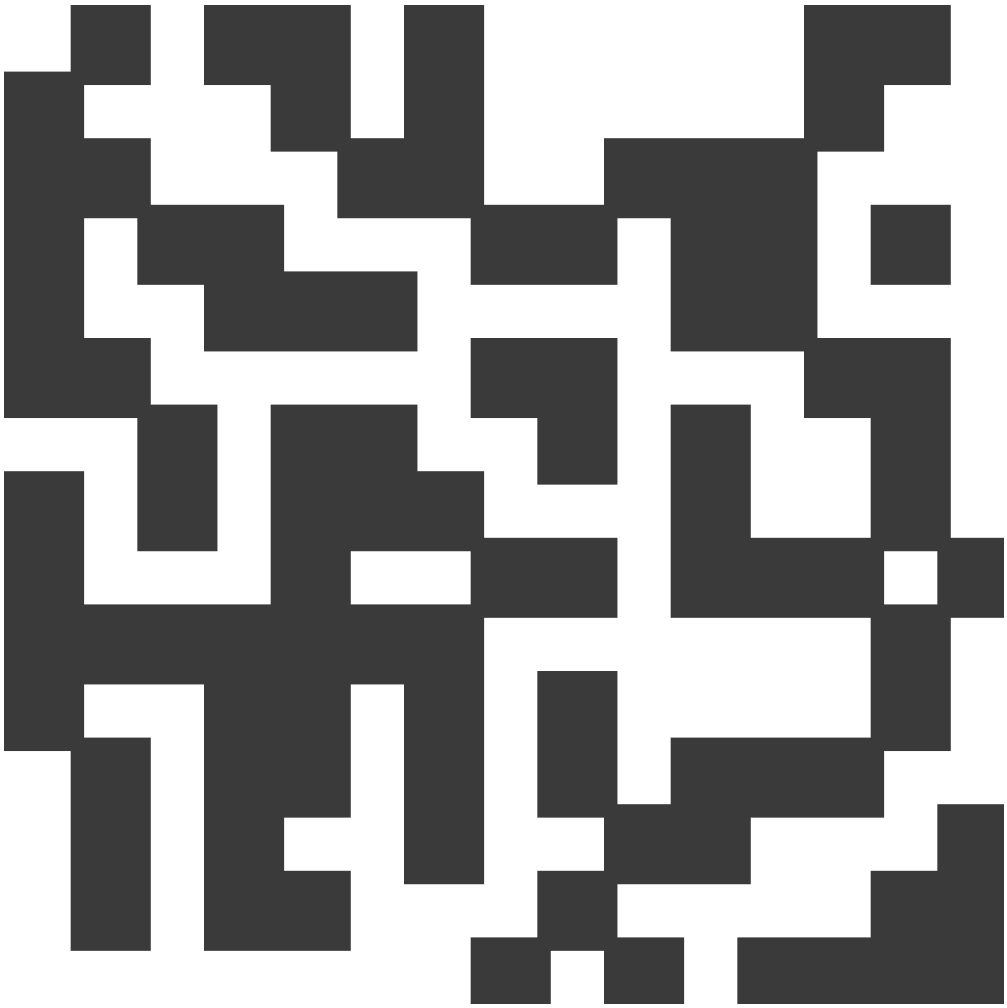

ML assisted circuit design using active learning

Master's thesis in Complex Adaptive System

Omar Bark

DEPARTMENT OF MICROTECHNOLOGY AND NANOSCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

ML assisted circuit design using active learning

Omar Bark

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

ML assisted circuit design using active learning

Supervisor: Martin Sjödin, Ericsson Research
Examiner: Christian Fager, of Microtechnology and Nanoscience

Master's Thesis 2025
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

ML assisted circuit design using active learning
Omar Bark
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

This thesis explores the use of active learning (AL) to reduce the amount of training data required for machine learning (ML) models used for circuit design by selective sampling of the most informative data points. In this study, an uncertainty-based AL approach was implemented. This method leverages the model's prediction uncertainty to selectively sample the most informative data points. A convolutional neural network (CNN) is trained to predict scattering parameters (S-parameters) from pixelated representations of 2-port passive microwave circuits. This method enables the ML model to act as a fast surrogate to electromagnetic (EM) solvers. The goal in this project is to speed up the training process for the ML models using AL.

The performance of the AL-based model is compared to a baseline model trained using random sampling. Evaluation is conducted on a fixed test set, as well as across different frequency ranges and S-parameter. Results show that AL consistently outperforms the baseline in terms of root mean square error (RMSE), particularly at higher frequencies where EM behavior becomes more complex.

Ensemble models were also investigated to assess their potential in improving the sampling strategy. However, they did not yield better results. Each ensemble run required over two weeks of computation, limiting further experimentation. An ensemble of models refers to a collection of multiple individual models whose predictions are combined to improve overall performance and robustness.

Finally, the models were tested in a design task where a genetic algorithm generated circuits from targeted S-parameters. The AL model achieved a 32.9% lower mean RMSE than the baseline when comparing predicted and simulated S-parameters.

These findings highlight AL as a promising approach for improving data efficiency in ML-based circuit design.

Keywords: active learning, machine learning, microwave circuits, surrogate modeling, S-parameters, convolutional neural networks, circuit simulation, electromagnetic modeling

Acknowledgements

I would like to express my sincere gratitude to Martin Sjödin at Ericsson Research for his supervision, continuous support, and valuable insights throughout this thesis project. His guidance has been essential to both the technical development and overall direction of the work.
I would also like to thank Professor Christian Fager at the Department of Microtechnology and Nanoscience at Chalmers University of Technology for serving as examiner and for his feedback and encouragement during the project.
I would also like to thank CEMWorks for sharing their simulation software, which was essential for this thesis project. Their tool played a critical role in generating the data used in this study.
This work was partly funded by Vinnova, and I am grateful for the opportunity to contribute to applied research in this area.
I also want to acknowledge my colleagues and peers at Ericsson for their helpful discussions, which greatly enriched the process.
Lastly, I would like to thank my partner for her patience, encouragement, and unwavering support during this intense and rewarding journey.

Omar Bark
Gothenburg, May 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial intelligens
ML	Machine Learning
EM	Electromagnetic
AL	Active learning
MW	Microwave
CNN	Convolutional neural network
S-parameter	Scattering parameters
GA	Genetic algorithm
RF	Radio frequency
MoM	Method of moments
BN	Batch normalization
NLL	Negative log-likelihood

Contents

List of Figures

1
Introduction

This chapter provides a brief introduction to the study. After reading it, the reader should have sufficient background on the topic and a clear understanding of the objectives and scope of this thesis.

1.1 Background

Traditional circuit design relies on predefined topologies, optimized through computationally expensive parameter sweeps. While effective, this approach limits design exploration and may overlook superior configurations.

Professor Kaushik Sengupta and his research group at Princeton University introduced a new approach that applied artificial intelligence (AI) in radio frequency (RF) integrated circuit design. The proposed concept suggests using a binary matrix to symbolize the circuit, where 1 stands for metal and 0 for non-metal, figure 1.1 illustrates a typical matrix layout. With deep learning models, they were able to automate and significantly improve the design process for broadband power amplifiers. Their method not only reduced design time but also produced unconventional circuit layouts that could outperformed traditional designs [1]. Similar AI-based inverse design techniques have also been extensively explored in the field of nanophotonics, particularly for the synthesis of optical passive components such as wavelength-division multiplexing (WDM) filters and metasurfaces. These methods leverage deep neural networks or adjoint-based optimization to enable rapid and efficient design of complex photonic structures with highly specific transmission characteristics. Notable examples include the use of deep learning for the inverse design of nonlinear nanophotonic devices and high-performance photonic structures [2–6].

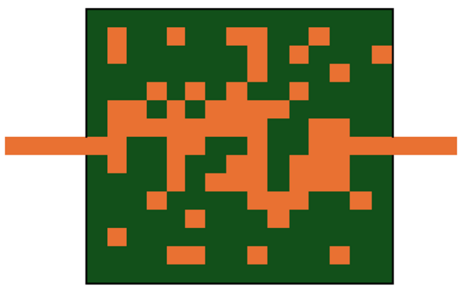

Figure 1.1: Binary circuit layout where orange squares represent **metal (1)** and green squares represent **non-metal (0)**.

Machine learning (ML) has emerged as a powerful tool for circuit design, predicting Scattering parameters (S-parameters) of circuits and reducing reliance on time consuming electromagnetic (EM) simulations. Instead of manually testing different designs, an inverse design process is used, where circuits are gradually improved through genetic optimization which relies on an ML-model trained to predict S-parameters of the circuits. The ML model is used as a surrogate for full-wave solvers in order to speed up the circuit design process.

A major challenge in ML-assisted circuit design is the large number of EM simulations needed for training the ML model and the time it takes to run them, especially for very fine simulation grids and large circuits, which significantly increases computational costs.

This thesis adopts a matrix-based representation to support ML-driven circuit design. Specifically, a 2-port 15×15 binary matrix where the ones represent metal and 0 the absence of metal, with the potential to scale to larger configurations. Prevous work have choosen the training data randomly but in this study an active learning (AL) approach is implemented, where the most informative samples are selected.

1.2 Problem Statement

This study evaluates the efficacy of AL relative to prior approaches, with particular emphasis on computational efficiency and predictive performance. It also looks at whether AL can serve as a practical approach for ML–based microwave (MW) design. The goal is to understand if AL can help reduce the amount of data needed, speed up the design process, and make models more reliable. The study also considers how using AL in this context might impact future research and open up new ways of working with EM design problems.

1.3 Research Questions

This section outlines the research questions that guide the investigation. These questions are directly linked to the research objectives and help define the scope and direction of the study.

1. What is the impact of AL on the amount of training data required for accurate S-parameter prediction in ML-based MW circuit design?
2. How does the performance of models trained with AL compare to the performance of models trained with traditional passive sampling in terms of accuracy and computational efficiency?
3. To what extent can AL generalize across different circuit topologies and frequency ranges in practical design scenarios?

1.4 Significance of the Study

This study holds substantial significance for the field of ML-based circuit design, particularly in the context of reducing the volume of data required for effective model training. One of the primary bottlenecks in this domain is the data generation process, which relies heavily on computationally expensive EM simulations. While generating a large amount of data for small circuits (e.g., up to 11×11) is relatively manageable, the simulation time for larger circuits (e.g., 23×23) can span several months depending on the number of required simulations, the frequency resolution, and the available hardware resources. If AL can significantly reduce the amount of data needed, while maintaining or improving predictive accuracy, this approach could greatly accelerate the design workflow. Consequently, the findings of this study have the potential to significantly enhance the scalability and practicality of ML-assisted circuit design.

1.5 Scope and Limitations

The scope of this study is deliberately narrowed to ensure feasibility within the available time and computational resources. The primary focus is on evaluating the performance of AL strategies for S-parameter prediction in ML-driven MW circuit design, using a constrained experimental setup.
Specifically, the study is limited to:

- A single circuit size: 15×15 grid topology.
- One fixed matrix configuration and ports that where symmetrically placed on the opposite side of each other.
- A defined frequency range of 500 MHz to 10 GHz, sampled at 20 linearly spaced points.
- A single EM solver configuration and stack-up structure.
- A limited set of AL strategies and one baseline (passive) model for comparison.

The main limitation of the study is time. Both EM simulation and ML model training are computationally intensive, especially for larger circuits. While the use of a fixed 15×15 topology enables a controlled evaluation of AL performance, it also restricts the generalizability of the findings to other circuit sizes or topological variations. Additionally, due to time constraints, only one matrix configuration was tested, and no parameter sweep or sensitivity analysis was performed.

Although these constraints limit the breadth of the study, they allow for a focused and in-depth analysis within a realistic timeframe. Future work could extend the scope by including multiple circuit sizes, varying matrix parameters, broader benchmarking across solvers and design spaces, and different AL strategies.

1.6 Thesis Structure

This master's thesis is structured into five main chapters, from background to conclusion. The first chapter provides an introduction to the topic, outlining the challenges in ML based circuit design. It defines the problem statement, presents the research questions, and explains the purpose of the study. The chapter also clarifies the scope and limitations of the work, and highlights its potential impact.

The second chapter presents the theoretical foundation for the research. It covers essential concepts such as S-parameters, EM modeling using the Method of Moments (MoM), and the principles of ML with a focus on convolutional neural networks (CNN). It also explains how AL can reduce the need for training data, and how genetic algorithms (GA) can be used to guide circuit optimization.

The third chapter describes the methodology employed in the study. This includes the generation of synthetic training data through EM simulation, the design of the neural network architectures, and the implementation of the AL cycle. The chapter also outlines how the models were trained and evaluated, and how the comparison between different approaches was carried out.

Chapter four presents the results of the experiments. It compares the performance of the baseline model and the model based on AL in terms of predictive accuracy and generalization. It also reports findings from ensemble models and includes the outcome of a circuit design task using genetic optimization.

Finally, the fifth chapter summarizes the main conclusions of the study. It reflects on the effectiveness of AL for improving data efficiency in ML assisted MW circuit design, discusses limitations of the current work, and proposes directions for future research. The thesis concludes with appendices that contain supporting material, such as simulation parameters, model configurations, and additional technical details.

