NV Center CNN Recovery Results

Dvir Jacobovich

March 2022

1 Abstract

I implemented an AI model based on Convolutional Neural Network (CNN) in order to improve the performance of the compressed sensing part in the NV center magnetic field recovery, over the traditional optimizers like NESTA, L1, or TV minimizers. Unlike these optimizers the CNN gets only the CS projection vector as input, as the sampling matrix is redundant in the sense that it doesn't add more information to any Machine Learning model. ML model can "learn" this matrix and the sampling basis features, by the projections vector. While this matrix plays a massive role in classic optimization algorithms as the optimizer does all sort of pre-defined mathematical manipulations on the matrix as part of the optimization.

The model is trained over a variable magnetic field magnitudes, wherein each one the simulated experiment is performed with Compressed Sensing. The CNN model is inputted with a constant number of measurements M for all magnetic fields magnitudes, and outputs the 8 targeted Lorentzian peak locations in MHz. During the training process, it performs validation test obtaining the Mean Absolute Error (MAE) for each sample with its corresponding simulated-peak locations - MAE of output-target matching.

Currently, and after few breakthroughs, I trained 3 different models with

$$M = 40,\ 50,\ 60$$

Correspondingly. The 3 average MAE over 120 testing samples for each model are

$$MAE \approx 0.021,\ 0.007,\ 0.012[MHz]$$

respectively. All were tested with 3 simultaneous MW frequencies, and with additive noise $\epsilon = 1e{-4}$. The 60 measurements model is still in progress and I'm currently doing some adjustments, since potentially it can and should produce better performance than the 50.

2 Experiment Simulation

As the title implies, most of the simulation up to the compressed sensing recovery part is almost the same as Chris and Galya experiment simulation. What I did differently was to generate samples of mock NV experiment[1], where each one has a different magnetic field magnitude. These magnetic fields are randomly generated from the uniform continuous distribution:

$$B \in [45, 125][Gauss],\ \text{with } \Theta = 30,\ \text{and } \Phi = 60.$$

The simulation obtains the 4 magnetic field projections over each one of the NV diamond orientations according to the lab frame. The diamond orientations unit vector were defined by:

[1]Generated as Matlab objects, meaning that each one has attributes needed for the deep learning model like its CS measurements vector $\hat{y}$ which is the model input, its 8 Lorentzians' peak locations (Targets), the magnetic field itself for analyzing the test results and more different fields that would be discussed later

$$D = \frac{1}{\sqrt{3}}\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$$

And the 4 projections were obtained by

$$B_{projs} = \sum_i (D \cdot B)_{i,j}, \quad i \in [4], \text{ and } j \in [3]$$

The simulation gets the 4 resonances' detunings by

$$\text{detunings} = 2.8 * B_{projs}$$

A discreet frequency window was defined for every magnetic field.
The **full peak window** was defined by the detunings' max-min subtraction in MHz and it's denoted by l:

$$l = max(\text{detunings}) - min(\text{detunings})$$

In the simulation, all samples are centered at $C = 2875[Mhz]$. The simulation obtains the symmetric 8 Lorentzian locations:

- 4 from the right of C corresponding to $|0\rangle \Rightarrow |1\rangle$ transition
- Their 4 symmetric detunings from C left corresponding to $|0\rangle \Rightarrow |-1\rangle$.

The **window full size**[2] is denoted by L, and the fraction $\frac{l}{L}$ by $\mathbf{r}$. [3]

Next, the amount of padding the simulation adds to each side of l is given by

$$\mathbf{P} = \frac{(1-\mathbf{r})}{2 * \mathbf{r}} * l$$

The simulation also defines the **window frequency resolution**, which is denoted by df and it has units of $\left[\frac{MHz}{sample}\right]$, so the **window number of points** N is simply given by l plus the padding addition divided by df:

$$N = \frac{l + 2P}{df}$$

Lastly, the **full discreet window** linspace is given by

$$\text{freqs} = [N \text{ points from } -0.5 * (l + \mathbf{P}), \text{ to } 0.5 * (l + \mathbf{P})] + C$$

As for future work, there may no need to set a discreet window for the compressed sensing, instead a possible option is to set N to be large enough. My intuition is that by doing that, N approaches infinity and the frequency resolution increases. Therefore, different CS measurements' potential information increases. Since it has nothing to do with the optimizer itself, but the sampling part of CS, I'm convinced that for the same reasons it led to improvement in the classic optimizer performance, it can improve any ML model as well. I'll test it and keep you posted.

