CS 229 Project Report

Project title: Machine learning application in optimization of flexible circuit configuration
Project category: Physical sciences

1 Motivation

Flexible circuits are of great interest to hardware manufacturers. Wearable electronics that stretch with human motion are a prime application of this technology. However, achieving desired stretchability is difficult since transistor elements are relatively stiff. Stretching them beyond some threshold will irreversible damage the parts. One solution is to place the transistors on a flexible polymer substrate which absorbs the required stretching deformation. However, a naive placement of the transistors on a large substrate is an inefficient use of space, so we wish to optimize the configuration of the transistors to minimize the area of the circuit substrate. We can use optimization to determine the configuration. Unfortunately, this problem is non-convex, and the current genetic algorithm implementation is computationally expensive. In a design setting where many circuits must be designed, this computational expense is unacceptable. This motivates us to use machine learning to predict a compact circuit configuration with minimal online runtime.

2 Problem Statement

This optimization problem has been successfully performed for a three-transistor circuit using particle swarm optimization (PSO). During PSO, 60 'particles' are tested in each iteration. Each particle is a circuit configuration with randomly placed transistor elements. We run a mechanical finite element simulation on each particle where the circuit is stretched by 50%. Each finite element simulation determines whether the transistors are damaged. The particles are naively assigned to a random location on the design space, looking for better placements and rejected worse placements. The optimization problem is formulated as

$$\begin{aligned} \min \quad & \text{area} \\ \text{s.t.} \quad & \text{no overlapping transistors} \\ & \text{transistors inside domain} \\ & \text{transistor strain energy below threshold} \end{aligned}$$

The Figure 1 shows two randomly generated configurations in the PSO which might lead to a compact final design. The dark gray boxes represent rectangular transistors, and the light gray area is the substrate domain.

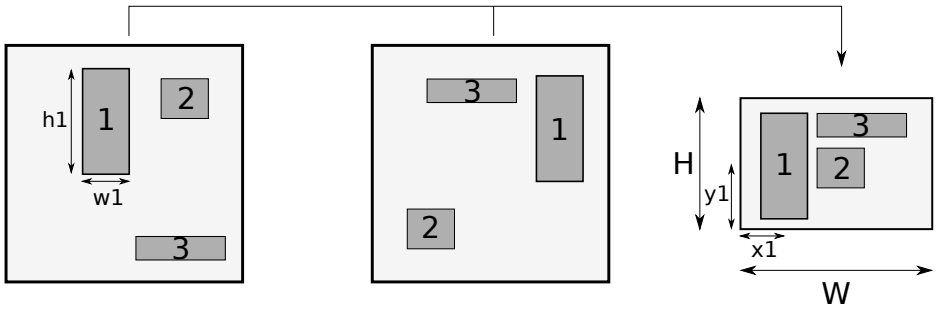


Figure 1: Representation of two 'particles' of possible configurations leading to a more compact design.

Each PSO simulation takes around 24 hours on a single core to predict the optimized solution. If we wish to apply this to larger circuits with a large number of transistors, then this is a very computationally expensive process. So, we wished to explore the application of ML techniques on this complex problem to see if we could predict fairly accurate optimized configurations in lesser time.

3 Data collection

To apply machine learning to this problem, we performed 1,000 optimizations to generate a dataset of near-optimum configurations. This section discusses the structure of the data and our collection procedure.

3.1 Features

Each particle swarm optimization was performed on a circuit with three transistors elements. The twelve input features quantify the geometry and material properties of each transistor. Specifically, each transistor has a specified width w_k, height h_k, Young's modulus E_k, and maximum allowable strain ϵ_{max_k} (for $k = 1, 2, 3$) which are formatted in the dataset as

$$x^{(i)} = [w_1, w_2, w_3, h_1, h_2, h_3, E_1, E_2, E_2, e_{max_1}, e_{max_2}, e_{max_3}]$$

The eight output parameters include the optimized overall dimensions of the circuit W and H and the centerline positions of each transistor x_k, y_k.

$$y^{(i)} = [W, H, x_1, y_1, x_2, y_2, x_3, y_3]$$

3.2 Simulations and data augmentation

We used Latin hypercube sampling to randomly generate our input features, ensuring an appropriate span of the feature space, where $w, h \in [0.2\text{mm}, 2\text{mm}]$, $E \in [5\text{MPa}, 50\text{MPa}]$, and $\epsilon_{max} \in [0.005, 0.01]$. This is implemented using the `pyDOE` module in Python. We generated 1,000 optimized samples, which took two weeks of wall-clock time on 72 cores in parallel.