2

Theory

This section provides the theoretical foundation for the research, covering essential concepts in S-parameters, MoM, ML, AL and GAs.

2.1 Scattering Parameters

S-parameters are used to describe the behavior of linear electrical networks in terms of incident and reflected waves at the network's ports. Unlike traditional parameters such as impedance (Z) or admittance (Y), S-parameters are based on the ratio of incoming (incident) to outgoing (reflected or transmitted) waves, and are represented as complex quantities (phasors) that encode both amplitude and phase information.

S-parameters exhibit an intrinsic dependence on frequency and are defined with respect to a specified reference impedance, typically 50 Ω in RF and MW applications [7–9]. The values of S-parameters vary as a function of frequency, and their interpretation is only meaningful when the reference impedance is defined. This normalization is essential, as the numerical values of the S-matrix depend on the impedance environment in which the network operates.

2.1.1 Advantages of S-parameters in high-frequency circuit design

At MW frequencies, traditional network parameters such as impedance and admittance become less practical due to fundamental theoretical challenges and difficulties in accurate measurement. Accurate measurement of Z and Y parameters requires ideal terminations such as open circuits (infinite Z) or short circuits (zero Z), which are challenging to implement across the frequency range considered in this study, specifically 0.5–10 GHz. Furthermore, measuring voltage and current directly at high frequencies is challenging due to unintended inductive and capacitive effects, as well as the fact that signals propagate as EM waves rather than discrete voltage and current. With S-parameters the need for open or short conditions is eliminated, which reduces the risk of instability, and enables accurate, broadband measurements. As a result, S-parameters are far more suitable for high-frequency and MW circuit analysis [10].

2.1.2 Mathematical Formulation and Interpretation

For an N-port network, the relationship between the the incident and reflected waves at each port can be expressed as:

$$\mathbf{b} = \mathbf{S}\,\mathbf{a}$$

where $\mathbf{a}$ and $\mathbf{b}$ are $N \times 1$ vectors containing the complex amplitudes of incident and outgoing waves at each port, respectively, and $\mathbf{S}$ is the $N \times N$ scattering matrix.

In the case of a two-port network ($N = 2$), the scattering matrix is:

$$\mathrm{S} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix}$$

In this example: S_{11} and S_{22} represent the input and output reflection coefficients, S_{21} is the forward transmission coefficient (from port 1 to 2) and S_{12} is the reverse transmission coefficient (from port 2 to 1)
For example, if $S_{11} = 0$, this indicates that no power is reflected back from port 1 when it is excited, meaning it is perfectly matched at that port.
For instance, a low-pass filter typically exhibits high transmission ($|S_{21}| \approx 1$) at low frequencies and significant attenuation ($|S_{21}| \ll 1$) at high frequencies. In contrast, a high-pass filter demonstrates the inverse behavior. Band-pass and band-stop filters are characterized by narrow frequency intervals of strong transmission or rejection, respectively.
By simulating a wide variety of circuit topologies and analyzing their resulting S-parameters, designers can assess whether a given layout meets the target frequency response. The structure of the S-matrix depends on several factors, including the physical geometry, material characteristics, and how ports are defined and connected.

2.1.3 Frequency-Dependent Modeling Complexity

As the frequency of operation increases, the EM behavior of MW circuits becomes significantly more complex. This complexity arises from the shorter wavelengths associated with high frequencies, which make circuits more sensitive to small-scale variations in geometry, materials, and layout discontinuities.

At lower frequencies, the wavelength is much larger than the typical feature sizes in the circuit, so the EM fields tend to vary smoothly. In this case, surrogate models or numerical solvers can often make accurate predictions using relatively coarse resolution.

But as the frequency increases, especially above 7 GHz, even small features can become a significant part of the wavelength. This introduces strong resonance, interference, and mode conversion effects, which require much finer geometric detail and higher computational resolution to capture correctly.

Wave propagation studies confirm that the complexity of wave propagation increases with frequency due to the shorter wavelengths involved, thereby making them more sensitive to small-scale heterogeneities, resulting in intricate interference patterns, dispersion, and stronger attenuation [11]. Similarly, high-frequency waveguide components such as those in the X- and Ku-band require sub-10 μm manufacturing precision to maintain target performance, highlighting how even tiny deviations can cause significant EM mismatches [12].

In surrogate modeling, this means that training data must be significantly richer, and models must resolve finer-scale patterns to maintain low prediction error at higher frequencies. Failure to do so leads to increased RMSE, as observed in this study.

2.2 Method of Moments

The MoM is a numerical method used to solve surface integral equations that arise in EM problems. It is particularly useful in high-frequency applications, such as analyzing how EM waves scatter from conductors in open or unbounded regions.

2.2.1 Integral equation formulation

The MoM begins by expressing Maxwell's equations as surface integral equations, which is particularly useful when modeling scattering from conductors. In time-harmonic problems, the most commonly used formulation is the Electric Field Integral Equation (EFIE). It links the known incident electric field $\mathbf{E}_{\text{inc}}$ to the unknown surface current density $\mathbf{J}$ induced on a conducting surface:

$$\mathbf{E}_{\text{inc}}(\mathbf{r}) = -j\omega\mu \int_S \mathbf{G}(\mathbf{r}, \mathbf{r}') \cdot \mathbf{J}(\mathbf{r}')\, dS'$$

In this equation, $\mathbf{G}(\mathbf{r}, \mathbf{r}')$ is the dyadic Green's function in free space, and the integration runs over the surface S of the object. The *dyadic Green's function* tells you how each component of a source vector (like current density $\mathbf{J}$) affects each component of the resulting field (like $\mathbf{E}$) in space. The goal is to solve for $\mathbf{J}$, which characterizes how the object interacts with the incoming wave.

2.2.2 Discretization

To solve the integral equation numerically, the unknown surface current $\mathbf{J}$ is approximated using a finite sum of known basis functions $\mathbf{f}_n$, each scaled by an unknown coefficient I_n:

$$\mathbf{J}(\mathbf{r}) = \sum_{n=1}^{N} I_n\, \mathbf{f}_n(\mathbf{r})$$

This expansion transforms the continuous integral equation into a system of linear equations:

$$\mathbf{ZI} = \mathbf{V}$$

Here, $\mathbf{Z}$ is the impedance matrix, whose entries describe how each basis function interacts with the others through the dyadic Green's function. The vector $\mathbf{I}$ contains the unknown coefficients I_n from the expansion of the current $\mathbf{J}$. The right-hand side vector $\mathbf{V}$ represents the projection of the incident electric field $\mathbf{E}_{\text{inc}}$ onto each basis function.
The system is typically constructed using the Galerkin method [13], where the same set of basis functions is used for both expansion and testing. These details are handled internally by the Emerald EM solver from CEMWorks [14].

2.2.3 Application for this study

The Emarald EM solver from CEMWorks [14], is a MoM-based EM solver used to calculate the S-parameters of 2-port MW circuits in this thesis. The simulated results form the training data for the ML models developed later in the work. Although the solver itself is proprietary, it follows the MoM framework described above.

2.3 Machine Learning

ML is a subfield of AI that focuses on developing algorithms capable of learning patterns from data. Instead of explicitly programming rules, the model discovers relationships through exposure to input–output pairs. In this thesis work, ML is used to predict S-parameters of circuit topologies represented as binary matrices.

2.3.1 Supervised Learning

Supervised learning is where a model learns from a dataset of labeled pairs $\{(x_i, y_i)\}_{i=1}^{N}$, with x_i representing the input (e.g., a circuit layout) and y_{pred} the output (e.g., S-parameters). The task is to learn a function $f : \mathcal{X} \to \mathcal{Y}$ such that:

$$y_{pred} = f(x_i; \theta) + \varepsilon \tag{2.1}$$

Here, θ denotes the trainable parameters of the model, and ε represents inherent noise or approximation error. The training process involves minimizing a loss function that quantifies the prediction error. A common choice for regression tasks is the loss function Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \|y_i - y_{pred}\|^2 \tag{2.2}$$

2.3.2 Neural Networks

Neural networks are a widely used model class for supervised learning. They are composed of interconnected layers of computational units called neurons, each of which calculates a weighted sum of its inputs, passes it through a non-linear activation function, and forwards the result. The architecture typically includes an input layer, one or more hidden layers, and an output layer.

Figure 2.1 below shows a simple feedforward neural network with one hidden layer:

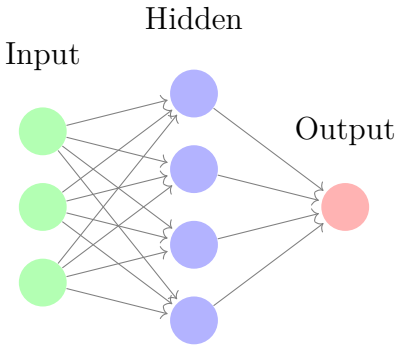

Figure 2.1: A simple feedforward neural network with one hidden layer.

For advanced tasks deeper networks with multiple hidden layers are needed in order to learn complex patterns.

2.3.3 Convolutional Neural Networks

In this work, CNNs are used to extract spatial features from binary circuit layouts, which are represented as grid-based matrices. Because circuits can be represented as binary matrices, CNNs are suitable for capturing local topological patterns that influence circuit behavior.

Unlike fully connected networks where every neuron is linked to every neuron in the next layer, CNNs use small filters (also called kernels) that move across the input to detect patterns. This reduces the number of parameters the model needs to learn and helps it recognize local patterns and spatial relationships, which is important when the position and structure of the data carry meaning.

2.3.3.1 The Convolution Operation

The convolution operation is the central mechanism in a CNN. The operation makes it possible for the network to detect patterns by applying small filters to local regions of the input data. This section goes through the mathematics of this operation in detail and explain how it works in practice.

Let the input be a two-dimensional matrix $x \in \mathbb{R}^{H \times W}$, where H is the height and W is the width of the input. This could be an image, a spectrogram, or the case presented in this study, a binary matrix representing a circuit layout. A filter is defined as $w \in \mathbb{R}^{k \times k}$, which is a small kernel that scans through the input. Typically, the filter size is smaller than the input size, although in applications with small inputs (e.g., 15×15), relatively large filters may be used in early layers.

To compute the convolution, the filter is applied to local regions of the input by sliding it across the spatial dimensions. At each location, the output is obtained by computing a weighted sum between the filter and the corresponding input patch. If

the output is denoted as z, and assuming zero-padding is applied such that the filter is centered at every position, the value at output location (i, j) is computed as:

$$z_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} w_{m,n} \cdot x_{i+m-p,j+n-p}$$

where $w_{m,n}$ are the filter weights, x is the padded input, $k \times k$ is the filter size, and $p = \left\lfloor \frac{k}{2} \right\rfloor$ is the padding size. This ensures that the filter is properly centered even at the borders of the input. The resulting value $z_{i,j}$ quantifies how strongly the pattern represented by the filter is present in that specific region of the input.

Typically in complex problem more than one filter is used. Each filter learns to detect a different type of pattern. When using F different filters, each one produces its own feature map, so the output of the convolutional layer becomes a 3D tensor with shape $(H - k + 1) \times (W - k + 1) \times F$, assuming stride 1 and no padding.

The idea is that in early layers, filters learn to detect simple patterns, while in deeper layers, they combine these to identify more abstract and complex features. This hierarchical representation is one of the main reasons CNNs are so effective in tasks involving spatial data.

2.3.3.2 Activation and Non-Linearity

Once the convolution operation has been applied, the outputs are passed through an activation function. This step is essential to introduce non-linearity into the model. Without it, no matter how many layers are stacked, the entire network would behave as a single linear transformation, which limits its ability to approximate complex functions. Non-linear activation functions enable the network to learn and represent non-linear relationships between the input and output, which is necessary for solving most real-world problems.

A widely used activation function is the Rectified Linear Unit, or ReLU, defined as:

$$\text{ReLU}(z) = \max(0, z) \tag{2.3}$$

This function keeps positive values unchanged while setting all negative values to zero. ReLU is simple and computationally efficient, which makes it a popular choice in convolutional networks. It also helps reduce the vanishing gradient problem [15], making training more stable.

However, ReLU is not perfect. Since it outputs exactly zero for all negative inputs, it can lead to dead neurons, meaning some units may stop updating their weights during training if they get stuck with negative values permanently. To address this issue, several alternative activation functions have been proposed.

One such alternative is the Exponential Linear Unit (ELU), which smooths the transition for negative values. It is defined as:

$$\mathrm{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(\exp(z) - 1) & \text{if } z \leq 0 \end{cases} \tag{2.4}$$

Here, α is a hyperparameter that controls the value to which negative inputs are mapped. Specifically, it determines the minimum saturation level for negative values, where larger α values result in more negative outputs for large negative inputs. ELU allows the output to have negative values, which can improve learning dynamics and help the model converge faster in some cases.
Another smooth variant is the Softplus function:

$$\mathrm{Softplus}(z) = \ln(1 + \exp(z)) \tag{2.5}$$

Softplus can be seen as a smooth approximation of ReLU. It never returns exactly zero and always outputs positive values. Although it avoids the issue of dead neurons, it is more computationally expensive, which makes it less common in deeper and more complex networks.

