הטכניון – מכון טכנולוגי לישראל
הפקולטה להנדסת חשמל
המעבדה לתקשורת

ספר פרויקט

תכן מסננים באמצעות מערכות לומדות

Final Report

Machine Learning Based Filter Design

מגישים:

אילון פררו

אריק גרין

בהנחיית:

דן פישלר

Final Project Report

Aylon Feraru, Eric Green, Dan Fishler

Acknowledgements

We would like to thank Dennis Dikarov for his advice and support throughout the project.
Emanuel Cohen for his ideas that propelled it forward and agreeing to meet us multiple times.
Benny Hadad for his help with AWR, from instructing us to pushing it in an unexpected direction.
Ori Bryt for his help with understanding the article and agreeing to meet us.
Omer Malka for his help with computing power.
We thank you all and are very appreciative of your help, dedication, and belief in us.
-Aylon and Eric

Contents

List of Figures

1 Abstract

Machine learning techniques have transformed numerous scientific disciplines - from alternative methods and enhanced performance to unlocking previously unavailable design spaces and capabilities, including in the field of RF design.
In this work, we built upon, and independently implemented, the Machine Learning based approach established by Emir Ali Karahan et al. in [1–3] to the problem of Radio Frequency filter design, specifically trying to design a filter that fulfills the following requirements:

$$s_{11} \leq -15dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$
$$s_{21} \geq -0.5dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$
$$s_{21} \leq -15dB \qquad 13.7 \leq f \leq 14.5 \text{ GHz}$$
$$s_{22} \leq -15dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$

To innovatively solve the problem, we've created a new design space by simulating and pixelating an RO3003 substrate and the metal layer over it in AWR. We've iterated over many random arbitrary such structures, aggregating them in a dataset of 450,000 16×16 (as well as some 8×8) samples to train a Neural Network that estimates the arbitrary 16×16 structure's scattering parameters with precision of 0.38 MAE in dB.
We created a genetic algorithm to find an optimal filter design over the new space. Our filter design has managed to get close to the desired target, and we're confident that with more training samples or an expanded design space, we'd have surpassed the requirements, as the model consistently shows improvement with added samples. A big barrier however, is long training time, and smaller but impressive dataset, due to a lack of sufficient computing resources. Our code can easily be adjusted to new dimensions and filter requirements, and once training is completed new designs can be generated in an hour.

2 Theoretical Background

2.1 Differences from Conventional Design Techniques

The process of designing electromagnetic structures for high-frequency circuits often begins with selecting a specific configuration of functional elements. These elements are then fine-tuned to achieve the desired scattering parameters, typically using extensive parameter sweeps, various optimization techniques, or leveraging the knowledge and expertise of seasoned professionals. This method presents some drawbacks - the parameter sweeps are time-consuming, and the finite set of elements isn't guaranteed to reach the best performance - essentially, the design space is limited, time-consuming, and requires expert knowledge.

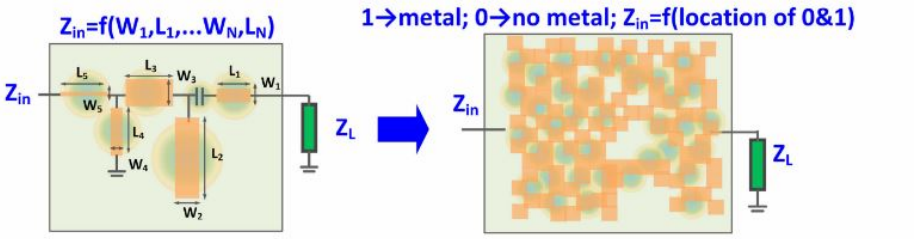

Figure 1: Usual design method in comparison with a ML based design method, taken from [1]

2.2 Relevant Works, and What Sets Ours Apart

The use of Machine learning in the field of RF has spawned several interesting studies we've examined and that related directly to the problem we're trying to solve. in 'Sophisticated Electromagnetic Scattering Solver Based on Deep Learning' and 'Predicting Scattering From Complex Nano-Structures via Deep Learning'- Yinpeng Wang et al. [4] [5] an accurate ElectroMagnetic scattering predictor model is constructed from an altered U-net architecture - the article solves a similar problem with similar means. Some of the techniques suggested in it such as Residual connections and Convolution Networks were applied in possible designs to our problem. In 'Deep-Learning-Based Inverse-Designed Millimeter-Wave Passives and Power Amplifiers' [1] Emir Ali Karahan et al. design a power amplifier using a CNN

and a genetic algorithm, in the work they estimate the real and imaginary s_{11}, s_{12}, and s_{22} over 9 frequency points. Their genetic algorithm optimizer design has inspired ours, and they predict all the scattering parameters like we do - but their predictions aren't in Decibel. In 'Tandem Neural Network Based Design of Multiband Antennas' - Aggraj Gupta et al. [2] arbitrary electromagnetic structures are designed to solve the problem of antenna design. As a result, they estimate s_{11} in Decibels, and their CNN construction proved a very good basis to our finished model. In 'Deep Learning based Modeling and Inverse Design for Arbitrary Planar Antenna Structures at RF and Millimeter-Wave' [3] Emir Ali Karahan et al. expand on how they've designed their CNN.
Our work is set apart from the relevant works by our unique implementation in different platforms, our problem of filter design which the articles don't tackle, and our solution method is a blend of techniques from different articles, and originality. We also improved on the design by introducing point-wise convolutions to great success in the prediction.

In comparison, the method of Machine Learning based optimization presented in [2] [1] offers a vast and previously untapped solution-space by pixelating the metal layer that sits on top of a substrate and acts as a new design space. The design space is very vast, for instance - a 16×16 pixel layer presents 2^{256} new electromagnetic structures that can act as Power Amplifiers, Antennas, and in our case: Filters.
Once the Neural Network is trained and the framework is complete, the design method we've implemented can create new designs without requiring external knowledge, in a negligible amount of time (about an hour). Our method can feasibly reach further optimized solutions than current ones, we can conclude that from seeing the performance consistently improve with the addition of data, and that the foundational works that implemented similar methods have exceeded state of the art performance.

2.3 Scattering Parameters

Scattering parameters, or S-parameters, are fundamental measurements used in electrical engineering to describe the behavior of electrical networks, particularly in high-frequency circuits. These parameters represent how signals behave as they encounter various components within the network, such as amplifiers, filters, and antennas. The parameter s_{11} represents the reflection coefficient at port 1, indicating how much of the signal entering port 1 is reflected back. The parameter s_{21} and s_{12} in the reverse direction) represents the transmission coefficient from port 1 to port 2, showing how much of the signal passes through the network from one port to the other. For passive structures, which do not amplify signals, s_{21} and s_{12} are equal due to the reciprocal nature of these components. Lastly, s_{22} is the reflection coefficient at port 2, similar to s_{11} but at the second port.

2.4 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of deep learning network specifically designed for processing structured grid data, such as images, and in our case the geometry of the metal layer in an arbitrary EM structure. CNNs are particularly effective for tasks involving image recognition and classification due to their ability to capture spatial hierarchies and patterns. CNNs contain Convolution layers, in which filters with learnable parameters perform convolution with the input. The input, output, and filter dimensions are described by height, width, and channels. Channels are the amount of values each pixel contains, for instance: A gray-scale image has one output channel. A colored image has three values, Red Blue and Green - one for each channel. As the network learns the channels are used to contain different features important to the image. CNNs also contain Fully Connected layers (FCs). Variations to the CNN structure can include Batch Normalization layers, Pooling Layers, different Padding to the convolution with the learnable filters, skip connections, choice of activation functions and more. We'll briefly discuss the layers Important to understanding our designs:

- Activation functions are point-operations, that are used to break linearity between proceeding layers.
- Pooling layers reduce the size of the input data by summarizing nearby outputs in a feature map. Max pooling selects the highest value, while average pooling computes the mean of values in the region. This reduces computation and helps make the model invariant to small translations in the input. They've been fundamental to the development process of the CNN, and also were eventually to the S-parameter predictor's detriment.
- Batch normalization (BatchNorm) layers are a type of layer used in neural networks to improve the training speed, stability, and performance. They work by normalizing the inputs to a layer for each mini-batch.
- Dropout layers are a type of regularization technique used in neural networks on FC layers to prevent over-fitting. They work by randomly setting a fraction of the outputs to activation functions to zero during training, which forces the network to learn more robust features and prevents reliance on specific neurons.

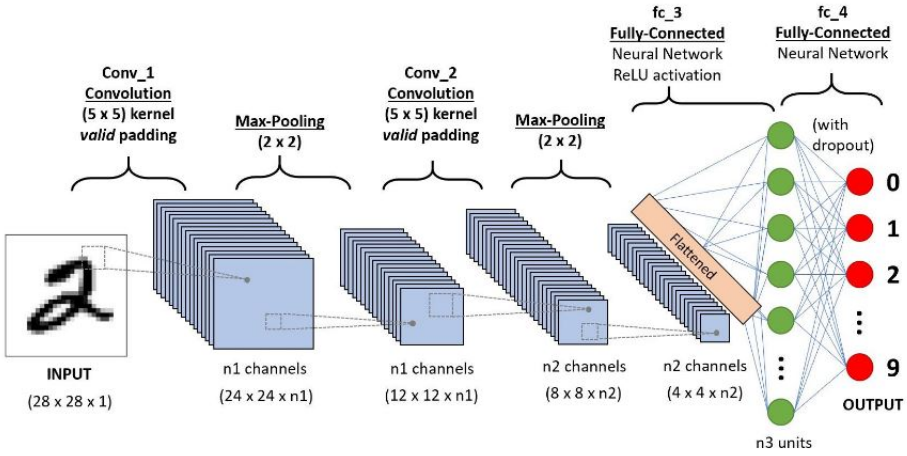

Figure 2: An Example of a CNN (MNIST digit classifier)

Maxwell's Equations and the Electromagnetic relations between the pixels from which the S-Parameters are derived depend heavily on the locations of pixels in relation to one another [1], which makes a CNN architecture a good choice for a rapid EM simulator.

2.4.1 Computing Convolution Dimensions

Motivation: When designing a CNN it's important to be mindful of how convolution layers affect the input sizes of subsequent layers. If one (such as me) does not pay attention, it may incur dimension error such as when the operands reduce the input dimensions to 0. Moreover, if one does not use Adaptive Pooling layers (which force the layers output to be a predetermined number), it's pertinent to know the dimensions of the input before connecting it to an FC layer as inserting a wrong value will cause an error.

$$W' = \left\lfloor \frac{W - K + 2 \cdot P}{S} \right\rfloor + 1$$

where W' is the width of the output layer, W is the width of the input layer, K is the Kernel size (for the kernel with learnable weights used in a convolution layer), P is the amount of padding used, and S is the stride (for a stride of two, the result of the convolution operator is registered only for every second pixel, creating a smaller image. A stride of one is a regular convolution).
When designing a Convolution layer it's important to keep in mind the following two results:

1. The number of channels in the input image = The number of channels in each learnable filter
2. The number of channels in the output image = The number of learnable filters

These results help us understand how many parameters our network and our output will contain, and help us design networks with correct dimensions.

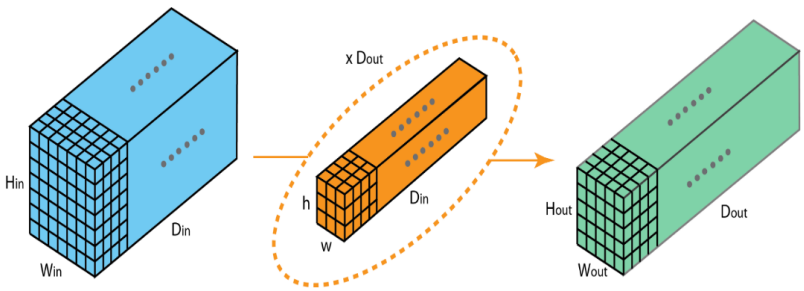

Figure 3: Convolution Input, Kernel, and Output Channels, taken from towardsdatascience.com

2.4.2 Prediction Problem Formalism

We'll later expand on the CNN's role in our design, but for now, let's define the problem we want it to solve. A CNN can be thought of as a function that takes an image and, based on weights w, produces an output (or several outputs) f_w. By training the function, we find the optimal weights that minimize another function called the loss function $\mathcal{L}(f_w)$.

It's important to note that our function takes in a matrix representing a metal layer and outputs a matrix that estimates S-parameters at different frequencies.

If we denote the arbitrary single-channel $N \times N$ binary matrix representing our metal layer as B, and the matrix of S parameters in our chosen frequencies as S, the problem we're trying to solve is:

$$\text{argmin}_w \, \mathcal{L}(f_w).$$

To formalize the problem using a CNN to predict S-parameters over chosen frequencies, we define our objective as minimizing the Mean Absolute Error (MAE) loss function:

$$\mathcal{L}_{\text{MAE}}(S, f_w(B)) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} |S_{ij} - f_w(B)_{ij}| \,,$$

where S_{ij} represents the true S-parameter values over the chosen frequencies, and $f_w(B)_{ij}$ represents the predicted frequency values from the CNN at position i, j.

To prevent overfitting and encourage generalization, we add L2 regularization to the loss function. L2 regularization penalizes large weights by adding a term proportional to the sum of the squared weights to the loss function:

$$\mathcal{L}_{\text{L2}}(w) = \lambda \sum_{i,j} w_{ij}^2,$$

where λ is the regularization parameter that controls the trade-off between the original loss and the regularization term, and w_{ij} are the weights of the CNN, including those in both convolutional and fully connected layers. The overall loss function becomes:

$$\mathcal{L}_{\text{total}}(f_w) = \mathcal{L}_{\text{MAE}}(S, f_w(B)) + \mathcal{L}_{\text{L2}}(w)$$

A typical value for λ is 10^{-5}, which was used in this project too.

Additionally, we consider dropout as another form of regularization added to the fully connected layers in our architecture to prevent overfitting. Dropout involves randomly setting a fraction p of the activations to zero during training, which helps in making the model more robust and less sensitive to the noise in the training data. During training, for each layer, we apply:

$$\text{Dropout}(h_{ij}) = \begin{cases} 0 & \text{with probability } p, \\ h_{ij} & \text{with probability } 1 - p \end{cases}$$

where h_{ij} are the activations of neuron i,j in a given layer.

In summary, the optimization problem we aim to solve with our CNN, considering MAE loss, L2 regularization, and dropout (only during training), is:

$$\text{argmin}_w \left(\frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} |S_{ij} - f_w(B)_{ij}| + \lambda \sum_{i,j} w_{ij}^2 \right)$$

Interestingly, regularization prevents double descent, yielding direct improvements when training samples are added. The loss function is used in an optimizer like ADAM, a version of batched gradient descent with moment estimations. Though we switched to ADAMW to decouple weights.

2.5 Genetic Algorithms

Genetic Algorithms (GAs) or Evolutionary Optimization Algorithms [6] are optimization algorithms that mimic natural selection and mutation. In these algorithms an array of possible solutions called Population. The population is updated and replaced by a new set of solutions produced by iterative use of genetic operators on individuals present in the population. The members of the population in one specific update are called a Generation. The fitness function is a function represents how well a particular solution solves the optimization problem, and is used to select members of a

generation for the genetic operators.
Every Generation gets repopulated by solutions that originate from:

- **Selection**: Every selection operator selects from the population two solutions, which are combined to create a new solution. A random operation called mutation is applied on the solution to encourage variety, and the solution is placed in the new generation.
- **crossover**: The fittest K performing members of a previous generation remain in the population. This ensures a monotonically rising fitness score in the best performing filter design.

3 Solution Overview

3.0.1 Solution Layout

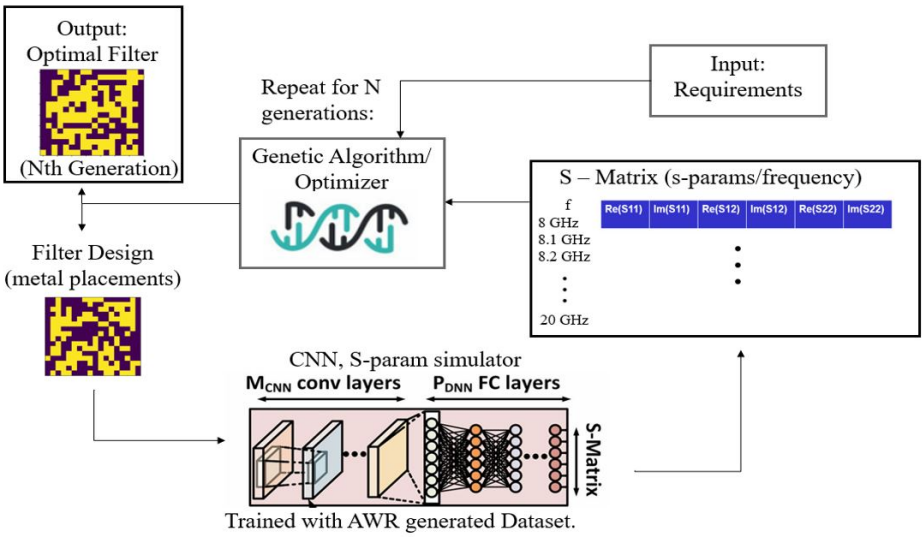

Figure 4: A schematic describing our filter design approach

The solution method is comprised of several programmatic components, that are built on top of each other.
first, we build a dataset of arbitrary Electromagnetic Structures inside AWR with an automated script that was coded in python using an API, and visual basic, inside of AWR itself. The script generates sets of samples, due to the amount of samples and complexity of EM simulation the set generation requires an immense amount of computation and generation time.
We export the sets to csv files, which we subsequently open in Python. Each such .csv file contains $10,000$ samples (unless a run was interrupted) and their scattering parameters over 195 frequency points.
Inside Python, we perform prepossessing - both changing the way the data is presented to be compatible with Python, and using passivity to triple the amount of samples we have.
The data, now saved as a PyTorch (A Python machine learning library) dataset, is used to train a convolutional neural network to estimate the frequency response parameters of the structures.
The weights of the trained model are saved, and the model is loaded into a Genetic Algorithm which optimizes designs based on their predicted frequency response by the model.

In this section, we'll delve into each component that partakes in our filter design approach - we'll start with the design process, choices and evolution the component has gone through, and we'll finish by presenting the final component.

3.1 Dataset Generation

To design the dataset we had to take into account the dimensions of the pixels, to induce the desired wave properties in them conducive to filter design. More specifically, we wanted pixels that are small enough in compare to the

typical wavelength λ of our frequency range, to avoid unwanted wave effects and in the same time large enough to be compatible for production. The middle frequency in the range of our interest is $f = 12.7$ [GHz]. the pixel size we chose is 10 [mil], or 0.000254[meter]. The Permittivity of our metal is $\varepsilon_r = 3.66$. so the following is fulfilled:

$$\lambda = \frac{C}{f \cdot \sqrt{\varepsilon_r}} = \frac{3 \cdot 10^8}{12.7 \cdot 10^9 \cdot \sqrt{3.66}} = 0.01234[meter]$$

$$\lambda = 0.01234\,[\text{meter}] \gg 0.00025\,[\text{meter}] = \text{pixel size}$$

The wavelength λ is also significantly larger compared to the whole structure of 16 pixels long:

$$\lambda = 0.01234\,[\text{meter}] \gg 0.004\,[\text{meter}] = \text{structure length}$$

3.1.1 Development Process

MATLAB

We explored several options in parallel before opting for the use of Python and AWR for the generation, one such option is the use of MATLAB's antenna toolbox. We managed to create script for creating random arbitrary structures in MATLAB and estimating its S-parameters.