[2]by size I mean in MHz units unlike number of frequencies samples.
[3]set to be 0.6 always.

3 Compressed Sensing Recovery using CNN

In the sensing part, I used Chris's random sampling bases $\mathbf{\Phi}$ to obtain the projections. I did few simulation trying to test the effect of different sampling bases on our results, but I coudn't see any benefit of any bases over the other (Hadamard and Gaussian were also tested). I guess the reason was we have only 3 simultaneous MW frequencies at each measurement in a much greater dimension like ~ 200 windows' size.
Additionally, there was no need to multiply the decomposition base $\mathbf{\Psi}$, and use the signal sparse representation as well like we often see in CS:

$$\hat{y} = \mathbf{\Phi}\Psi \cdot \hat{x} + \vec{\epsilon}$$

Since like mentioned before any ML model can learn this base representation as part of its weights adjusting.

3.1 Fully-connected layer in CS Recovery model

As it will explained soon, we wish to have at least one Fully-Connected layer in our model for any CS recovery problem.[4] The FC layer is equivalent to matrix multiplication with fixed and pre-defined dimensions. Therefore, the CNN input dimensions have to be the same for all inputs. Otherwise different inputs with different sizes would reach the FC layer which can multiplied with only specific size.
Therefore, I set a constant number of measurements for every model I trained. So like I mentioned for now I got three models that each one is trained on fixed M, where for now $M \in \{40, 50, 60\}$.

The reason we want our model to start with a matrix multiplication[5] layer is that according to one of the compressed sensing sampling assumptions, good measurements with potentially higher chances to better results needs to be more incoherent with the signal decomposition basis $\mathbf{\Psi}$.
Random bases play this role as they are the most incoherent[6] with every decomposition basis, therefore, by using them we expect the signal information to be spread incoherently over the projection vector.

In a matrix-multiplication-input-layer[7] every neuron is connected to all input's elements, in our case these elements are the CS projections $\hat{y}$. For example the matrix projects $\hat{y}$ into an higher dimension vector $\hat{Y}$, every neuron represents $\hat{Y}$ connected to all of $\hat{y}$ elements. This enables the model to perform projections analysis with the adjusting of every pair of neurons weight connection. So the model can learn the projections dependencies and adjust its weights correspondingly. Unlike non-fully connected layer, which are missing some connections between different projections and like we said the information is incoherently spread over all projections.

3.2 Convolutional layers in CS Recovery model

As opposed to FC layers, FCN - Fully-Convolutional Network, is a model that one can use without specifying its input dimension, since there is no FC layer, instead all we have is Convolutional layers. Typically, these layers put more focus on local features by learning filters weights with pre-defined sizes.
In deep Convolutional models, a Neuron receptive field can be described by the model's input information a single Neuron has access to. Good demonstration is shown in Fig 1 As the model is deeper it can focus more on general features as the we go deeper in the network.
With all that in mind, we would like our CS recovery model to combine both, dense layer as the first layer, and then deep Convolutional architecture that uses that fully-connected projections analysis, and then, using deep Convolutional path learns to manipulate this information into the experiment features, and ultimately into the NV diamond resonances.

[4] more particularly the input layer

[5] trainable-elements matrix of-course

[6] as I understand different randomized bases can lead to better results in certain CS cases, but the reason isn't they are more incoherent necessarily, but rather they are normalized and focus on different signal's features possibly more important ones, like the Hadamard and Dragon bases.