In some specific applications, the exponential function is used directly in output layers where only positive predictions are allowed. However, due to its tendency to produce very large outputs and gradients, it is rarely used in hidden layers.

Two functions that are also very common are sigmoid and tanh functions. The sigmoid function maps input values to the range between 0 and 1 and is defined as:

$$\mathrm{Sigmoid}(z) = \frac{1}{1 + \exp(-z)} \tag{2.6}$$

The tanh function is similar but maps inputs to the range between -1 and 1:

$$\mathrm{Tanh}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \tag{2.7}$$

These functions were commonly used in early neural networks but tend to suffer from vanishing gradients, which can slow down or prevent learning in deep networks. Because of this, they are now less common in convolutional architectures but still appear in recurrent networks and some specific use cases.

In practice, ReLU is often a reliable default choice, but depending on the depth of the network, the dataset, and the optimization landscape, alternatives like ELU or Softplus can provide better performance or smoother training.

2.3.3.3 Pooling and Downsampling

To reduce the spatial resolution of the feature maps while retaining the most important information, pooling layers are introduced. A common variant is max pooling, where each 2×2 block (or any other chosen window size) is replaced by its maximum value:

$$p_{i,j} = \max_{(m,n) \in \mathcal{R}_{i,j}} z_{m,n} \tag{2.8}$$

where $\mathcal{R}_{i,j}$ denotes the receptive field associated with pooled location (i, j).

2.3.3.4 The Adam Optimizer

Training a neural network involves updating its parameters so that the predictions it makes become more accurate over time. This is done by minimizing a loss function using an optimization algorithm. One of the most widely used optimizers in deep learning today is called *Adam*, which stands for Adaptive Moment Estimation [16]. Adam combines ideas from two other popular optimization methods: momentum and RMSProp. At its core, it keeps track of both the average of the gradients (first moment) and the average of the squared gradients (second moment), and uses these estimates to adapt the learning rate for each parameter individually.
More formally, given a parameter θ, its gradient at time step t is denoted by g_t. Adam maintains two running averages:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.9}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.10}$$

Here, m_t is the exponentially decaying average of past gradients, and v_t is the average of past squared gradients. The constants β_1 and β_2 are typically set to 0.9 and 0.999, respectively.
Since both m_t and v_t are initialized at zero, they are biased toward zero in the early steps of training. To correct this, Adam uses bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.11}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.12}$$

The expressions β_1^t and β_2^t are used in the bias correction terms to compensate for the fact that the moving averages m_t and v_t are initialized as zero and thus biased towards zero in early iterations. This correction ensures that the estimates $\hat{m}_t$ and $\hat{v}_t$ are unbiased, particularly during the initial steps of training.
The parameters are then updated using the following rule:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.13}$$

Here, α is the learning rate (often set to 0.001 by default), and ϵ is a small constant (such as 10^{-8}) added to avoid division by zero.
Adam is popular because it works well in practice for a wide range of problems. It automatically adapts the learning rate for each parameter, requires little tuning, and tends to converge faster and more smoothly than basic methods like standard stochastic gradient descent (SGD).

2.3.3.5 Dropout as Regularization

Dropout is a commonly used technique to reduce overfitting in neural networks. During training, a random set of neurons is temporarily turned off in each iteration. This means the network learns to solve the task without relying too heavily on any specific neuron, which leads to more robust and general representations.
Mathematically, each activation z_i is modified as

$$\tilde{z}_i = z_i \cdot r_i, \quad r_i \sim \text{Bernoulli}(p),$$

where p is the probability of keeping a neuron active, and r_i is a randomly sampled binary variable. At test time, all neurons are used, but their outputs are scaled by p to maintain consistent behavior with training.
By forcing the network to work under slightly different conditions each time, dropout helps prevent it from learning patterns that only work for the training data. It acts like training many smaller networks and averaging their results, which improves generalization [17].

2.3.3.6 Batch Normalization

Training deep neural networks can be challenging due to issues such as internal covariate shift, where the distribution of inputs to each layer changes during training. Batch normalization (BN) addresses this problem by normalizing the inputs to each layer across a mini-batch, thereby stabilizing and accelerating the training process [18].
Given a mini-batch of activations $\{x_1, x_2, \ldots, x_m\}$ for a particular layer, BN computes the mean and variance as

$$\mu_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma^2_{\text{batch}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\text{batch}})^2,$$

and normalizes each input:

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \varepsilon}},$$

where ε is a small constant added for numerical stability. To retain the network's representational power, BN introduces two learnable parameters, γ and β, and produces the final output as:

$$y_i = \gamma\hat{x}_i + \beta.$$

BN reduces the sensitivity of the network to weight initialization and allows for higher learning rates. It has become a standard component in modern deep learning architectures.

2.3.3.7 Layer Normalization

While BN has proven effective in many deep learning architectures, it relies on the statistics of a mini-batch, which can be problematic in scenarios with small batch

sizes or variable sequence lengths. Layer normalization (LN), introduced by Ba et al. [19], addresses this limitation by normalizing across the features of each individual data point rather than across the batch.

Given a layer input vector $\mathbf{x} = (x_1, x_2, \ldots, x_H)$ for a single training example with H hidden units, layer normalization computes the mean and variance as:

$$\mu = \frac{1}{H}\sum_{i=1}^{H} x_i, \quad \sigma^2 = \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu)^2,$$

and normalizes each feature as:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}},$$

where ε is a small constant for numerical stability. Similar to BN, learnable parameters γ and β are introduced to allow the model to scale and shift the normalized output:

$$y_i = \gamma\hat{x}_i + \beta.$$

Layer normalization is particularly effective in recurrent and transformer-based architectures, as it provides consistent normalization behavior independent of the batch size.

2.3.3.8 CNN Pipeline

Figure 2.2 shows an overview of a typical CNN architecture. This kind of setup is commonly used in image classification tasks, such as recognizing handwritten digits in datasets like MNIST. The network takes in a grayscale image and passes it through two convolutional layers, labeled Conv_1 and Conv_2, each followed by a max-pooling layer that reduces the spatial dimensions.

Each convolutional layer learns to extract useful patterns from the image by sliding filters across it, picking up features as described in Section 2.3.3.1. This helps the network focus on the most important information while reducing computational complexity. As we move deeper into the network, the spatial resolution shrinks, but the features become more abstract and meaningful.

Once the final pooling layer is applied, the resulting 3D tensor is flattened into a 1D vector, which is then passed into one or more fully connected layers. These layers combine all the extracted features to form the final prediction.

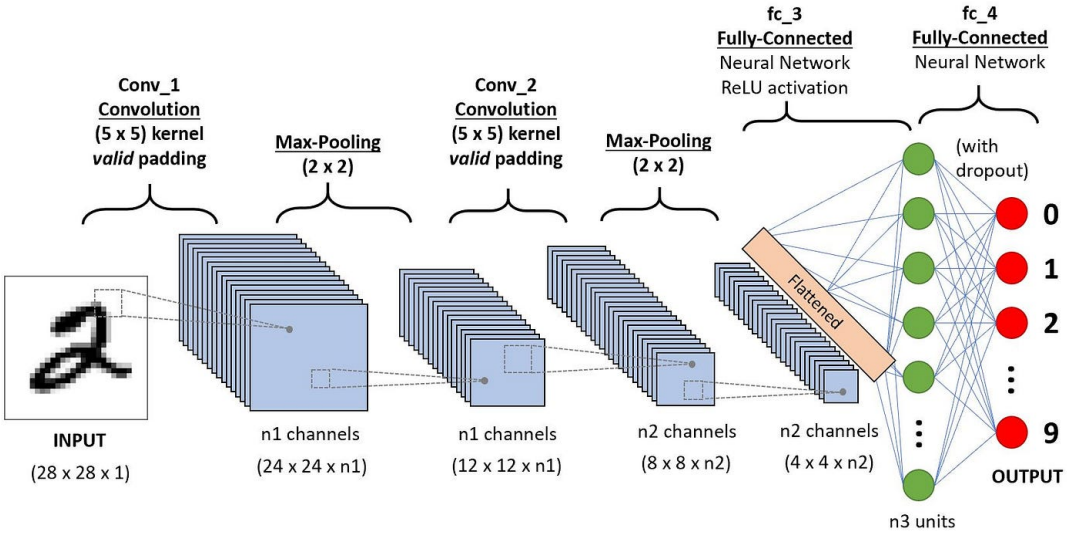

Figure 2.2: Overview of a standard CNN architecture for image classification. The input is processed through a sequence of convolutional and pooling layers, followed by flattening and fully connected layers [20].

While the network shown here is designed for a classification problem that outputs probabilities for digits 0 through 9 using a softmax layer, the same structure also forms the basis for typical regression models. Instead of predicting a class, the final part of the network can be modified to output continuous values, such as predicting S-parameters.

2.3.3.9 Advantages of CNNs for circuit design

CNNs are effective when the data has some kind of spatial or grid-like structure and circuit layouts can be presented in a suitable way for the CNNs. One key advantage of CNNs is that they can detect patterns anywhere in the input using the same set of filter weights. Instead of learning a unique weight for every individual input value, CNNs use small filters that move across the input and identify features regardless of position. This makes them much more efficient and scalable than fully connected networks, especially when working with large inputs. Because of this weight sharing, CNNs can handle complex inputs without requiring a huge number of parameters, which also helps keeping the computations manageable.

2.3.4 Modeling Uncertainty

In many engineering problems, it's not just about getting a prediction. It's equally important to understand how confident the model is in that prediction. Instead of having the model output a single fixed value, we can let it predict an entire probability distribution. This way, the model provides both an estimate and a measure of its own uncertainty. Gaussian output distributions are assumed, where each output is characterized by a predictive mean $\mu(x)$ and variance $\sigma^2(x)$:

$$y \sim \mathcal{N}(\mu(x), \sigma^2(x)) \tag{2.14}$$

Training such a model involves minimizing the negative log-likelihood (NLL) of the Gaussian distribution:

$$\mathcal{L}_{\text{NLL}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{d} \left[\log \sigma_{ij}^2 + \frac{(y_{ij} - \mu_{ij})^2}{\sigma_{ij}^2} \right] \tag{2.15}$$

Here, N denotes the number of training examples, and d is the dimensionality of each output vector (i.e., the number of predicted target variables per example). The double summation accounts for the fact that the model predicts a full d-dimensional Gaussian for each input x_i, with individual means μ_{ij} and variances σ_{ij}^2 for each component j of the output.
The NLL makes it possible to predict both the S-parameter prediction and its own uncertainty. In this study, it is particularly important because it enables the AL approach introduced in Section 2.4.

2.3.5 Evaluation Metrics

Model performance can be evaluated on the basis of how accurate the predictions are and how well the uncertainty is captured.
The main measure of accuracy is the Root Mean Squared Error (RMSE), which shows how far, on average, the model's predictions are from the true values. It is computed as:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \|y_i - \mu(x_i)\|^2} \tag{2.16}$$

In this formula, y_i is the actual value, $\mu(x_i)$ is the predicted mean, and N is the number of samples. Another important metric is how well the model captures its own uncertainty. This can be evaluated by calculating the correlation between the predicted variance σ^2 and the actual squared errors $(y - \mu)^2$. A strong correlation indicates that the model assigns higher uncertainty to less reliable predictions, which is a desirable behavior.

2.3.6 Advantages in circuit design

ML can be a powerful tool in circuit design, particularly when used to approximate expensive simulations. Once trained, a model can produce predictions almost instantly, in contrast to traditional full-wave solvers that often require substantial computational time. This speed makes it possible to rapidly evaluate thousands of design candidates, enabling efficient exploration of the design space.

2.4 Active Learning

AL is a subfield of ML that aims to reduce the need for labeled data by strategically selecting which unlabeled samples should be annotated next. It is very effective

in situations where data sampling is expensive or time-consuming, such as physical simulation, medical annotation, or engineering design tasks [21, 22].

2.4.1 How Active Learning is Formulated

Let $\mathcal{X} \subset \mathbb{R}^d$ represent the input space and $\mathcal{Y} \subset \mathbb{R}^k$ the output space. In the context of AL, the starting point is typically a small labeled dataset

$$\mathcal{L} = \{(x_i, y_i)\}_{i=1}^{n}$$

along with a much larger pool of unlabeled data points

$$\mathcal{U} = \{x_j\}_{j=1}^{m}, \quad \text{with} \quad \mathcal{L} \cap \mathcal{U} = \emptyset.$$

The idea is to gradually expand the labeled set by selecting the most informative samples from $\mathcal{U}$. More specifically, at each step, the labeled dataset $\mathcal{L}$ is updated by adding a newly labeled point selected from $\mathcal{U}$ based on the current model f_θ :

$$\mathcal{L}^{(t+1)} = \mathcal{L}^{(t)} \cup \{(x^*, y^*)\}, \quad \text{where} \quad x^* = \arg\max_{x \in \mathcal{U}} \phi(x; f_\theta).$$

Here, ϕ is an sampling function that scores how informative each unlabeled point is, based on the current state of the model. The point with the highest score is selected, labeled, and added to the training set. Depending on the strategy, the model may be retrained after each acquisition or after a batch of new labeled points has been added. This process is then repeated.
This approach allows the model to improve efficiently by focusing labeling efforts on the data that is expected to be most helpful for learning.