The 1,000 samples collected was the maximum feasible size for our available computational resources. To increase the size of the dataset, we developed an augmentation technique of the features and outputs. To do this, we permuted the order of each transistor's data. For example, one training sample can be permuted $3! = 6$ times since the data contains information for three transistors as

$$\{1, 2, 3\} \rightarrow \{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}$$

This gave us an additional larger dataset of $3! \times 1,000 = 6,000$ samples. We used both the regular data and the augmented data for learning to compare the effect of augmentation.

4 Supervised learning approach

We first try predicting an optimized output configuration using a supervised learning approach, considering that it is an easier benchmark to set. We separate the data into 80% training set, 10% validation set, and 10% test set.

4.1 Multivariate linear regression

MLR is a learning technique that generates a model to linearly predict an array of outputs of form

$$\hat{Y}_j^{(i)} = \hat{\beta}_0 + \hat{\beta}_{1j} x_1^{(i)} + \cdots + \hat{\beta}_{nj} x_n^{(i)}$$

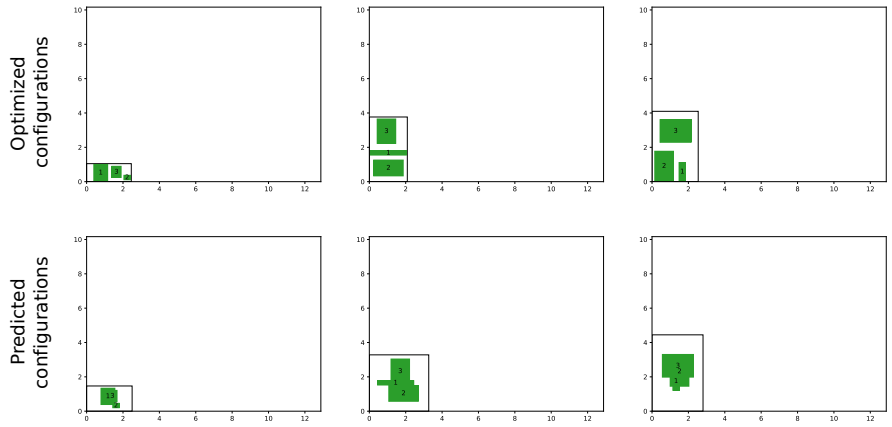
for the ith observation and jth feature. The MLR model was implemented using the `sklearn` module in Python. After training, we compute the mean squared error (MSE) and absolute difference (AD) for both the regular and augmented dataset as

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y^{(i)} - \hat{Y}^{(i)})^2 \;\; ; \;\; \text{AD} = \frac{1}{n} \sum_{i=1}^{n} |Y^{(i)} - \hat{Y}^{(i)}|$$

The difference in errors between the training and test data are small, so MLR has low bias. However, the augmented data set appears to somewhat help prediction performance.

MLR	MSE	MSE (aug data)	AD	AD (aug data)
Training data	2.102	2.338	1.095	1.187
Test data	2.717	2.236	1.194	1.178

Visualizing the predicted configurations gives additional insight into the shortcomings of MLR for our application. The figure below shows three optimized configurations above and their corresponding predictions below. The overlapping transistors in the predicted outputs are problematic since they violate the constraints integrated into the PSO algorithm. The outputs are generated by a nonlinear genetic optimization process and are therefore not a linear span of the inputs. This led us to use MARS, a method better suited to modeling nonlinearities.



--

Figure 2: Examples of test set optimized configurations and their corresponding predicted configurations.

4.2 Multivariate adaptive regression splines

MARS performs regression on a set of features and searches for nonlinear interactions in the training set by performing two stages. The forward stage looks for points to minimize MSE, and the pruning stage finds a subset of terms by minimizing a cross-validation score. The model is then comprised of a constant term, linear functions, and hinge functions. We used the `pyearth` Python package. MARS gives the results in the table below. We note that the MARS overfits the 1,000 sample data, shown by the MSE being much larger for the test data versus the training data. This overfitting is not manifest in the augmented dataset. Overall, the MARS model does not appear to perform significantly better than MLR.