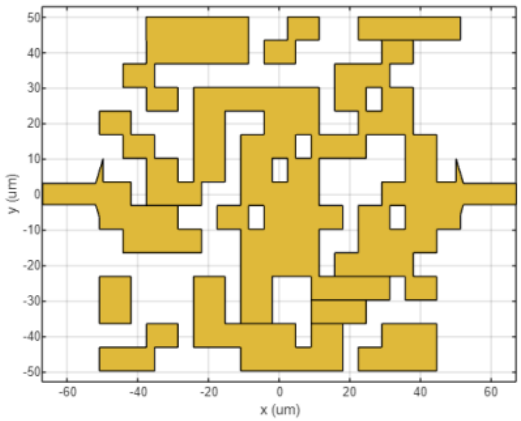

Figure 5: An arbitrary structure simulated in MATLAB

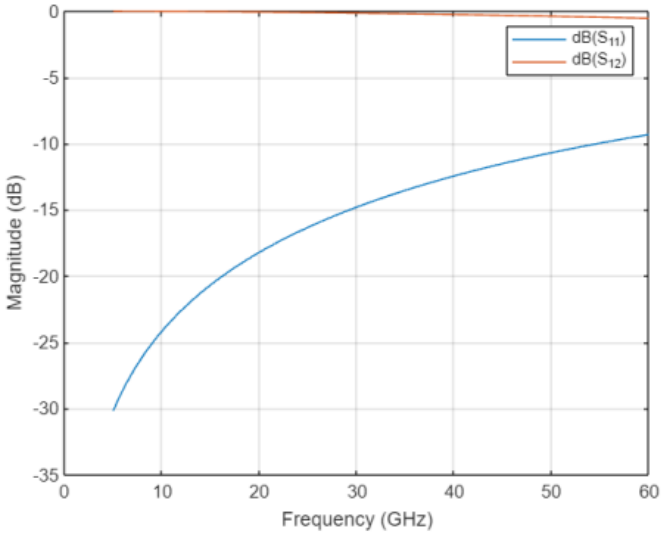

Figure 6: Matlab-Simulated S-parameters over an arbitrary structure

We eventually opted out of using MATLAB in favour of the other approach due to Python's interact-ability with AWR and PyTorch, and we opted for PyTorch for the ML sections of the project due to its very robust community, and

personally finding it more intuitive.

Choosing AWR's simulation mode:
A different development hurdle is, as stated in the abstract, the long simulation times in conventional EM simulators. While our approach intends to circumvent these simulation times, we still need to generate hundreds of thousands of simulations of these arbitrary EM structures. To simulate them, we opted for the 2.5D electromagnetic Method of Moments as opposed to the full 3D solver AXIEM, becuase comparing the two we found that MoM boasts faster simulation times and has no significant deviations in the Scattering Parameters for the structures, as shown below:

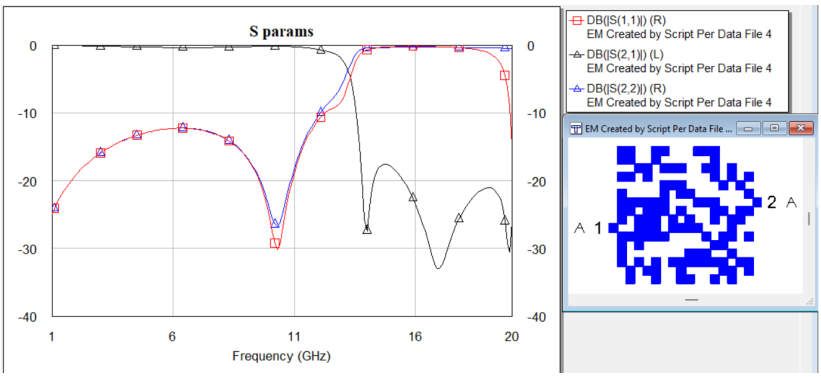

Figure 7: EM simulation results using AXIEM

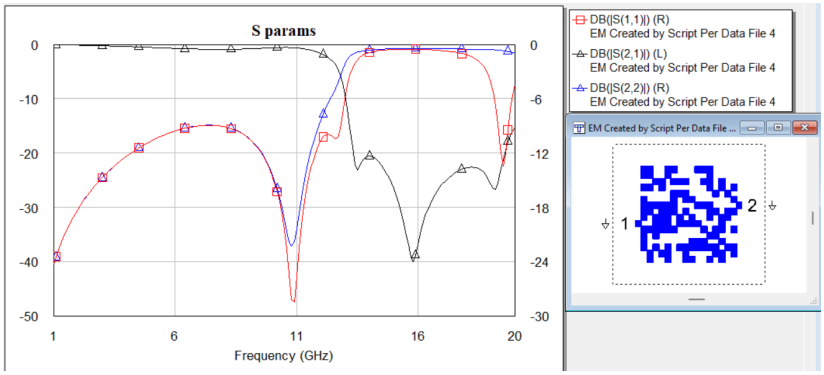

Figure 8: EM simulation results using Analyst

It's important to note, that the resources allocated to us hadn't been able to generate a dataset in a comparable size to the ones used in the articles [2] [1]. We're also interested in predicting $(s_{21})dB$, which is far more volatile in its behaviour and magnitude than s_{11}, the prediction in Decibels induces volatility as well, as we've seen when comparing predictions in Re/Im and dB.

3.1.2 Definitive Approach

In the following subsection we will describe the entire process of generating our dataset. Each underline contains the actions that were performed in the corresponding environment.

Python

We began by establishing connection between Python and AWR, using the pyawr Python package. We subsequently initialized a random matrix of $N \times N$ (16×16 in our final solution). The matrix presents the metal layer's layout on the Filter. The binary matrix that contains the values '0' or '1' corresponding to 'metal' and 'no metal', and a vector which holds two port locations (two numbers between 1'and 16), for the height of the input and output of the structure. The matrix and the ports vector are then saved to two separate dedicated files. These files are linked to AWR so it can read their contents for later use.

AWR

After saving and linking the two files, we use Python to run a Visual Basic script we wrote inside AWR. This script loads the created matrix and ports vector from the mentioned files into two variables. Then it sets the environment for EM (electromagnetic) simulation, such as substrate, type of metal, simulator (AXIEM), number of pixels in each dimension of the structure ($X = 16$, $Y = 16$), size of every pixel (10[mil]), size of the overlap between neighbor pixels (1[mil]) and more. All listed parameters can be easily adjusted to different values, in order to match any physical requirements of the structure. After setting all the definitions for a simulation, our VB (Visual Basic) script begins to implement the loaded structure from previously loaded files. Both the matrix and the port vector are loaded as a single row text string and are held in separate variables. Then they are converted into an array of doubles. We implement these variables as EM structure. The structure will appear as a 16x16 matrix of square metal pixels, located on a substrate. Every '0' in the given matrix represents an empty square pixel (without metal) on the substrate. The ports of the EM structure will be set in the same way, according to the ports vector.
We start by positioning two ports on the edges of both sides of the area where the EM structure will be situated. As mentioned above, the matrix now appears as one dimensional array of doubles. So in order to implement it as a two dimensional structure, we used two nested for loops with a common parameter 'i' to run over the array, so 'i' represents the index of a current array member. The first loop determines the row number (X_{location}, or X_location in code) of a current member in the array by calculating:

$$X\_\text{location} = (i \mod 16)$$

The second loop determines the column number (Y_location) of the current array member by calculating:

$$Y\_\text{location} = \lceil (i+1) \mod 16 \rceil$$

After obtaining the desired X and Y location of an array member, if it's value is '1', we proceed to place a metal pixel in the correspondent place in the EM structure. For a '0' we leave it empty. Completing the process results in an EM structure. After the structure is created, the VB script runs AXIEM simulation inside AWR and obtains the structure's S-Parameters.

For example, given the following matrix:

$$\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0
\end{array}$$

and ports vector: $(9, 7)$

The created EM structure will be:

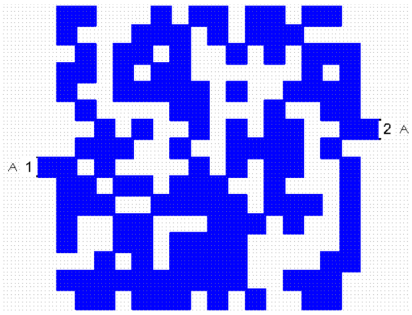

Figure 9: An Example of an EM structure, generated for a given matrix and ports vector

The results of the EM simulation of the shown EM structure, as they are presented in AWR:

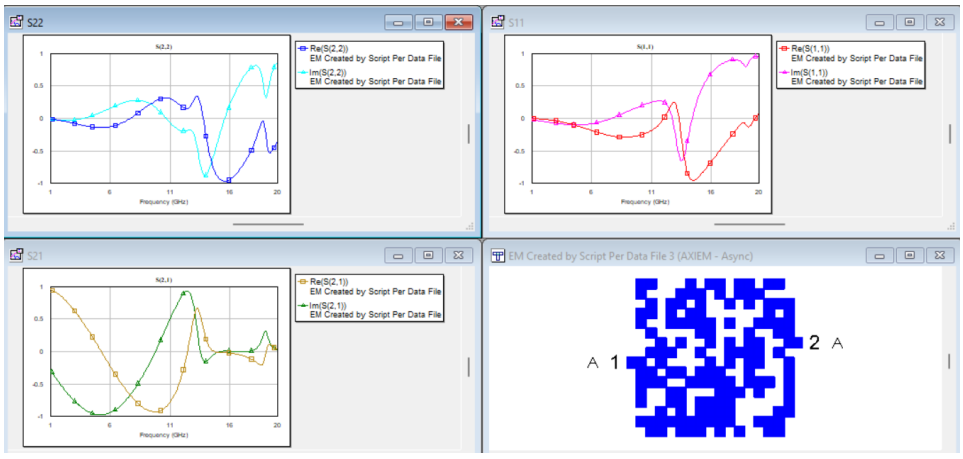

Figure 10: Simulation results for the shown example

Python

Our Python script is set to wait until it receives an indication that the simulation is completed. Then we import the data for every S Parameter from AWR, to our code in Python. For each one of them we store the data in two dimensional vectors. The first vector holds the Real part of the S Parameter against the corresponding frequency. The

second vector holds the Imaginary part of the S Parameter against the corresponding frequency. Thus in total six vectors are extracted.

PostgreSQL

After obtaining the data for a given EM structure, we store it in a database called PostgreSQL. We chose it because it is one of the most used databases among data analysts. It supports storage of a very large amount of data and provides easy access to it. We import the data for every created EM structure, from Python to Postgres, with the use of psycopg2 library in Python. This library allows us to communicate between Python and Postgres, as well as using SQL commands inside Python to save the data of every EM Structure inside PostgreSQL. All the data is saved in table, where every row contains data of a different EM Structure measurement. In order to work with the data, we used PgAdmin. PgAdmin is an open-source management tool which helps to interact with PostgreSQL databases through a graphical user interface (GUI). After all data is created, simulated, gathered and stored in our Database, we can present as following:

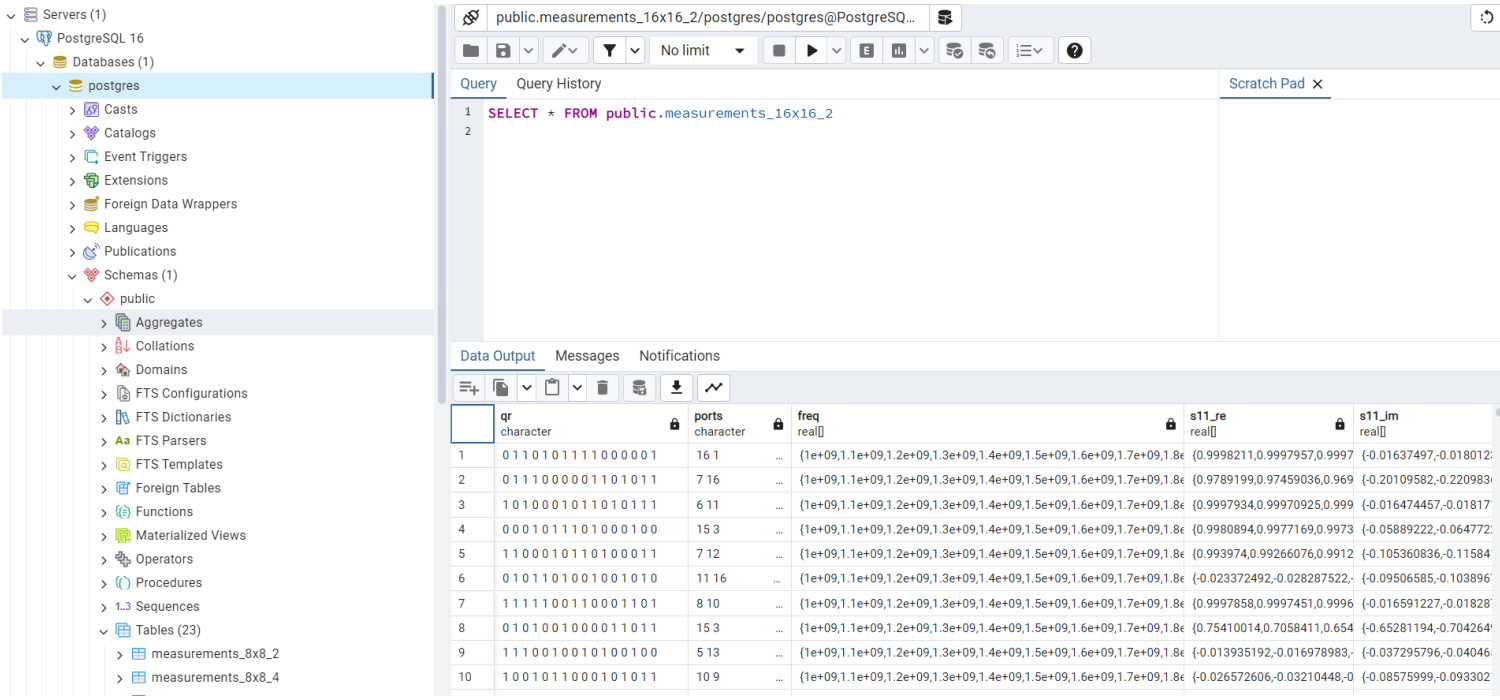

Figure 11: The way our Dataset is saved and can be viewed in PgAdmin

3.2 Dataset Preprocessing

The datasets generated in AWR are imported to Python via csv files (each contains about $10,000$ samples. each file is the output of a separate AWR run). A Python script Using Pandas (a python data analysis and manipulation package) then combines them to create a single dataset, going through each sample and changing its formatting from AWR's to a Python format.
for example, an array of S-parameters in AWR looks like $\{a, b, c, \ldots, d\}$.
Converting it to Numpy's format: array([a,b,c, ..., d]).
A similar but more complex process is performed on the matrices that represent the metal layer. At this point we recombine the matrix with its frequency response inside a PyTorch Dataset structure, from which we'll subsequently load it and train/validate/test the CNN network with it.

Another interesting operation we perform on the dataset at the pre-processing stage is tripling it by using Both Horizontal and Vertical s symmetries to learn the scattering parameters of different structures without having to simulate them , due to the passive nature of our component $s_{21} = s_{12}$. This lets us significantly cut down on the extensive dataset creation time.

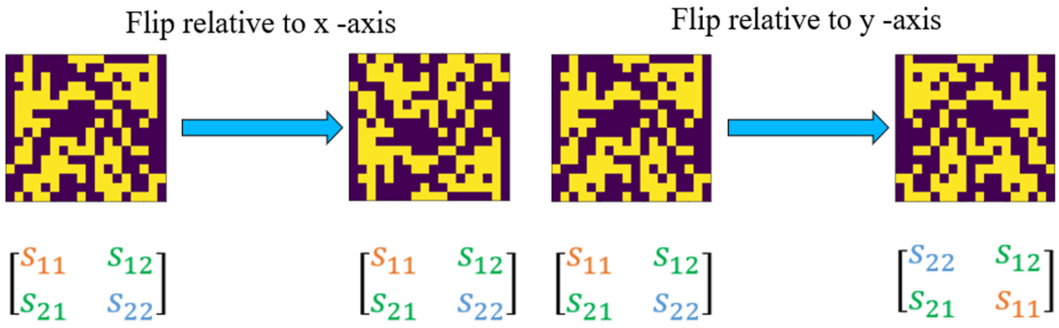

Figure 12: Using Symmetry and Passivity to triple the dataset

3.3 The Convolutional Neural Network

3.3.1 Development Process

The development process for the neural network was very long, the network has undergone many different version changes and score functions based on our changing understanding of the problem. For the sake of brevity, I'll only highlight the significant stages in its development, as well as explain the development workflow.

Simplified_CNN

At the start of the CNN development process, the goal was to estimate the scattering parameters in their Cartesian forms, as we've largely based our efforts on [1].The thinking was that a good prediction would translate into a good prediction in Decibels as well. Our dataset was smaller $(10,000)$.

At that point in the project we didn't have computing resources as well - not for training the models nor for AWR's simulations (intel i5 processor and no Graphical Processing Unit). As a result training initially failed: our first model, design inspired by [1], which had 12 convolution layers and 5 FC ones has overfit dramatically - outputting noise.

To alleviate the problem we opted for a simpler model we named Simplified_CNN, a convolutional neural network that had pooling layers which gradually reduced the Image dimensions. Simplified_CNN had 7 Convolution layers and 1 FC layer. We also used convolutions with a kernel size of 3 and padding of 1, which maintain input dimensions. We kept a tanh activation function in the end since S-parameters are confined to $[-1,1]$. We trained the model with the Adam optimizer (industry standard) and discouraged overfit with aggressive dropout of 0.7. At that point we wanted to use a 16×16 arbitrary structure size, but their simulation times with our computers proved unfeasible at the time. After a meeting with Emanuel we'd decided to start by generating much smaller 8×8 structures, this had allowed us to build a dataset faster.

After implementing the symmetry into the pre-processing stage, and building a marginally bigger dataset of 8×8 structures, our training samples tripled, and at that point the network learned well, and we were pleased with the initial results.

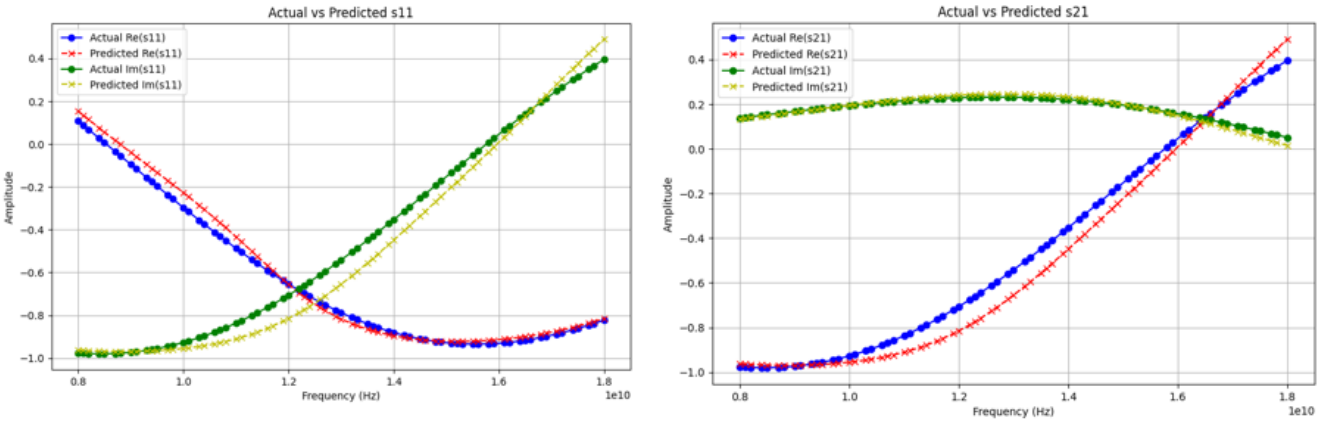

Figure 13: Simplified_CNN's performance on a good sample

Though upon further inspection of individuals the CNN had missed several test samples by a wide margin, and conver-

sion to Decibels seems to have completely wrecked the model's prediction unlike what we'd hoped - as the conversion "blows up small differences".

To improve accuracy, we ran trained and compared several different variants of Simplified_CNN to see if we can improve performances. Though sometimes we'd gained improvement, the performance stayed relatively the same in Re/Im, and bad in dB (around $20 - 30$ dB MSE).