[7] with priority to an up-sampling matrix - increasing the input dimension to bring the model more adjusting neuron-space to learn the compressed sensing

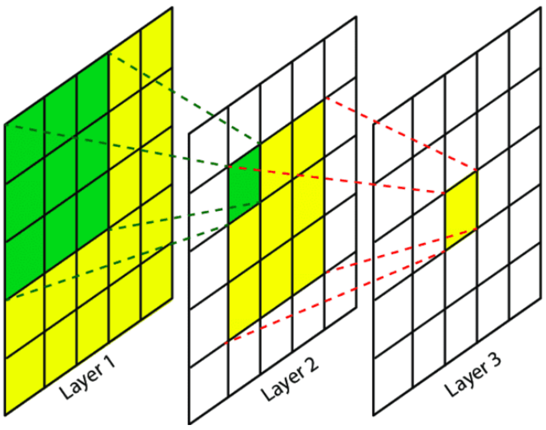

Figure 1: Here the filter size is (3, 3) and one can see that as the deeper the Neuron is located in the network, it has greater view and information on the input. Thus, the Neuron learns features which are less local and more general.

3.3 CS Sanity test

We know that the greater the magnetic field magnitude is, the greater its window size[8] L, and the number of frequencies in his window N. So we can say that for a given fixed **number of measurements**, for great magnetic field magnitudes the ratio between its **number of measurements** M to its **number of frequencies** N becomes smaller. Consequently, a good sanity test would be[9] checking if the model is getting increasing error as we test greater magnetic fields.

4 The CNN Model

4.1 Technical details

I used Pytorch packages to implement the model. This model's training process can be performed on at least 10Gb graphical memory, otherwise with less graphical memory should not be enough for large batch size and in any CPU the process would last forever. For that task I used Google Colab Nvidia Tesla T4 GPU which which has a free graphical memory of 14Gb. Colab is a friendly workspace to any deep learning. project.

For any post-training usage there is no need of so much memory but the process has to run on a supported GPU, since the model's device is set to 'cuda' and canot be changed to 'cpu'.

The full project can be found on my Github:

https://github.com/DvirJacobovich/Nitrogen-vacancy-center-deep-learning

The main file is NV_fixed_cs_measures.ipynb

4.2 Training Process

The training is performed by batches of samples. In every iteration the machine feed-forwards each of the batch samples without changing the weights[10]. Then, it takes the averaged optimized Gradient direction and updates the weights according to all batch samples average. Thus, the larger the batch size the more accurate and stable the process would be. I tried to maximize the

[8]both full window size, and peak full size.
[9]has been done
[10]For that reason GPUs are very recommend. They are much more paralel then CPUs and have their own memory.

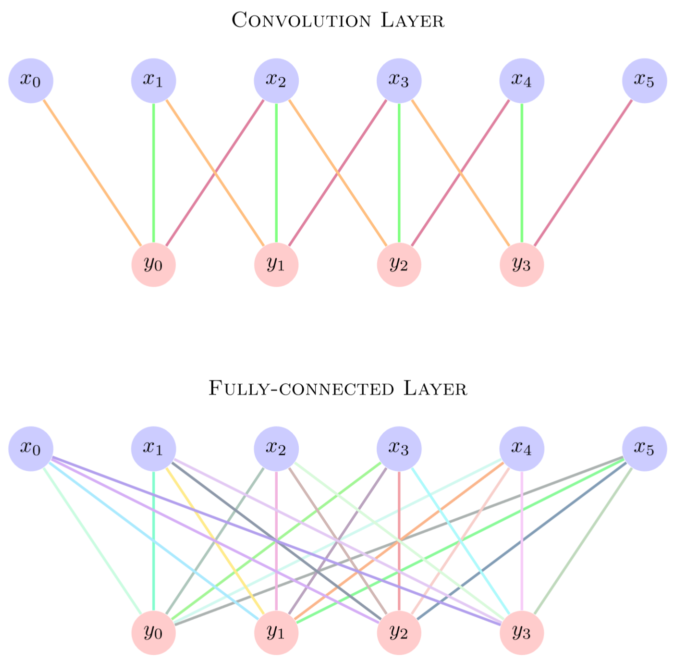

Figure 2: On the top 1D covolutional layer and you can see every neuron in the output y_i covers three local neighbours, while in the FC layer underneath every y_i is connected to all x_j.

batch size by reducing the network depth. It ended up with batch size of 128 which is great. Without going into details, I used ADAM optimizer to make a better convergence and more stable. Learning rate is set to be $lr = 1e - 4$ in almost all processes, and all other pre-defined parameters are in my flags.py file.