2.4.2 Uncertainty Estimation

A central concept in AL is *uncertainty*, representing how unsure the model is about its prediction for a given sample. In regression settings, the model may be extended to output both the predicted mean $\mu(x)$ and predictive variance $\sigma^2(x)$, allowing it to quantify different types of uncertainty. Two common types are *epistemic* and *aleatoric* uncertainty [23]. Epistemic uncertainty refers to uncertainty in the model parameters due to limited training data and can be reduced by collecting more data. Aleatoric uncertainty, on the other hand, captures inherent noise in the observations and cannot be reduced even with additional data. In Section 2.3.4 a commonly used loss function in AL is described.

There are several alternative ways to estimate uncertainty in neural networks. One popular approach is to use *deep ensembles*, where multiple models $\{f^{(k)}\}_{k=1}^{J}$ are trained independently. Each model makes its own prediction, and the uncertainty is estimated by analyzing the variation between their outputs. This method has been shown to be both simple and effective [24].
Let $\mu_i(\mathbf{p})$ and $\sigma_i^2(\mathbf{p})$ denote the predicted mean and variance from the i-th model for an input $\mathbf{p}$. The ensemble mean and variance are then estimated as follows:

$$\mu_*(\mathbf{p}) = \frac{1}{J}\sum_{i=1}^{J}\mu_i(\mathbf{p}), \tag{2.17}$$

$$\sigma_*^2(\mathbf{p}) = \frac{1}{J}\sum_{i=1}^{J}\left(\sigma_i^2(\mathbf{p}) + \left(\mu_i^2(\mathbf{p}) - \mu_*^2(\mathbf{p})\right)\right), \tag{2.18}$$

The total predictive uncertainty $\sigma_*^2(\mathbf{p})$ thus includes both the individual model uncertainty and the epistemic uncertainty due to model disagreement [25].
Another common technique is Monte Carlo Dropout. Instead of turning dropout off after training, it is kept active during inference, and multiple stochastic forward passes are performed. This mimics sampling from a Bayesian posterior and provides an estimate of uncertainty [26].

2.4.3 Sampling Strategies

To decide which data points from the unlabeled pool $\mathcal{U}$ should be labeled next, different sampling strategies can be used. Each strategy defines a scoring function $\phi\colon \mathcal{X} \to \mathbb{R}$, which ranks the points based on how informative they are expected to be for the model.

2.4.3.1 Uncertainty Sampling

One of the simplest and most commonly used strategies is uncertainty sampling. Here, the model selects the data point for which its prediction is the most uncertain, which is typically measured using the predictive variance:

$$x^* = \arg\max_{x\in\mathcal{U}} \sigma^2(x)$$

This method encourages the model to focus on areas where it currently has the least confidence, which often leads to faster improvement.

2.4.3.2 Query-by-Committee (QBC)

This strategy involves training a group of models, often called a committee, denoted $\{f^{(k)}\}$. For each candidate point x, all models in the committee make a prediction, and the disagreement among them determines how informative the point is [27]. In regression tasks, this disagreement is often quantified using the variance of the predictions:

$$\phi_{\mathrm{QBC}}(x) = \mathrm{Var}\left(\left\{f^{(k)}(x)\right\}_{k=1}^{K}\right)$$

A high variance means that the committee disagrees strongly, which suggests that labeling this point would help the models to predict better.

2.4.3.3 Expected Model Change and Error Reduction

Some AL strategies aim to select samples that are expected to have the greatest impact on the model if labeled and included in the training set. These approaches typically quantify the benefit of a candidate point in terms of either how much it would change the model (*expected model change*) or how much it would reduce prediction error on a given dataset (*expected error reduction*).

For example, in the *Expected Gradient Length* (EGL) method [28], the model estimates the expected norm of the gradient that would result from training on a candidate point with all possible labels. The sample with the largest expected gradient norm is selected, as it would induce the largest change to the model's parameters.

Similarly, the *Expected Error Reduction* (EER) strategy [29] selects the sample that is expected to most reduce the model's generalization error, often estimated on a held-out validation set. This requires simulating how the model would perform after being trained on the newly labeled data.

While both methods are theoretically well-motivated, they are computationally expensive in practice, especially for deep models or large unlabeled pools, since they involve retraining or approximating training effects for multiple candidate points. As a result, they are less commonly used in large-scale applications.

2.4.3.4 Adversarial Sampling for Predictive Uncertainty

Adversarial sampling is a technique for generating perturbed inputs that intentionally challenge a neural network's predictions. Originally introduced by Goodfellow et al. [30], the core idea is to add small, structured perturbations that maximize the model's predictive loss, thereby probing regions where the model is uncertain or overconfident.
Formally, the optimal perturbation δ^* is given by:

$$\delta^* = \arg\max_{\delta \in \mathcal{P}(x)} \ell(\theta, x \oplus \delta),$$

where $\ell(\theta, x \oplus \delta)$ is the loss function and $\mathcal{P}(x) = \{\delta \mid C(x \oplus \delta) = 1 \text{ and } \|\delta\|_0 \leq k\}$ defines the set of valid perturbations. The function $C(\cdot)$ encodes structural constraints (e.g., port connectivity), and the norm constraint $\|\delta\|_0 \leq k$ ensures the perturbation is sparse, i.e., limited to at most k bit flips.
The corresponding adversarial sample is then:

$$x_{\text{adv}} = x \oplus \delta^*.$$

In the context of uncertainty estimation, adversarial sampling serves as an effective regularization strategy. By guiding the model to produce smoother predictive distributions around challenging regions, it helps uncover vulnerabilities and improves calibration. When combined with deep ensembles [24], adversarial training increases robustness to out-of-distribution (OOD) data without requiring full Bayesian inference, while maintaining competitive performance and expressiveness.

2.4.3.5 Diversity-Driven and Random Sampling

If the sampling strategy focuses only on uncertainty, it may repeatedly select samples from the same region of the input space, especially where the model is most uncertain. This can lead to redundant samples and poor coverage of the full distribution, which may hurt generalization.

To avoid this, many practical approaches combine uncertainty with other criteria such as diversity or representativeness. The idea is to ensure that the selected samples are not only informative but also well spread across the input space.

Diversity-based sampling aims to select points that are not only uncertain but also spread out across the input space. This can be achieved using clustering methods or distance-based measures.

Random sampling adds an element of exploration by occasionally picking a point at random. For example, with probability p, the next sample is chosen uniformly from $\mathcal{U}$, regardless of its uncertainty. This helps prevent the model from getting stuck in local regions and improves coverage over the entire input space.

2.4.4 Illustration of the AL Cycle

The AL process proceeds iteratively in a closed loop, as shown in Figure 2.3. Starting from an initial labeled set, a model is trained to predict both the output and its associated uncertainty. This model is then applied to the unlabeled data pool $\mathcal{U}$, from which the most informative samples are selected according to a query strategy $\phi(x)$, such as uncertainty sampling.

These selected samples are passed to a labeling process (e.g., simulation or human annotation), and added to the training set $\mathcal{L}$. The model is then retrained on the expanded dataset. This cycle continues until a stopping condition is met, such as performance saturation or a labeling budget being exhausted.

This iterative sampling framework allows the model to focus on the most relevant parts of the input space, thereby improving sample efficiency and generalization performance. The process is particularly beneficial in domains where labels are expensive, as it minimizes the total number of labels needed to reach a target accuracy.

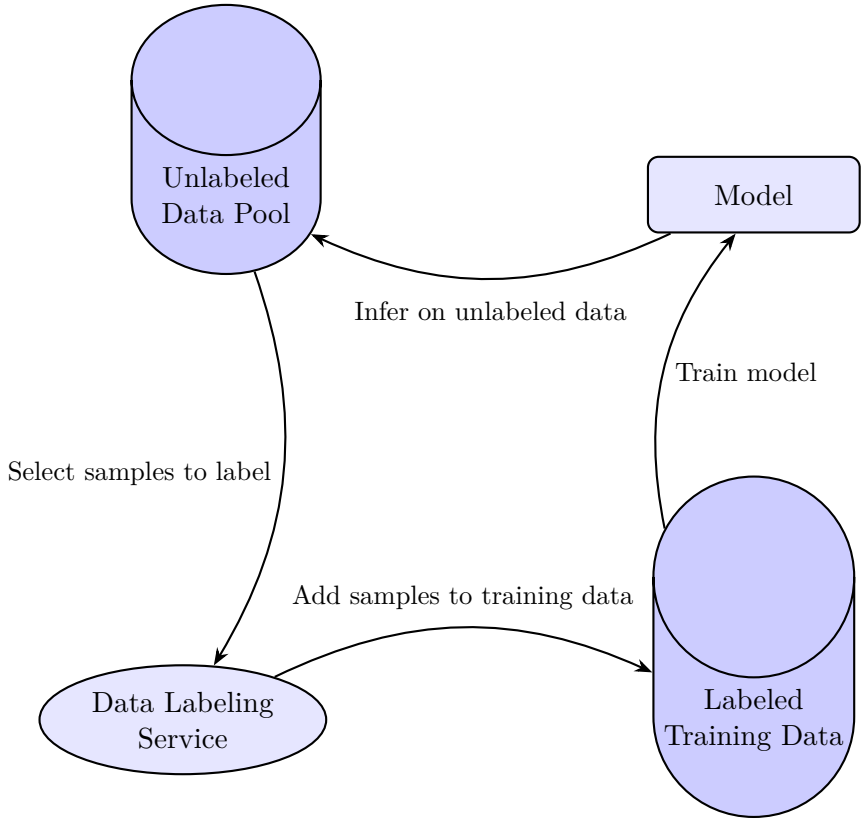

Figure 2.3: Cyclic overview of the AL process. The model identifies uncertain samples in the unlabeled pool, sends them for labeling, updates the training data, and retrains iteratively.

2.4.5 Scientific Applications

In scientific domains such as MW engineering, sampling labeled data (e.g., via full-wave simulation) is computationally costly. By leveraging AL strategies in combination with surrogate modeling, recent work has demonstrated significant reductions in the number of required simulations [31]. This accelerates the design loop for multi-port circuits and other EM structures.

2.5 Genetic Algorithms

GAs are a class of evolutionary algorithms inspired by the principles of natural selection and genetics. They are designed to solve optimization problems by evolving a population of candidate solutions through mechanisms that mimic biological reproduction. As a population-based, stochastic search method, GAs are particularly suitable for complex, high-dimensional, and non-differentiable objective functions.

2.5.1 Encoding, Fitness, and Selection

A GA operates on a population of individuals, where each individual represents a candidate solution to a specific optimization problem. The encoding of these

individuals, known as genotypes, can take different forms such as binary strings, real-valued vectors, or permutations, depending on the nature of the problem. The search space is implicitly defined by the set of all valid genotypes.

Each individual is evaluated using a fitness function, which assigns a scalar value indicating the quality of the solution. The fitness function is problem-specific and reflects how well an individual satisfies the optimization objective. These fitness scores guide the selection process, where individuals are chosen to reproduce and contribute to the next generation. Common strategies include roulette wheel selection, tournament selection, and rank-based selection. These methods favor individuals with higher fitness, following the idea of "survival of the fittest".

2.5.2 Variation, Generations, and Convergence

New individuals are created using two main variation operators: crossover and mutation. Crossover combines parts of two or more parents to produce offspring, while mutation introduces random changes to maintain diversity in the population. These operations help the algorithm explore different regions of the search space and avoid premature convergence.
GAs proceed over multiple generations. In each generation, a new population is formed through selection, crossover, and mutation. Elitism is often applied to ensure that the best-performing individuals are carried over unchanged. Over time, the population ideally converges toward high-quality solutions.
The algorithm stops when a predefined condition is met. Typical stopping criteria include reaching a maximum number of generations, achieving a certain fitness level, or detecting stagnation (i.e., no significant improvements over several generations).

2.5.3 Strengths and Limitations

GAs are inherently heuristic and therefore do not offer formal guarantees of finding a globally optimal solution. Despite this limitation, they are widely valued for their robustness and flexibility. Unlike gradient-based optimization methods, GAs do not rely on derivative information and are thus well suited for problems with non-smooth, discontinuous, or discrete search spaces.

One of the key strengths of GAs is their ability to explore a broad range of possible solutions by maintaining diversity within the population. This makes them particularly effective in avoiding local optima. However, to perform well, a GA must balance two opposing forces: *exploration*, which promotes diversity and broad search, and *exploitation*, which focuses on refining high-quality solutions. If this balance is not maintained, the algorithm may either converge too early or fail to converge at all [32].

3

Methods

The methodology is structured into four distinct phases: data generation, model development, AL, and model evaluation.

3.1 Synthetic Data Generation

To enable supervised training and evaluation of the proposed ML model, a large-scale dataset of MW circuit layouts and corresponding S-parameters was generated. The complete data generation process is outlined below.