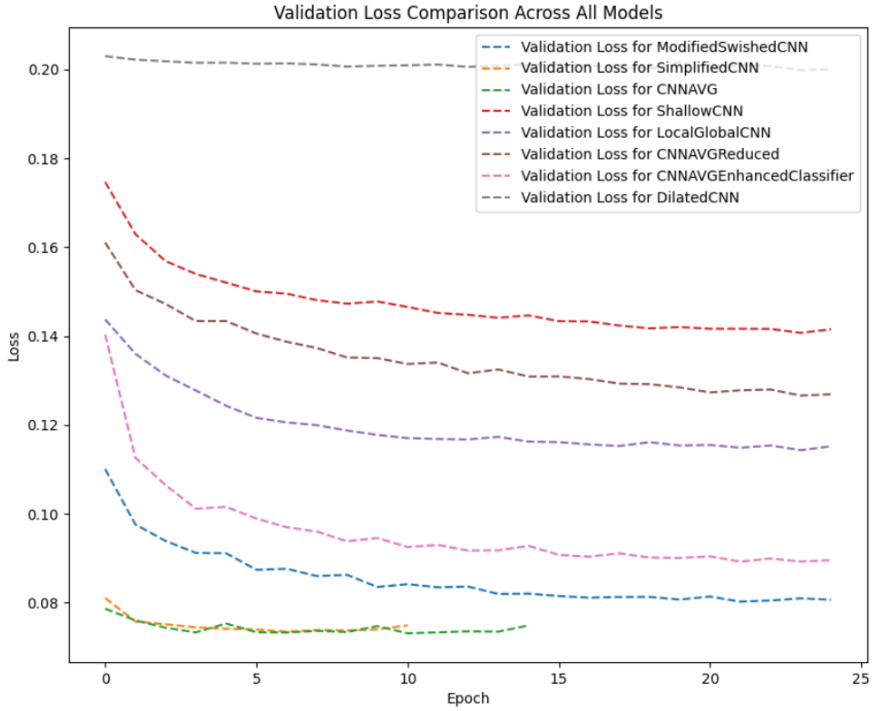

Figure 14: An example of a run testing CNN model variants, one of many we did.

Conversion to dB in forward Pass and Custom Loss Functions Since attempts to improve on Simplified_CNN didn't seem to yield success, we tried instead to implement the conversion to Decibels in the forward pass of the model, before instead of the tanh and after it. And subsequently compute loss during training in Decibel.
The idea was that training the model with a decibel output instead of training it with a conventional loss function over Re/Im would yield better results.
The training took too long to complete in the former case, in the latter case the model had outputted static noise around 0 - training wasn't efficient. I later came to realize this was likely due to vanishing gradient, though at the time I'd considered it over-fit as it seemed similar to the outputs in other cases. Since training took significantly longer I decided to pursue different avenues.
One such Idea was to train with a cost function that takes Decibels into account, but also the Re/Im loss.

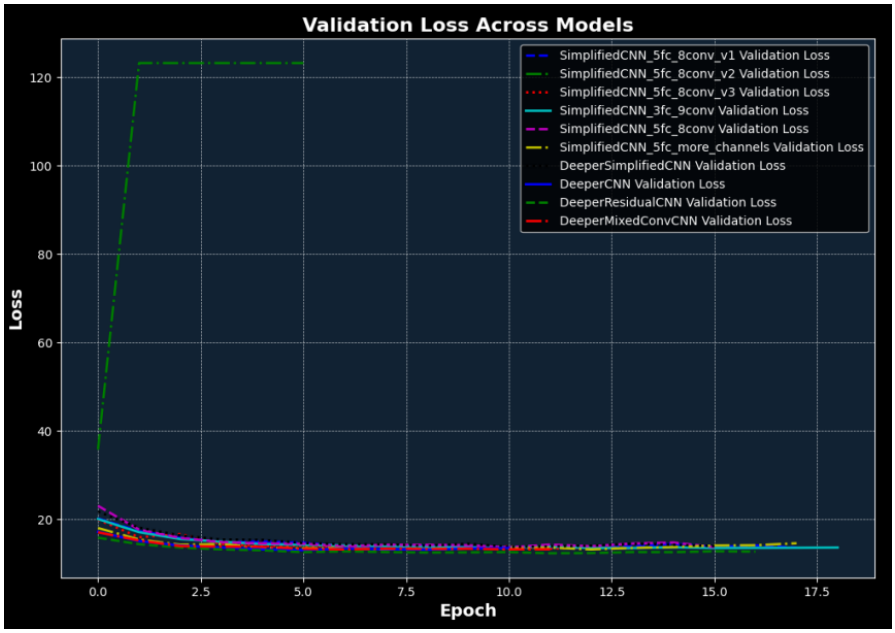

Figure 15: Training Graph with a Combined Cost Function

The approach yielded improvements, but the predictions were still far below a level we're comfortable optimizing over.

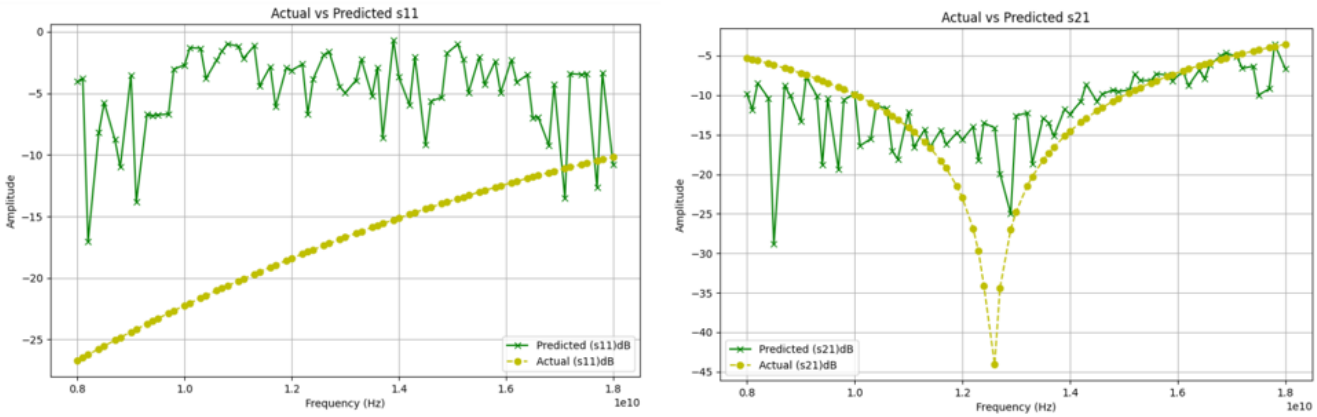

Figure 16: s_{11} and s_{21} Prediction over a sample of a model trained with a dB based loss function

As you can see, the predictions in dB seem jittery, as opposed to the smooth predictions in Re/Im the models provide, so as an attempt to get the best of both world's I've ran a simulation with a cost function that's a weighted sum of the two. The weighted sum heavily favoured the Re/Im loss, as we'd reasoned the model will have an easier time training over it, and would optimize over the finer details with dB.

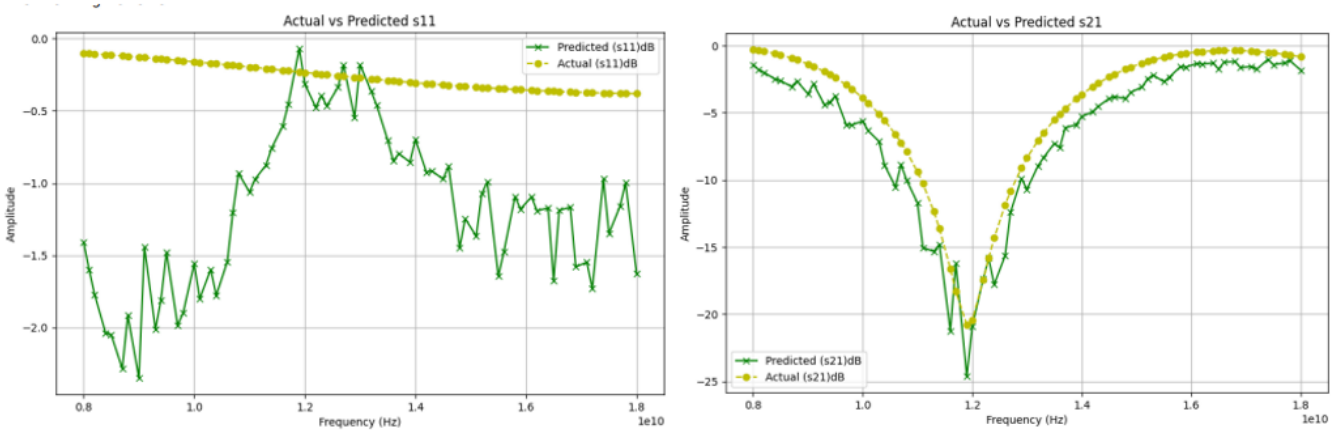

Figure 17: A test sample and prediction over the combined loss in dB

The hypothesis did work somewhat, though the predictions leave much to be desired (this is a relatively good test sample).
Additionally, we tried to penalize jitter by adding an error component based on whether two following predictions are far away from one another - but that penalty resulted in smooth predictions that don't track the parameters well.

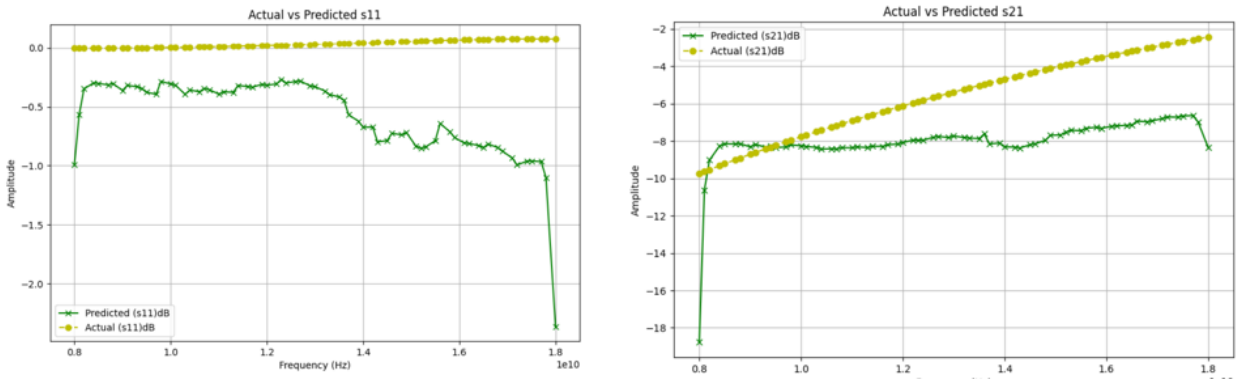

Figure 18: Prediction with a smoothness penalty.

We'd then attempted to use a Huber Loss function instead of MSE/MAE to make the model less sensitive to outliers. Performance was yet again not satisfactory: no significant improvement over the other loss functions, when viewing test samples.

Changing Output Dimensions

While tinkering with loss functions seems to have somewhat improved predictions, we're still a long ways off from a prediction accuracy that'd let us optimize over without encountering "garbage in garbage out".
As a result, we'd decided to rework the output format and the GA code built on top of it - in the hopes that perhaps presenting the S-parameters differently would help the model in the prediction task.
Our prediction model was trained to predict 70 frequency points per structure, and the training was done in batches of 32 structures.
An input matrix batch for arbitrary EM structures, with dimensions 8×10 (2 additional columns for ports) is formatted as torch.size([32,1,8,10]) - where 32 is for the batch size and 1 is to make the dimensions compatible with different PyTorch operands.
the output is at dimensions torch.size([32,420]), where 420 is an s parameter vector of the format

$$\vec{s} = [\mathbf{Re}\{s_{11}\} - \mathbf{Im}\{s_{11}\} - \mathbf{Re}\{s_{12}\} - \mathbf{Im}\{s_{12}\} - \mathbf{Re}\{s_{22}\} - \mathbf{Im}\{s_{22}\}]$$

(the real and imaginary values of 70 frequency points concatenated, the $-$ denotes concatenation).
The output gets formatted in this method because of the create augmented dataset function in the pre-processing

stage.
As a first attempt, if instead we were to present $\vec{s}$ as [32,6,70] instead of [32,420] the separation could possibly help the algorithm predict better.

$$\vec{s} = [\mathbf{Re}\{s_{11}\}, \mathbf{Im}\{s_{11}\}, \mathbf{Re}\{s_{12}\}, \mathbf{Im}\{s_{12}\}, \mathbf{Re}\{s_{22}\}, \mathbf{Im}\{s_{22}\}]^T$$

each row corresponds to an S parameter's real or imaginary value, and each column corresponds to the frequencies it sweeps across.
Modifying the CNN architecture to accommodate the new output size, and running several model variants we discovered no significant improvement over the previously established Re/Im prediction's MSE of 0.08.

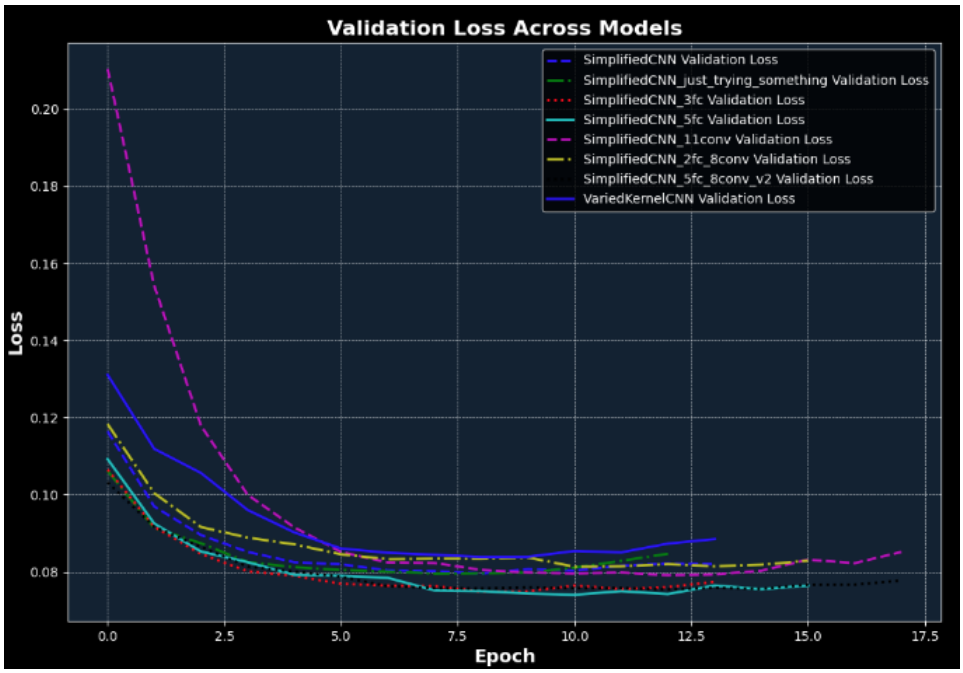

Figure 19: Training run with new output dimensions yielded no significant improvement over best predictor

We'd seen no improvement, so we decided to modify the pre-processing stage and the modeles to predict the output directly in Decibels.

$$\vec{s} = [dB(S_{11}), dB(S_{12}), dB(S_{22})]^T$$

This approach had initially appeared not to yield significant improvement, looking at the training graph:

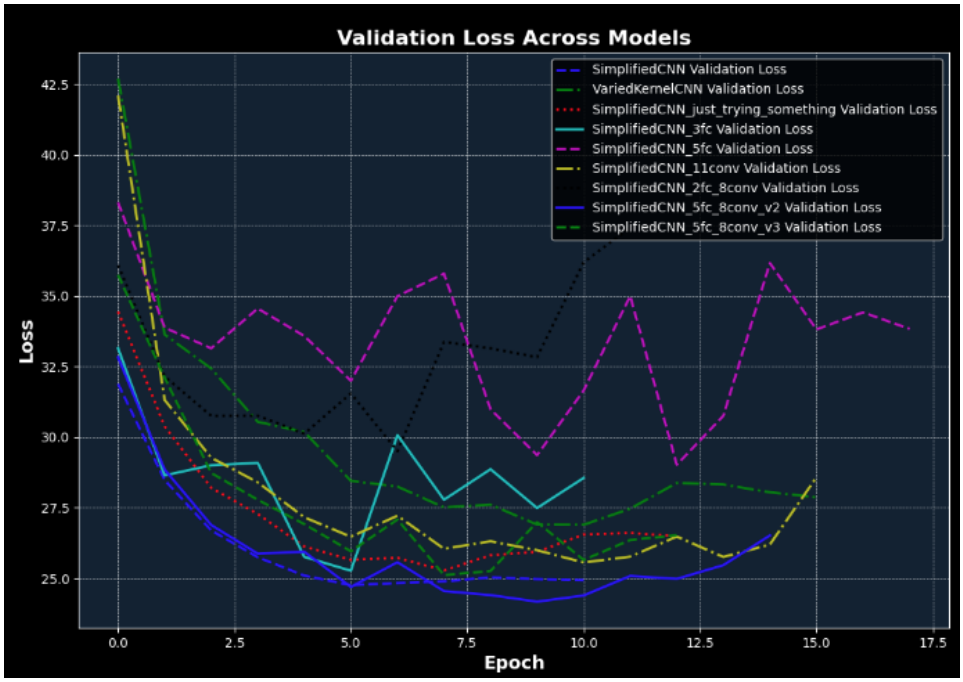

Figure 20: Direct dB prediction Training graph

However, when looking at individual samples the results look a lot more promising. Even in failed predictions, there appears to be no Jitter unlike the other approaches.