4.2.1 Training and Validation Losses

During the process I tested two errors: training and validation, and all errors were set to be **Mean Absolute Error (MAE)**. Validation error was obtained by 120 samples that were not part of the training data set. This test indicates the model current generalization. The process saved a copy of the model every time it tested a new minimal validation error. The validation test was performed every 3 iterations, by setting Pytorch Auto-grad to false so the process dont change the Auto-grad, and acts like the network is fixed.

The 120 validation samples were set **in one batch**, so the process maximize the generalization prediction error to be the average of all validation samples errors.

4.2.2 Number of samples - real data problem and solution

I trained each model with 2000 simulated-samples, which for real experiment can take time to generate this amount of sample. However, I thought about two ML techniques that can significantly reduce the number of real-experiment samples needed:

- **Data Augmentation -** technique for increasing the amount of data by modifying copies of the already existing data. This technique acts like another regularizer as it helps to reduce over-fitting and gets more generalization.
- **Transfer-Learning -** In Transfer-learning we take pre-trained model and using it to different but related problem (like slightly different family of images). This is being done by training once again the pre-trained model on the new data, but now the training changes only some of the model weights we assume are relevant to the new problem. This process saves lots of

new-related data samples since we are using an already trained model and we need to adjust some of its features that are relevant to the new-related data. We may find it useful if we take our simulated-trained models and with much less real-experiment data samples to train only the first part of the network e.g. FC path. I assume the projections' relevant information is learned in this part of the network. This information is being separated from non-relevant information like noises, and since we expect the real setup to have different noises we might be interested to re-adjust this part.

4.3 Model additional layers

PReLU activation, and batch normalization are followed by every convolutional or FC layer. The batch normalization has showed to improve the convergence, and the PReLU is a version of the popular LeakyReLU activation, except that unlike LeakyReLU, PReLU learns the small leak parameter which can be adapted to the other learnable parameters - weights and biases.
Additionally, to improve model's generalization, and to avoid over-fitting to the training data set, I used dropout layers with probability of 0.5. The dropout has significant effect on the validation loss. When validation is performed we remove dropouts layer as just like batch normalization these layer has an important role during the training process, and they are not integral part of the network. As the batch normalization is good to improve convergence, dropout layers vanishes every neuron's output with pre-defined probability, what actually changes the network at every iteration. Therefore, unlike typical over-fitting scenario, where we have too many neurons for a quite simple task, here the neurons' non-deterministic vanishing doesn't let the model to over-fit its training data. Instead, the network learns the dependencies of different neurons every time, so in the validation test, the process test the model with all those dependencies together.

4.4 Model architecture

The input size is [batch size, 1, M]. Typically in 1D problems the second dimension is the feature dimension and it's meant for expanding our data into multiple channels like we going to see soon in the Convolutional path.

1. The network input is split into two paths:
   (a) Fully-connected path as mentioned with that projects the input measurement vector to a much higher dimension - 1200, using a FC layer with matrix dimension of $[M, 1200]$.[11]
   (b) In the other path the input goes through a square matrix-FC layer $[M, M]$ that keeps the input dimension followed by the Pytorch build-in **Multi-Head-Self-Attention**. This set of layers is used to give the model the right architecture to focus on input's certain properties. It was most common in sequential inputs like NLP problems but in recent years it showed great improvements in many CNN regression problem as it shows here. We want the model to focus on the 8 Lorentzian peak locations, but for that to happen it needs to give priorities to certain properties obtained in the square $[M, M]$ FC layer.
2. The two paths are combined with a *Blinear* layer that gets two inputs x_1, x_2 and produces

   $$y = x_1^T A x_2 + b$$

   The model learns A and b, and in our case the output is of size 4096.
3. The Convolutional path - at first we have blocks of 1D Convolutional layers with

   kernel size = 4, stride = 2, padding = 1

   to reduce the dimension from 4906 to 8 in each block the dimension is cut by half, and instead the model expands its feature dimension by two starting from 32 to 4096 - when it reaches

[11] I basically implement it in few steps, first increase the dimension to 200, then 500, and lastly 1200. This path architecture has proven to perform better, but each FC layer has an heavy time-space cost

dimension of 8.