3.1.1 Circuit Matrix Sampling

Each circuit layout is represented as a binary matrix $A \in \{0,1\}^{15\times15}$, where $A_{ij} = 1$ indicates a metallic region and $A_{ij} = 0$ denotes a non-metallic (empty) region. The metallic fill ratio p_{metal} for each sample is drawn from a normal distribution:

$$p_{\text{metal}} \sim \mathcal{N}(\mu, \sigma), \quad \mu = 0.5,\ \sigma = 0.15$$

To ensure physical plausibility and control over mesh density, we constrain the sampled metallic fill ratio p_{metal} to lie within the range:

$$p_{\text{metal}} \in [0.2,\ 0.8]$$

Note that the actual metal coverage of each matrix, defined as Density(A), may deviate slightly due to the stochastic nature of the sampling process.

The pixels are enlarged relative to the grid they are placed on to improve diagonal connectivity. The grid pitch is 900 μm, while the pixel size is 1080 μm, corresponding to a 20% increase.

Each valid matrix is also assigned fixed port locations. In the 15×15 configuration used here, port 1 is placed at position $(0, 7)$ and port 2 at $(14, 7)$ see figure 3.1.

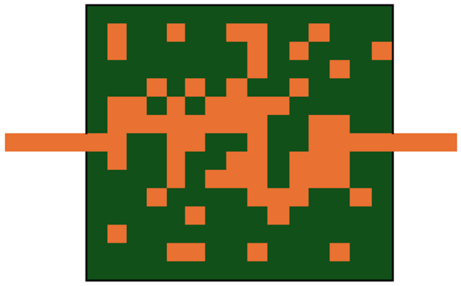

Figure 3.1: Binary circuit layout where orange squares represent **metal (1)** and green squares represent **non-metal (0)**.

To ensure that each sampled circuit matrix exhibits physical feasibility and electrical functionality, a connectivity constraint is enforced. Specifically, a valid sample must contain at least one continuous metallic path between predefined input and output ports, which are fixed at designated boundary positions on the grid.

Connectivity is evaluated using a recursive traversal algorithm that expands through adjacent metallic elements. If a valid path exists between the port regions, the matrix is accepted; otherwise, it is discarded. In that case, a new matrix is resampled using the same p_{metal} value. This constraint ensures that all simulated circuits represent physically meaningful transmission structures.

3.1.2 Electromagnetic Simulation

The EM behavior of each accepted circuit layout was simulated using **CEMWorks Emerald**, a proprietary full-wave solver based on the MoM, as detailed in Section 2.2. This simulation approach is particularly well-suited for high-frequency planar structures, enabling accurate computation of S-parameters across the desired frequency range.

The circuit layout is designed as a three-layer structure, consisting of a bottom copper ground plane, a Rogers 4350 dielectric substrate, and a top copper signal layer. The copper layers serve as the conductive planes, while Rogers 4350 acts as the dielectric material separating them. The stackup configuration is illustrated in Figure 3.2, and the material properties are summarized in Table 3.1.

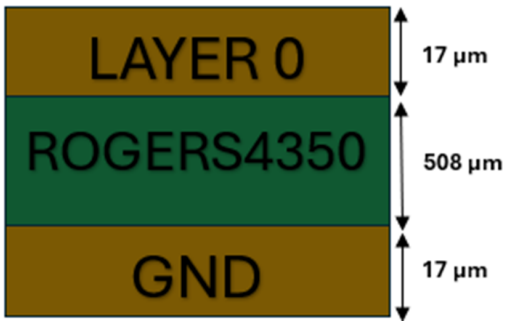

Figure 3.2: Three-layer PCB stackup with Rogers 4350 dielectric and copper layers.

Table 3.1: Material properties for the PCB stackup

Material	Type	Property	Value
Copper	Metal	Conductivity	5.88×10^7 S/m
Rogers 4350	Dielectric	Relative Permittivity (ε_r)	3.66
		Loss Tangent ($\tan\delta$)	0.004
		Relative Permeability (μ)	1

Each geometry is exported as a GDSII layout and compiled into an XML-based configuration file specifying the simulation parameters, including frequency sampling, stack-up structure, mesh density, and material properties. Simulations are executed via a batch interface, and the resulting frequency-dependent scattering matrix $S(f) \in \mathbb{C}^{2\times 2}$ is extracted in Touchstone format (`.s2p`).

3.1.3 Storage and Access

All simulation results and associated metadata are stored in a MongoDB collection to facilitate efficient access and distributed processing [33]. Each document in the database contains:

- The binary layout matrix A
- Simulation identifier and storage path
- Frequency vector f
- Real, imaginary, and magnitude components of selected S-parameters (S_{11}, S_{12}, S_{22})

MongoDB was chosen for its document-based structure and network accessibility, enabling seamless integration with remote components in the AL pipeline. In particular, the database allows a separate virtual machine to query, retrieve, and label new samples dynamically during model training and evaluation.

Data Generation Pipeline

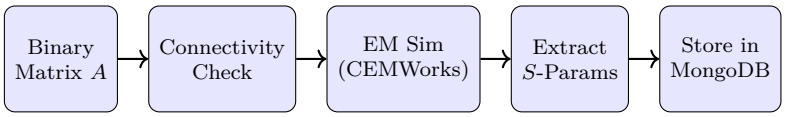

Figure 3.3: Overview of the data generation pipeline used to synthesize and simulate circuit layouts.

3.1.4 Data Augmentation

To enhance the generalization ability of the neural network and address the limited number of original samples, data augmentation was applied to the simulated circuit layouts.

Three types of geometric transformations were used:

- **Horizontal flip**: The layout is mirrored along one of the horizontal axis. Since the ports are symmetrically positioned along this axis, this transformation preserves the physical behavior of the circuit and leaves the corresponding S-parameters unchanged.

- **180-degree rotation**: The layout is rotated half a turn around the vertical axis. This operation effectively interchanges the input and output ports, which requires a corresponding update of the S-parameters such that $S_{11} \leftrightarrow S_{22}$.

- **Combined flip and rotation**: The layout is both flipped and rotated. This transformation is equivalent to flipping the circuit around the opposite horizontal axis in the circuit plane. As with the pure rotation, the port locations are swapped, and the associated S-parameters are updated accordingly.

Each transformed variant is temporarily stored as a new sample and labeled accordingly. The augmentation process increases the dataset size by a factor of four (original plus three augmented versions), while maintaining the underlying physical consistency of the data. This expanded and diversified dataset improves the robustness of the model and helps reduce overfitting during training.

3.2 Machine Learning Method

This study explores deep learning approaches for predicting complex S-parameters of 15×15 binary circuit topologies. To evaluate the impact of architectural choices and uncertainty modeling, three different CNN models were developed:

- A baseline model predicting only the S-parameters

- A joint-output model predicting both S-parameters and associated variances using shared layers

- A dual-branch model with separate layers for predicting S-parameters and variances

The latter two architectures are illustrated in Figure 3.4. The left diagram shows the dual-branch model with separate output heads for mean and variance, while the right diagram shows the joint-output model with shared layers.

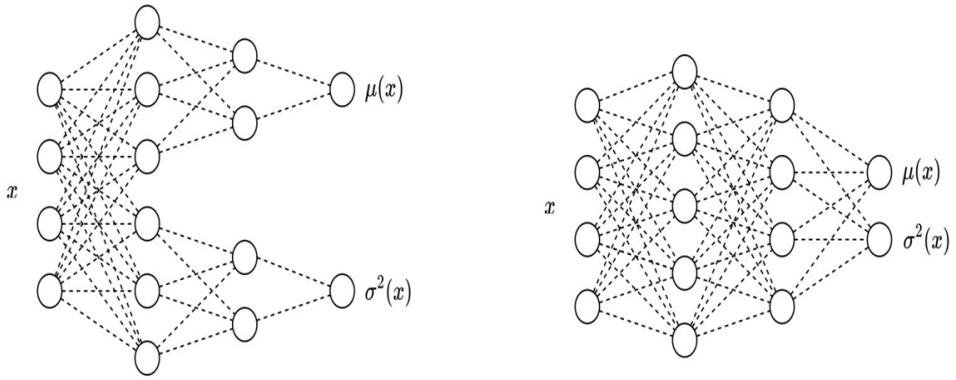

Figure 3.4: Two architectures for modeling predictive mean $\mu(x)$ and variance $\sigma^2(x)$. Left: separate output heads. Right: shared architecture with joint output. Adapted from Gal [26].

The reasoning behind each model architecture, as well as the implementation details and training strategies, are described in the following sections.

3.2.1 Baseline CNN for Initial Validation

To verify the feasibility of predicting EM responses directly from binary circuit layouts, a standard CNN was developed and trained using supervised learning. This baseline model served as a preliminary proof of concept, with the primary goal of confirming that accurate predictions of S-parameters could be obtained before introducing more advanced components such as uncertainty modeling and AL.

The model takes as input a binary matrix of size 15×15, representing the circuit layout. The network architecture consists of multiple convolutional layers followed by a series of fully connected layers. Convolutional feature extraction is performed using progressively smaller kernel sizes, enabling hierarchical pattern recognition across different spatial resolutions. BN and dropout are applied throughout to stabilize training and reduce overfitting.

Pooling is applied after convolutional layer 13, allowing the earlier layers to retain more spatial detail during feature extraction. Once the convolutional part of the network is complete, the features are passed through three fully connected layers, each with 1024 neurons. An ELU activation function was used for the convulotional layers and a linear activation for the fully connected layers. The final layer consists of 120 output neurons with a linear activation function. These outputs represent the real and imaginary components of the S parameters S_{11}, S_{12}, S_{22}, evaluated at 20 frequency points.

The model is optimized using the Adam optimizer with a learning rate of 0.005 and trained to minimize the MSE loss function. In this model, no uncertainty estimates are produced, and all outputs are treated as deterministic point predictions.

The model was implemented using a set of hyperparameters that have previously

been shown to result in stable training. The goal here was not to fine tune for the best possible performance, but rather to ensure that the entire pipeline from data generation and preprocessing to training was functioning correctly. A full list of the hyperparameter settings can be found in Appendix D.

3.2.2 CNN with Shared Layers for the outputs

As a first step toward incorporating predictive uncertainty, a CNN was built to estimate both the S-parameters and their associated variances using a shared architecture. This was done by extending the baseline CNN with an additional output branch, allowing the model to learn both the mean and the variance of the output distribution at the same time, using the same set of convolutional and dense layers.

The idea behind this approach was to keep the model simple and compact, without introducing too many extra parameters. To train the network, a custom loss function based on the NLL in (2.15) of a Gaussian distribution was used, following the method originally proposed by Nix and Weigend [34].

In practice, however, the results were disappointing. The predicted variances were unstable and did not reflect meaningful uncertainty. The model also performed worse on the validation set compared to the baseline. This outcome aligns with findings from earlier studies, which suggest that using a shared representation for both mean and variance often causes interference between the two learning processes [24, 34].

Recent work has shown that better uncertainty estimates are achieved when the network uses separate branches for predicting mean and variance [24, 34, 35]. Based on this, the shared-layer model was abandoned in favor of architectures that treat mean and variance as distinct learning targets, as described in the next subsection.

3.2.3 CNN with Separated Branches

Unlike the previous shared-layer approach, this model separates the two tasks into distinct branches after the feature extraction layers. The structure is based on the probabilistic regression framework proposed by Nix and Weigend [34], where the network produces both a prediction and an estimate of how uncertain that prediction is. Each branch uses its own fully connected layers to learn either the mean or the variance, allowing the model to handle the two learning objectives independently. This setup is useful when the data contains heteroscedasticity (input-dependent noise).

By keeping the mean and variance learning pathways separate, the model avoids the interference problems seen in the shared-layer version. This design has also been supported by more recent research [24, 35], which shows that separated branches often lead to better uncertainty calibration and overall performance.

Architecture Overview. The model consists of a shared convolutional feature extractor followed by two separate fully connected branches: one dedicated to estimating the mean $\mu(x)$, and the other to estimating the standard deviation $\sigma(x)$, from which the variance $\sigma^2(x)$ is derived.

The input is a tensor of shape $H \times W$, representing the binary layout of a circuit sample. After initial BN, the input is passed through a sequence of convolutional layers with ReLU activations, dropout regularization, and optional max pooling at specified layer indices. The convolutional features are then flattened and split into two independent dense pathways.

Mean Branch. The mean estimation branch consists of several dense layers with `tanh` activations as used in [34], followed by BN and dropout. The final layer outputs 120 real values representing the predicted real and imaginary components of the S-parameters:

$$\hat{\mu}(x) = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n]$$

Each $\hat{y}_i$ corresponds to either the real or imaginary part of a specific S-parameter at a given frequency point. For example, $\hat{y}_1 = \mathrm{Re}[S_{11}(f_1)]$, $\hat{y}_2 = \mathrm{Im}[S_{11}(f_1)]$, $\hat{y}_3 = \mathrm{Re}[S_{12}(f_1)]$, and so on.

