-Gonzalez et al., 2020), chemical molecules(Fout et al., 2017; Jin et al., 2018b; Do et al., 2019), and traffic networks (Guo et al., 2019; Cui et al., 2019). For readers who are unfamiliar with GNNs, recent reviews and tutorials exist (Zhou et al., 2018; Wu et al., 2020). Related to structural engineering domain, (Hamrick et al., 2018) uses reinforcement learning to train a graph neural network policy that learns to glue pairs of

blocks to stabilize a block tower under gravity in a physics engine. However, the scale and the complexity of the tower (maximum of 12 blocks) are much smaller than real-world buildings.

Though some papers have applied GNNs to solve combinatorial optimization problems (Bello et al., 2016; Kool et al., 2018; Prates et al., 2019; Li et al., 2018), applying GNN to solve design optimization problem is underexplored. For molecular design optimization problems, both (Jin et al., 2018a) and (You et al., 2018) train GNNs to generate a new molecular graph that optimizes objectives subject to constraints. (Jin et al., 2018a) trains a junction tree variational autoencoder to obtain the latent embedding of a molecular graph and iteratively revise the latent embedding based on a neural prediction model. Constraints on the revision similarity is later enforced over a population of the revised molecular graphs to find the one that has the best predicted property. Similar to (Hamrick et al., 2018), (You et al., 2018) also chooses reinforcement learning approach and trains a graph convolutional policy network which outputs a sequence of actions to maximize the property reward. Compared to these two papers, our approach trains NeuralSizer to directly propose cross-sections, and uses NeuralSim as a differentiable simulator to provide a back-propagable loss.

2.3. Differentiable Approximator for Design

Differentiable approximators are commonly used to model the non-differentiable loss function of interest and to provide gradients in back-propagation. Taking drawing task as an example, StrokeNet (Zheng et al., 2018) and Canvas Drawer (Frans & Cheng, 2018) optimizes the unsupervised reconstruction loss with the aid of a differentiable renderer. The differentiable renderer models the relation between stroke actions and the output drawing. Given a target scene of 3D shapes, (Tian et al., 2019) trains a neural program executer with a pre-trained scene generator, which maps the input code to the corresponding output scene. Outside of image generation domain, (Zhou et al., 2016) takes a model-based approach by training a hand pose generation model followed by a non-linear layer, which acts as a differentiable approximator for forward kinematics computation.

3. Formulation

3.1. Size Design Optimization Problem

In this paper, we target the size design optimization problem in the structural design process. After structure engineers complete the skeleton design, they have to decide the cross-sections for all the columns and beams, which directly impact the performance of the building, including the material usage, stability, constructability, etc. The size design optimization problem is formulated with the following objective and constraints:

- Mass Objective: We want to minimize the total material mass of the building.

- Drift Ratio Constraint: Building code regulations require a building to satisfy a set of constraints to ensure its stability and safety. The drift ratio constraint requires the drift ratios in all stories (visualized and defined in the grey text box in Figure 3) to be less than some limit under lateral seismic loads.

- Variety Constraint: This constraint comes from the constructability requirement which sets a maximum number for different cross-section types used. Using too many different cross-section types leads to higher manufacturing and transportation cost.
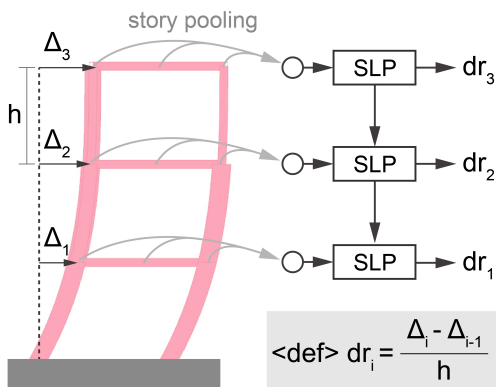

Figure 3. Drift Ratio is defined (in the grey text box) as the ratio of the relative lateral displacement of a story to the story height.

In most cases, using stronger columns or beams improves stability, but leads to a larger total mass. The optimal design should satisfy both constraints and have a minimal building mass. Mathematical equations for the objective and constraints are provided in Section 4.2.2.

3.2. Data Generation

Due to the lack of real structural design data, we synthesize a dataset that contains building skeletons with randomly sampled cross-sections in real-world scale. We also use Autodesk Robot Structural Simulator (RSA), a simulation software widely used in the industry, to compute the structural simulation results for the synthetic dataset. Various loads are considered in the simulation: 1. Self-weight load of the building structure, 2. Surface loads on floor panels which are distributed to the underlying beams, 3. Surface loads on the roof story, and 4. Linear loads at the boundary beams for external walls. NeuralSim is then trained with the building skeletons with paired cross-sections and simulated drift ratio. Please refer to the supplementary material for more details.