Every such block is followed by PReLU and batch normalization as well as another Convolutional block with

kernel size=3, stride=1, padding=1

With same feature dimension to give the model more depth.

When the model reaches the final dimension - 8, it has 4096 features. One great way to reduce the number of features is to use blocks of

kernel size = 1, stride = 1, padding = 0

So the model uses such block until it reaches the expected total dimension [batch size, 1, 8] to match with the real locations target.

5 Results

I show here three models with: $df = 3$. In every measurement we have 3 simultaneous MW frequencies, and $M = 40, 50, 60$.

1. **40 measurements:** We have averaged MAE over 120 testing samples of **0.021[MHz]**. Here I show plot of these results: MAE Vs. The magnetic field magnitude in Fig 3.
   Full 120 testing performance can be seen here:

   The 40 measures model performance results

2. **50 measurements:** The best results so far were obtained by the model of 50 measurements, with averaged MAE of **0.0078[MHz]** over 120 testing samples. The results can be shown in Fig 4. Full 120 testing performance can be seen here:

   The 50 measures model performance results

3. **60 measurements:** Here the average MAE over 120 testing samples is **0.012[MHz]**. I'm currently doing dimensional adjustments to improve these results. The results are shown in Fig 5 and here:

   The 60 measures model performance results

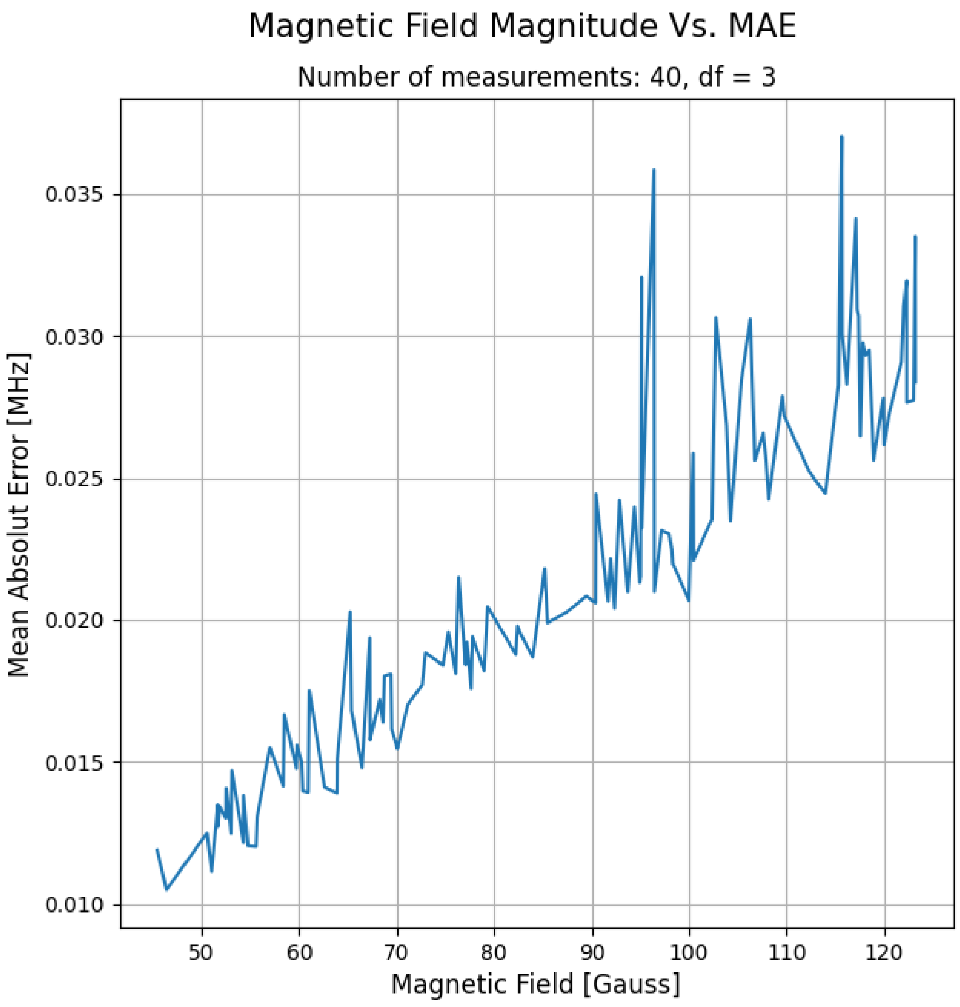

Figure 3: $M = 40$, Up to fluctuations we can see already here that the MAE increases as the magnetic field magnitude is getting bigger like we mentioned before and expected.