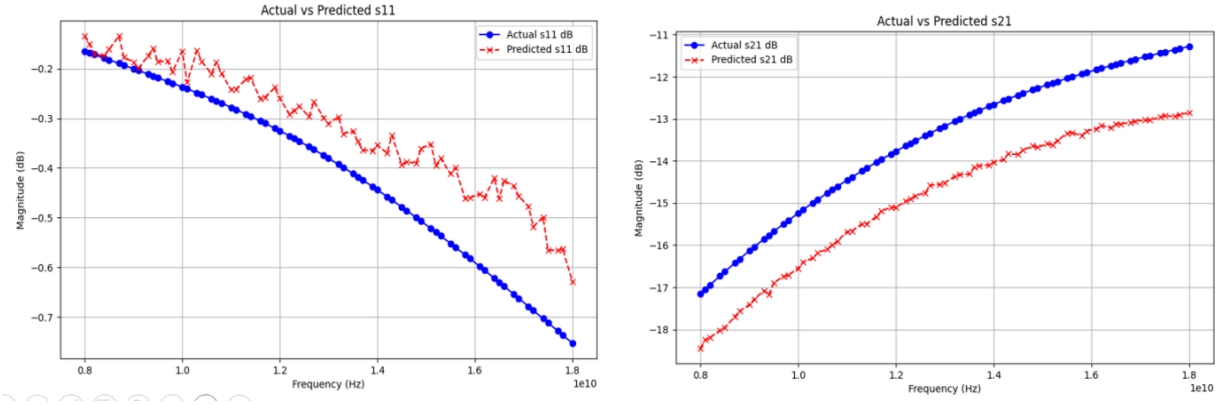

Figure 21: A good prediction in Decibel

On the other hand, while the samples look smooth and approximate, it seems as though sometimes the model misses the mark completely, for instance:

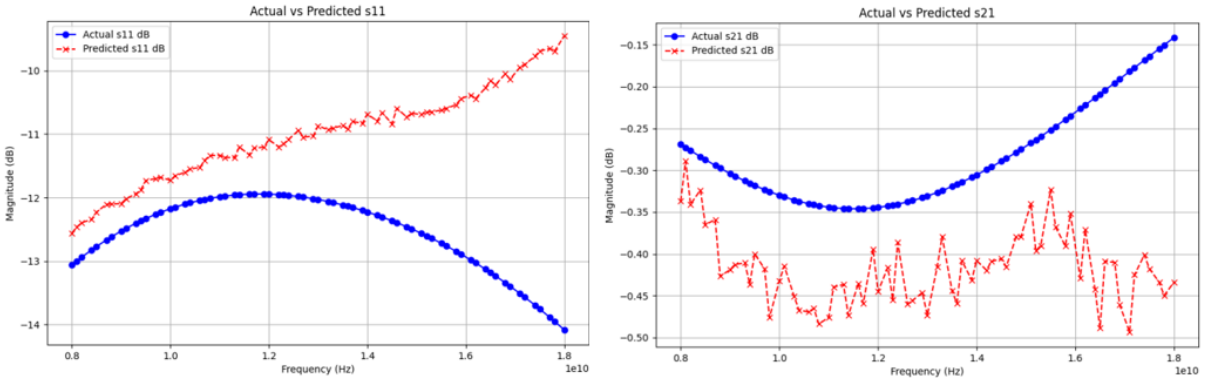

Figure 22: A missed prediction sample

This had seemed quite odd to me, and inspecting the samples the frequency responses of s_{11} looked quite identical to s_{22}'s.
We've found a bug inside the dataset pre-processing stage that copied s_{11} into s_{22} instead of swapping the two when generating samples flipped relative to the y-axis.
At that point we've fixed the bug, and switched over to the 16×16 dataset as we wanted a bigger design space. Another major improvement was gained from re-examining the article at [2], since they'd faced a similar problem to the one we had (prediction of s_{11} in dB), their CNN architecture was more suited. When adapting it the results yielded:

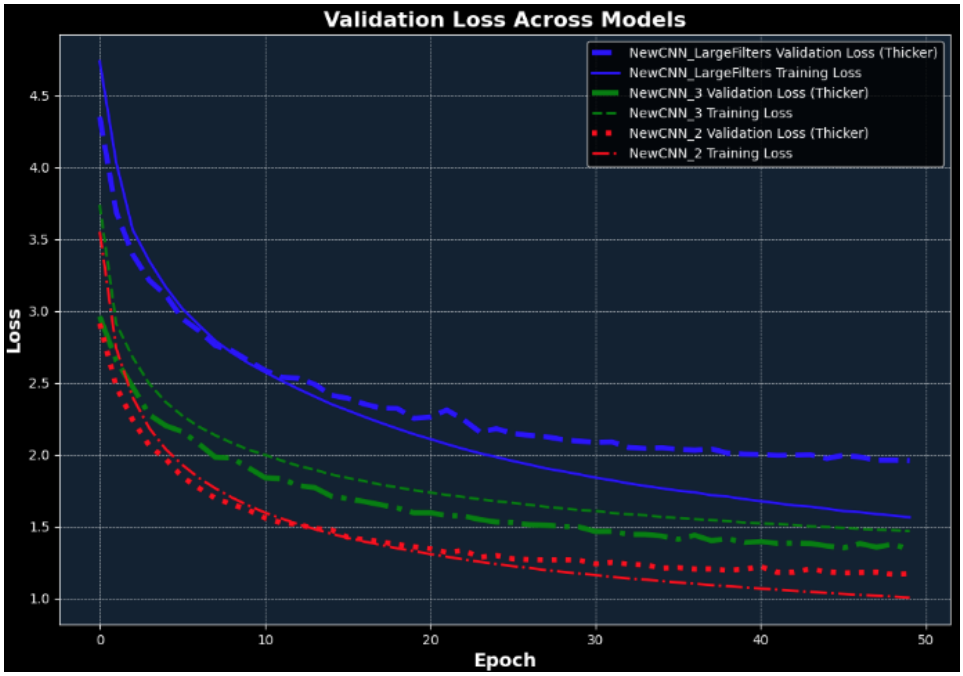

Figure 23: 8×8 samples training run, post bug fix, direct dB prediction

We'd managed to achieve a training error of 1.57 MAE over the 16×16 samples and 1.17 for the 8×8 ones in dB, which is a dramatic improvement. At that point we'd returned to the relevant works, and implemented a CNN structure similar to the one in [2] since the antenna design relied upon predicting $dB(s_{11})$. For time considerations and focused design we'd decided to only train for fewer output frequencies directly relevant to the requirements, removing the need for some reverse passes in gradient computations done by PyTorch.
implementing the design in that paper had yielded a result of 0.9 MAE over the points we cared about, and the addition of two training samples sets that had just finished generating from AWR got us an accuracy of 0.81 MAE in dB over the points we're concerned with.
It's important to note that the S-parameter behaviour in the 16×16 samples as opposed to the 8×8 samples appears far more volatile, meaning our prediction abilities are great, and only set to improve with the addition of samples.

3.3.2 Penultimate Approach

After many iterations and model architectures, output formats, and cost functions, the architecture that produced the most success is the following:

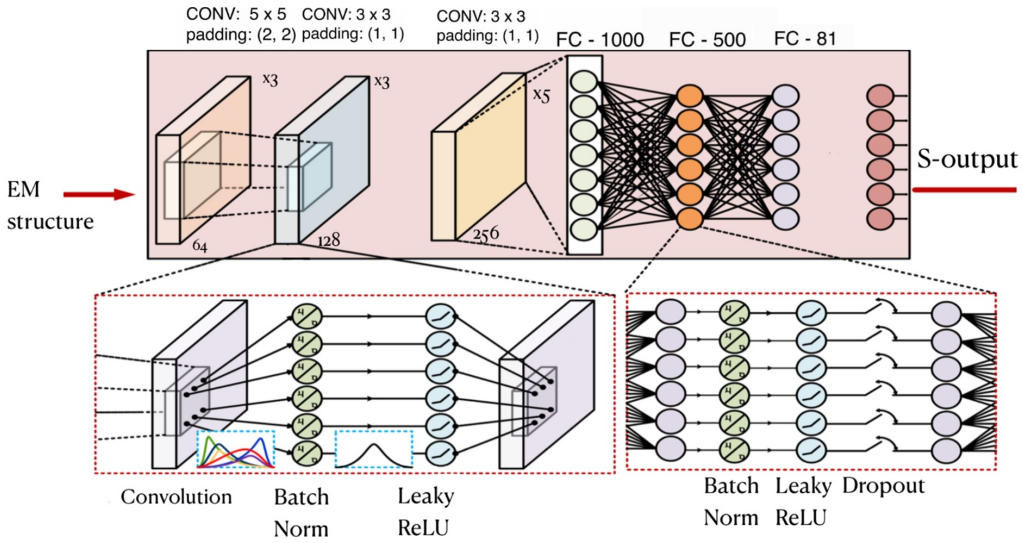

Figure 24: Image of the chosen CNN architecture for estimating S-parameters

11 Convolution layers, separated by a batch norm, and a LeakyReLU activation function. Padded to preserve image dimensions, followed by three FC layers separated by LeakyReLU, Batch Normalizations and Dropout layers of 0.4.
We conducted a final run with additional samples, switched the optimizer to AdamW so we'd be able to decouple weight decay. Training with weight decay, reduced learning rate on plateau, and early stopping has yielded an MAE of 0.72 dB, a direct improvement over 0.81MAE, and 0.9MAE which we got from running the same architecture but with less samples and Adam.
The training stopped at the 47th Epoch after no notable improvement for 8 Epochs, training time was 47150 seconds, 13.09 hours. I'd have tested variations of the model but due to the large training time and limited GPU availability I cannot.

This has overall led us to believe that the addition of more training samples, as well as perhaps minor changes to the architecture (such as adding or removing layers that follow the same rules outlined in the design) could lead us to state of the art results. The results are really good, however, especially in comparison to previous attempts.With fewer samples than the models in the studies we analyzed, and strong demands to estimate all 3 S-parameters in dB which the other studies did not contend with, we managed to gain a suitable prediction accuracy for optimization over all the parameters that could be improved with the addition of more samples.

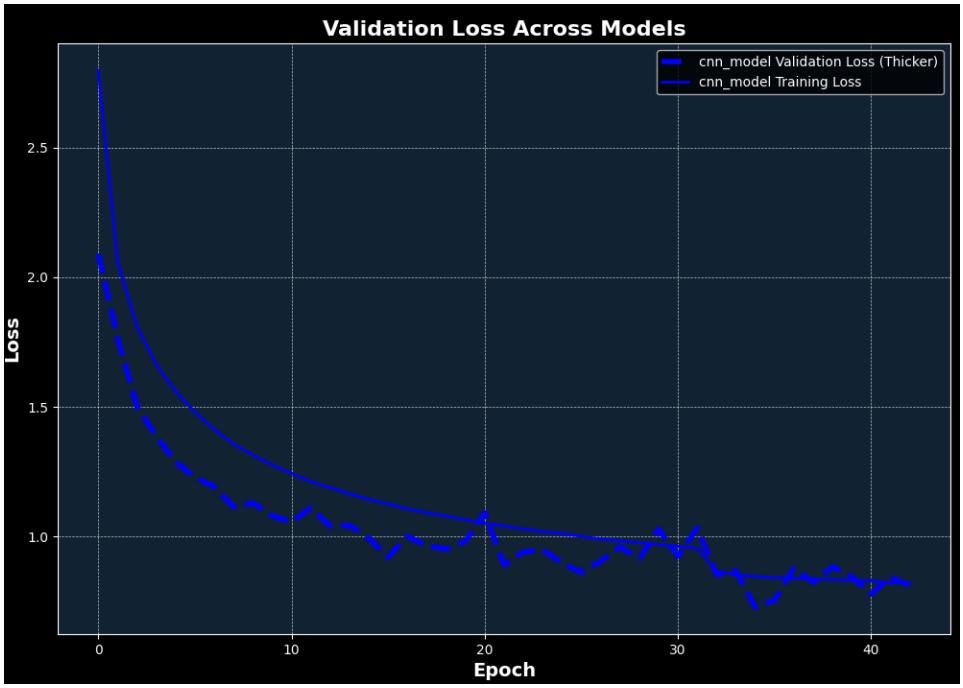

Figure 25: The final CNN run, early stopping at 47 epochs, 0.72 dB MAE

Let's examine a random test sample:

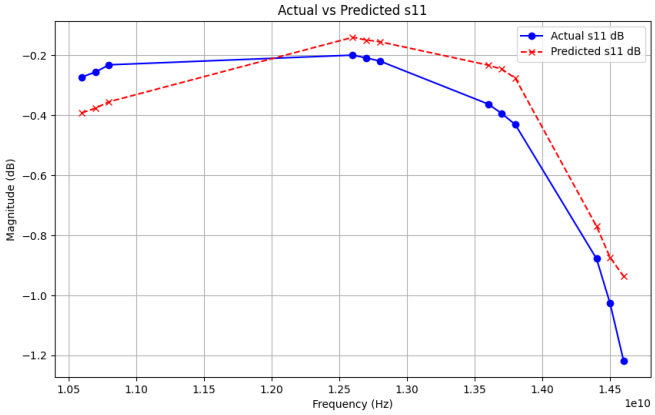

Figure 26: s_{11} and its prediction by our CNN

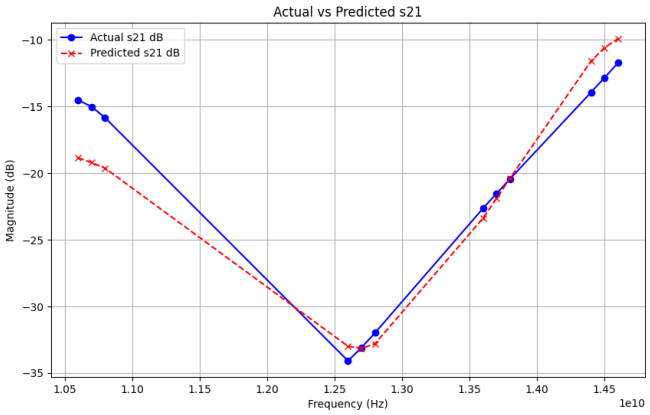

Figure 27: s_{21} and its prediction by our CNN

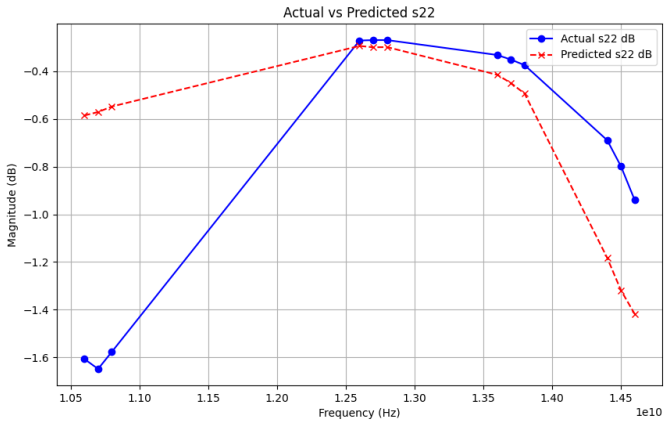

Figure 28: s_{22} and its prediction by our CNN

This accuracy is impressive considering the sample's behaviour is chaotic (and increasingly so with the addition of pixels), even though it mildly missed s_{22}'s lower frequencies it accurately captured the vast majority of the data. We think it highly plausible that the addition of more training samples, and hyper-parameter sweeps, could increase the model's performance to a very high degree. We reason as such due to noticing consistent improvements with the addition of more samples, structural changes, and we note more accurate results with far greater datasets in [1] [2] [3].

3.3.3 Improving On The Design Even Further

To improve further on the design we've decided to replace the CNN's Fully Connected layer with a much smaller one by first conducting a Point-wise convolution (a Convolution with a kernel that is 1x1). This let's us first generate a single channel image which will flatten to a smaller vector, and as a result will be fed into a more compact FC structure with fewer parameters.
This approach should lead to faster training, and a less agressive risk of overfit, and testing it shows that it has.

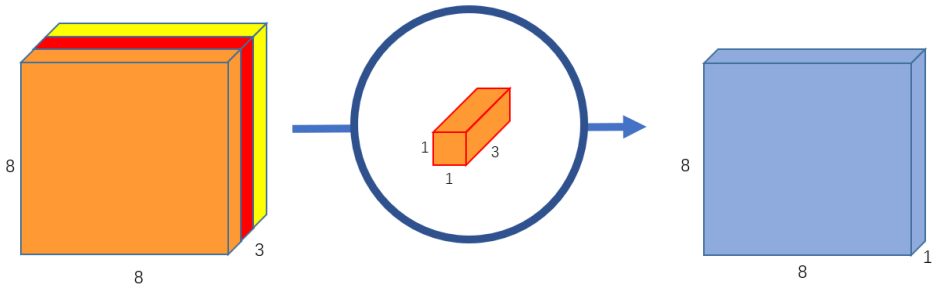

Figure 29: An example of point-wise convolution: using a 1x1 filter converts a 3 channel image into 1 channel. Image Credit: Chi-Feng Wang

This approach offered significant improvements, leading to 0.38 MAE dB accuracy! We also noticed how early stopping did not activate for 150 epochs.

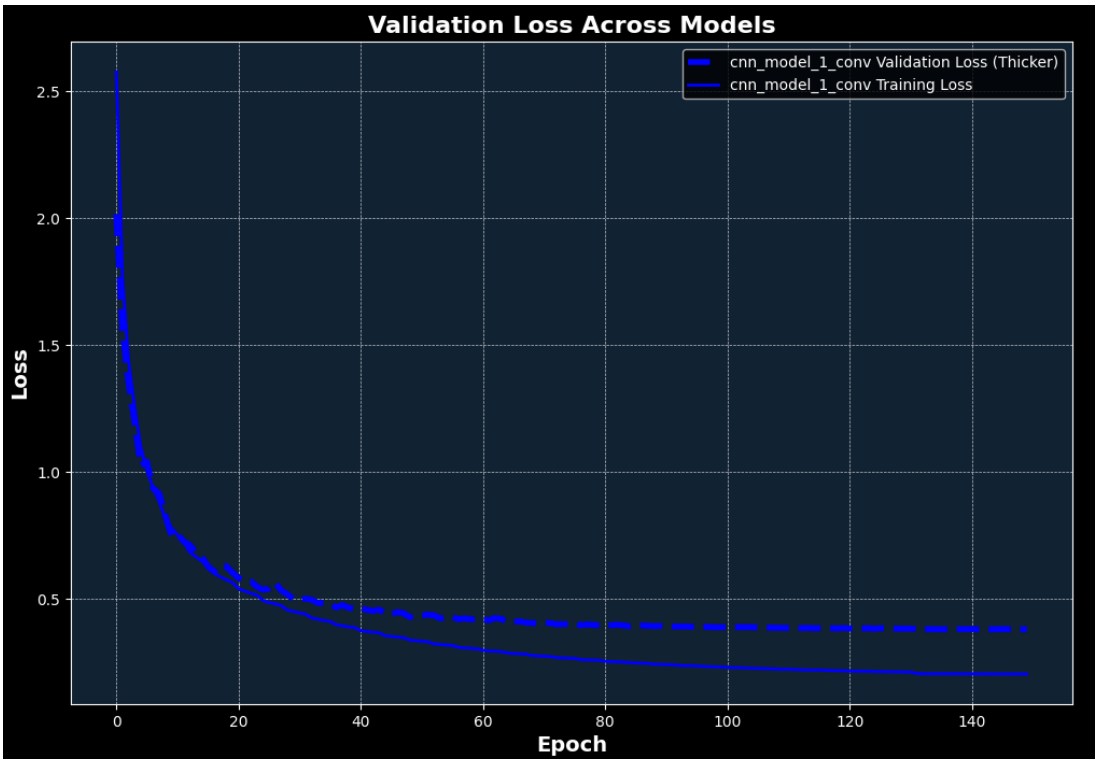

Figure 30: Final CNN training run, with point-wise convolutions, results in 0.38 MAE in dB

We find the results extremely impressive, and we've demonstrated both potential as well as two avenues for improvement: Increasing the dataset, and modifying the model structure further.

3.4 The Genetic Algorithm

3.4.1 Development Process

The Genetic algorithm was initially constructed in parallel to other components and as such independently, though with the intention to eventually connect it to a Neural Network and test it once one has been trained. Unlike Previous models which used pooling and adaptive pooling this model retains the image dimensions throughout the entire computation. Its vast layer structure makes it slow to train, but its performances are better. We suspect the reasons for that are increased regularization, and the fact that the model computes problem in space and so reducing the image size prevents it from estimating the parameters accurately. The model requires adjustment when changing the input size due to the lack of Adaptive Pooling layers that make the model size agnostic.

Dummy Problem - a matrix with ones

Initially, the genetic population had a population of binary matrices (not yet representing the metal layer), a Roulette Wheel selection operator, a basic mutation operator, and no crossover. The two selected matrices were used to create a hybrid by creating a random cutoff point every row and copying the structure from one parent into the offspring until the cutoff, and from the second parent after the cutoff.

The mutation genetic operator operates on a filter by iterating through each pixel and rolling a random number in a uniform $\mathcal{U}_{[0,1]}$ distribution. If the number is lower than an assigned constant called Mutation Rate, the pixel is flipped ($0 \to 1$ or $1 \to 0$).

A roulette wheel selection operator works by assigning a probability of being selected based on the fitness function. We'll denote P_j as the probability member j of being selected, and $\mathcal{F}_j$ as his fitness function value. For a population of N members:

$$P_j = \frac{\mathcal{F}_j}{\sum_{i=0}^{N} \mathcal{F}_i}$$

The intention was to see if I could optimize a small randomly initialized population to the point where I get a matrix of ones, and learn from the experimentation.

Experimenting with the fitness function yielded the following results on a population of 10×10 binary matrices with a population size of 50:

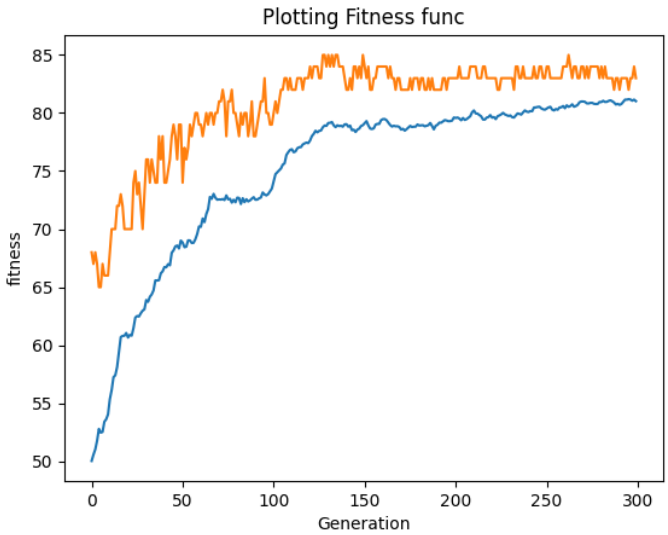

Figure 31: Number of ones in the Population - GA. Blue: Average. Red: Best Performer.

As you can see, no member of the population managed to reach the goal and become a matrix with ones. Though the optimization trend definitely exists until it plateaus. Repeating the experiment with a 16×16 population but with a better fitness function yielded the following results:

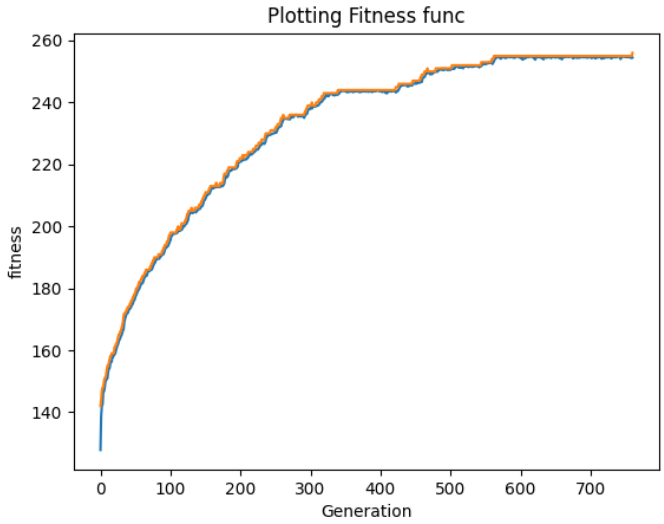

Figure 32: Fitness Score as a function of Generations in the second experiment

The fitness function that solved the problem is the following:

$$\mathcal{F}_i = e^{-(N^2 - M_{ones})}$$

where M_{ones} is the number of ones in the binary matrix. The insight gleaned from the experimentation, which we were later evinced to find was conventional wisdom, is that the fitness function worked much better because the Selection operator *heavily* favors the slightly better solutions, whereas previously the distinction was slight and so past a certain point no improvement was reached.

Dummy Problem - trying to emulate a GAN

At this point in the project, we still didn't have a trained CNN serving as an S-parameter predictor, so to learn how to deploy and link a trained machine learning model we devised the following dummy problem. The second problem we attempted to solve was to emulate a digit generator (specifically create the number 7).

We had a trained MNIST digit classifier which classifies images of handwritten digits with an accuracy of 95%, connected it to the genetic algorithm, and optimized the population based on the classifier's confidence a chosen solution actually is a 7.

Initial optimization efforts yielded some success, with the classifier confidence at 30%. The results looked unintelligible to the human eye, however. As a result, I implemented both crossover (sorting the population and directly transferring the best members) and a tournament selection operator instead of the roulette one - in which the two members selected to "reproduce" are the ones with the highest fitness scores from a random group of members drawn randomly from the population. The group of members is called a Tournament, and its size is the Tournament Size.

With the additional changes, the genetic algorithm successfully optimized over the classifier's confidence that a member is a representation of 7 - the GA successfully achieved a confidence of 99.94%. And, the result looks like:

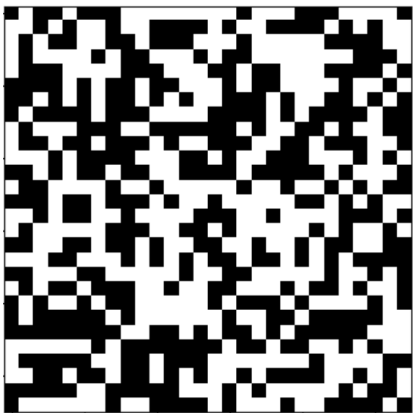

Figure 33: An image of the number 7, according to a digit classifier...

What happened? "CNN's don't see like us". The genetic algorithm optimized the CNN's prediction of a number 7, however its cost function didn't compel it to create something that to humans would truly look like the number 7. The algorithm picked the easiest optimization route available to it. We essentially accidentally created an adversarial example, an example that wouldn't fool a human but would absolutely convince a model. This concept is prevalent in securing, specifically attacking AI systems by perturbing images with a small amount of noise to fool CNN classifiers into classifying them confidently to a false category.

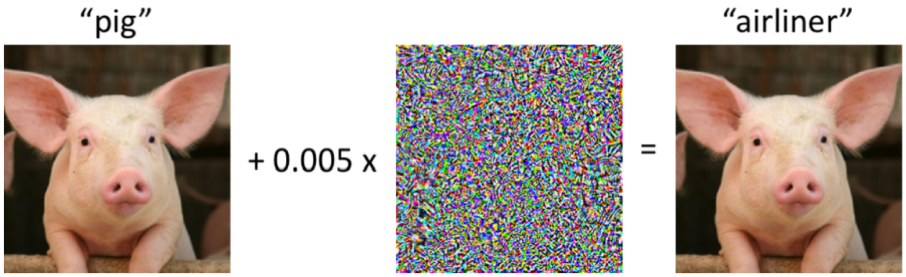

Figure 34: An adversarial example image, taken from https://gradientscience.org/intro_adversarial/

It isn't very important to our project (interesting though). Importantly, we did show we can connect a GA to a CNN and optimize over it.

3.4.2 Definitive component - S-parameter Optimizer Genetic Algorithm

After developing a GA that is capable of connecting to an ML model, we connected our GA to a trained CNN that predicts S-parameters. Our GA uses a tournament selection scheme, and an improved mutation scheme which is based on the one outlined in the previous subsection. We modified the genetic algorithm to be more in line with [1] by reducing the mutation rate every epoch. We also tweaked the algorithm so that the solutions it generates require and maintain a two-port structure - modeled by the first and last columns of the solution matrices being empty apart from one square denoting the port location. We modified the mutation operator so that it nudges the port locations randomly, and we modified the process of creating new members from the selected members such that ports are chosen randomly from each parent.

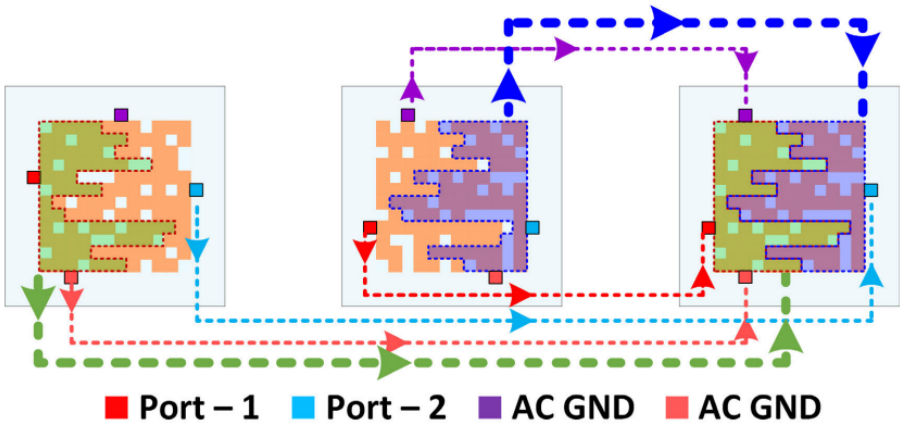

Figure 35: Port selection Process for a 4 port structure, we adapted it for a 2 port structure. Picture from [1]

Another change we made to the genetic algorithm is to replace the sorting we did to the array to find the fittest K members of the population, with a Min Heap implementation with reduced complexity. $\mathcal{O}(N \log N) \to \mathcal{O}(N \log(K))$. This change dramatically reduced generation time - it cannot be understated.
It's worth noting that we considered and implemented a Depth First Search graph to disincentivize members without a direct path between the two ports, we ultimately decided against it because we didn't want to count off solutions that appear counter-intuitive and the optimization worked without it.
other additions we've made to the genetic algorithm are making a not insignificant part of each generation's members completely random, and instead of creating two new population members from each tournament selection we opted to discard one and have twice as many tournaments. The impetus behind these choices is to induce variety, and explore more of the solution-space before converging.

The score function: we took the prediction of the CNN for each population member, which we'll denote as $\hat{S}$ (in the code: output_matrix.)
Every row in $\hat{S}$ represents an S-parameter, and each column represents frequency in which we estimate it.
We look at every requirement in the set and construct the following 'S_score' for each parameter's requirement. For instance if we want to require from s_{11} A requirement we'll call r:

$$s_{11} \leq -15dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$

We'll denote an indicator function for not fulfilling the specific requirement r at the desired frequencies as

$$\mathbf{I}_r\{\hat{s}_{11}(f_i)\} := \mathbf{I}\{\{f_i \in [10.7, 12.7]\} \cap \{\hat{s}_{11}(f_i) \geq -15\}\}$$

We'll write a component in the following way:

$$S_r^{score} = \exp\left(-w_r \sum_i |\hat{s}_{11}(f_i) - (-15)| \cdot \mathbf{I}_r\{\hat{s}_{11}(f_i)\}\right)$$

denoting R as the set of all requirements we wish to enact, the score function for a structure is:

$$\mathcal{F} = \prod_{r \in R} S_r^{score}$$

The score function is the product of all the score components derived from every requirement. Where $\mathbf{I}$ is the indicator function. w_r is the weight of each requirement r- we can use it to de-emphesize or emphesize requirements based on how important they are to us. f_i iterates over the frequencies in $\hat{S}$ for which we estimated a value for s_{11}: $\hat{s}_{11}$.

4 Results

With the following design requirements

$$s_{11} \leq -15dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$

$$s_{21} \geq -0.5dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$
$$s_{21} \leq -15dB \qquad 13.7 \leq f \leq 14.5 \text{ GHz}$$
$$s_{22} \leq -15dB \qquad 10.7 \leq f \leq 12.7 \text{ GHz}$$

and told to de-emphasize s_{22}. We weighted its S_r with 0.5, and we weighted every other requirement as 1. We'll present the run now:

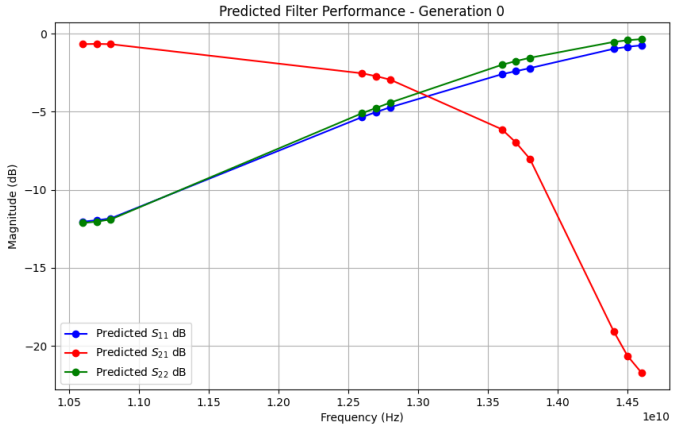

Figure 36: The S-parameters of the best EM structure at generation 0

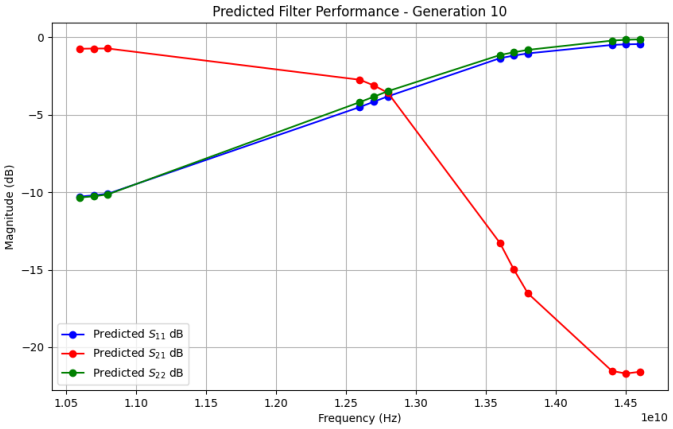

Figure 37: The S-parameters of the best EM structure at generation 10

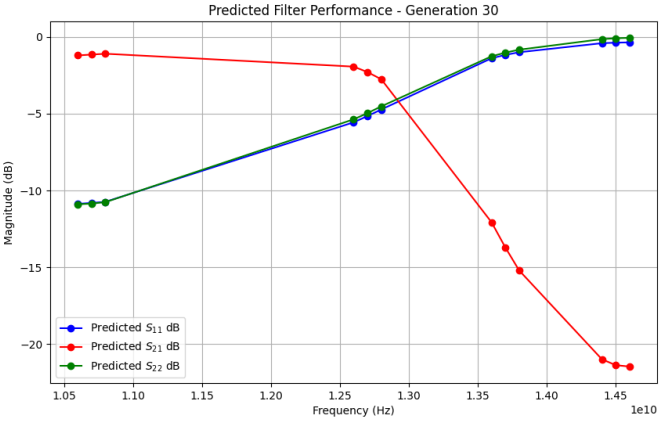

Figure 38: The S-parameters of the best EM structure at generation 30

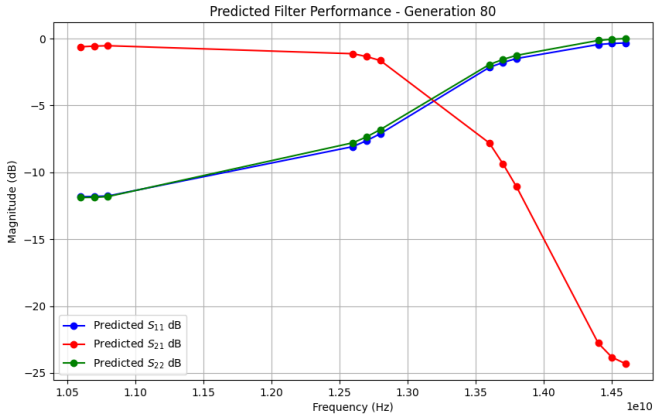

Figure 39: The S-parameters of the best EM structure at generation 80

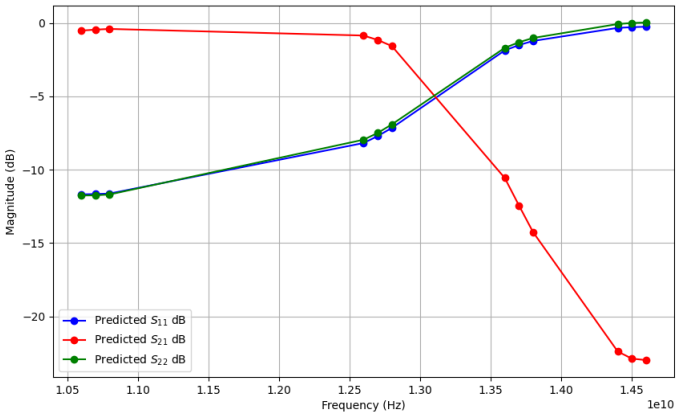

Figure 40: The S-parameters of the best EM structure at generation 100

Observing the final design's frequency response in AWR:

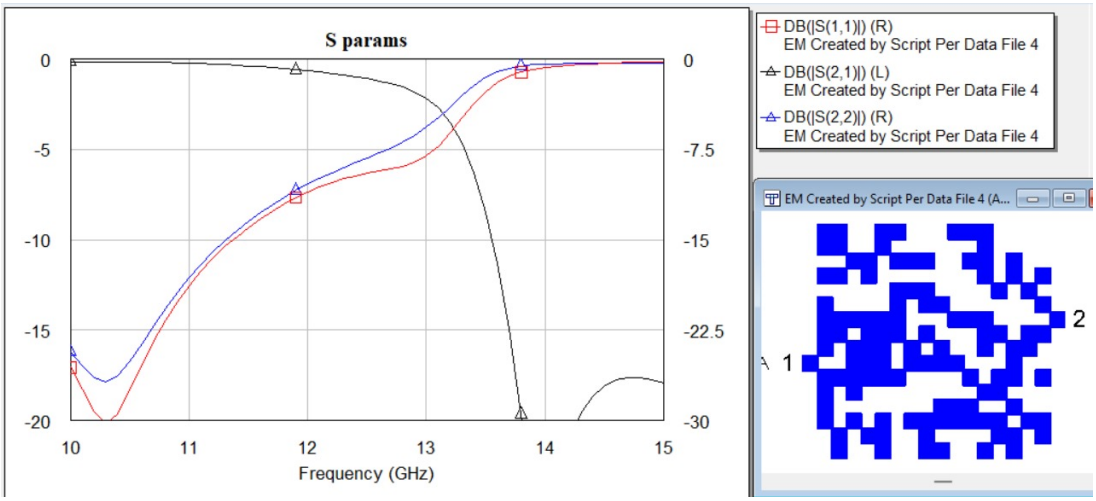

Figure 41: The final filter design's frequency response

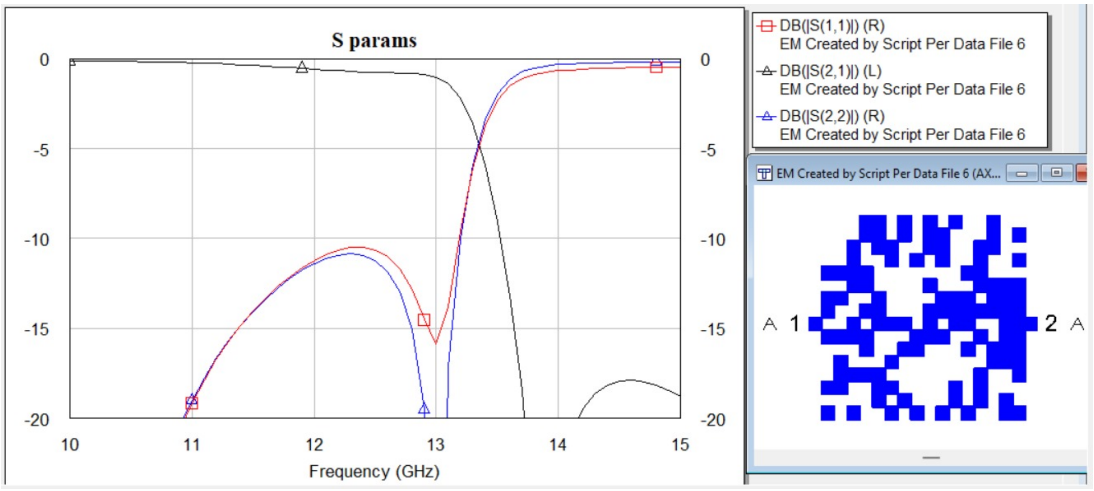

Figure 42: The result of a differnet training run

You can see how the score function appears to rise exponentially as we designed it, it plateaus sometimes until a design exceeds the best one either via mutating it beneficially or stumbling across a better solution by combining two solutions.

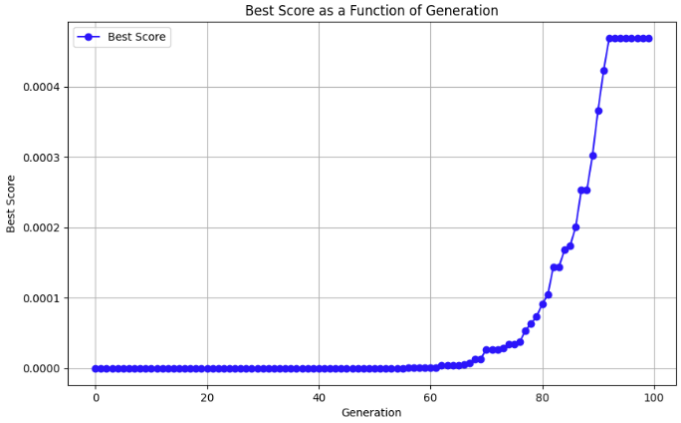

Figure 43: Best score as a function of generation, using the exponential scoring function

While we didn't quite fulfill all the requirements, the requirements are fairly demanding and the model has gotten close. We've also noticed direct improvement in prediction ability with the addition of more samples. With a GPU better than an RTX-4060, these results successfully demonstrate that our method is viable, and with our code now easily operable. With the addition of Samples, and computing power to perform hyper-parameter sweeps, the very same pipeline could be used to design better filters more accurately. Which for a final project, I personally believe is really impressive.