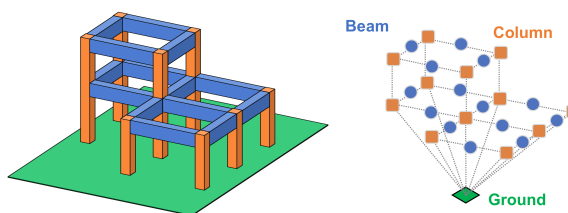## 3.3. Representing Building Structures as Graphs

Figure 4. An example building structure and its structural graph representation.

We represent building geometries as structural graphs. Every bar (column or beam) is represented as a graph node. An edge connects two nodes if the two corresponding bars are joined together. Information of bar i is stored as node feature $v_i = [p_1, p_2, B, T, L]$, where p_1 and p_2 locates the two endpoints of the bar, B indicates if the bar is a beam or a column, T is a one-hot vector representing the cross-section, and L provides auxiliary loading condition information, including 1. if the bar is on the roof story, 2. if the bar is on the boundary, and 3. the surrounding floor penal areas which are multiplied by the per-area loads when computing the surface loads. A pseudo ground node is connected to all first-story columns and the values of its feature vector are all -1. The structural graph of a simple example structure is illustrated in Figure 4. Story level indices of each bar are also saved.

4. Graph Neural Networks

4.1. NeuralSim

4.1.1. NETWORK STRUCTURE

Inspired by GraphNet (Battaglia et al., 2018), NeuralSim also contains three steps: encoder, propagation, and decoder. The encoder first maps the input node features into the embedding space. Then, in the propagation step, each node embedding is iteratively updated based on the aggregated

message, which is computed using the neighbor embeddings as well as the current node embedding. We also experiment a model variant called NeuralSim + PGNN, which extends the message by integrating the position-aware message (You et al., 2019). Though the classic message function can model the forces and reaction forces between bars, position-aware message can further provide global information that helps identify loading conditions, such as if a bar is on the roof story or boundary. After a fixed number of propagation steps, we obtain the final embeddings of all nodes. Given the story indices, story embeddings are computed by pooling over all final embeddings of nodes in the same story.

Instead of using a standard multi-layer perceptron (MLP) as a decoder, we design a *Structured Decoder* (SD) for outputting drift ratios in each story. The illustration of SD is visualized in Figure 3. Starting from the roof story, SD updates each story embedding in the top-down order. The update module takes in the embedding of a story and the updated embedding of the above story to replace the current embedding. The design of SD is intended to mimic the algorithmic structure of the drift ratio's definition. Moreover, given the physics nature that the lower the story, the higher the story drift, drift ratio in the same story will have different distributions if two buildings have different total numbers of stories. After updating the story embeddings, we pass them to two MLP decoders. One decoder outputs the predicted drift ratios while the other classifies if the ground-truth drift ratios exceed the drift ratio limit 0.015 or not.

4.1.2. Loss

Given the two kinds of output from the decoders, our multi-task loss is defined as the sum of an L1 loss for the drift ratio output and a binary cross-entropy loss for the classifier output. Later in the result section, we show that this multi-task loss helps NeuralSim learn a better embedding.

4.1.3. Training

We split the total 4000 data into 3200, 400, and 400 for training, evaluation, and testing purposes. Adam Optimizer is used with learning rate 1e-4 and weight decay 5e-4. Batch size is set to 1 and the number of epoch is 5.

4.2. NeuralSizer

4.2.1. Network Structure

Similar network structure is adopted by NeuralSizer. It has the same encoder and propagation step. SD is not used since there isn't a strong bias in the size design. Instead, a graph embedding is computed to provide the information of the entire structure. In the end, an MLP decoder is used to map the final node embeddings concatenated by the graph embedding to the probability over cross-sections using hard

Gumbel Softmax function (Jang et al., 2016). The hard version returns deterministic samples as one-hot vectors to ensure consistency of NeuralSim inputs, but in back-propagation, the derivatives are calculated as if they are the soft samples.

4.2.2. Loss

Given the objective and constraints of the optimization problem in 3.1, we formulate them as differentiable losses.

- Mass Objective obj: The total mass of a bar is the product of its length, the area of its cross-section, and the material density. The length is derived from the two endpoint locations of the bar and we assume all bars are made of the same steel. The total mass is normalized by the number of bars in the structure.