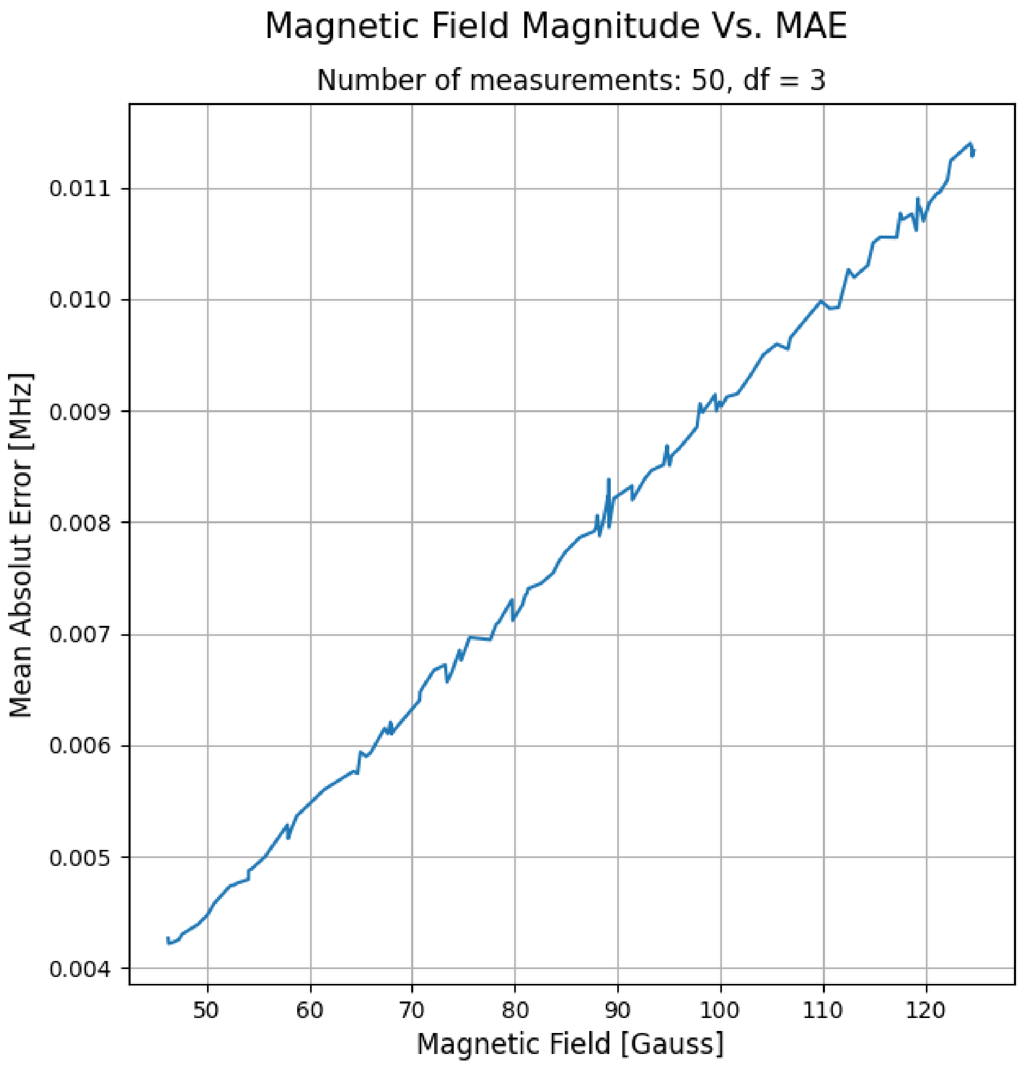

Figure 4: $M = 50$, In this case the fluctuations are less significant and we can also see that as the magnetic field magnitude is getting bigger we have increasing MAE values like we expected, and mentioned in previous sections.

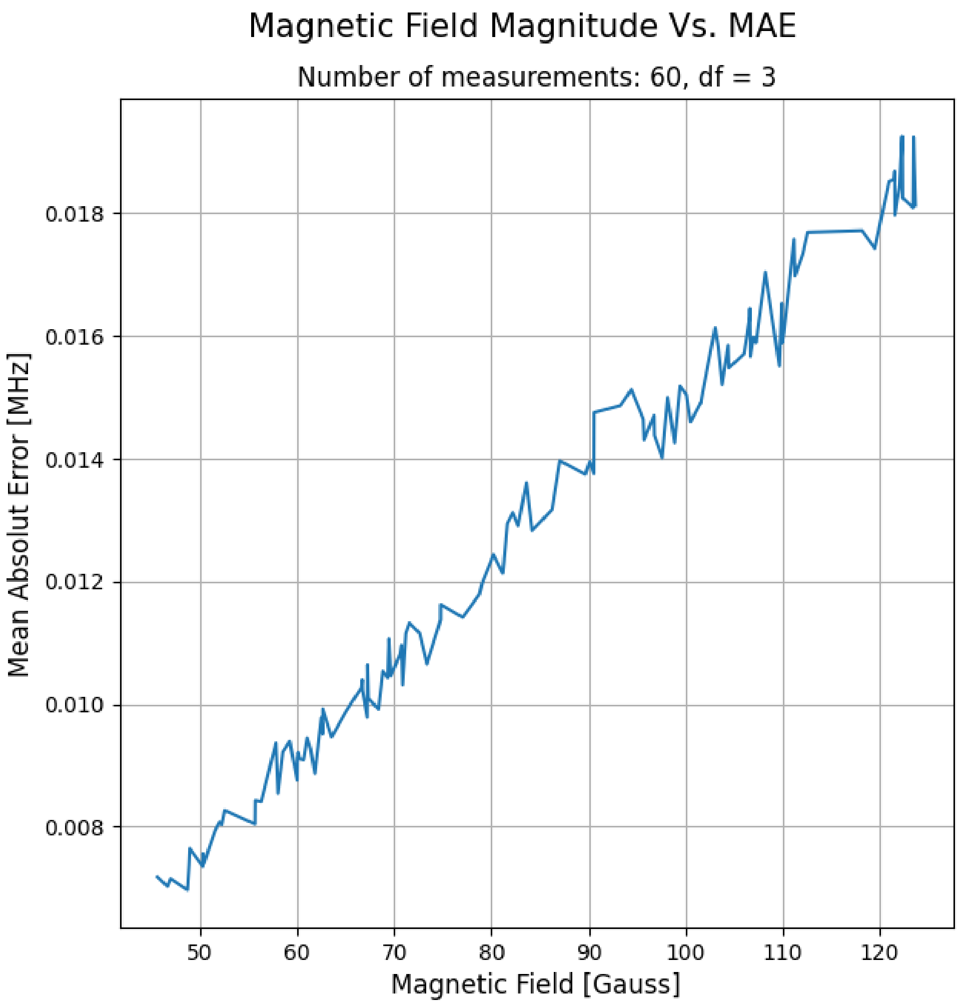

Figure 5: $M = 60$, the performance is also passing the sanity test we mentioned as the magnetic field increases with the MAE. Currently trying to improve this model.