Variance Branch. The variance branch follows an identical structure but with separate weights. Each dense layer is followed by layer normalization and dropout. The final output layer uses a softplus activation as described in Section 2.3.3.2. The softplus activation function was used due to it ensuring positive variance values and promoting a more stable training process than the exponential activation function. To ensure stable training, the bias of the final variance layer is initialized using a transformation of a predefined standard deviation σ_{init}, via:

$$z_{\text{init}} = \log(\exp(\sigma_{\text{init}}) - 1)$$

where σ_{init} was empirically evaluated and set to 0.1.

Model Output The final output is the concatenation of the predicted mean and variance vectors:

$$\hat{y}_{pred}(x) = [\hat{\mu},\ \hat{\sigma}]$$

3.2.4 Two-Stage Training Strategy

To improve the reliability and calibration of uncertainty estimation, a two-stage training strategy was used. This approach separates the optimization of the predictive mean and variance into distinct phases, reducing interference between learning objectives and improving numerical stability.

Phase 1: Training Mean Only. In the first phase, the model is trained to predict only the mean values μ, while all layers associated with the variance estimation σ^2 are frozen. The model is optimized using mean squared error (MSE)

loss and the Adam optimizer with a learning rate of 0.001. This setup ensures that the shared convolutional and dense layers learn meaningful feature representations without destabilizing gradients from the variance branch.

Phase 2: Joint Optimization of Mean and Variance. Once the mean branch has converged, the variance layers are unfrozen and both branches are trained jointly using the NLL loss. The learning rate is reduced to 0.0005 in this phase to account for the sensitivity of the variance outputs, which are prone to instability when exposed to large gradient updates. This careful scheduling promotes smoother convergence and improves the calibration of the predicted uncertainties.

Training Schedule. The number of epochs for each phase was determined empirically. Based on repeated experimentation, 300 epochs were found sufficient for Phase 1 to achieve convergence of the mean predictions, while 400 epochs in Phase 2 allowed sufficient fine-tuning of both mean and variance outputs without overfitting.

3.2.5 Developed Loss functions

Accurate uncertainty estimation in deep learning models requires loss functions that not only ensure precise predictions but also promote meaningful variance outputs. Several candidate loss functions were implemented and tested in this work. The final selection was based on empirical performance in terms of correlation between predicted variance and squared error. Apart from the NLL in (2.15), these functions where tested.

Correlation-Penalized NLL. To encourage alignment between predicted variance and squared error, a penalty term was introduced based on their empirical correlation:

$$\mathcal{L}_{\text{Corr-NLL}} = \mathcal{L}_{\text{NLL}} \cdot \left(1 + \frac{1}{\rho^2}\right)$$

where ρ is the Pearson correlation coefficient between σ^2 and $(y - \mu)^2$ [36]. While effective in increasing correlation, this loss destabilized training.

Soft Threshold Correlation Penalty (Final Loss) The final loss function added a soft penalty only when correlation fell below a target threshold:

$$\mathcal{L}_{\text{Final}} = \mathcal{L}_{\text{NLL}} + \alpha \cdot \max(0, \rho_{\text{threshold}} - \rho)^2$$

This formulation consistently yielded correlations in the range of 0.55–0.65 during validation, without significantly harming predictive performance.
These results are comparable to prior work such as [37], which reported similar correlation magnitudes for probabilistic CNNs in regression tasks. A full list of tested loss functions is given in Appendix B.

3.2.6 Output Activation for Variance Estimation

To ensure that the predicted variances are strictly positive, two activation functions were tested for the output of the variance branch: the exponential function and the softplus function.

The exponential activation, $\sigma(x) = \exp(z)$ used in [34], enforces positivity but often leads to unstable gradients during training. In practice, this caused erratic variance estimates and made the model sensitive to initialization and learning rate.

To address this, the softplus function discussed in Section 2.3.3.2 was used. The softplus function guarantees positive outputs but avoids the steep gradients of the exponential function. It behaves more smoothly and is nearly linear for large inputs, which helps keep training stable.

Models using softplus showed more consistent training progress, better calibration of predicted uncertainty, and fewer numerical issues. Based on these observations, softplus was chosen as the final activation function. This decision is also supported by prior work [23, 34], where softplus is commonly used in heteroscedastic regression tasks.

3.2.7 Evaluation Protocol

To find a well suited architecture for predicting both S-parameters and variance, 18 different versions of the two-branch CNN were tested. These varied in depth, number of filters, kernel sizes, pooling, dense layers, and regularization.

Each model was trained using the two-stage training approach and evaluated based on RMSE. The model that performed best was chosen for the rest of the project.

This final setup is referred to as the *Large Baseline Model*, and its structure is shown in Table 3.2. Details of all tested configurations can be found in Appendix C.

Table 3.2: Final selected configuration (Large Baseline Model)

Component	**Configuration**
Convolutional kernel sizes	[8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2]
Number of filters per layer	[128 (×12), 64 (×2)]
Pooling	Max pooling after layer index 12
Activation function (CNN/dense)	ELU / ELU
Dense layers (mean branch)	[1024, 1024, 1024]
Dense layers (variance branch)	[1024, 1024, 1024]
Dropout (CNN/dense)	0.1 / 0.1
L2 regularization (CNN/dense)	0.0001 / 0.0001

3.3 Active Learning Loop

To minimize the number of required EM simulations while maintaining predictive performance, an iterative AL loop was developed. The design of the loop follows the theoretical framework introduced in Section 2.4, where uncertainty is leveraged to select the most informative samples for simulation. The loop was implemented using TensorFlow/Keras [38, 39] for model training and MongoDB [33] for data storage and management.

3.3.1 Initialization and Data Partitioning

The loop begins with the construction of an initial dataset from a large collection of pre-simulated 15×15 MW circuit layouts. These data were distributed across two MongoDB instances hosted on separate servers. A total of 7000 samples were allocated for initial training and 1500 for validation. An additional 12000 examples were set aside as a static test set, and the remainder (300 000) formed the unlabeled pool. This test set was fixed for all experiments.

3.3.2 Model Training

The predictive model consisted of a convolutional neural network with a dual-branch architecture for mean and variance prediction, as described in Section 3.2.3. A two-stage training strategy was employed (Section 3.2.4): first optimizing the mean branch using MSE loss, followed by joint optimization of both branches using a custom loss describe in (Section 3.2.5). The model was trained for 300 epochs in Phase 1 and 400 epochs in Phase 2 using the Adam optimizer with learning rates 0.001 and 0.0005 respectively.
Training checkpoints, early stopping, and learning rate schedulers were used to ensure convergence and avoid overfitting. For ensemble learning experiments, 5 models were independently trained and their outputs aggregated to improve robustness and uncertainty estimation.

3.3.3 Uncertainty-Based Sample Acquisition

After training, the model was used to infer predictive variance on a randomly drawn batch of 35000 unlabeled samples from the pool. For single-model settings, the uncertainty score for each sample was defined as the average predicted variance across all S-parameters and frequencies. In ensemble settings, uncertainty was estimated by combining model disagreement and individual variances as described in Equation 2.18.
In each iteration, the 8500 most informative samples were selected and split into a training set of 7000 samples and a validation set of 1500. This ensured that the amount of new data added in each round matched the size of the initial labeled set. The selected samples were then augmented as previously described and added to the labeled pool. To ensure a fair evaluation, the samples that were not selected in a given iteration were discarded from the pool and did not participate in future

selection rounds of the AL loop. To keep the process consistent and reproducible across iterations, all data splits were saved for later use.

3.3.4 Model Update and Iteration

In each iteration, the model was retrained from scratch using all labeled training data collected up to that point. This process was repeated for a total of 8 iterations, which resulted in a final training set with 56 000 samples and a validation set of 12 000.
At the end of each iteration, the model's RMSE was evaluated on the fixed test set and recorded for comparison.

3.4 Model Comparison

To evaluate whether AL actually led to more data-efficient training, the final AL model was compared to a baseline model trained on the same amount of data, but with samples selected randomly instead of based on uncertainty. Both models used the same architecture described in Section 3.2.3, but the baseline was trained in a single stage and did not use the two-phase training setup. The second phase does not improve the RMSE therefore it was not included.
To make the comparison fair, both models started from the same initial dataset, used the same fixed test set, and were trained with identical hyperparameters see Appendix D. The only difference was how the additional training data was selected.

3.4.1 Experimental Design

The baseline model was trained on datasets of the same size as those used in the AL loop, but the additional samples were selected randomly from the same unlabeled pool. Both the baseline and the AL model started with identical training and validation sets. While the AL model expanded its labeled data step by step using uncertainty sampling, the baseline simply added randomly chosen samples with the same batch sizes.
To keep the comparison fair, both models were evaluated on the same test set containing 12,000 samples. For each model, the RMSE per example was calculated for every iteration and then averaged across the entire set. In addition, separate RMSE values were computed for each individual S-parameter (S_{11}, S_{12}, S_{22}) at the final iteration.

3.4.2 Generalization Through Genetic Optimization

To further evaluate how well the trained models generalized beyond their training data, they were tested in a practical circuit design task using a GA. In this setup, the model was not retrained or fine-tuned. Instead, it was used as a surrogate model to predict the S-parameters of thousands of candidate circuits during the optimization process.

The GA's goal was to find circuit layouts that matched specific S-parameter targets. In total, 100 target S-parameters were chosen from the test set randomly. For each target, the model was used to guide the search by scoring candidate circuits based on how closely their predicted S-parameters matched the target. The best circuits from the GA process were then passed on to EM simulations for validation.

To evaluate how well the model handled new designs, two types of errors were used. First, the predicted S-parameters of the generated circuit were compared to the target values used in the optimization. This showed how well the model could guide the GA toward a circuit that met the design goals. Second, the predicted S-parameters were compared to the results from EM simulation of the same circuit. This provided a measure of how accurate the model's predictions were when applied to entirely new circuit layouts.

4

Results

4.1 RMSE comparison between the models

Figure 4.1 shows the average RMSE on the test set for both models at each iteration in the AL. The same test set was used throughout, ensuring a fair comparison. As seen in the plot, the AL model consistently outperforms the base model trained with randomly sampled data. The RMSE of the base model quickly plateaus, which may be due to the training samples being less informative. Multiple runs were performed, and the base model consistently showed similar behavior.

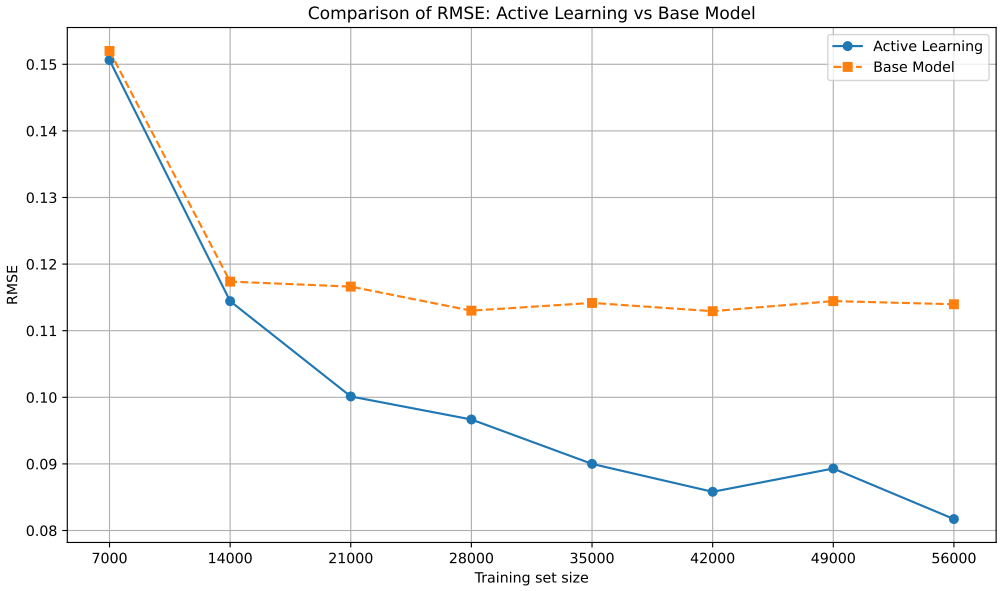

Figure 4.1: Comparison of RMSE for the base model and the AL model as a function of training set size.

Figures 4.2, 4.3, and 4.4 present the RMSE across frequency for the real and imaginary parts of the S-parameters (S_{11}, S_{12}, and S_{22}) at the final iteration of the AL loop. The AL model again shows superior performance over the baseline model across all components.

A notable trend is the increase in RMSE at higher frequencies, particularly above 7 GHz. This behavior is discussed further in Section 2.1.3. These results highlight the benefits of adaptive sampling strategies like AL, which focus more on difficult

regions and thus maintain better accuracy even in challenging frequency ranges.