5 Conclusion and Suggestions

In this project, we've created a complete and adjustable machine learning based filter design pipeline, that can be modified over a vast range of frequencies, utilizes a previously untapped design space, and does not require conventional design techniques or RF expertise to operate.
We created an impressive dataset, that's easily expandable, and, went through numerous network designs, and built from scratch a genetic algorithm. All these interlinking components built with different platforms synergize successfully to generate new and revolutionary filter designs. The performance of the filters produced by the pipeline can be improved with relative ease (but high initial computing power requirements) by expanding the dataset, and the design space by increasing filter count. Similar works with more resources have exceeded state of the art, and ours can too.
We created a script that generates a dataset, trains a CNN on it to predict scattering parameters, and optimizes over the scattering parameters with impressive accuracy that can be improved by expanding the dataset. We implemented a Genetic Algorithm based on the CNN, that rapidly generates and tests possible designs.
We're both extremely proud of the final result, we're happy we got to experiment and learn new design environments such as AWR, and PyTorch.
We got to read and re-read studies, conducting this research has had a direct positive impact on the project many times and showed us the importance of it. We also really learned to value and frankly chase after the opinion and advice of experts.
Without asking for help, both with computing and with the material, this project wouldn't have been able to flourish. And the addition of more computing power both for dataset generation and hyper-parameter sweeps (Such as net architectures) can very plausibly lead to state of the art results.
We think that this project can be developed in many different directions, research-wise, our mentor wishes to analyze the EM properties of RF structures. Performance wise, it can always be improved by adding more samples, and increasing the search space by adding more pixels.
A different optimizer and Estimator structures could also be experimented with, for instance in [2] the optimizer is a Fully Connected Multi Layer Perceptron. Many new developments in the field of ML could be implemented as well: from Generative Adversarial Networks, to Particle Swarm algorithms and Diffusion Models.
The project could also be improved in its Robustness - the code could be adjusted for stitching together several filter designs to generate a Diplexer.
Specifically the field of PINNs, Physics informed neural networks, could perhaps offer better prediction methods. Perhaps RNNs could offer a novel approach as well.
As the product of over 9 months of work, we're very proud of the project and the time we dedicated to developing it. It was a massive undertaking, and we dealt with it successfully.
We really hope you found our project report interesting!
- Aylon and Eric

References

[1] E. A. Karahan, Z. Liu, and K. Sengupta, "Deep-learning-based inverse-designed millimeter-wave passives and power amplifiers," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 11, pp. 3074–3088, 2023.

[2] A. Gupta, E. A. Karahan, C. Bhat, K. Sengupta, and U. K. Khankhoje, "Tandem neural network based design of multiband antennas," *IEEE Transactions on Antennas and Propagation*, vol. 71, no. 8, pp. 6308–6317, 2023.

[3] E. A. Karahan, A. Gupta, U. K. Khankhoje, and K. Sengupta, "Deep learning based modeling and inverse design for arbitrary planar antenna structures at rf and millimeter-wave," in *2022 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting (AP-S/URSI)*, 2022, pp. 499–500.

[4] Y. Wang and Q. Ren, "Sophisticated electromagnetic scattering solver based on deep learning," in *2021 International Applied Computational Electromagnetics Society Symposium (ACES)*, 2021, pp. 1–3.

[5] Y. Li, Y. Wang, S. Qi, Q. Ren, L. Kang, S. D. Campbell, P. L. Werner, and D. H. Werner, "Predicting scattering from complex nano-structures via deep learning," *IEEE Access*, vol. 8, pp. 139 983–139 993, 2020.

[6] V. Chahar, S. Katoch, and S. Chauhan, "A review on genetic algorithm: Past, present, and future," *Multimedia Tools and Applications*, vol. 80, 02 2021.

6 Appendix: Code And Use Instructions

To use the python code you need to make sure your Python is version 3.11.7 (or any other version in which PyTorch, Pandas, Numpy, and Heapq and the other packages are implemented and installed.)
The Machine learning script also utilizes CUDA (for Graphical Processing Units), so either make sure you have CUDA installed, or the trying to train a model will be *painfully slow!*.
Generating a filter with an RTX4060 GPU takes around 1-2 hours.
You need all the imported files to be present to run each piece of code.
To train the model over different frequencies you need to have the dataset files present in your Python project.
The neural nets weights are saved in a .pth file
The preprocessed dataset is saved in a .pth file as well, to add more samples you need to reprocess it from the .csv files (which are heavy). but If you have the .pth file for the dataset, you don't need the .csv files to train a network on the preselcted frequencies. compare_neural_nets_2.py is used to train the nets
model's_name.pth contains the weights for a model with that name, that'll be loaded.
cnn_model.py contains the models (structure and the like).
To run the GA go to main.py
To change the requirements of the model open genetic_algorithm_size_adjusted.py, tweak the score function score(self) with the help of signed distance lower/higher inside the class member to create a product over the frequencies you're interested in. To change the frequencies the model predicts (you're going to have to retrain it) open dataset_importer_2.py, uncomment these lines:
print(plot_freqs)
for i in range(len(plot_freqs)):
print(f'freqs_string[i] = plot_freqs[i] / 1e9 Ghz')
they will tell you which frequency does every index in the 195 index array represent.
Take the frequencies you care about, and augment indices (defined a few lines above) to contain them. Notice that now you have to write a new score function inside genetic_algorithm_size_adjusted.py
because you effectively changed the positions of the frequencies in indices.
If you're tweaking the score function (read above), note that output_matrix is structured as follows:
[0,0,x] - will output the value of s_{11} at the frequency corresponding to the x frequency point inside indices.
similarly with [0,1,x] and s_{21}, and [0,2,x] for s_{22}.

While it can be daunting at first, once you're versed in the code, you can add/remove/shift frequency points with ease and little effort.
There's also CustomDataset.py and others that contain the preprocessing phase of the code - for any regular application that doesn't fundementally change the code they shouldn't normally be touched.
To get the dataset's .csv files and access the drive for previous versions, contact dan.

6.1 Code for generating the Dataset

In the following figures we present the code we used to generate our dataset of 150,000 simulated EM structures. Our code is provides with explanations, embedded within every figure.

Here are the libraries we used in Python:

```
import time
import pyawr.mwoffice as mwo
from pathlib2 import Path
import numpy as np
import random
import psycopg2
```

Figure 44: Python libraries that were used to generate the dataset

Here are the functions we use in the process. The explanations regarding the arguments and functions usage are listed in the definition of each function:

```
def replace_entire_file(new_content, path):
    # arguments: the content we want to write to the file, the path of the file we want to change
    # the path must include the name of the file. CHANGE '\' TO '/' IN THE PATH.
    file = Path(path)# Opening the file using the Path function
    file.write_text(new_content) #Writing the new content to the file
    return "File content replaced"

def generate_matrix(pix_num):
    # every row of the matrix must end with "space", EXCEPT THE LAST ROW!!
    # Initialize the grid with random 1s and 0s
    grid = [[random.choice([0, 1]) for _ in range(pix_num)] for _ in range(pix_num)]
    # Convert grid to string with spaces between elements
    grid_str = ""
    for row_idx, row in enumerate(grid):
        grid_str += " ".join(map(str, row))
        if row_idx != len(grid) - 1:  # Add space at the end of each row except the last one
            grid_str += " "
        grid_str += "\n"
    return grid_str

def ports_height(pix_num):
    # Generate two random numbers between 1 and 16
    number1 = random.randint( a: 1, pix_num)
    number2 = random.randint( a: 1, pix_num)
    # Convert the numbers to strings and concatenate them
    numbers_string = str(number1) + ' ' + str(number2)
    return numbers_string, number1, number2
```

Figure 45: Functions presented: file content updating, generation of random matrix of '0' and '1', generation of random port location vector

```python
import psycopg2

#create connection to a givven server
connection = psycopg2.connect(
    host='localhost',
    dbname='postgres',
    user='postgres',
    password='mypgdbpass',
    port=5432)
#creating a table iside the dataset using SQL
sql_query="""
    CREATE TABLE IF NOT EXISTS measurements_16x16(
    id INT NOT NULL,
    QR CHAR(550),
    PORTS CHAR(255),
    FREQ REAL[],
    S11 REAL[],
    S21 REAL[],
    S22 REAL[] )"""

#run the sql query in the postgres database
cur = connection.cursor()#cursor is an object that travvels the data between python and postgreSQL
cur.execute(sql_query)
connection.commit() #
connection.close()
print("Table is created")
```

Figure 46: Function presented: creating a table in postgres

```python
def import_data_to_datafile(F, S11_Re,S11_Im, S21_Re,S21_Im, S22_Re,S22_Im, id):
    #establish connection with the postgres server
    connection = psycopg2.connect(
        host='localhost',
        dbname='postgres',
        user='postgres',
        password='mypgdbpass',
        port=5432)
    #after the word "into" bellow, insert the name of the dataset you want to save the created samples in
    insert_sql_query = """
    insert into measurements_16x16_X(id,QR,PORTS,FREQ,S11_Re,S11_Im, S21_Re,S21_Im, S22_Re,S22_Im)
    values(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)"""
    # casting before storing in the dataset
    f = [float(x) for x in F]
    s11_Re = [float(x) for x in S11_Re]
    s11_Im = [float(x) for x in S11_Im]
    s21_Re = [float(x) for x in S21_Re]
    s21_Im = [float(x) for x in S21_Im]
    s22_Re = [float(x) for x in S22_Re]
    s22_Im = [float(x) for x in S22_Im]

    cur = connection.cursor()
    cur.execute(insert_sql_query, vars: (id, QR_matrix, Ports_location_string, f, s11_Re, s11_Im, s21_Re,s21_Im,s22_Re,s22_Im))
    connection.commit()
    print("record is created")
    connection.close()
```

Figure 47: Function presented: Saving the data into the dataset

Here we establish connection between Python environment and our AWR project. Then we implemented a "for" loop with 5000 iterations, creating 5000 different EM Structures, as explained in the notations along the code:

```
awrde = mwo.CMWOffice() #creating link between AWR and Python
Project = awrde.Project
start_time0 = time.time()

for i in range(1,5001): # <--the number inside range() stands for the amount of samples to be created.
    pix_num = 16 #number of pixels in the created EM Structure
    Ports_location_string, left_port_height, right_port_height = ports_height(pix_num)
    QR_matrix = generate_matrix(pix_num)
    print(QR_matrix)
    print(left_port_height, right_port_height)

    #uppdating the files with the created matrix and ports vector
    print(replace_entire_file(QR_matrix, path: 'C:/Users/ericg/OneDrive - Technion/שולחן העבודה/project/awr/QR matrix Re Im.txt'))
    print(replace_entire_file(Ports_location_string, path: 'C:/Users/ericg/OneDrive - Technion/שולחן העבודה/project/awr/ports heights Re Im.txt'))


    if Project.EMStructures.Exists('EM Created by Script Per Data File'):   #check if the EM structure exists
        Project.EMStructures.Remove('EM Created by Script Per Data File')   #deletes it if exists

    code_module = awrde.Project.ProjectScripts('QR_creation').Routines('Main') #runs the Visual Basic script in AWR Project
    code_module.RunWithArgs(None)
```

Figure 48: creating dataset, part 1

```
time.sleep(3)#wait for the EM structure to be created and for the simulation to start from the script

ping_interval = 1 #time between simulation state querry
max_time = 200  #max time in seconds before giving up
measurement_done = False
elapsed_time = 0
start_time = time.time()
#awrde.Project.Simulator.Analyze()
while not measurement_done or elapsed_time>max_time:
    sim_status = awrde.Project.Simulator.AnalyzeState
    if sim_status == mwo.mwAnalyzeStateType.mwAST_SimulationEnded.value:
        measurement_done = True
    time.sleep(ping_interval)
    elapsed_time = time.time() - start_time

print('Simulation Done.')
```

Figure 49: creating dataset, part 2

```
#add measurements to the corresponding graphs inside AWR Project
S11graph = Project.Graphs('S11')
# Parameters for the function below are: (EM structure name given inside AWR, Measurement Name)
S11graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Re(S(1,1))')
S11graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Im(S(1,1))')
S21graph = Project.Graphs('S21')
S21graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Re(S(2,1))')
S21graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Im(S(2,1))')
S22graph = Project.Graphs('S22')
S22graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Re(S(2,2))')
S22graph.Measurements.Add( SourceDoc: 'EM Created by Script Per Data File', Measurement: 'Im(S(2,2))')

measS11_Re = S11graph.Measurements[0]  #Access by index
measS11_Im = S11graph.Measurements[1]
measS21_Re = S21graph.Measurements[0]
measS21_Im = S21graph.Measurements[1]
measS22_Re = S22graph.Measurements[0]
measS22_Im = S22graph.Measurements[1]


measS11_Re_Tuple = measS11_Re.TraceValues(1)  #Returns tuple for each data point
measS11_Im_Tuple = measS11_Im.TraceValues(1)
measS21_Re_Tuple = measS21_Re.TraceValues(1)
measS21_Im_Tuple = measS21_Im.TraceValues(1)
measS22_Re_Tuple = measS22_Re.TraceValues(1)
measS22_Im_Tuple = measS22_Im.TraceValues(1)
```

Figure 50: creating dataset, part 3

```
    measS11_Re_Array = np.asarray(measS11_Re_Tuple)  #numpy function to covert tuple into an array
    measS11_Im_Array = np.asarray(measS11_Im_Tuple)
    measS21_Re_Array = np.asarray(measS21_Re_Tuple)
    measS21_Im_Array = np.asarray(measS21_Im_Tuple)
    measS22_Re_Array = np.asarray(measS22_Re_Tuple)
    measS22_Im_Array = np.asarray(measS22_Im_Tuple)


    F = measS11_Re_Array[:,0]       #extract frequency vector
    S11_Re = measS11_Re_Array[:,1]    #extract S11_Re vector
    S11_Im = measS11_Im_Array[:,1]    #extract S11_Im vector
    S21_Re = measS21_Re_Array[:,1]
    S21_Im = measS21_Im_Array[:,1]
    S22_Re = measS22_Re_Array[:,1]
    S22_Im = measS22_Im_Array[:,1]


    import_data_to_datafile(F,S11_Re,S11_Im,S21_Re,S21_Im,S22_Re,S22_Im,i) #save data

print("--- process ended in %s seconds ---" % (time.time() - start_time0))
```

Figure 51: creating dataset, part 4

Here is the Visual Basic code which we activate within Python and running inside AWR, to build and simulate the EM Structure:

```
Sub Main
    Debug.Clear

    'reload datafiles with matrix and ports vector
    Dim MatrixFile As DataFile
    Set MatrixFile = Project.DataFiles("QR matrix")
    MatrixFile.Reload()
    Dim PortsFile As DataFile
    Set PortsFile = Project.DataFiles("ports heights")
    PortsFile.Reload()


    'Base Structure of the EM: trace, rectangle, trace with ports at two ends.
    Dim EMS As EMStructure
    Dim initRec As EMInitializationRecord
    Set initRec = Project.EMStructures.CreateInitRecord
    initRec.Type = mwEMI_Stackup
    initRec.GlobalDocumentName = "Global Definitions"
    initRec.StackupName = "STACKUP.Top"
    Set EMS = Project.EMStructures.AddInit("EM Created by Script Per Data File",mwEMS_AwrAxiem,initRec)

    'convert datafile with "ports hight" into an array
    content = PortsFile.DataAsText
    Dim contents() As String
    contents() = Split(content, " ")
    ' Initialize a double array to store the converted values:
    Dim PortsIndex() As Double
    ReDim PortsIndex(UBound(contents))
    ' Convert each number string to a double and store in the array:
    For i = 0 To UBound(contents)
        PortsIndex(i) = CDbl(contents(i))
    Next
```

Figure 52: VB code - Part 1

```
'convert datafile with 0,1 vector into an array
content2 = MatrixFile.DataAsText
Dim contents2() As String
contents2() = Split(content2, " ") 'vbLf vbCrLf

' Initialize a double array to store the converted values
Dim MatrixVector() As Double
ReDim MatrixVector(UBound(contents2))

' Convert each number string to a double and store in the array
For i = 0 To UBound(contents2)
    MatrixVector(i) = CDbl(contents2(i))
Next


'set default values for QR matrix
Dim pix_sizeX As Double
Dim pix_sizeY As Double
Dim overlap As Double
Dim pix_sizeX_eff As Double
Dim pix_sizeY_eff As Double
Dim pix_num As Double

pix_sizeX = UnitScaleMil2Base(10)' <-- insert values in mil here
pix_sizeY = UnitScaleMil2Base(10)
overlap = UnitScaleMil2Base(1)
pix_sizeX_eff = pix_sizeX + overlap
pix_sizeY_eff = pix_sizeY + overlap
pix_num = 16
```

Figure 53: VB code - Part 2

```
    EMS.Shapes.AddRectangle(0,-pix_sizeY*PortsIndex(0), pix_sizeX_eff, pix_sizeY_eff, "Metal01")
    EMS.Shapes.AddFace(0,-pix_sizeY*(1+PortsIndex(0)),-1,0)
    EMS.Shapes.AddRectangle((pix_num + 1)*pix_sizeX, -pix_sizeY*PortsIndex(1), pix_sizeX_eff, pix_sizeY_eff, "Metal01")
    EMS.Shapes.AddFace((pix_num + 2)*pix_sizeX + overlap, -pix_sizeY*(1+PortsIndex(1)), (pix_num + 2)*(pix_sizeX) + 1 + overlap, 0)


    Dim X_location As Double
    Dim Y_location As Double

    'Debug.Print "i" & "    " & "X" & "   " & "temp" & " " & "Y"

    For i = 0 To UBound(contents2)
        If (MatrixVector(i)=1) Then
            X_location = (i Mod pix_num)
            Y_location = CustomCeiling(i+1,pix_num)
            'Debug.Print i & "    " & X_location & "    " & temp & "    " & Y_location
            EMS.Shapes.AddRectangle(pix_sizeX + (X_location)*pix_sizeX, (Y_location)*(-pix_sizeY), pix_sizeX_eff, pix_sizeY_eff, "Metal01")
        End If
    Next

    Project.Simulator.Analyze
End Sub
'unit normalization - converting length of a square pixel from mil into normalized unitles unit.
Function UnitScaleMil2Base(MilValue As Double) As Double
    Dim BaseValue As Double
    BaseValue = MilValue * 2.54e-5
    UnitScaleMil2Base = BaseValue
End Function
'function that executes the ceiling operation
Function CustomCeiling(ByVal dividend As Double, ByVal divisor As Double) As Double
        Dim result As Double
        result = dividend / divisor
        Dim roundedResult As Double
        roundedResult = CDbl(Fix(result))
        If roundedResult >= result Then
            CustomCeiling = roundedResult
        Else
            CustomCeiling = roundedResult + 1
        End If
    End Function
```

Figure 54: VB code - Part 3

6.2 Code for Training the Model

Python Code Snippets

Compute resources utilization during a run

for an NVIDIA GeForce RTX 4060 GPU, and an intel core i5 CPU
The last training run has utilized the GPU, using nvidia-smi:

```
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                     TCC/WDDM  | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4060      WDDM  |   00000000:01:00.0  On |                  N/A |
| 32%   66C    P2             N/A /  115W |    2461MiB /   8188MiB |     98%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

training time takes about 13 Hours. Genetic Algorithm optimization time takes about an hour.

Dataset Importing Code: dataset_importer_2.py

This file combines the .csv files into a preprocessed dataset, by reading them, changing the formatting from AWR's to PyTorch's. In this file the frequency points are also picked. The dataset needs to be reprocessed if you want to shift the chosen frequencies.

```python
import ast
import pandas as pd
import numpy as np
import torch
from CustomDataset import CustomDataset
from sklearn.model_selection import train_test_split


def dB_np(x_re, x_im):
    return 20 * np.log10(np.sqrt(x_re ** 2 + x_im ** 2))  # Removed the previously A


def load_and_combine_csv_files(filenames):
```

```
    df_list = [pd.read_csv(filename) for filename in filenames]
    combined_df = pd.concat(df_list, ignore_index=True)
    for column in ['s11_re', 's11_im', 's21_re', 's21_im', 's22_re', 's22_im']:
        combined_df[column] = combined_df[column].apply(
            lambda x: np.array(ast.literal_eval(x.replace('{', '[').replace('}', ']'
    return combined_df


def preprocess_qr_and_ports(qr_string, ports_string):
    ports = list(map(int, ports_string.split()))
    port1, port2 = ports[0] - 1, ports[1] - 1
    # Filter out rows that are not strictly binary before converting
    qr_array = np.array(
        [[int(bit) for bit in row.strip() if bit in '01'] for row in qr_string.split
        dtype=np.float32)
    l_vect = np.zeros((qr_array.shape[0], 1))
    r_vect = np.zeros((qr_array.shape[0], 1))
    l_vect[port1, 0] = 1
    r_vect[port2, 0] = 1
    return np.hstack([l_vect, qr_array, r_vect])


def create_augmented_dataset(df, indices):
    augmented_data = []
    for _, row in df.iterrows():
        qr_processed = preprocess_qr_and_ports(row['qr'], row['ports'])
        s_params = {
            's11_re': row['s11_re'][indices],
            's11_im': row['s11_im'][indices],
            's21_re': row['s21_re'][indices],
            's21_im': row['s21_im'][indices],
            's22_re': row['s22_re'][indices],
            's22_im': row['s22_im'][indices]
        }

        # Convert S-parameters to dB
        dB_s11 = dB_np(s_params['s11_re'], s_params['s11_im'])
        dB_s21 = dB_np(s_params['s21_re'], s_params['s21_im'])
        dB_s22 = dB_np(s_params['s22_re'], s_params['s22_im'])

        # Combine into a single matrix with dB S-parameters as rows
        s_parameters_matrix = np.stack([dB_s11, dB_s21, dB_s22])

        augmented_data.append((qr_processed, s_parameters_matrix))
    return augmented_data
```

```
filenames = [
    "measurements_16x16_1.csv",
    "measurements_16x16_2.csv",
    "measurements_16x16_3.csv",
    "measurements_16x16_4.csv",
    "measurements_16x16_5.csv",
    "measurements_16x16_6.csv",
    "measurements_16x16_7.csv",
    "measurements_16x16_8.csv",
    "measurements_16x16_9_4146 semples.csv",
    "measurements_16x16_10.csv",
    "measurements_16x16_11.csv",
    "measurements_16x16_12_4343 semples.csv",
    "measurements_16x16_13_2320  samples.csv",
    "measurements_16x16_14.csv",
    "measurements_16x16_16_1374 samples.csv",
    "measurements_16x16_18_ 3094 samples.csv",
    "measurements_16x16_17.csv",
    "measurements_16x16_19.csv",
    "measurements_16x16_24.csv",
    "measurements_16x16_21.csv",
    "measurements_16x16_26.csv",
    "measurements_16x16_20.csv",
    "measurements_16x16_22.csv",
    "measurements_16x16_28.csv",
    "measurements_16x16_30.csv",
    "measurements_16x16_23_175 samples.csv",
    "measurements_16x16_25.csv",
    "measurements_16x16_15.csv"

]

# for the 8x8:
# filenames = ['measurements_8x8_0.csv', 'measurements_8x8_1.csv', 'measurements_8x8
#            'measurements_8x8_4.csv', 'measurements_8x8_5.csv', 'measurements_8x8_6
#           'measurements_8x8_8.csv', 'measurements_8x8_9.csv', 'measurements_8x8_10
#          'measurements_8x8_12.csv']
combined_df = load_and_combine_csv_files(filenames)
print(f"Number of samples in combined DataFrame: {len(combined_df)}")

# start, end, pts = 74, 174, 100
indices =  [100,101,102, 120, 121, 122, 130, 131, 132, 138, 139,140 ] #np.linspace(s
freqs_string = combined_df.loc[0, 'freq']
freqs_list = ast.literal_eval(freqs_string.replace('{', '[').replace('}', ']'))
plot_freqs = np.array(freqs_list)[indices]
```

```
augmented_data = create_augmented_dataset(combined_df, indices)
print(f"Number of samples in augmented data: {len(augmented_data)}")

# print(plot_freqs)
# for i in range(len(plot_freqs)):
#     print(f'freqs_string[{i}] = {plot_freqs[i] / 1e9} Ghz')
# Splitting dataset

train_data, test_data = train_test_split(augmented_data, test_size=0.1, random_state
test_data, val_data = train_test_split(test_data, test_size=0.5, random_state=42)
train_dataset = CustomDataset(train_data)
val_dataset = CustomDataset(val_data)
test_dataset = CustomDataset(test_data)
# After preprocessing and extracting plot_freqs
print(len(train_dataset), len(test_dataset), len(val_dataset))
torch.save({
    'train_dataset': train_dataset,
    'validation_dataset': val_dataset,
    'test_dataset': test_dataset,
    'output_w': len(indices),
    'plot_freqs': plot_freqs,
    # Additional information like selected frequencies # for the 70 points, the freq
}, 'preprocessed_data_16.pth')
```

CustomDataset.py - packages the data into a PyTorch dataset structure, and generates more samples with symmetry

---

A file used by dataset_importer_2.py

PYTHON

```
import numpy as np
import torch
from torch.utils.data import Dataset


class CustomDataset(Dataset):
    def __init__(self, augmented_data):
        self.augmented_data = augmented_data

    def __len__(self):
        # Triple the dataset length for each item being seen three times (original,
        return len(self.augmented_data) * 3

    def __getitem__(self, idx):
```

```
        original_idx = idx // 3
        input_matrix, target_matrix = self.augmented_data[original_idx]

        # Create a copy to avoid modifying the original data
        input_matrix = np.array(input_matrix)
        target_matrix = np.array(target_matrix)

        # Apply flipping based on the index
        if idx % 3 == 1:
            input_matrix = np.flipud(input_matrix).copy()  # Vertical flip\
        elif idx % 3 == 2:
            input_matrix = np.fliplr(input_matrix).copy()  # Horizontal flip
            # Assuming target_matrix is structured as [dB(S11), dB(S12), dB(S22)]
            # Swap S11 and S22 components (only)
            target_matrix[[0, 2]] = target_matrix[[2, 0]]

        # Convert numpy arrays to PyTorch tensors
        input_tensor = torch.tensor(input_matrix, dtype=torch.float32).unsqueeze(0)
        target_tensor = torch.tensor(target_matrix, dtype=torch.float32)

        return input_tensor, target_tensor
```

cnn_model.py

contains model designs – cnn_model is the best one.
cnn_model_1_conv uses a 1x1 convolution layer to cut down on the parameters in the FC down the line and speed up training time.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import time
import json
import os
class cnn_model_1_conv(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_1_conv, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
```

```python
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
```

```
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.conv1x1 = nn.Conv2d(256, 1, kernel_size=1, bias=False)  # Reduce to 1 c

        # Output size calculation based on input dimensions        self.fc_input_siz
        self.fc = nn.Linear(self.fc_input_size, 3 * output_w)

        self.output_w = output_w

    def forward(self, x):
        x = self.conv(x)
        x = self.conv1x1(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 1, 3, self.output_w)  # Reshape to (batch_size, 1, 3,
        return x

class cnn_model_elu(nn.Module):  #REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_elu, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
```

```
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ELU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ELU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ELU(),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 16 * 18, 1000),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.ELU(),
            nn.Dropout(0.4),
```

```
            nn.Linear(500, 3 * output_w),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x
class cnn_model(nn.Module):  #REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 16 * 18, 1000),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(500, 3 * output_w),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x

class cnn_model_short(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_short, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
```

```
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 16 * 18, 1000),  # input size is 16x18 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
```

```
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(500, 3 * output_w),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x

import torch
import torch.nn as nn

class skip_and_pool(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(skip_and_pool, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05)
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 2 * 2, 1000),  # Adjust the input size based on the pool
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(500, 3 * output_w)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x) + x  # Skip connection
        x = self.conv3(x)
        x = self.conv4(x) + x  # Skip connection
        x = self.conv5(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x

class cnn_model_more_conv(nn.Module):  #REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_more_conv, self).__init__()
        self.conv = nn.Sequential(
            #7x7
            nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.01),

            # 5x5
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.01),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.01),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.01),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.01),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.01),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.01),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.01),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.01),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),

        )
        self.fc = nn.Sequential(
```

```
            nn.Linear(256 * 16 * 18, 1000),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(500, 200),
            nn.BatchNorm1d(200),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(200, 3 * output_w),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x

class cnn_model_1(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_1, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.PReLU(),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.PReLU(),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.PReLU(),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.PReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.PReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
```

```
            nn.PReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.PReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.PReLU(),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.PReLU(),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 8 * 10, 1000),  # input size is 16x18 after conv layers
            nn.BatchNorm1d(1000),
            nn.PReLU(),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.PReLU(),
            nn.Dropout(0.4),
            nn.Linear(500, 3 * output_w),
        )
```

```
    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x




class cnn_model_og_study(nn.Module):  #REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_og_study, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=12, stride=1, padding=6), #17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=10, stride=1, padding=5),  # 18x20
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=8, stride=1, padding=4),  # 19x21
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=6, stride=1, padding=2),  # 18x20
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),  # 18x20
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),  # 18x20
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=1),  # 17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=1),  # 16x18
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
```

```
            nn.Conv2d(64, 64, kernel_size=4, stride=1, padding=2),  # 17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),  # 17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),  # 17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),  # 17x19
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
        )
        self.fc = nn.Sequential(
            nn.Linear(64 * 17 * 19, 500),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),

            nn.Linear(500, 500),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),

            nn.Linear(500, 500),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),

            nn.Linear(500, 500),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),

            nn.Linear(500, 3 * output_w),
            #no tanh function becuase we're going for a dB prediction
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
```

```
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x


class cnn_model_more_fc(nn.Module):  #REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_more_fc, self).__init__()
        self.conv = nn.Sequential(
            # 5x5
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3, second part
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
```

```
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 8 * 10, 1000),  # input size is 16x16 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),
            nn.Linear(1000, 750),
            nn.BatchNorm1d(750),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),
            nn.Linear(750, 500),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),
            nn.Linear(500, 200),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.4),
            nn.Linear(200, 3 * output_w),
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x


class cnn_model_2(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
    def __init__(self, output_w):
        super(cnn_model_2, self).__init__()
        self.conv1 = nn.Sequential(
```

```
            nn.Conv2d(1, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
        )
        self.conv6 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
        )
        self.conv7 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
        )
        self.conv8 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.conv9 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.conv10 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
```

```
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.conv11 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.conv12 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )

        self.fc = nn.Sequential(
            nn.Linear(256 * 8 * 10, 1000),  # input size is 16x18 after conv layers
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.4),
            nn.Linear(500, 3 * output_w),
        )

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1) + x1  # Residual connection
        x3 = self.conv3(x2) + x2  # Residual connection
        x4 = self.conv4(x3)
        x5 = self.conv5(x4) + x4  # Residual connection
        x6 = self.conv6(x5) + x5  # Residual connection
        x7 = self.conv7(x6)
        x8 = self.conv8(x7)
        x9 = self.conv9(x8) + x8  # Residual connection
        x10 = self.conv10(x9) + x9  # Residual connection
        x11 = self.conv11(x10) + x10  # Residual connection
        x12 = self.conv12(x11) + x11  # Residual connection

        x = x12.view(x12.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x

class cnn_model_3(nn.Module):  # REMEMBER - INPUT IS ACTUALLY 16X18!
```

```
    def __init__(self, output_w):
        super(cnn_model_3, self).__init__()
        self.conv = nn.Sequential(
            # 7x7
            nn.Conv2d(1, 64, kernel_size=7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),
            nn.Conv2d(64, 64, kernel_size=7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.05),

            # 5x5
            nn.Conv2d(64, 128, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),
            nn.Conv2d(128, 128, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.05),

            # 3x3
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.05),
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 8 * 10, 1000),  # input size is 16x18 after conv layers,
            nn.BatchNorm1d(1000),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.5),
            nn.Linear(1000, 500),
            nn.BatchNorm1d(500),
            nn.LeakyReLU(0.05),
            nn.Dropout(0.5),
            nn.Linear(500, 3 * output_w),
        )

    def forward(self, x):
```

```
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        x = x.view(x.size(0), 3, -1)  # Reshape to (batch_size, 3, output_w)
        return x
```

compare_cnns_2.py

Allows one to train multiple CNN networks sequentially, the results of the run are saved into a .json file inside a directory called training_results.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import time
import json
import os
import gc
import torch.backends.cudnn as cudnn
from torch.cuda.amp import GradScaler, autocast
from torch.utils.data import DataLoader
from cnn_model import cnn_model, cnn_model_more_fc, cnn_model_more_conv, cnn_model_2
    cnn_model_short, cnn_model_og_study
from simplified_cnn_model import Combined_H_SMOOTH_Loss as dB
from torch.optim.lr_scheduler import ReduceLROnPlateau, StepLR

def train_loop_with_lr_reduction(model, train_loader, validation_loader, optimizer,
                                 run_name=""):
    global early_stopping_counter
    start_time = time.time()
    best_val_loss = float('inf')
    early_stopping_counter = 0
    train_losses = []
    validation_losses = []

    # Initialize the learning rate scheduler
    scheduler_plateau = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patienc
    scheduler_step = StepLR(optimizer, step_size=4, gamma=0.9)

    for epoch in range(num_epochs):
        model.train()
```

```
        running_loss = 0.0
        for data, targets in train_loader:
            data, targets = data.to(device), targets.to(device)
            optimizer.zero_grad()

            # You can disable CuDNN for specific forward pass if needed
            with torch.backends.cudnn.flags(enabled=False):
                outputs = model(data)

            targets = targets.view_as(outputs)
            loss = criterion(outputs, targets)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)  # Grad
            optimizer.step()
            running_loss += loss.item()

        avg_train_loss = running_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        # Validation phase
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for data, targets in validation_loader:
                data, targets = data.to(device), targets.to(device)
                outputs = model(data)
                targets = targets.view_as(outputs)
                loss = criterion(outputs, targets)
                val_loss += loss.item()

        avg_val_loss = val_loss / len(validation_loader)
        validation_losses.append(avg_val_loss)

        # Step the scheduler
        scheduler_plateau.step(avg_val_loss)
        scheduler_step.step()

        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            torch.save(model.state_dict(), f"{run_name}_{model.__class__.__name__}.p
            early_stopping_counter = 0
        else:
            early_stopping_counter += 1
            if early_stopping_counter >= 8:
                print(f"Early stopping at epoch {epoch + 1}. Best validation loss: {
                break
```

```
        # Clear memory
        del data, targets, outputs, loss
        torch.cuda.empty_cache()
        gc.collect()

        if (epoch % 10 == 0):
            print(
                f'Epoch [{epoch + 1}/{num_epochs}], Train Loss: {avg_train_loss:.4f}

    end_time = time.time()
    total_time = end_time - start_time
    print(f'Training completed in {total_time:.2f} seconds. Best validation loss: {b
    return best_val_loss, train_losses, validation_losses, total_time


def save_results_to_json(model_name, results, directory="training_results", run_name
    filepath = os.path.join(directory, f"{run_name}.json")
    if os.path.exists(filepath):
        with open(filepath, "r") as file:
            all_results = json.load(file)
        all_results[model_name] = results  # Update existing dictionary with new res
    else:
        all_results = {model_name: results}
    with open(filepath, "w") as file:
        json.dump(all_results, file, indent=4)


def plot_results_from_json(filepath):
    with open(filepath, "r") as json_file:
        all_results = json.load(json_file)
    plt.figure(figsize=(12, 8), facecolor='black')
    styles = ['-', '--', '-.', ':']
    colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

    with plt.style.context('dark_background'):
        for idx, (model_name, results) in enumerate(all_results.items()):
            style_idx = idx % len(styles)
            color_idx = idx % len(colors)
            train_style = colors[color_idx] + styles[style_idx]
            val_style = colors[color_idx] + styles[(style_idx + 1) % len(styles)]

            plt.plot(results['validation_losses'], val_style, linewidth=4,
                     label=f"{model_name} Validation Loss (Thicker)")
            plt.plot(results['train_losses'], train_style, linewidth=2, label=f"{mod
```

```
        plt.xlabel('Epoch', fontsize=14, fontweight='bold')
        plt.ylabel('Loss', fontsize=14, fontweight='bold')
        plt.title('Validation Loss Across Models', fontsize=16, fontweight='bold')
        plt.legend(loc='upper right', fontsize=10)
        plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.75)
        ax = plt.gca()
        ax.set_facecolor('#112233')
        plt.show()


if __name__ == '__main__':
    saved_data = torch.load('preprocessed_data_16.pth')
    train_dataset = saved_data['train_dataset']
    validation_dataset = saved_data['validation_dataset']
    output_w = saved_data['output_w']
    run_name = input("Please enter the run name: ")

    models = [cnn_model, cnn_model_short, cnn_model_more_conv, cnn_model_og_study]

    torch.cuda.empty_cache()

    batch_size = 32
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, nu
    validation_loader = DataLoader(validation_dataset, batch_size=batch_size, shuffl
                                   pin_memory=True)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("Device:", device)

    # Check if CUDA is actually being used
    if torch.cuda.is_available():
        print(f"CUDA is available. Using GPU: {torch.cuda.get_device_name(0)}")
    else:
        print("CUDA is not available. Using CPU.")

    results_directory = "training_results"
    os.makedirs(results_directory, exist_ok=True)

    for i in range(len(models)):
        model = models[i](output_w=output_w).to(device)
        model_name = model.__class__.__name__
        print(f'Currently training: {model_name}')
        optimizer = optim.AdamW(model.parameters(), weight_decay=1e-5)
        criterion = nn.L1Loss()

        best_val_loss, train_losses, validation_losses, train_time = train_loop_with
```

```
            model, train_loader, validation_loader, optimizer, criterion, device, nu
        )

        results = {
            'model': model_name,
            'best_val_loss': best_val_loss,
            'train_losses': train_losses,
            'validation_losses': validation_losses,
            'train_time': train_time
        }

        # Save results for the current model
        save_results_to_json(model_name, results, run_name=run_name)

    plot_results_from_json(f"training_results/{run_name}.json")
```

seed_generator.py

contains supporting functions for the genetic algorithm, including the mutation and the random generation of structures.

PYTHON

```python
import numpy as np
import matplotlib.pyplot as plt
import random


def random_seed(rows, cols):  # generates a random seed with random port locations.
    l_port_location = random.randint(0, rows - 1)
    r_port_location = random.randint(0, rows - 1)
    l_vect = np.zeros(rows)
    l_vect[l_port_location] = 1
    r_vect = np.zeros(rows)
    r_vect[r_port_location] = 1
    middle = np.random.randint(2, size=(rows, cols - 2))
    seed = np.c_[l_vect, middle, r_vect]
    return (seed)

def stack_seed(center,l_port_location, r_port_location,rows=16, cols=18):
    l_vect = np.zeros(rows)
    l_vect[l_port_location] = 1
    r_vect = np.zeros(rows)
    r_vect[r_port_location] = 1
    seed = np.c_[l_vect, center, r_vect]
```

```
    return (seed)

def strip_seed(member): #returns the matrix and two port locations
    rows, cols = member.shape
    idx_l= 0
    idx_r = 0
    center = member[:, 1:cols - 1]
    l_vect = member[:,0]
    r_vect = member[:, cols  -1 ]
    # print(f'l_vect: {l_vect}')
    # print(f'r_vect: {r_vect}')    for i in range(len(l_vect)):
        if l_vect[i] == 1:
            idx_l = i
            break
    for j in range(len(r_vect)):
        if r_vect[j] == 1:
            idx_r = j
            break
    return center, idx_l, idx_r

def seed_mutation(seed, mutation_rate):
    rows, cols = seed.shape
    center = seed[:, 1:cols - 1]  # only mutate the random seed in the middle, not t

    # Apply mutation to center based on mutation_rate    mask = np.random.uniform(0,
    m_cent = np.where(mask, 1 - center, center)

    # Mutate left and right ports
    l_vect = seed[:, 0]  # get the first column
    r_vect = seed[:, cols - 1]  # get the last column
    if np.random.uniform(0, 1) <= mutation_rate:  # port mutation
        l_vect = np.roll(l_vect, np.random.randint(-2, 2))  # randomly choose cyclic
    if np.random.uniform(0, 1) <= mutation_rate:  # port mutation
        r_vect = np.roll(r_vect, np.random.randint(-2, 2))  # randomly choose cyclic

    # Concatenate the mutated parts to form the mutated seed    mutated_seed = np.co
    return mutated_seed
```

genetic_algorithm_size_adjusted.py

---

contains the bulk of the genetic algorithm code, defines two classes: Member - which represents an arbitrary electromagnetic structure, and

Population, which holds an array of such members.
also contains the selection scheme, score function and more.

```python
from typing import List
import matplotlib.pyplot as plt
import numpy as np
import torch
import random
import heapq
import cnn_model
import seed_generator as sg
import math
import genetic_algorithm_size_adjusted as ga

# CNN INITIALIZATION
device = 'cuda' if torch.cuda.is_available() else 'cpu'
loaded_model = cnn_model.cnn_model(output_w=12)
loaded_model.load_state_dict(torch.load('pre_final_run_cnn_model.pth'))
loaded_model.to(device)
loaded_model.eval()


# dB Function
def dB(Re, Im):
    h = math.sqrt(Re ** 2 + Im ** 2)
    return 20 * math.log(h, 10)

# GENETIC ALGORITHM CLASS

def signed_distance_lower(s_param, constraint):
    S_score = 0
    for i in range(len(s_param)):
        if s_param[i] > constraint:
            S_score += abs(s_param[i] - constraint)
    return np.exp(-S_score)

def signed_distance_higher(s_param, constraint):
    S_score = 0
    for i in range(len(s_param)):
        if s_param[i] < constraint:
            S_score += abs(s_param[i] - constraint)
    return np.exp(-S_score)

class Member:
    def __init__(self, seed):
        self.seed = seed
```

```
        self.rows = seed.shape[0]
        self.columns = seed.shape[1]
        self._score = None  # Cached fitness score

    def score(self):
        if self._score is None:  # Calculate score only if not already done
            seed_tensor = torch.FloatTensor(self.seed).unsqueeze(0).unsqueeze(0).to(
            output_matrix = loaded_model(seed_tensor).cpu().detach().numpy()
            S_11_low = output_matrix[0, 0, 0:5]
            S_21_low = output_matrix[0, 1, 0:5]
            S_21_high = output_matrix[0, 1, 6:]
            S_22_low = output_matrix[0,2,0:5]
            s_11_score = signed_distance_lower(S_11_low, -15)
            s_21_score_1 = signed_distance_higher(S_21_low, -0.5)
            s_21_score_2 = signed_distance_lower(S_21_high, -15)
            S_22_score = (signed_distance_lower(S_22_low,-15))**0.5 #to set w =0.5
            self._score = s_11_score * s_21_score_1 * s_21_score_2*S_22_score

            # Add logging for score
            # print(f"Calculated score for member: {self._score}")        return sel

    def mutate(self, mutation_rate):
        self.seed = sg.seed_mutation(self.seed, mutation_rate)
        self._score = None  # Reset score cache after mutation

def create_offsprings(member_1: Member, member_2: Member) -> list[Member]:
    center_1, l_1, r_1 = sg.strip_seed(member_1.seed)
    center_2, l_2, r_2 = sg.strip_seed(member_2.seed)
    new_seed = np.zeros(center_1.shape)
    for i in range(center_1.shape[0]):  # iterate over rows
        j = np.random.randint(1, center_1.shape[1] - 2)  # create a random cutoff po
        new_seed[i] = np.append(center_1[i][:j], center_2[i][j:])
    random_number_1 = np.random.randint(2)  # Generates either 0 or 1
    random_number_2 = np.random.randint(2)
    if random_number_1:  # Generates either 0 or 1
        l_1_prime = l_1
    else:
        l_1_prime = l_2
    if random_number_2:
        r_1_prime = r_1
    else:
        r_1_prime = r_2
    return [Member(sg.stack_seed(new_seed, l_1_prime, r_1_prime))]

class Population:
    mutation_rate: float
```

```
    def __init__(self, population_size, rows, columns):
        self.fitness = None
        self.generation_count = 0
        self.population_size = population_size
        self.rows = rows
        self.columns = columns
        self.mutation_rate = 0.1  # initial mutation rate
        self.members = [Member(sg.random_seed(rows, columns)) for _ in range(populat
        self.update_population_fitness()

    def update_population_fitness(self):
        total_fitness = sum(member.score() for member in self.members)
        self.fitness = [member.score() / total_fitness for member in self.members]

    def print_best_seed(self):
        best_member = max(self.members, key=lambda x: x.score())
        return best_member

    def choose_parents(self, num_of_parents):
        # Use a min-heap to keep track of the top num_of_parents members
        heap = [(member.score(), id(member), member) for member in self.members[:num
        heapq.heapify(heap)

        for member in self.members[num_of_parents:]:
            score = member.score()
            if score > heap[0][0]:
                heapq.heappushpop(heap, (score, id(member), member))

        parents = [member for _, _, member in heap]

        # Log the scores of the chosen parents
        for parent in parents:
            print(f'scores are: {parent.score()}')
        return parents

    def mutate(self, member_list):
        for member_inst in member_list:
            member_inst.mutate(mutation_rate=self.mutation_rate)
        return member_list

    def tournament_selection(self, tournament_size, offsprings_num):
        new_pop = []
        for _ in range(offsprings_num):
            competitors = random.sample(self.members, tournament_size)
            sorted_competitors = sorted(competitors, key=lambda x: x.score(), revers
```

```
            parent1, parent2 = sorted_competitors[0], sorted_competitors[1]
            new_pop.extend(create_offsprings(parent1, parent2))  # 2 winners of each
        new_pop = self.mutate(new_pop)  # mutate the kids.
        return new_pop

    def new_generation(self):  # this function creates a new generation.
        crossover_num = 8  # P =8 in PA article.
        tournament_size = 400
        random_num = 50  # Adjusted number of random members for diversity

        crossovers = self.choose_parents(crossover_num)
        offsprings_num = self.population_size - crossover_num - random_num  # Adjust

        new_pop = self.tournament_selection(tournament_size=tournament_size, offspri
        new_pop.extend(crossovers)

        # Introduce random members for diversity
        for _ in range(random_num):
            new_pop.append(Member(sg.random_seed(self.rows, self.columns)))

        self.members = new_pop
        self.update_population_fitness()
        self.mutation_rate = max(0, self.mutation_rate - 0.001)  # ensure mutation r
        print(f"Just finished generation {self.generation_count}!")
        self.generation_count += 1
```

genetic_algorithm_size_adjusted.py – GPU supported version with progressive score function

---

PYTHON

```
from typing import List
import matplotlib.pyplot as plt
import numpy as np
import torch
import random
import heapq
import cnn_model
import seed_generator as sg
import math

# CNN INITIALIZATION
device = 'cuda' if torch.cuda.is_available() else 'cpu'
loaded_model = cnn_model.cnn_model(output_w=12)
loaded_model.load_state_dict(torch.load('final_cnn_model.pth'))
```

```
loaded_model.to(device)
loaded_model.eval()

# dB Function
def dB(Re, Im):
    h = math.sqrt(Re ** 2 + Im ** 2)
    return 20 * math.log(h, 10)

# GENETIC ALGORITHM CLASS

def signed_distance_lower(s_param, constraint):
    S_score = 0
    for i in range(len(s_param)):
        if s_param[i] > constraint:
            S_score += abs(s_param[i] - constraint)
        else:
            S_score -= abs(s_param[i] - constraint)
    return -S_score

def signed_distance_higher(s_param, constraint):
    S_score = 0
    for i in range(len(s_param)):
        if s_param[i] < constraint:
            S_score += abs(s_param[i] - constraint)
        else:
            S_score -= abs(s_param[i] - constraint)
    return -S_score

class Member:
    def __init__(self, seed, model, device):
        self.seed = seed
        self.rows = seed.shape[0]
        self.columns = seed.shape[1]
        self._score = None
        self.model = model
        self.device = device

    def score(self):
        if self._score is None:
            seed_tensor = torch.FloatTensor(self.seed).unsqueeze(0).unsqueeze(0).to(
            output_matrix = self.model(seed_tensor).cpu().detach().numpy()
            S_11_low = output_matrix[0, 0, 0:5]
            S_21_low = output_matrix[0, 1, 0:5]
            S_21_high = output_matrix[0, 1, 6:]
            # S_22_low = output_matrix[0, 2, 0:5]
            s_11_score = signed_distance_lower(S_11_low, -15)
```

```
            s_21_score_1 = signed_distance_higher(S_21_low, -0.5)
            s_21_score_2 = signed_distance_lower(S_21_high, -15)
            self._score = s_11_score + s_21_score_1 + s_21_score_2
        return self._score

    def mutate(self, mutation_rate):
        self.seed = sg.seed_mutation(self.seed, mutation_rate)
        self._score = None

def create_offsprings(member_1: Member, member_2: Member) -> List[Member]:
    center_1, l_1, r_1 = sg.strip_seed(member_1.seed)
    center_2, l_2, r_2 = sg.strip_seed(member_2.seed)
    new_seed = np.zeros(center_1.shape)
    for i in range(center_1.shape[0]):
        j = np.random.randint(1, center_1.shape[1] - 2)
        new_seed[i] = np.append(center_1[i][:j], center_2[i][j:])
    random_number_1 = np.random.randint(2)
    random_number_2 = np.random.randint(2)
    l_1_prime = l_1 if random_number_1 else l_2
    r_1_prime = r_1 if random_number_2 else r_2
    return [Member(sg.stack_seed(new_seed, l_1_prime, r_1_prime), member_1.model, me

class Population:
    mutation_rate: float

    def __init__(self, population_size, rows, columns, device):
        self.fitness = None
        self.generation_count = 0
        self.population_size = population_size
        self.rows = rows
        self.columns = columns
        self.mutation_rate = 0.1
        self.device = device
        self.model = cnn_model.cnn_model(output_w=12).to(device)
        self.model.load_state_dict(torch.load('final_cnn_model.pth'))
        self.model.eval()
        self.members = [Member(sg.random_seed(rows, columns), self.model, self.devic
        self.update_population_fitness()

    def update_population_fitness(self):
        total_fitness = sum(member.score() for member in self.members)
        self.fitness = [member.score() / total_fitness for member in self.members]

    def print_best_seed(self):
        best_member = max(self.members, key=lambda x: x.score())
        return best_member
```

```
    def choose_parents(self, num_of_parents):
        heap = [(member.score(), id(member), member) for member in self.members[:num
        heapq.heapify(heap)
        for member in self.members[num_of_parents:]:
            score = member.score()
            if score > heap[0][0]:
                heapq.heappushpop(heap, (score, id(member), member))
        parents = [member for _, _, member in heap]
        return parents

    def mutate(self, member_list):
        for member_inst in member_list:
            member_inst.mutate(mutation_rate=self.mutation_rate)
        return member_list

    def tournament_selection(self, tournament_size, offsprings_num):
        new_pop = []
        for _ in range(offsprings_num):
            competitors = random.sample(self.members, tournament_size)
            sorted_competitors = sorted(competitors, key=lambda x: x.score(), revers
            parent1, parent2 = sorted_competitors[0], sorted_competitors[1]
            new_pop.extend(create_offsprings(parent1, parent2))
        new_pop = self.mutate(new_pop)
        return new_pop

    def new_generation(self):
        crossover_num = 8
        tournament_size = 400
        random_num = 50
        crossovers = self.choose_parents(crossover_num)
        offsprings_num = self.population_size - crossover_num - random_num
        new_pop = self.tournament_selection(tournament_size=tournament_size, offspri
        new_pop.extend(crossovers)
        for _ in range(random_num):
            new_pop.append(Member(sg.random_seed(self.rows, self.columns), self.mode
        self.members = new_pop
        self.update_population_fitness()
        self.mutation_rate = max(0, self.mutation_rate - 0.001)
        print(f"Just finished generation {self.generation_count}!")
        self.generation_count += 1
```

main.py

loads a trained neural net and its weights (you have to specify the weights you want to load - weights are contained in a pth file). Contains the main loop for running a genetic algorithm and a function that prints out the relevant designs.

```python
import matplotlib.pyplot as plt
import numpy as np
import torch
import random
import cnn_model
import seed_generator as sg
import simplified_cnn_model
import math
import genetic_algorithm_size_adjusted as ga

def generate_matrix(grid):
    # Convert grid to string with spaces between elements
    grid_str = ""
    for row_idx, row in enumerate(grid):
        grid_str += " ".join(map(str, row))
        if row_idx != len(grid) - 1:  # Add space at the end of each row except the
            grid_str += " "
        grid_str += "\n"

    return grid_str

generations = 100
# Load the data
saved_data = torch.load('preprocessed_data_16.pth')
output_w = saved_data['output_w']
dataset = saved_data['test_dataset']
plot_freqs = saved_data['plot_freqs']

# CNN INITIALIZATION
device = 'cuda' if torch.cuda.is_available() else 'cpu'
loaded_model = cnn_model.cnn_model(output_w=12)
loaded_model.load_state_dict(torch.load('final_cnn_model.pth'))
loaded_model.to(device)
loaded_model.eval()

# Initialize genetic algorithm parameters
rows = 16  # 16 X 18, 16x16 random with 2 port vectors
columns = 18
population_size = 8000  # original size: 4096. N  #   NEED TO BE CHANGED IN "new_gen
```

```python
# Training loop
pop = ga.Population(population_size=population_size, rows=rows, columns=columns)
best_scores = []

for generation in range(generations):
    pop.new_generation()
    best_member = pop.print_best_seed()
    best_image = best_member.seed
    best_scores.append(best_member.score())

    print(best_image)
    print(generate_matrix(best_image))
    print(f'Best score in generation {generation}: {best_member.score()}')

    # Prepare the new input data
    new_data = torch.FloatTensor(best_image).unsqueeze(0).unsqueeze(0).to(device)

    # Forward pass to get predictions
    with torch.no_grad():
        output_matrix = loaded_model(new_data).cpu().numpy()

    dB_S11 = output_matrix[0, 0, :]
    dB_S21 = output_matrix[0, 1, :]
    dB_S22 = output_matrix[0, 2, :]
    if (generation %10 == 0) or (generation == 99):
        # Plot the results
        plt.figure(figsize=(10, 6))
        plt.plot(plot_freqs, dB_S11, 'o-', color='b', label='Predicted $S_{11}$ dB')
        plt.plot(plot_freqs, dB_S21, 'o-', color='r', label='Predicted $S_{21}$ dB')
        plt.plot(plot_freqs, dB_S22, 'o-', color='g', label='Predicted $S_{22}$ dB')
        plt.title(f'Predicted Filter Performance - Generation {generation}')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.legend()
        plt.grid(True)
        plt.show()


# Plot the best score as a function of generation
plt.figure(figsize=(10, 6))
plt.plot(range(generations), best_scores, 'o-', color='b', label='Best Score')
plt.title('Best Score as a Function of Generation')
plt.xlabel('Generation')
plt.ylabel('Best Score')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

main.py – GPU supported version

PYTHON

```
import matplotlib.pyplot as plt
import numpy as np
import torch
import random
import cnn_model
import seed_generator as sg
import simplified_cnn_model
import math
import genetic_algorithm_size_adjusted as ga

def generate_matrix(grid):
    grid_str = ""
    for row_idx, row in enumerate(grid):
        grid_str += " ".join(map(str, row))
        if row_idx != len(grid) - 1:
            grid_str += " "
        grid_str += "\n"
    return grid_str

generations = 100
saved_data = torch.load('preprocessed_data_16.pth')
output_w = saved_data['output_w']
dataset = saved_data['test_dataset']
plot_freqs = saved_data['plot_freqs']

# CNN INITIALIZATION
device = 'cuda' if torch.cuda.is_available() else 'cpu'
loaded_model = cnn_model.cnn_model(output_w=12)
loaded_model.load_state_dict(torch.load('final_cnn_model.pth'))
loaded_model.to(device)
loaded_model.eval()

# Initialize genetic algorithm parameters
rows = 16
columns = 18
population_size = 4000

# Training loop
```

```
pop = ga.Population(population_size=population_size, rows=rows, columns=columns, dev
best_scores = []

for generation in range(generations):
    pop.new_generation()
    print(generation)
    if (generation == 99) or (generation == 100):
        best_member = pop.print_best_seed()
        best_image = best_member.seed
        best_scores.append(best_member.score())

        print(best_image)
        print(generate_matrix(best_image))

        new_data = torch.FloatTensor(best_image).unsqueeze(0).unsqueeze(0).to(device
        with torch.no_grad():
            output_matrix = loaded_model(new_data).cpu().numpy()

        dB_S11 = output_matrix[0, 0, :]
        dB_S21 = output_matrix[0, 1, :]
        dB_S22 = output_matrix[0, 2, :]
        plt.figure(figsize=(10, 6))
        plt.plot(plot_freqs, dB_S11, 'o-', color='b', label='Predicted $S_{11}$ dB')
        plt.plot(plot_freqs, dB_S21, 'o-', color='r', label='Predicted $S_{21}$ dB')
        plt.plot(plot_freqs, dB_S22, 'o-', color='g', label='Predicted $S_{22}$ dB')
        plt.title(f'Predicted Filter Performance - Generation {generation}')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.legend()
        plt.grid(True)
        plt.show()

plt.figure(figsize=(10, 6))
plt.plot(best_scores, 'o-', color='b', label='Best Score')
plt.title('Best Score as a Function of Generation')
plt.xlabel('Generation')
plt.ylabel('Best Score')
plt.legend()
plt.grid(True)
plt.show()
```

s-param comparator

for when you want to see in your eyes how good the trained CNN model performs on an unseen random test sample.

```python
import torch
import numpy as np
import matplotlib.pyplot as plt
from cnn_model import cnn_model as SimplifiedCNN  # Adjusted import statement

# Load the data
saved_data = torch.load('preprocessed_data_16.pth')
output_w = saved_data['output_w']
dataset = saved_data['test_dataset']
plot_freqs = saved_data['plot_freqs']

# Initialize the model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimplifiedCNN(output_w=output_w).to(device)
model.load_state_dict(torch.load('final_cnn_model.pth'))
model.eval()


def plot_s_comparison(model, dataset, sample_idx, plot_freqs, s_param='s11'):
    s_param_indices = {'s11': 0, 's21': 1, 's22': 2}  # Adjusted for direct dB outpu
    index = s_param_indices[s_param]

    with torch.no_grad():
        data, targets = dataset[sample_idx]
        data = data.unsqueeze(0).to(device)
        outputs = model(data)  # Output is [batch_size, 3, output_w]

        predicted_db = outputs[0, index, :].cpu().numpy()

        actual_db = targets[index, :].numpy()

        plt.figure(figsize=(10, 6))
        plt.plot(plot_freqs, actual_db, 'o-', color='b', label=f'Actual {s_param} dB
        plt.plot(plot_freqs, predicted_db, 'x--', color='r', label=f'Predicted {s_pa

        plt.title(f'Actual vs Predicted {s_param}')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.legend()
        plt.grid(True)
        plt.show()
```

```python
# Example usage
sample = np.random.randint(0, len(dataset))
plot_s_comparison(model, dataset, sample, plot_freqs, s_param='s11')
plot_s_comparison(model, dataset, sample, plot_freqs, s_param='s21')
plot_s_comparison(model, dataset, sample, plot_freqs, s_param='s22')
```