- Drift Ratio Constraint l_{dr}: This constraint requires the absolute value of all drift ratio dr_i to be less than a limit lim. Therefore, we penalize the mean of how much the drift ratio in each story exceeds the limit:

$$l_{dr} = Mean\{\text{LeakyReLU}(|dr_i| - lim)\} \leq 0$$

- Variety Constraint l_{var}: The variety constraint penalizes the number of cross-section usages more than 6. We compute the usage percentage of each cross-section p and expect the sum of top 6 percentages to be 1. In other words, we can formulate the constraint as below:

$$l_{var} = 1 - \text{SumTop6}(p) = 0$$

- Entropy Constraint l_H: To avoid quick overfitting to some undesired local minimum, we introduce this entropy constraint inspired by maximal entropy reinforcement learning (RL) (Haarnoja et al., 2018). Denote the entropy of NeuralSizer output of each bar as H_i, the maximum entropy over n different cross-sections H_{max}, and a target ratio $\alpha = 0.6$. The entropy constraint is formulated as:

$$l_H = Mean\{H_i\}/H_{max} - \alpha = 0$$

Without this entropy constraint, NeuralSizer converges within 50 iterations in the experiment and always uses one cross-section type for all bars.

The total loss L equals $w_0 obj + w_1 l_{dr} + w_2 l_{var} + w_3 l_H$. Instead of fine-tuning the weights w_i manually, we automate the process by optimizing the dual objective and approximating dual gradient descent(Boyd & Vandenberghe, 2004). This technique has shown successful results in soft actor-critic algorithms (Haarnoja et al., 2018) and reward constrained policy optimization (Achiam et al., 2017). A brief explanation of the method is given in the supplementary material.

4.2.3. TRAINING

In each epoch, a new structural graph is randomly generated and fed to NeuralSizer to get the design output. The output cross-sections are concatenated to the node embeddings in the structural graph, which is passed to NeuralSim for structural simulation. Given the design output and drift ratio output, the total loss is computed. NeuralSizer updates based on the back-propagation gradients once every 5 epochs, and runs 50,000 epochs for training. Though having the best accuracy, NeuralSim + PGNN has a much longer inference time than NeuralSim due to the computation of the position-aware message. Therefore, we use a frozen pre-trained NeuralSim, which also shows high accuracy throughout the training.

5. Experiments

All training and testing run on a Quadro M6000 GPU.

5.1. NeuralSim Results

We compare NeuralSim to four other graph neural network models: GCN (Kipf & Welling, 2016), GIN (Xu et al., 2018), GAT (Veličković et al., 2017), and PGNN (You et al., 2019). Table 1 shows the L1 loss and the relative accuracy of the drift ratio output as well as the classification accuracy of the classifier output. NeuralSim trained with the Structured Decoder (SD) outperforms GCN, GIN, GAT, and PGNN in all three metrics. Moreover, integrating the position-aware message from PGNN helps further improve the performance.

Ablation study results are also included in Table 1. NeuralSim shows better performance when trained with SD since the imposed inductive bias of SD models the increasing drift ratios in lower stories. Moreover, adding the classifier output and the binary cross-entropy loss helps NeuralSim learn a better embedding and thus improves the performance.

A plot of all learning curves is included in the supplementary material. Training takes around 3.5 hours and a forward propagation of NeuralSim for one design takes 6.789 milliseconds in average. Compared to our simulation software Autodesk RSA, which takes 13 seconds, NeuralSim is 1900 times faster with 97.36% accuracy. NeuralSim + PGNN takes 43.92 milliseconds, which is 300 times faster.

In Table 2, NeuralSim demonstrates its generalizability to test data beyond the training scope. We split the test data into 3 buckets based on the numbers of stories: 1~3, 4~7, and 8~10 story. One model is trained with 1~10 story buildings while the other is trained with 4~7 story buildings. Both models are trained with the same amount of training data and tested against each bucket. The results in the first row show that the performance variation across different buckets

is small. The second row demonstrates that NeuralSim also performs well on extrapolated data. The learned message passing module in NeuralSim models the physics of force propagation, which is universal across buildings of different numbers of stories. As a result, NeuralSim shows strong generalizability to extrapolated data.

5.2. NeuralSizer Results

Two scenarios are created. The high safety factor scenario has a drift ratio limit 0.015 while the low safety factor scenario has a drift ratio limit 0.025. For each scenario, two experiments are conducted with different weights on the mass objective. Table 3 summarizes the results of all experiments. All constraints are close to zero, indicating that NeuralSizer learns to satisfy the hard constraints. In particular, the drift ratio constraint, which measures how much the drift ratio exceeds the limit, is negligible compared to the magnitude of drift ratios. Several designs are visualized in Figure 5. Training takes around 2.5 hours and a forward propagation of NeuralSizer for one design takes 10.07 milliseconds in average.

To test the generalizability of NeuralSizer, we pick the high safety factor scenario and objective weight = 10 for an experiment. Again, we compare two models (training with 1~10 story and 4~7 story buildings) and measures the performance of three test buckets (1~3, 4~7, and 8~10 story buildings). The results are summarized in Table 4. Both models satisfy the constraints well and show similar performances across the three different buckets. NueralSizer trained with 4~7 story buildings also shows generalizability to buildings with more and less numbers of stories than the training range.

5.3. User Feedback and User Study

Figure 5 visualizes the design outputs of NeuralSizer (high safety factor + objective weight 10) for various buildings. A cross-section with stronger structural properties is visualized in a darker color and a thicker stick.

To understand the quality from a professional perspective, we show 10 different designs to structural engineers and ask for their feedback. They say that the cross-section choices look natural to them, except that the design is too sophisticated as they usually assign same cross-sections in groups. Interestingly, structural engineers reveal some of the design rules that NeuralSizer reasons and learns and that they think are reasonable. These rules are listed and explained below.

- During an earthquake, columns must support vertical gravity loads while undergoing large lateral displacements. Therefore, to satisfy the drift ratio constraint, NeuralSizer learns to distribute masses on columns more than beams.

Table 1. NeuralSim Performance compared to Other Models

Model	L1 Loss ×1e−4	Relative Accuracy	Classification Accuracy
GCN	16.01	94.86	89.22
GIN	33.85	89.62	84.27
GAT	10.87	96.41	93.35
PGNN	9.39	96.72	94.83
NeuralSim	**7.57**	**97.36**	**95.64**
NeuralSim + PGNN	**5.01**	**98.22**	**96.43**
NeuralSim(no SD)	10.24	96.65	92.71
NeuralSim(only L1 loss)	16.47	95.24	n/a

Table 2. NeuralSim Generalizability Results

Train Data	Test Data	L1 Loss ×1e−4	Relative Accuracy	Classification Accuracy
1∼10 story (Baseline)	1∼3 story	6.09	98.09	95.01
	4∼7 story	6.67	97.55	96.31
	8∼10 story	8.73	96.34	96.79
4∼7 story	1∼3 story	23.97	91.70	85.75
	4∼7 story	15.49	92.93	96.40
	8∼10 story	26.43	91.11	84.33

Table 3. NeuralSizer Results under Different Scenarios

Scenario	Objective Weight	Objective	Constraints	
		Mass Objective	Drift Ratio Constraint	Variety Constraint
High Safety Factor	1	0.870	6.00×1e−7	0.01×1e−8
	10	0.735	1.34×1e−7	1.04×1e−8
Low Safety Factor	1	0.592	6.42×1e−5	1.67×1e−8
	10	0.596	3.32×1e−5	1.78×1e−8

Table 4. NeuralSizer Generalizability Results (High Safety Factor, Objective Weight = 10)

Train Data	Test Data	Objective	Constraints	
		Mass Objective	Drift Ratio Constraint	Variety Constraint
1∼10 story (Baseline)	1∼3 story	0.738	1.62×1e−7	0.80×1e−8
	4∼7 story	0.725	1.28×1e−7	0.97×1e−8
	8∼10 story	0.711	1.69×1e−7	1.06×1e−8
4∼7 story	1∼3 story	0.773	2.96×1e−7	1.30×1e−8
	4∼7 story	0.746	3.50×1e−7	1.25×1e−8
	8∼10 story	0.728	3.68×1e−7	1.01×1e−8

- Since gravity loads are carried down the building structure, the loads accumulate and increase on lower stories. Columns for lower stories have a higher strength requirement than for higher stories. This can also be observed from NeuralSizer's design outputs.

- It is reasonable to have similar patterns in design outputs of different buildings. Given the objective and constraints in the optimization problem, structural engineers will probably design with similar patterns.

One user testing is conducted to compare our results with human design process. Given a building and 30 minutes, a structural engineer tries five design iterations. The first three iterations are able to satisfy all the constraints while the fourth and the fifth cannot. The best out of the three valid designs is used to compare with NeuralSizer results. The result shows NeuralSizer's design output has a better performance. The quantitative user testing results are provided in the supplementary material. The goal of this testing is not to show NeuralSizer can replace professional struc-
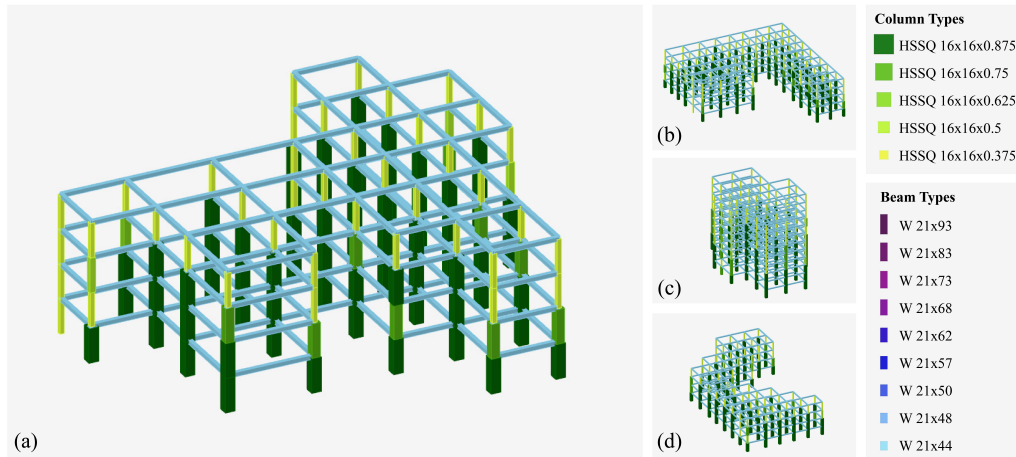
Figure 5. Visualization of NeuralSizer's Design Output

tural engineers, but to show that NeuralSizer can speed up the iterative design process by providing a better starting point.

5.4. Comparison with Genetic Algorithm(GA)

5.4.1. GA SETUP

In this subsection, we compare our method with the genetic algorithm (GA), which is a widely-used algorithm in the structural optimization research. GA contains six steps:

1. Initialization: To start GA, a population of 100 candidate solutions is generated. A candidate has a chromosome which encodes the cross-sections in genes.

2. Evaluation: We compute the score of each candidate as measured by the total loss equation. The same initial weights for individual losses are used as when training NeuralSizer, except they are now fixed.

3. Selection: The top five candidates are directly passed to the next generation without crossover and mutation.

4. Crossover: We use a selection mechanism that random samples two candidates and outputs the better one, called selected candidate. With a crossover rate of 0.9, this selected candidate pairs with another selected candidate to breed a children candidate using crossover operation; otherwise, the selected candidate becomes the children candidate. In total, ninety-five children candidates are generated for the next population.

5. Mutation: The genes in the ninety-five children candidates have a probability of 0.01 to randomly change to a different cross-section. After the mutation process, the ninety-five children candidates and the top five candidates form a new population for the next generation.

Table 5. Time Comparison of GA under Different Setups

Setup	Time	Total Iterations	Time / Iteration
NeuroSizer	10.07 ms	-	-
GA + RSA	24 hr	30	-
$\rightarrow (estimated)$	2 weeks	1000	20.16 mins
GA + NeuralSim	30 mins	1000	0.03 mins

6. Termination: The iteration process is terminated after a certain number of iterations or a limited amount of time.

5.4.2. GA TIME COMPARISON

In the evaluation step, computing scores requires running simulation tools, which is time-consuming. However, by replacing simulation tools with NeuralSim, we can reduce the running time of GA. Table 5 lists running times of GA solving a building with 622 bars under different setups. NeuralSizer outputs a design within 10.07 milliseconds. If we evaluate GA candidates with Autodesk RSA (GA + RSA), we can run 30 iterations in 24 hours. By estimation, it will take up to 2 weeks to finish 1000 iterations. However, if NeuralSim is used for evaluation(GA + NeuralSim), it only takes around 30 minutes to complete 1000 iterations.

5.4.3. USING NEURALSIZER OUTPUTS AS GA SEEDS

We first run GA + NeuralSim with random seeds for 1000 iterations as a baseline. Due to the stochastic nature of GA, We execute 10 runs and save the result that has the minimal total loss. Next, we use the NeuralSizer output as GA seeds. Since NeuralSizer outputs the probability of cross-sections, there are two seeding strategies: best seed

and sampled seeds. Best seed finds the best design output based on the highest probability and populates it to the initial population. On the other hand, sampled seeds sample from the probability to generate different seeds as the initial population. We also run GA + RSA runs for 24 hours each with random seeds and sampled seeds.

Figure 6 plots the performance curves of GA using different seeding approaches. All curves of GA using NeuralSizer seeds (orange, green, and purple curves) show lower losses at the beginning of GA. Moreover, after running the same number of iterations, curves starting with NeuralSizer seeds end up in better solutions than those starting with random seeds. However, the difference between using best seed and sampled seeds is little.

To quantitatively measure the performance across different designs, we run GA with NeuralSizer using sampled seeds and random seeds (once for each) over 20 different designs. Three metrics are defined and illustrated in Figure 7.

For GA using random seeds, we denote the starting and ending losses as $(rand\_start, rand\_end)$. Likewise, for GA using NeuralSizer sampled seeds, they are denoted as $(sizer\_start, sizer\_end)$.

1. $M_1 = \dfrac{(rand\_start - sizer\_start)}{(rand\_start - rand\_end)}$
   The metric measures the percentage of improvement gained by initializing GA with NeuralSizer sampled seeds. If M_1 is larger than 100%, it means that the best performance of the NeuralSizer sampled seeds at iteration 0 beats the best performance of GA with random seeds at iteration 1000.

2. $M_2 = \dfrac{(rand\_end - sizer\_end)}{rand\_end}$
   The metric compared the best performances between GA with NeuralSizer sampled seeds and random seeds, both at 1000 iterations. A positive M_2 indicates that the final performance of GA with NeuralSizer sampled seeds is better.

3. M_3 is the first iteration when $sizer\_start$ is less than $rand\_end$. Note that $M_3 = 0$ whenever $M_1 > 100\%$.

The quantitative results are summarized in Table 6. From M_1 results, we can see that some performance of Neural-Sizer seeds are almost the same as the best performance of GA with random seeds at iteration 1000, and some are even better. This proves that NeuralSizer solves the size design optimization problem and that the results are comparable to GA. M_2 results show the capability of further optimizing the NeuralSizer output to obtain solutions which are better than optimizing with random seed after the same amount of iteration. Lastly, the maximum of the six M_3 values in Table

6 is 128. Thus, compared to the total 1000 iterations, using NeuralSizer seeds provide a speedup by at least 8 times.

6. Conclusion

In this paper, we propose an end-to-end learning pipeline to solve the size design optimization problem. Trained as a neural approximator for structural simulation, NeuralSim shows $\sim 97\%$ accuracy and is 1900 times faster than simulation tools. As a consequence, it can not only provide gradients to upstream models due to the differentiable nature, but also be used to provide quick evaluation in evolutionary solvers or instant feedback after every human design decision. NeuralSizer is trained to output optimal cross-sections subject to the optimization objective and constraints. The design outputs of NeuralSizer satisfy the constraint and are comparable to or sometimes better than the optimal solution of the genetic algorithm. Moreover, they can be used as initial seeds for the genetic algorithm, which speeds up the convergence and further optimizes the design. Last but not least, the design outputs demonstrate reasonable design rules and thus can also be used as the initial design for structural engineers to save design iterations.

The limitations and future work can be defined in three categories: first, the skeleton design phase in Figure 1 can be included into this pipeline and defined as a graph generation problem. Second, except from columns and beams, our graph representation does not cover all the structural components, such as walls and panels. Handling the heterogeneous attributes across the components is worth investigating. Lastly, although the general structural constraints can be applied to any buildings in the world, the local building codes are varied. Converting the building code as part of the input is a potential solve.

Buildings account for 40 percent of the global carbon dioxide emission (International Energy Agency, 2017). Minimizing the mass of a building, we can not only reduce the material cost, but also decrease the carbon dioxide emission during the fabrication, transportation, and construction process, and therefore, make a huge impact to the environment. However, this domain is under-explored in the machine learning community. Through this work, we hope to arouse more research interest to explore this valuable direction.

Figure 6. Performance Curves of GA using Different Seeding Approaches
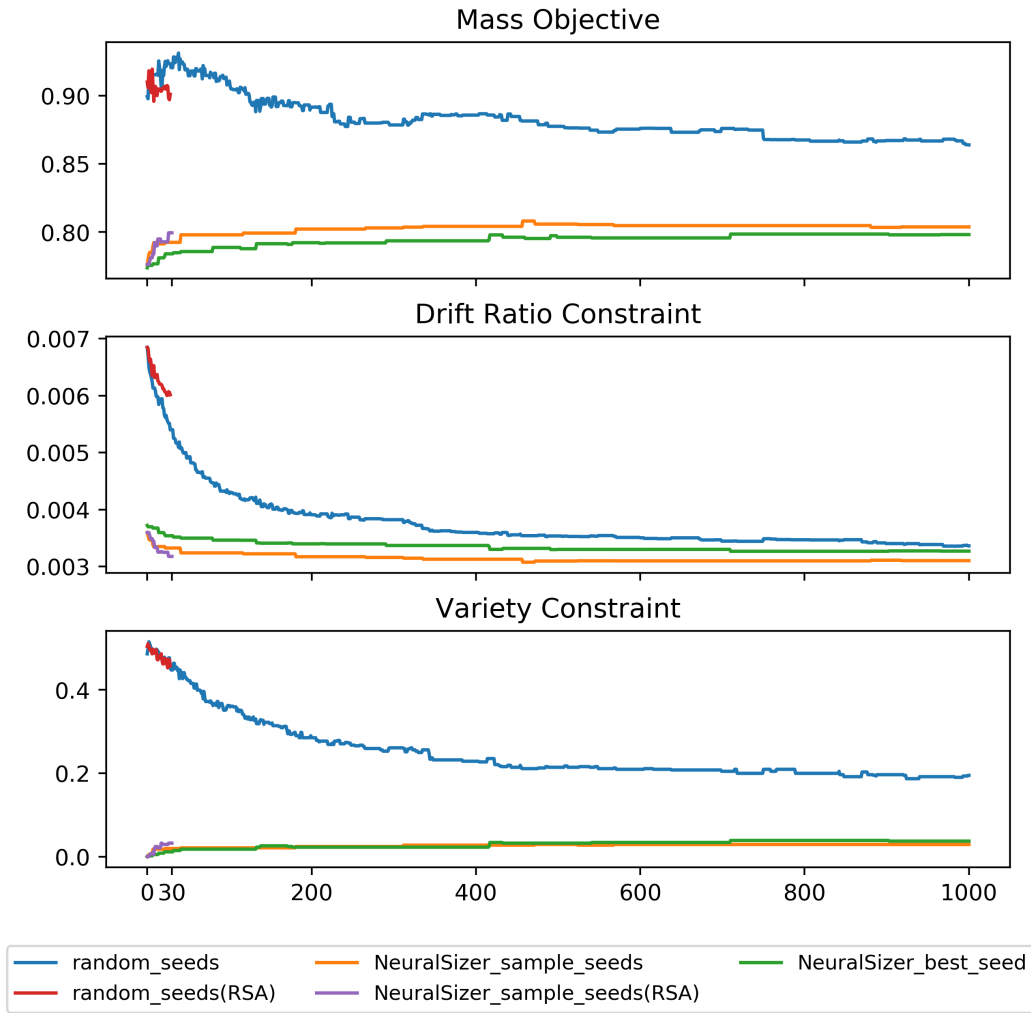
Table 6. NeuralSizer Seeding Performance

Metric	Mass Objective	Drift Ratio Constraint	Variety Constraint
High Safety Factor			
1	232.60%	115.30%	186.20%
2	7.43%	25.70%	95.82%
3	0	25.6	0
Low Safety Factor			
1	83.15%	95.35%	156.22%
2	4.16%	49.22%	32.53%
3	128	0	0

Figure 7. Performance Metrics Illustration
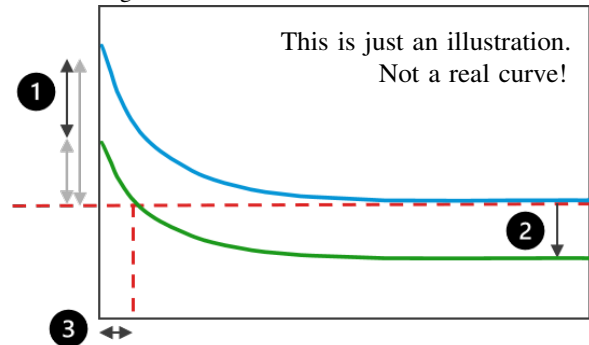
References

Achiam, J., Held, D., Tamar, A., and Abbeel, P. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp.

22–31. JMLR. org, 2017.

Balogh, T. and Vigh, L. Genetic algorithm based optimization of regular steel building structures subjected to

seismic effects. In *Proceedings 15th world conference on earthquake engineering*, pp. 1–10, 2012.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Boyd, S. and Vandenberghe, L. *Convex optimization*. Cambridge university press, 2004.

Cheng, Y., KOU, C.-h., Li, C., et al. Optimal design of steel columns with axial load using artificial neural networks. *DEStech Transactions on Engineering and Technology Research*, (amme), 2017.

Cui, Z., Henrickson, K., Ke, R., and Wang, Y. Traffic graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

Do, K., Tran, T., and Venkatesh, S. Graph transformation policy network for chemical reaction prediction. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 750–760, 2019.

Fout, A., Byrd, J., Shariat, B., and Ben-Hur, A. Protein interface prediction using graph convolutional networks. In *Advances in neural information processing systems*, pp. 6530–6539, 2017.

Frans, K. and Cheng, C.-Y. Unsupervised image to sequence translation with canvas-drawer networks. *arXiv preprint arXiv:1809.08340*, 2018.

Greco, L. Machine learning and optimization techniques for steel connections. In *Proceedings of IASS Annual Symposia*, volume 2018, pp. 1–8. International Association for Shell and Spatial Structures (IASS), 2018.

Guo, S., Lin, Y., Feng, N., Song, C., and Wan, H. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 922–929, 2019.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

Hamrick, J. B., Allen, K. R., Bapst, V., Zhu, T., McKee, K. R., Tenenbaum, J. B., and Battaglia, P. W. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.

Hasançebi, O. and Dumlupınar, T. A neural network approach for approximate force response analyses of a bridge population. *Neural Computing and Applications*, 22(3-4):755–769, 2013.

Imran, M., Shi, D., Tong, L., and Waqas, H. M. Design optimization of composite submerged cylindrical pressure hull using genetic algorithm and finite element analysis. *Ocean Engineering*, 190:106443, 2019.

International Energy Agency. *World Energy Balances 2017*. 2017. doi: https://doi.org/https://doi.org/10.1787/world_energy_bal-2017-en. URL https://www.oecd-ilibrary.org/content/publication/world_energy_bal-2017-en.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Jin, W., Barzilay, R., and Jaakkola, T. Junction tree variational autoencoder for molecular graph generation. *arXiv preprint arXiv:1802.04364*, 2018a.

Jin, W., Yang, K., Barzilay, R., and Jaakkola, T. Learning multimodal graph-to-graph translation for molecular optimization. *arXiv preprint arXiv:1812.01070*, 2018b.

Kipf, T., Fetaya, E., Wang, K.-C., Welling, M., and Zemel, R. Neural relational inference for interacting systems. *arXiv preprint arXiv:1802.04687*, 2018.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Kool, W., Van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Li, Z., Chen, Q., and Koltun, V. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pp. 539–548, 2018.

Prates, M., Avelar, P. H., Lemos, H., Lamb, L. C., and Vardi, M. Y. Learning to solve np-complete problems: A graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4731–4738, 2019.

Rajeev, S. and Krishnamoorthy, C. Genetic algorithms-based methodologies for design optimization of trusses. *Journal of structural engineering*, 123(3):350–358, 1997.

Sanchez-Gonzalez, A., Heess, N., Springenberg, J. T., Merel, J., Riedmiller, M., Hadsell, R., and Battaglia, P. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. W. Learning to simulate complex physics with graph networks. *arXiv preprint arXiv:2002.09405*, 2020.

Tamura, T., Ohsaki, M., and Takagi, J. Machine learning for combinatorial optimization of brace placement of steel frames. *Japan Architectural Review*, 1(4):419–430, 2018.

Tian, Y., Luo, A., Sun, X., Ellis, K., Freeman, W. T., Tenenbaum, J. B., and Wu, J. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019.

Torky, A. A. and Aburawwash, A. A. A deep learning approach to automated structural engineering of prestressed members. 2018.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Watters, N., Zoran, D., Weber, T., Battaglia, P., Pascanu, R., and Tacchetti, A. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, pp. 4539–4547, 2017.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

You, J., Liu, B., Ying, Z., Pande, V., and Leskovec, J. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in neural information processing systems*, pp. 6410–6421, 2018.

You, J., Ying, R., and Leskovec, J. Position-aware graph neural networks. *arXiv preprint arXiv:1906.04817*, 2019.

Zheng, N., Jiang, Y., and Huang, D. Strokenet: A neural painting environment. 2018.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

Zhou, X., Wan, Q., Zhang, W., Xue, X., and Wei, Y. Model-based deep hand pose estimation. *arXiv preprint arXiv:1606.06854*, 2016.