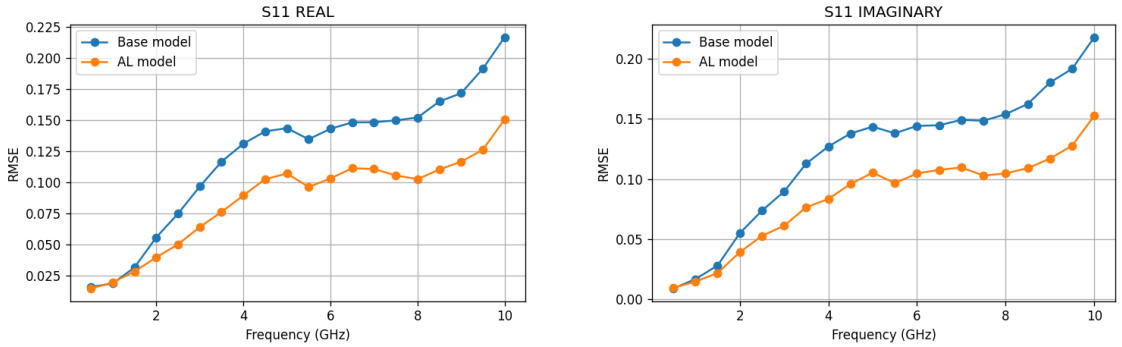

Figure 4.2: RMSE for S_{11} over frequency.

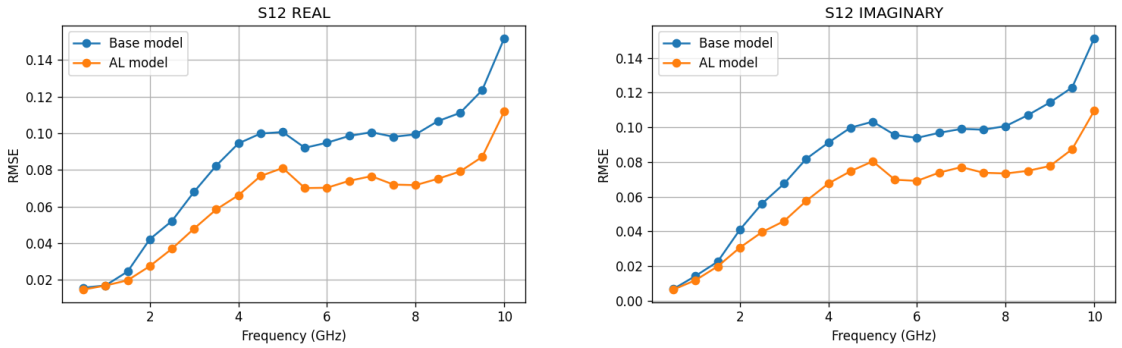

Figure 4.3: RMSE for S_{12} over frequency.

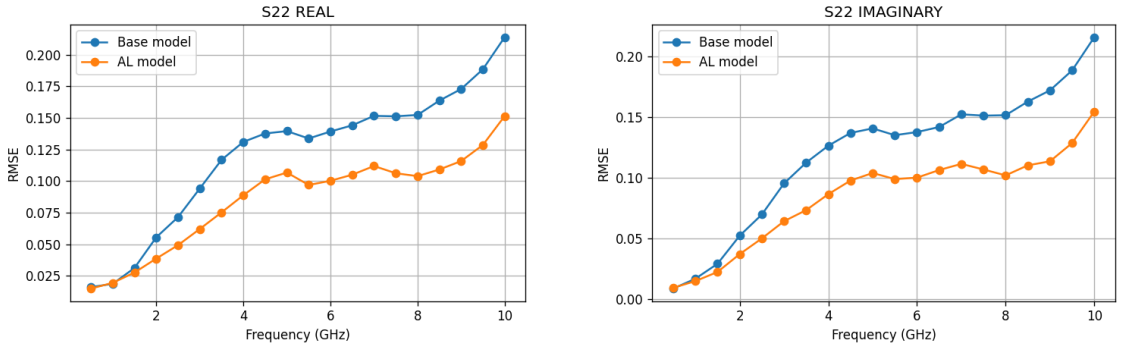

Figure 4.4: RMSE for S_{22} over frequency.

4.2 RMSE for Ensembles of Models

Figure 4.5 shows how the RMSE developed over AL iterations for each individual ensemble model. Interestingly, the ensemble approach did not give better results compared to using a single model.

One reason for this could be how the variance was calculated and used during training and prediction. As shown in (2.17), the ensemble mean is computed as the average of individual model means. The corresponding variance is given in (2.18). The way uncertainty is combined across ensemble members affects how the model samples new data points, and in this case, it may have led to less informative choices. This aspect was not explored further because running a full ensemble takes around 16 days. It remains a promising direction for future work, as better methods for handling uncertainty could potentially improve performance.

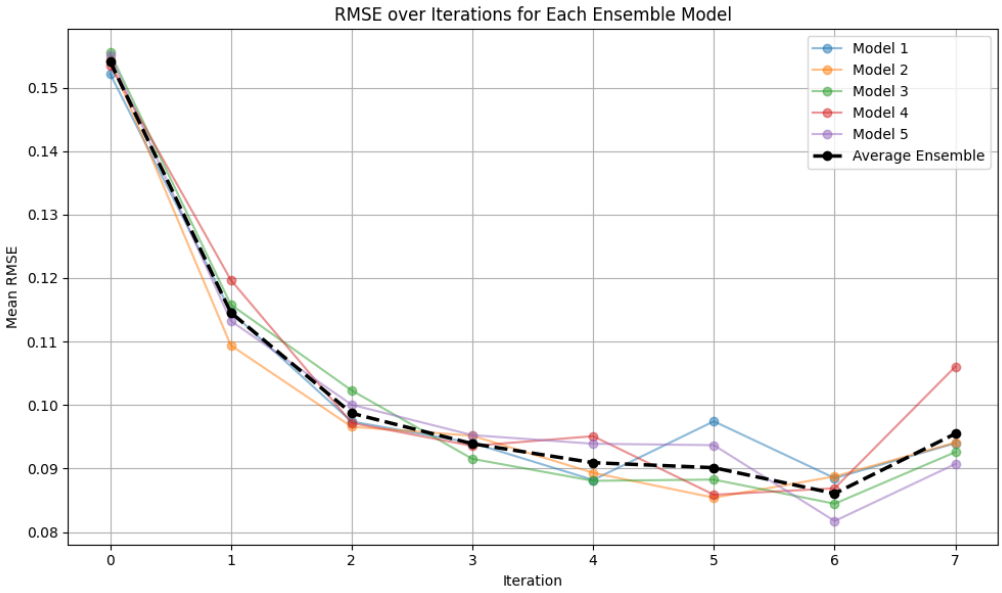

Figure 4.5: Mean RMSE over AL iterations for each ensemble model.

4.3 Results from Circuit Design

The results presented in Table 4.1 are based on a downstream task where target S-parameters were provided as input to a GA. The GA returned circuit layouts for target S-parameters. Each layout was then simulated to extract its S-parameters using the EM solver described in Section 3.1.2. The simulated S-parameters were compared to the predictions from both the base model and the AL model. The RMSE was used to quantify the difference between the predicted and simulated responses.As shown in the table, the AL model achieved a significantly lower mean RMSE than the base model, with a reduction of 32.9%.In addition to the lower mean RMSE, several other improvements can be observed. The standard deviation of the RMSE is notably smaller for the AL model (0.0644 vs 0.1496), indicating more consistent performance across different circuit configurations. The median RMSE is also lower (0.1102 vs 0.1425), suggesting that the typical prediction error is reduced, not just the average. Furthermore, the worst-case error (Max) is significantly lower in the AL model (0.4525 vs 0.9463), highlighting improved robustness and reliability, especially in edge cases. This suggests that the AL model generalizes better to unseen circuit configurations, making it more suitable for design tasks guided by

target specifications.

Metric	AL	Base
Mean	0.119	0.178
Std	0.064	0.149
Median	0.110	0.142
Min	0.049	0.041
Max	0.453	0.946

Table 4.1: Summary of RMSE metrics comparing the AL model and the base model.

4.4 Filter Design and Final Results

In this experiment, a MW filter was designed with a specific target behavior for the S_{12} parameter, as illustrated in Figure 4.6. The goal was to allow transmission up to approximately 7 GHz, after which the magnitude should drop sharply, forming a low-pass response with strong attenuation.

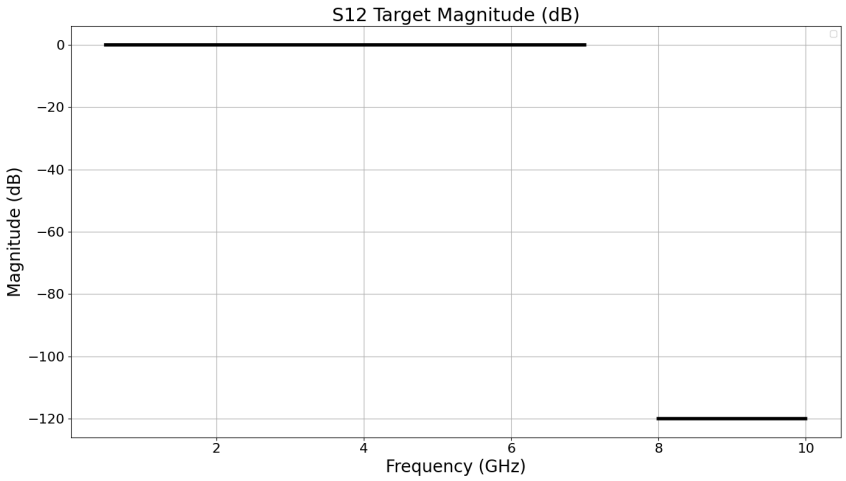

Figure 4.6: Target magnitude response for S_{12} in dB.

Using both the base model and the AL model, 10 circuit candidates were generated that attempt to meet this design objective and the best of each model where choosen. The corresponding metal layouts for the best designs from each model are shown in Figures 4.7 and 4.8. Each pixel represents the presence of metal, with a darker shade indicating conductive regions.

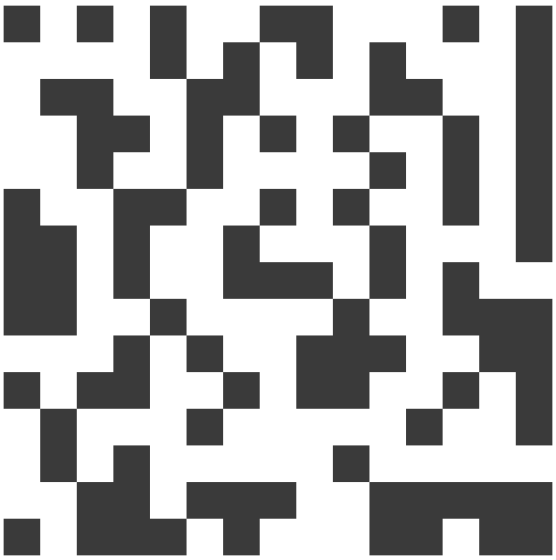

Figure 4.7: Metal layout from the base model.

Figure 4.8: Metal layout from the AL model.

Figure 4.9 compares the magnitude of S_{12} for the simulated and predicted S-parameters from each model. As shown, the design found using AL significantly outperforms the baseline. It closely matches the sharp transition at 7 GHz and achieves much stronger attenuation beyond this point.

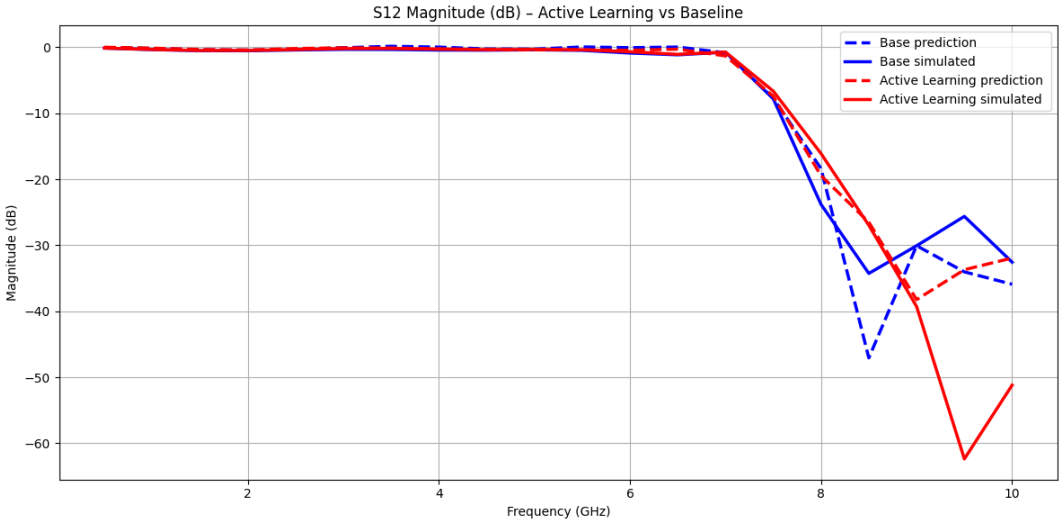

Figure 4.9: Comparison of the predicted and simulated magnitude of S_{12} for the best candidate designs from each model.

5
Conclusion

5.1 Conclusion

This study investigated the use of AL for improving the data efficiency and generalization of ML models in the context of design of passive MW circuits. Through a series of experiments comparing AL with a baseline passive learning approach, several conclusions can be drawn.

First, the results clearly show that AL significantly reduces the RMSE on the test set compared to the base model trained with randomly sampled data. This improvement was consistent across training set sizes and S-parameter components, indicating that AL enables the model to learn more efficiently by focusing on informative samples.