MLR	MSE	MSE (aug data)	AD	AD (aug data)
Training data	1.745	2.341	0.997	1.189
Test data	2.815	2.234	1.242	1.181

The box plots below show the distribution of errors. Both MLR and MARS suffer from high outliers of particularly bad predictions. The distribution of MSE and AD on test samples is roughly the same for both MLR and MSE for the regular and augmented data. The time taken for these algorithms to fit and predict was under 10 minutes of wall-clock time for our datasets. But the major problem is that the constraints are being violated.

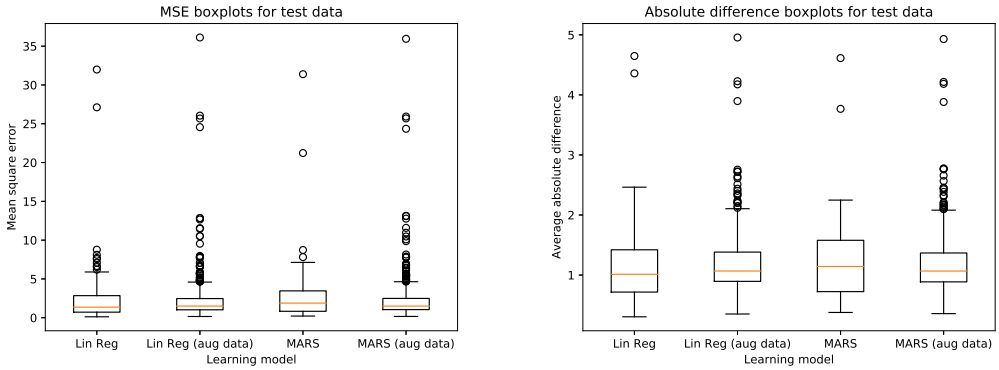


Figure 3: Boxplots of test set errors. Each model has roughly the same distribution of errors.

4.3 Linear scaling

The figure of the predicted configuration shows overlapping transistor elements. This is one limitation of a supervised learning approach: there is no simple way to enforce constraints. One workaround to obtain a valid circuit configuration is to linearly scale the configuration until no overlap is observed as shown below. Although this leads

to a safer configuration (as the substrate area has increased), performing this procedure on the MLR test set results in a mean size increase of 802% over all test set entries compared to the optimized shape. This is not a satisfactory result, which motivates a reinforcement learning approach.

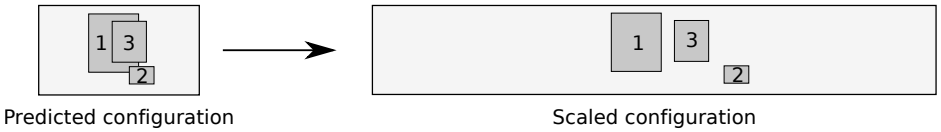


Figure 4: Example of configuration with overlapping transistors that is scaled until constraints are satisfied.

5 Reinforcement learning approach

Our optimization problem ideally falls into the category of continuous state space and continuous action space. The problem has additional complexity in that the state space for the transistor coordinates changes each time the area decreases. To circumvent these complexities, we simplify the model by fixing the area and instead maximizing the factor of safety (FS) of transistors. We define the FS as

$$\text{FS} = \frac{\text{maximum allowable strain energy in transistor}}{\text{average strain energy in transistor}} .$$

In the RL algorithms, positive rewards are given to the current global maximum state and high negative rewards for violated constraints. The state with the highest reward is output as the optimized solution. We found that the formulation of the reward function and initialization of the optimum value function are the most challenging aspects of developing this model. The reasons for this are that the rewards for a particular state are not known a priori and the problem is non-convex.

5.1 Value iterations for discretized space

The simplest way to apply RL to a complex continuous problem is by discretizing it. Both the space and actions are discretized. The optimal value function V^* is learned using Bellman's equation.

$$V^*(s) = R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V^*(s')$$

The value iteration method was both implemented manually (using the framework of cart-pole problem from CS229 PS4) and using `mdptoolbox` package. Though the python module resulted in faster iterations, the time taken for its initialization nullified this advantage.

Initially, high positive rewards were given to the current global maximum states. But this led to premature convergence to local maxima. So, small positive rewards are chosen, and the optimal value function is initialized to a slightly higher value so that other states are also checked for maxima. Random initializations contribute to uniform search over the domain. A sample simulation by discretizing the space into 4,097 states and actions into 125 states produced an optimized configuration as shown below.

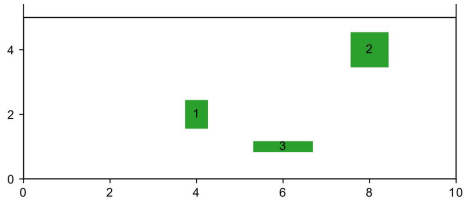


Figure 5: Predicted configuration of value iteration by discretization method

This method suffers from the curse of dimensionality, with even a simple coarse discretization yielding 125 action states and 4,097 space states. Even though this algorithm predicts a fairly accurate optimized shape, convergence requires over 4,000 iterations and 19 hours of wall-clock time on the coarse discretization. Hence, this is a feasible but not desirable method for this problem.

5.2 Fitted value iteration for continuous space

In this method, the space is considered as continuous but the actions are still discretized. In this method, we approximate the optimal value function as a function of the states. Initially, linear regression is used for the approximation. As this problem is non-convex, we found that linear and quadratic functions of the state weren't good enough for approximating the optimum value function over the whole domain. So, we are currently trying out different reward and state functions so as to get a converged solution. We believe that this is a stepping stone towards formulating the RL for a continuous space and action problem for minimizing the area.

6 Conclusion

The goal of this project was to use machine learning to speed up a relatively slow genetic algorithm. We developed learning models using MLR and MARS. Both MLR and MARS consider the output features to be independent of each other and depend only on the input features. This assumption does not hold in this problem and therefore resulted in predictions with violated constraints. The data we get by optimization is bereft of failed/unsafe configurations. Thus, supervised learning algorithms cannot learn about the safety of a resulting configuration.

Linear scaling of the configuration to remove overlap is a possible solution for the problem of violated constraints. However, this technique results in a very high increase in area, making it undesirable. Due to lack of data, we couldn't implement Recurrent Neural Networks (RNN) which considers the dependence between the output features, by taking them as some of the inputs.

This urged us to look at reinforcement learning. Though value iterations by discretization considers all the constraints, this approach may not be a good algorithm for non-convex optimizations, as the algorithm is time consuming and computationally expensive. Though we didn't find an optimal method for predicting configurations in this project, we believe that we successfully evaluated the behavior of some of the ML techniques for such optimization problems.

For future work, we wish to arrive at a good RL technique for this continuous space and action problem of minimizing the area of substrate. If we can produce more data, then a deep learning framework with RNNs can also be tried. Finally, we wish to apply these techniques for optimizing the design of large circuits with a large number of transistors, which was our motivation for this project.

7 Contributions

Ren and Prajwal worked closely together on this project, meeting regularly to discuss ideas and our progress. We developed the idea of the project together. Specifically, Prajwal wrote the majority of the Python scripts for training and testing the data. He also developed the RL algorithm. Ren worked on debugging the PSO code, pre and post processing the data, and generating figures. Responsibilities for writing the paper and creating the poster were evenly divided.

8 Code base

The source code can be found here. The `README.md` file contains a description of the various Python files we used.

9 Acknowledgements

- Reza Rastak, CEE PhD candidate, author of PSO algorithm used in this project, offered advice and access to his code.

10 References

[1] Friedman, J. Multivariate adaptive regression splines. The annals of statistics, 19(1), 1991.
[2] Olsson, A, et al. On Latin hypercube sampling for structural reliability analysis. Structural Safety 25 47-68. 2003.
[3] `pyDOE` Python package. Latin-Hypercube. https://pythonhosted.org/pyDOE/.
[4] `sklearn.linear-model` Python package. LinearRegression. https://scikit-learn.org/stable/modules/linear-model.html.
[5] `pyearth` Python package. Earth. https://github.com/scikit-learn-contrib/py-earth
[6] `mdptoolbox` Python package. MDP. https://github.com/sawcordwell/pymdptoolbox/tree/master/src/mdptoolbox.
[7] CS 229, PS 4 Problem 6, 2018