Second, the analysis across frequency ranges revealed that prediction accuracy tends to degrade at higher frequencies, which is expected. Still the AL model maintained lower RMSE even in these challenging regions, highlighting its advantage in identifying valuable training points.

Although ensemble models were also evaluated, they did not outperform the single AL model in this study. One potential explanation lies in how uncertainty was computed across ensemble members, which may have led to suboptimal sample selection. Due to computational constraints, each ensemble run required approximately 16 days. Due to time limitations further exploration of ensemble uncertainty remains an area for future work.

Finally, in the downstream design task involving circuit generation using a GA, the AL model again demonstrated superior performance. With AL the predicted S-parameters of GA-generated circuits resulted in a 32.9% lower mean RMSE than the base model.

Altogether, these findings support the conclusion that AL is an effective and scalable strategy for reducing data requirements and improving model generalization in ML-based MW circuit design.

5.2 Future Work

There are several interesting directions for future work based on the findings of this study. One is to explore better sampling strategies within the AL framework. The current approach could likely be improved by using smarter ways of selecting training data, such as combining uncertainty with sample diversity, using adveserial training (see Section 2.4.3.4), or looking at different ways to compute the uncertainty.

Another direction is to apply the method to larger and more complex circuit layouts.In this study the AL approach was only tested on one circuit size and layout, and it would be valuable to see how well the approach scales when the design space becomes bigger. Finally, a long-term goal is to use this method as a part of the design workflow.

Bibliography

[1] Z. Liu, E. A. Karahan, and K. Sengupta, "Deep Learning-Enabled Inverse Design of 30–94 GHz $P_{\mathrm{sat,3dB}}$ SiGe PA Supporting Concurrent Multiband Operation at Multi-Gb/s," *IEEE Microwave and Wireless Components Letters*, vol. 32, no. 6, pp. 724–727, Jun. 2022.

[2] S. Molesky, Z. Lin, A. Y. Piggott, W. Jin, J. Vučković, and A. W. Rodriguez, "Inverse design in nanophotonics," *Nature Photonics*, vol. 12, no. 11, pp. 659–670, 2018.

[3] D. Liu, Y. Tan, E. Khoram, and Z. Yu, "Training deep neural networks for the inverse design of nanophotonic structures," *ACS Photonics*, vol. 5, no. 4, pp. 1365–1369, 2018.

[4] S. So, T. Badloe, J. Noh, J. Bravo-Abad, and J. Rho, "Deep learning enabled inverse design in nanophotonics," *Nanophotonics*, vol. 9, no. 5, pp. 1041–1057, 2020.

[5] W. Ma, F. Cheng, Y. Xu, X. Wang, Y. Liu, Y. Liu, L. Zhang, and D. Liu, "Deep learning for the design of photonic structures," *Nature Photonics*, vol. 15, no. 2, pp. 77–90, 2021.

[6] T. W. Hughes, M. Minkov, I. A. Williamson, and S. Fan, "Adjoint method and inverse design for nonlinear nanophotonic devices," *ACS Photonics*, vol. 5, no. 12, pp. 4781–4787, 2018.

[7] D. M. Pozar, *Microwave Engineering*, 4th ed., Wiley, 2012.

[8] R. E. Collin, *Foundations for Microwave Engineering*, 2nd ed., IEEE Press, 2001.

[9] K. Kurokawa, "Power Waves and the Scattering Matrix," *IEEE Transactions on Microwave Theory and Techniques*, vol. 13, no. 3, pp. 194–202, May 1965.

[10] A. Ferrero and M. Pirola, "Generalized Mixed-Mode S-Parameters," *IEEE Transactions on Microwave Theory and Techniques*, vol. 54, no. 1, pp. 458–463, Jan. 2006.

[11] A. Gholami and A. Nayebi, "Wave propagation and frequency-dependent sensitivity in heterogeneous media," *Springer Nature Applied Sciences*, vol. 2, pp. 323–334, 2020.

[12] Y. Tan et al., "Design and manufacturing challenges of high-precision X-band waveguide components," *IEEE Transactions on Microwave Theory and Techniques*, vol. 70, no. 1, pp. 82–90, 2022.

[13] W. C. Gibson, *The Method of Moments in Electromagnetics*, 2nd ed., CRC Press, Boca Raton, FL, 2014.

[14] CEMWorks Inc., *Emerald Cloud-Based Electromagnetic Simulation Platform*. Available online: `https://cemworks.com/`

emerald-cloud-electromagnetic-em-simulation-platform/ (accessed June 2, 2025).

[15] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011, pp. 315–323.

[16] Kingma, D. P. and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[18] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.

[19] J. Ba, J. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[20] Saturn Cloud, "A Comprehensive Guide to Convolutional Neural Networks: The ELI5 Way," *Saturn Cloud Blog*, 2023. [Online]. Available: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/. [Accessed: June 2, 2025].

[21] B. Settles, *Active Learning*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool, 2012.

[22] D. D. Lewis and W. A. Gale, "A Sequential Algorithm for Training Text Classifiers," in *Proc. 17th Annual Int. ACM SIGIR Conf.*, pp. 3–12, 1994.

[23] Kendall, A. and Gal, Y., "What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?", in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[24] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[25] R. Pestourie, Y. Mroueh, T. V. Nguyen, P. Das, and S. G. Johnson, "Active learning of deep surrogates for PDEs: application to metasurface design," *npj Computational Materials*, vol. 6, no. 1, pp. 1–9, 2020.

[26] Y. Gal, "Uncertainty in Deep Learning," PhD thesis, University of Cambridge, 2017.

[27] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by Committee," in *Proc. 5th Annual Workshop on Computational Learning Theory*, pp. 287–294, 1992.

[28] Settles, B., Craven, M., and Friedland, L., "Active Learning with Real Annotation Costs", in *Advances in Neural Information Processing Systems (NeurIPS)*, 2008.

[29] Roy, N., and McCallum, A., "Toward Optimal Active Learning through Sampling Estimation of Error Reduction", in *Proceedings of the 18th International Conference on Machine Learning (ICML)*, 2001.

[30] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proc. International Conference on Learning Representations (ICLR)*, 2015. Available: `https://arxiv.org/abs/1412.6572`

[31] F. Garbuglia, J. Qing, N. Knudde, D. Spina, I. Couckuyt, D. Deschrijver, and T. Dhaene,
"Bayesian active learning for multi-objective feasible region identification in microwave devices,"
Electronics Letters, vol. 57, no. 10, pp. 400–403, May 2021.
DOI: 10.1049/ell2.12022

[32] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, 1996.

[33] MongoDB Inc. *MongoDB: The application data platform*, 2024. Available at: `https://www.mongodb.com`

[34] D. A. Nix and A. S. Weigend, ``Estimating the mean and variance of the target probability distribution," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN)*, 1994.

[35] M. Karahan *et al.*, "Deep Learning-Enabled Generalized Synthesis of Multi-Port Electromagnetic Structures and Circuits for mmWave Power Amplifiers," in *IEEE MTT-S Int. Microwave Symp. (IMS)*, 2024. (Accepted)

[36] K. Pearson, "Note on Regression and Inheritance in the Case of Two Parents," *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895.

[37] S. Sinha, S. Ebrahimi, T. Darrell, and R. Volpi, "Towards Calibrated and Uncertainty-Aware Neural Networks," in *MERL Technical Report*, TR2019-117, Mitsubishi Electric Research Laboratories, 2019.

[38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available at: `https://www.tensorflow.org/`.

[39] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: `https://keras.io/`

A

Baseline CNN Hyperparameters

The following hyperparameters were used for the initial baseline CNN model:

- **Convolutional Layers:**
  - Kernel sizes: [8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2]
  - Filters per layer: [128 (×12), 64 (×2)]
  - Activation: ELU
  - Dropout rate: 0.1
  - L2 regularization: 0.0
  - Pooling applied after layer index: 12
- **Dense Layers:**
  - Neurons per layer: [1024, 1024, 1024]
  - Activation: ELU
  - Dropout rate: 0.1
  - L2 regularization: 0.0
- **Output Layer:**
  - Neurons: 66
  - Activation: Linear
  - Loss function: Mean Squared Error (MSE)
- **Training Settings:**
  - Dataset split seed: 42
  - Optimizer: Adam (lr = 0.005, $\beta_1 = 0.95$, $\beta_2 = 0.999$)

B

Loss Functions Used

- **Mean Squared Error (MSE)**:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu_i)^2$$

- **Gaussian Negative Log-Likelihood (NLL)**:

$$\mathcal{L}_{\text{NLL}} = \frac{1}{N}\sum_{i=1}^{N}\left[\log \sigma_i^2 + \frac{(y_i - \mu_i)^2}{\sigma_i^2}\right]$$

- **Correlation-Penalized NLL (Corr-NLL)**:

$$\mathcal{L}_{\text{Corr-NLL}} = \mathcal{L}_{\text{NLL}} \cdot \left(1 + \frac{1}{\rho^2}\right)$$

- **Soft Correlation-Threshold NLL (Final)**:

$$\mathcal{L}_{\text{Final}} = \mathcal{L}_{\text{NLL}} + \alpha \cdot \max(0, \rho_{\text{threshold}} - \rho)^2$$

C

Evaluated Model Configurations

The following list summarizes all 18 CNN configurations tested during model selection. Each entry specifies convolutional depth, filter counts, dense layers, regularization, and activation strategies used.

Large Baseline Model (Full Capacity) `Conv kernels: [8,8,7,7,6,6,5,5,4,4,3,3,2,2]`
`Filters: [128×12, 64×2]`
`Dense (mean): [1024, 1024, 1024]`
`Dense (variance): [256, 128, 64]`
`Dropout (CNN/Dense): 0.1 / 0.1`
`L2 (CNN/Dense): 0.0 / 0.0`
`Activation: ELU`

Medium Model (Reduced Layers & Dense) `Conv kernels: [8,7,6,5,4,3,2,2]`
`Filters: [128×4, 64×2, 32×2]`
`Dense (mean): [512, 256]`
`Dense (variance): [128, 64]`
`Dropout: 0.15 / 0.2`
`L2: 0.0 / 0.0`
`Activation: ELU`

Lightweight Model (Even Smaller) `Conv kernels: [8,6,4,3,2]`
`Filters: [128,128,64,32,16]`
`Dense: [256,128]`
`Variance: [64,32]`
`Dropout: 0.2 / 0.3`
`L2: 0.0001 / 0.0001`
`Activation: ReLU`

Alternative Activation (LeakyReLU) `Conv kernels: [8,7,6,5,4,3,2,2]`
`Filters: [128×4, 64×2, 32×2]`
`Dense: [512, 256]`
`Variance: [128, 64]`
`Activation: LeakyReLU`

Stronger Regularization `Dropout: 0.3 / 0.4`
`L2: 0.001 / 0.001`

Fewer Conv Layers, More Dense Capacity `Conv kernels: [8,7,6,4,3]`
`Dense: [1024, 512]`

Deep Regularized CNN `Conv kernels: [7,6,5,4,3,2,2]`
`Filters: [128×3, 64×2, 32×2]`

Dropout: 0.3 / 0.4
L2: 0.001 / 0.001
Activation: ReLU

Small Efficient CNN Conv kernels: [5,4,3]
Filters: [64, 64, 32]
Dropout: 0.4 / 0.5
L2: 0.002 / 0.002
Activation: LeakyReLU

Wide Kernel CNN Conv kernels: [9,7,5,3,2]
Activation: Swish

Hybrid CNN Conv kernels: [8,4,3,2,2]
Dense: [512,128]
Variance: [128,32]

Aggressive Dropout CNN Dropout: 0.5 / 0.6

Depth Over Width CNN Conv kernels: [5,5,4,4,3,3,2,2]
Filters: [64×4, 32×2, 16×2]
Dropout: 0.3 / 0.3

MobileNet Inspired Conv kernels: [3,3,3,3]
Filters: [64,64,128,128]
Dense: [256,128]

ResNet Inspired Conv kernels: [7,5,5,3,3,3]
Filters: [64,128,128,128,64,32]

Transformer Hybrid Conv kernels: [5,5,3]
Activation: Swish

Large Feature Extraction Conv kernels: [9,7,5,3,2]

Super Small Efficient Conv kernels: [5,3]
Dense: [128, 64]
Dropout: 0.5 / 0.5

Optimized Small Efficient CNN Conv kernels: [5,4,3]
Dropout: 0.45 / 0.5
L2: 0.003 / 0.003

D

Final Model Configuration

The table below summarizes the full hyperparameter configuration used for the final model:

- **Convolutional layers:**
  - Kernel sizes: 8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2
  - Number of filters: 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 64, 64
  - Activation: `elu`
  - Use pooling: `True`
  - Pooling applied after layer index: 12
- **Dense layers (mean and variance branches):**
  - Number of neurons: 1024, 1024, 1024
  - Activation: `elu`
- **Regularization:**
  - Dropout (CNN layers): 0.1
  - Dropout (dense layers): 0.1
  - L2 regularization (CNN layers): 0.0001
  - L2 regularization (dense layers): 0.0001

DEPARTMENT of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY