Estimating Signal Frequency Components using LSTM

Juan Jimenez*, Ajay Thakkar†
Department of Electrical and Computer Engineering, Stevens Institute of Technology
Hoboken, NJ
Email: *jjimene6@stevens.edu, †athakka5@stevens.edu

***Abstract*—Accurately extracting frequency components from a given signal is a fundamentally critical issue in the domain of signal processing. In most signal processing applications mathematical techniques like the Fast-Fourier-Transform (FFT) are employed to calculate the discrete Fourier transform (DFT) of a given signal to analyze its frequency components in the frequency domain. These techniques can be computationally expensive and require the dedication of a significant amount of computational resources to meet the requirements of real-time systems. This paper proposes a RNN based network which learns to estimate the frequency components of time domain signals based on deep learning methods. Single and combined periodic signals are generated for training our model, with the addition of uniform and Gauss white noise as well as phase differences to make the model more robust and have greater generalization potential. Our model shows significant predictive potential, as evaluated through our extensive testing with synthetic and real signals. Click here to see the GitHub repository.**

***Index Terms*—Signal Processing, FFT, LSTM, Frequency Estimation, RNN.**

I. Introduction

The accurate extraction of frequency components from signals constitutes a foundational challenge within the realm of signal processing. These extracted periodic signals are imperative in many real world issues, such as electrocardiography and signal gathering from communication systems, robots, and radar. Traditionally, mathematical techniques such as the Fast Fourier Transform (FFT) have been pivotal for calculating the discrete Fourier transform (DFT) of signals, offering insights into their frequency domain characteristics. These methods are able to achieve great performance in accuracy, however these methods often pose computational challenges, particularly in applications where real-time processing is imperative. Additionally latency can be introduced into the system as the FFT can only be performed after receiving the entire signal. The demand for significant computational resources to implement these techniques effectively as well as the added latency issues have prompted exploration into alternative approaches.

In response to these challenges, this paper introduces a novel solution: a Recurrent Neural Network (RNN)-based network designed to learn and estimate frequency components of time-domain signals through deep learning methodologies. Departing from traditional mathematical algorithms, our proposed model takes in a given signal and learns to identify the specific frequency components of that signal.

To train our RNN model, we generate single and combined periodic signals, augmenting them with uniform and Gauss white noise, as well as introducing phase differences. This augmentation not only enhances the model's robustness but also broadens its generalization potential, allowing it to effectively handle diverse and complex signal scenarios.

Our evaluation focuses on synthetic signals, and the results demonstrate the significant predictive potential of our model. Through extensive testing, we showcase the effectiveness of our approach in accurately identifying frequency components, even in the presence of noise and phase variations.

II. Contributions

Juan Jimenez

- Wrote abstract, introduction, tools and technologies sections, multiple technologies, and problems sections
- Assisted in coding the python script used to generate the synthetic data
- Helped setup toolchains and workflow for experimenting
- Co-wrote both single and multiple frequency models
- Helped in collecting real data in ECE lab
- Optimized multiple frequency model

Ajay Thakkar

- Wrote methodology, data section, single frequency, results, and conclusion sections
- Wrote a majority of the python script used to generate the synthetic data
- Co-wrote both single and multiple frequency models
- Helped in collecting real data in ECE lab
- Optimized single frequency model

III. Methodology

Temporal dependencies play a crucial role in signal frequency estimation due to the inherent dynamic nature of signals. In the context of signals, especially those containing multiple frequencies, the relationship between different points in the time sequence is essential for accurately capturing the periodic patterns associated with each frequency component. Traditional Fourier-based methods, such as Fast Fourier Transforms (FFT), treat signals as stationary and fail to account for variations over time. In contrast, the use of Long Short-Term Memory (LSTM) layers in the proposed architecture allows the model to capture and learn the temporal dependencies within the signal sequence. This is particularly significant because

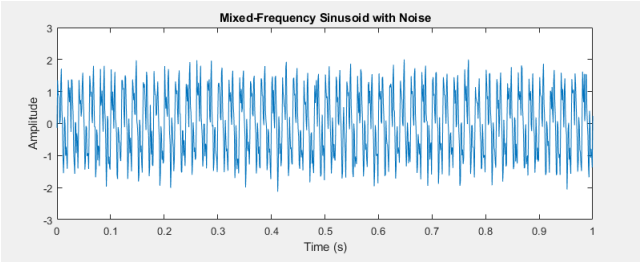

Fig. 1. Time-Domain Sinusoid with Noise

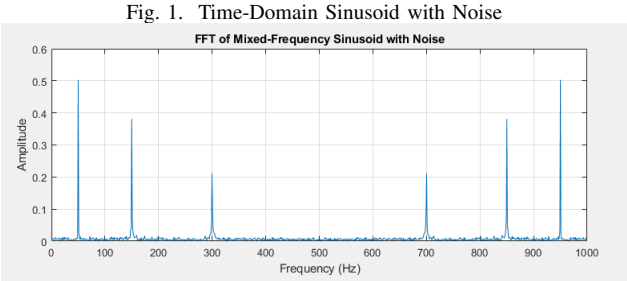

Fig. 2. Frequency Components of Sinusoid

the presence of multiple frequencies introduces complexities, such as frequency modulations and variations in amplitude and phase, which are inherently time-dependent.

LSTMs, with their memory cells and gating mechanisms, excel in preserving information over extended time intervals, making them well-suited for modeling the evolving dynamics of signals. The ability to consider both past and future points in the sequence enables the model to discern how frequencies interact over time, providing a more nuanced understanding of the signal's structure. This temporal context is crucial for accurately predicting the frequencies present in a signal, as the influence of past and future points informs the model about the evolving nature of the signal and the relationships between different frequency components. Consequently, incorporating LSTM layers in the architecture enhances the model's capacity to handle the time-varying complexities inherent in signals with multiple frequencies, making it more adept at precise signal frequency estimation.

A fully connected layer enables the model to learn complex relationships and interactions between the features extracted by the LSTM layers. This is crucial when dealing with signals containing an unpredictable number of frequencies, as the fully connected layer allows the model to discern patterns and combinations of frequency components. The dense connections between neurons in the fully connected layer facilitate the learning of intricate relationships between the encoded features, enabling the model to adapt to various signal complexities. In the context of an unknown number of frequencies, the flexibility provided by the fully connected layer is invaluable. Unlike fixed-size output layers, a fully connected layer allows the model to dynamically adjust its output based on the encoded information. This flexibility is crucial for achieving accurate and versatile signal frequency estimation across diverse signal scenarios.

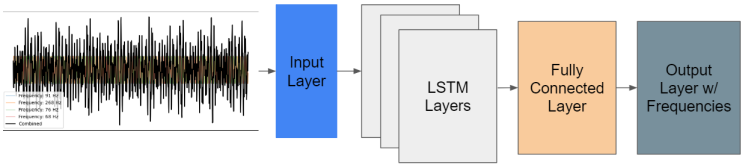

Fig. 3. Caption

IV. Data

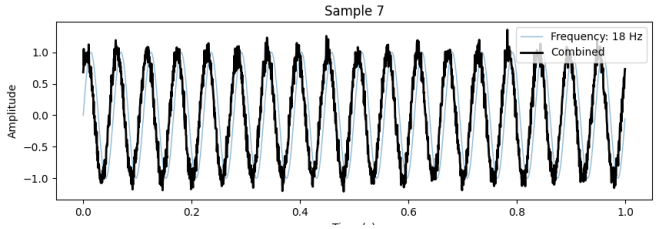

Fig. 4. Single Frequency Sinusoid

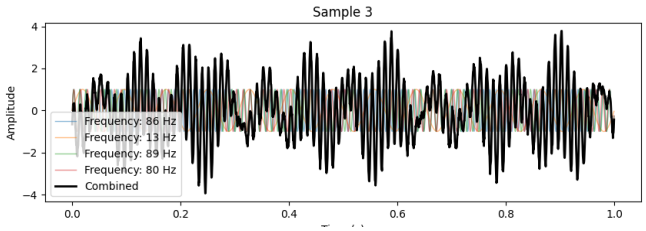

Fig. 5. Mixed Frequency Sinusoid

Our approach to the training data was to create synthetic sinusoids which should mimic real signals that something like an antenna to capture. To do this, we use python to create a string of numbers that represent a sinusoid and label it with the frequency components that make up that sinusoid. There are various parameters which we can tune to narrow or expand our dataset's scope.

To create a sinusoid, we follow the simple equation

$$y = A * sin(2\pi ft + \phi) + N \tag{1}$$

where A represents amplitude, f represents the frequency, ϕ represents the phase, N represents noise, and t represents the current time increment. The time increment corresponds with how many sample points will make up one sinusoid, but this has some constraints on it such as the Nyquist sampling theorem:

$$fNyquist = 2 * fMax \tag{2}$$

The Nyquist sampling theorem dictates that the number of samples that make up the sinusoid must be at least two times the max frequency component in the sinusoid in order for the discrete form of the signal to accurately reflect the signal's frequency. We use oversampling to give more resolution to

the digitized signal, and so for a max frequency of 100 Hz we use 299 points.

We include to replicate real-life signals, and this is done with a random number generator with a normal distribution. The amplitude is chosen with a random number generator with a range from 20 to 100. The phase is also chosen with a random number generator with a range from 0 to 2π.

We wanted to simulate sinusoids with up to 3 frequency components, so first a random number is selected to determine the number of frequency components in the sinusoid and their values. Then a sinusoid is made for each of the frequency components and added together and normalized to create the combined sinusoid.

To test the model's accuracy with both single frequency sinusoids and mixed frequency sinusoids, we made one dataset with 100,000 points which contained single frequencies ranging from 0-100Hz. We used a split of 80-20, yielding a training set of 80,000 sinusoids and a test set of 20,000 sinusoids. We then had a validation split of 20% of our training data.

Our other dataset consisted of 200,000 points which contained mixed frequencies from 0-10Hz in which we followed the same split. This yielded a training set of 160,000 sinusoids and a testing set of 40,000 sinusoids with a validation set of 20% of our training set.

V. Real Data

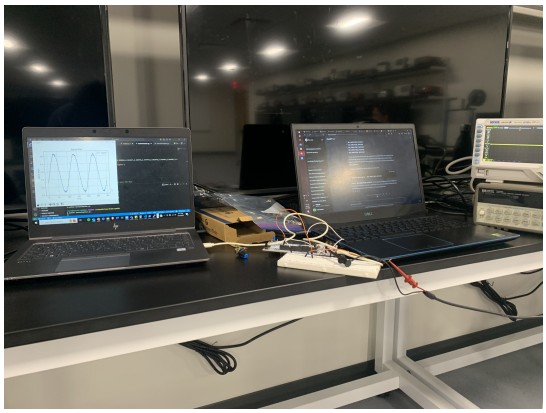

Fig. 6. Real Sinusoid Acquisition

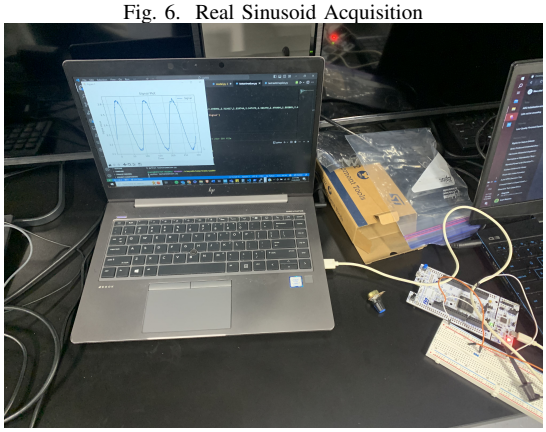

Fig. 7. Real Sinusoid Acquisition

While our training dataset is all synthetic, we wanted to make sure that this would be applicable to real discrete sinusoids. To do this, we used an STM32 microcontroller and an Analog-to-Digital Converter to sample an analog signal from a waveform generator. The waveform generator allowed us to add noise and phase to replicate a signal that would be seen in real-world applications, and the STM32 allowed us to sample at the same rate as was used in our synthetic dataset.

In the end, we arbitrarily picked a few frequencies between 1-100 Hz with varying levels of noise and sampled them at a rate of 299 samples per second. This matches the rate at which we made the synthetic data, allowing us to use our trained model to predict the frequency of our real sinusoid. After testing that the Analog-to-Digital converter was working, we transmitted 299 points of data over UART to a computer and hand labelled the data.

For multiple frequencies, it was hard to generate a true analog multi-frequency sinusoid, however in theory adding two of our real signals would yield what a multi-frequency analog signal would look like. Therefore, we planned to simply add the single frequency sinusoids we sampled to create a mixture of analog multi-frequency sinusoids to evaluate our model with.

VI. Tools and Technologies

Several software packages were used in the creation of this project. For the generation of synthethic signals python and numpy were utilized. For pre-processing, we utilized Pandas and scikit-learn to import our data from a CSV and normalize the data. The actual model was created using TensorFlow and Keras. Training was carried out on physical hardware on an NVDIA GeForce RTX 2070 SUPER and an NVIDIA GeForce RTX 3060. The model was tested on an Intel Core i7-7700 @ 3.60Ghz CPU. To evaluate our model on real data, we utilized an STM32F412GZ Board and their integrated STMCube toolchain for compiling our code and flashing onto the board.

VII. Experiments

A. Single Frequency Sinusoid

At first, we just wanted our model to accomplish the basic task of predicting the frequency of a sinusoid given varying amplitudes, noise, and phase in some frequency range. To do this, we used Keras to make a stacked LSTM layer. We came up with the model in Figure 8. The model had 631635 trainable parameters which translated to 2.41 MB in memory.

The variables surrounding this task can be described through Table I

Metric	Value
Frequency	1-100Hz
Sampling Rate	299
# of Sinusoids	100,000

TABLE I
1-100Hz Details

```
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 299, 299)          359996

batch_normalization_3 (Bat   (None, 299, 299)          1196
chNormalization)

lstm_4 (LSTM)                (None, 299, 128)          219136

batch_normalization_4 (Bat   (None, 299, 128)          512
chNormalization)

lstm_5 (LSTM)                (None, 64)                49408

batch_normalization_5 (Bat   (None, 64)                256
chNormalization)

flatten_1 (Flatten)          (None, 64)                0

dense_2 (Dense)              (None, 32)                2080

dense_3 (Dense)              (None, 1)                 33

...
Trainable params: 631635 (2.41 MB)
Non-trainable params: 982 (3.84 KB)
```

Fig. 8. Single Frequency Model Iteration 1

We used a loss function of mean average error, which we believed was the best way to quantify loss since we wanted our predictions to be as close to our labels as possible. After training in 20 epochs with the Adam optimizer, a learning rate of .001, and a batch size of 256, we achieved a loss of .7364. Training time was a total of 17 minutes.

After noticing that in 20 epochs our model did not converge, we implemented some optimization techniques such as learning rate callbacks and early stopping which yielded the final model we decided to showcase in the results section.

After seeing how accurate the model was for predicting a frequency range of 1-100 Hz, we wanted to see how good it was at predicting a higher frequency range. This would require increasing the number of samples that make up a sinusoid due to the Nyquist Sampling Theorem. So, we tried increasing the frequency range to 1-250 Hz and the number of samples to 500. Since there are now many more frequencies for the model to learn, we wanted there to be ample reference for the model, so we also increased the number of data points to 200,000. This yielded new variables for our new task found in Table II.

Metric	Value
Frequency	1-250Hz
Sampling Rate	500
# of Sinusoids	200,000

TABLE II
1-250HZ DETAILS

We theorized that having one neuron in the LSTM for each sample in the sinusoid helps the model place importance on certain points and therefore determine the frequency more accurately, and so we increased the number of neurons in the first LSTM layer of the stack to 500. However, when trying to train we had over a million trainable parameters and with more data points it would take a long time to train. So, we tried adding a dense layer at the beginning with 500 neurons connecting to an LSTM stack with less neurons hoping this would still allow for a way to place importance on certain points.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 500, 500)          1000

dropout (Dropout)            (None, 500, 500)          0

lstm (LSTM)                  (None, 500, 64)           144640

batch_normalization (Batch   (None, 500, 64)           256
Normalization)

lstm_1 (LSTM)                (None, 500, 32)           12416

batch_normalization_1 (Bat   (None, 500, 32)           128
chNormalization)

lstm_2 (LSTM)                (None, 8)                 1312

batch_normalization_2 (Bat   (None, 8)                 32
chNormalization)

flatten (Flatten)            (None, 8)                 0

dense_1 (Dense)              (None, 1)                 9

=================================================================
Total params: 159793 (624.19 KB)
Trainable params: 159585 (623.38 KB)
Non-trainable params: 208 (832.00 Byte)
```

Fig. 9. Single Frequency Model Iteration 2

Using the model described in Figure 9, we essentially tried using a smaller model for a wider range of frequencies to achieve a more general model. After training with the same parameters of the first experiment, we achieved a test loss of 1.0344 in 25 epochs after 32 minutes of training.

B. Multiple Frequency Sinusoid

The original objective of our project was to predict a signal composed of multiple frequency components. Once we had experimented with our model with a single sinusoid component we determined it was appropriate to see if our model would be able to predict signals made up of up to three frequencies. In order to experiment with this we changed our synthetic data set generation to create signals of one, two, and three components at random and modified to last layer from Figure 8 to have a final dense layer of three nodes so as to change the model to multi-output regression. We also added twice as many samples to the data set (for a total of 200,000 samples) as we reasoned that the model might benefit from more samples to learn from as there would be more possible combinations of frequencies with the added three frequency components. The labels of our data were also zero-padded to account for the fact that not all signals had up to three frequency components. The hyperparameters of these models were the same as those used in the single frequency section running with the same number of epochs and optimizer. The results of this test were rather disappointing, as we discovered that our model did not have the predictive capabilities which it had displayed for single frequency components. The model had no trouble at

predicting signals with one frequency component, even being able to guess the zeros from the padded labels. However, when it came to predicting the frequencies of multiple components, it was completely wrong, an example of this phenomenon can be seen below:

```
[[9 6 0]]
1/1 [==============================] - 0
    ↪ s 36ms/step
[[7.4612494  7.3376575  0.18250757]]
[[1 0 0]]
1/1 [==============================] - 0
    ↪ s 36ms/step
[[ 1.1530275   0.02986774 -0.01399648]]
[[4 0 0]]
1/1 [==============================] - 0
    ↪ s 35ms/step
[[4.038736   0.18943879 0.04057801]]
[[6 0 0]]
1/1 [==============================] - 0
    ↪ s 39ms/step
[[ 6.0909653   0.18605796 -0.02926266]]
[[ 5 10  1]]
1/1 [==============================] - 0
    ↪ s 39ms/step
[[5.037971  4.941992  5.1115823]]
[[9 0 0]]
1/1 [==============================] - 0
    ↪ s 37ms/step
[[8.784639   0.26631105 0.10548395]]
[[7 6 0]]
1/1 [==============================] - 0
    ↪ s 36ms/step
[[ 6.5200205  6.292043  -0.0309853]]
```

Listing 1. Output of Multiple Frequency Signal

After some tuning of hyperparameters and trying various optimization methods including normalization, changing the density of our LSTM layers, adding more data, and adjusting our learning rate and optimization functions we continued to observe the averaging behavior in our results. We realized that there was a massive problem with the current setup of the model: the deep learning model which we had created was taking order into account, for language models the order of words is extremely important, but for our combined frequencies whats important is not the particular order of the frequencies but the actual value of the frequencies that are contained within the combined frequency. We used the mean absolute error as our loss function for our model and we realized that this was the source of the averaging behavior we were witnessing in our predicted results. In order to alleviate this problem we wrote our own custom loss function, where we sort our predicted and true labels individually, calculate the mean absolute error label by label, and then take the mean of all of those errors as our loss, the code below is the python implementation of our modified loss function:

```
def permutation_invariant_loss(y_true, y_pred)
    :
    # Sort the true and predicted values along
        the last axis
    y_true_sorted = tf.sort(y_true, axis=-1)
    y_pred_sorted = tf.sort(y_pred, axis=-1)

    # Calculate the mean absolute error
        between the sorted true and predicted
        values
    return K.mean(K.abs(y_true_sorted -
        y_pred_sorted), axis=-1)
```

Listing 2. Custom Loss Function

This modified loss function allowed the model to focus on being accurate with the frequencies rather than worrying about the position of the predictions it was making. We also modified the architecture of the deep learning model we created to better represent our problem. Instead of having stacked LSTM layers we decided to go instead for a branching LSTM layer approach. Our thinking was that each branch would represent one of the three frequency components which could make up our signal and we hoped that each branch would be able to abstract each single frequency component and then have the predictions concatenated at the end. Figure 10 is a graphic displaying the high level architecture of our frequency estimator model. Figure 11 shows a summary of the parameters used in the model as well as the memory usage on the computer. Despite being a relatively small model with only 64 nodes per LSTM layer, we reasoned that for frequencies ranging from 1-100 Hz the model should have the complexity necessary to prove the effectiveness of our new approach. Running our model on 15 epochs without any optimizations yielded the following results and proved the effectiveness of this architecture at predicting mixed signals:

```
[[60 79  0]]
1/1 [==============================] - 0
    ↪ s 34ms/step
[[7.9380295e+01 5.5823364e+01 9.7100623e
    ↪ -04]]
[[20  0  0]]
1/1 [==============================] - 0
    ↪ s 37ms/step
[[-0.24019669 19.448805    0.26118606]]
[[51  0  0]]
1/1 [==============================] - 0
    ↪ s 42ms/step
[[1.9001137e-02 5.1232677e+01 2.1893805e
    ↪ -01]]
[[56  0  0]]
1/1 [==============================] - 0
    ↪ s 28ms/step
[[ 0.14844789 56.208225    0.22355312]]
[[17 28  0]]
1/1 [==============================] - 0
    ↪ s 41ms/step
[[14.959453 28.089502  4.972579]]
```

```
[[22  0  0]]
1/1 [==============================] - 0
    ↪ s 40ms/step
[[-0.15763746 21.468786    0.2873538 ]]
[[78 98 50]]
1/1 [==============================] - 0
    ↪ s 32ms/step
[[97.570656 78.67236  49.6501  ]]
```

Listing 3. Branching Model Output Predictions

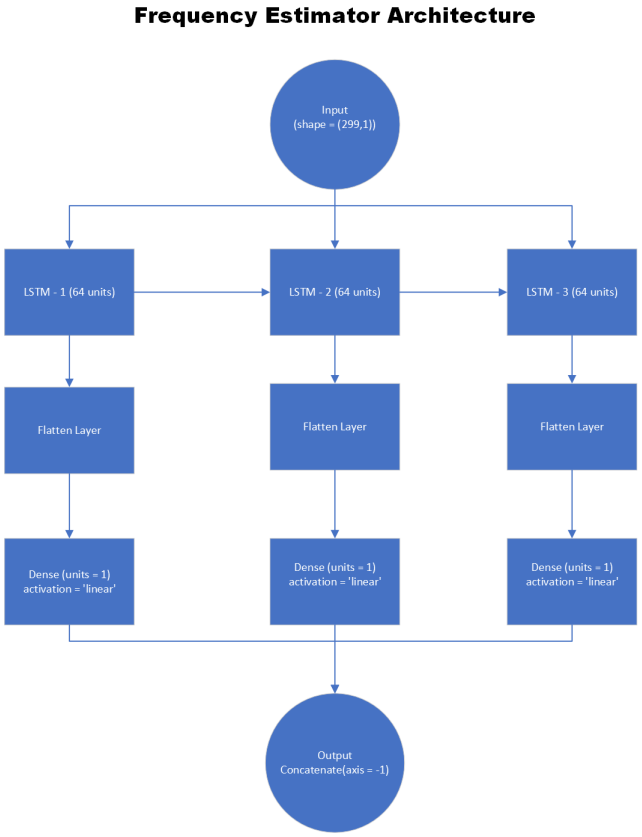

Fig. 10. High level overview of the Frequency Estimator Architecture

The results of our testing and experimentation proved that this was the correct model for dealing with mixed signal frequencies and that it should serve as the baseline model for our mixed signal frequency prediction. The results from this test provided a test loss of around 1.311, however the model proved its effectiveness at learning to predict combined frequency components which was the major roadblocked which we faced in our implementation of our frequency predicting model.

VIII. Results

A. Single Frequency Sinusoid

The baseline model for our single frequency sinusoid can be found in Figure 8, and the training details can be found in Table III.

Predictions from the baseline model on our real data were very impressive as can be seen below:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================
 input_1 (InputLayer)        [(None, 299, 1)]          0         []

 lstm (LSTM)                 (None, 299, 64)           16896     ['input_1[0][0]']

 lstm_1 (LSTM)               (None, 299, 64)           33024     ['lstm[0][0]']

 lstm_2 (LSTM)               (None, 299, 64)           33024     ['lstm_1[0][0]']

 flatten (Flatten)           (None, 19136)             0         ['lstm[0][0]']

 flatten_1 (Flatten)         (None, 19136)             0         ['lstm_1[0][0]']

 flatten_2 (Flatten)         (None, 19136)             0         ['lstm_2[0][0]']

 output_branch_1 (Dense)     (None, 1)                 19137     ['flatten[0][0]']

 output_branch_2 (Dense)     (None, 1)                 19137     ['flatten_1[0][0]']

 output_branch_3 (Dense)     (None, 1)                 19137     ['flatten_2[0][0]']

 concatenate (Concatenate)   (None, 3)                 0         ['output_branch_1[0][0]',
                                                                  'output_branch_2[0][0]',
                                                                  'output_branch_3[0][0]']

=================================================================
Total params: 140355 (548.26 KB)
Trainable params: 140355 (548.26 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Fig. 11. Branching Model Summary Report

Metric	Value
Optimizer	Adam
Learning Rate	.001
Batch Size	256
Epochs	20
Training Time	17 min
MAE	.7364
MSE	1.0260
MAPE	2.7091

TABLE III
SINGLE FREQUENCY UNOPTIMIZED RESULTS

```
Real Sinusoid Tests:
[[1]]
1/1 [==============================] - 1
    ↪ s 725ms/step
[[1.594873]]
[[10]]
1/1 [==============================] - 0
    ↪ s 73ms/step
[[9.389566]]
[[19]]
1/1 [==============================] - 0
    ↪ s 33ms/step
[[18.276806]]
[[27]]
1/1 [==============================] - 0
    ↪ s 41ms/step
[[26.469257]]
[[36]]
1/1 [==============================] - 0
    ↪ s 65ms/step
[[35.501297]]
[[45]]
1/1 [==============================] - 0
    ↪ s 34ms/step
[[45.40043]]
```

```
[[54]]
1/1 [==============================] - 0
    ↪ s 36ms/step
[[53.19606]]
[[90]]
1/1 [==============================] - 0
    ↪ s 34ms/step
[[91.0962]]
```

Listing 4. Single Frequency Model Real Outputs

To reduce overfitting and optimize our model, we decided to introduce callbacks to our model and train for more epochs. An early stop was implemented which would stop training if the loss did not decrease over four epochs, and the learning rate was decreased by a factor of .1 if the valuation loss did not decrease in two epochs. We also trained for 40 epochs to ensure convergence. This yielded the results found in Table IV, and a plot of the learning rate over epoch iterations can be found in Figure 12.

Metric	Value
Optimizer	Adam
Learning Rate	.001
Batch Size	256
Epochs	40
Training Time	33 min
MAE	.1125
MSE	.0408
MAPE	.4565

TABLE IV
SINGLE FREQUENCY OPTIMIZED RESULTS

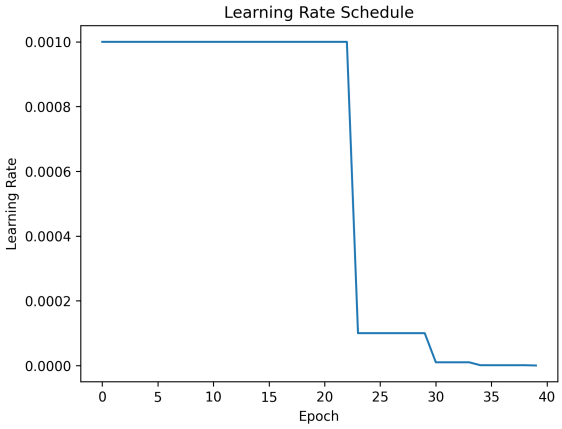

Fig. 12. Learning Rate Schedule Single Frequency

The predictions from our optimized model on our real data can be seen below:

```
Real Sinusoid Tests:
[[1]]
1/1 [==============================] - 1
    ↪ s 754ms/step
[[1.5843158]]
[[10]]
1/1 [==============================] - 0
    ↪ s 60ms/step
[[8.991657]]
[[19]]
1/1 [==============================] - 0
    ↪ s 63ms/step
[[18.054453]]
[[27]]
1/1 [==============================] - 0
    ↪ s 45ms/step
[[27.597923]]
[[36]]
1/1 [==============================] - 0
    ↪ s 68ms/step
[[36.59767]]
[[45]]
1/1 [==============================] - 0
    ↪ s 69ms/step
[[45.43515]]
[[54]]
1/1 [==============================] - 0
    ↪ s 99ms/step
[[54.40014]]
[[90]]
1/1 [==============================] - 0
    ↪ s 58ms/step
[[90.656]]
```

Listing 5. Single Frequency Model Real Outputs

B. Multiple Frequency

The training details for the baseline multiple frequency model can be found in Table V.

Metric	Value
Optimizer	Adam
Learning Rate	.001
Batch Size	256
Epochs	20
Training Time	28 min
Custom Loss	.3556

TABLE V
MULTIPLE FREQUENCY UNOPTIMIZED RESULTS

To reduce overfitting, we attempted using the same optimization techniques as the single frequency model with details in Figure VI and a learning rate schedule detailed 13.

Metric	Value
Optimizer	Adam
Learning Rate	.001
Batch Size	256
Epochs	40
Training Time	33 min
Custom Loss	.6660

TABLE VI
MULTIPLE FREQUENCY OPTIMIZED RESULTS

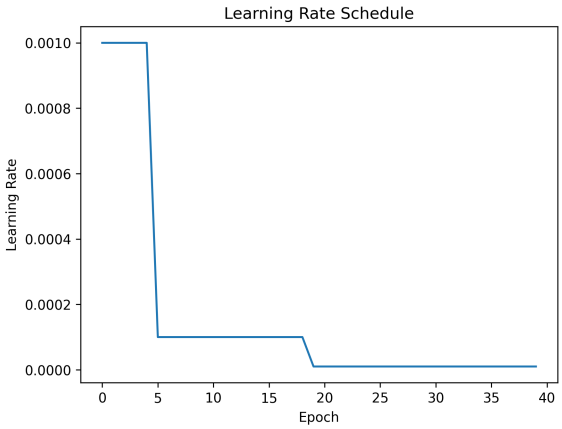

Fig. 13. Learning Rate Schedule Multiple Frequency

We found that the optimization did not help much in terms of overall loss, but in both instances we were able to predict synthetic data very well as can be seen below:

```
Synthetic Sinusoid Tests:
[[69  0  0]]
1/1 [==============================] - 0s 34ms/step
[[ 6.887474e+01  8.094144e-02 -7.955432e-03]]
[[60 75 31]]
1/1 [==============================] - 0s 89ms/step
[[76.56334  57.653023 29.73457 ]]
[[78  0  0]]
1/1 [==============================] - 0s 84ms/step
[[ 7.8195938e+01 -7.9611748e-02  4.8662648e-03]]
[[52  0  0]]
1/1 [==============================] - 0s 49ms/step
[[5.1761131e+01 5.3212315e-02 2.7601570e-02]]
[[91 93  0]]
1/1 [==============================] - 0s 87ms/step
[[91.03203   93.37774    0.0978315]]
```

Listing 6. Multiple Frequency Model Synthetic Outputs

IX. Problems/Issues

In regards to the technologies there was little to no problems setting up the various software libraries and getting the model to run. During our experiment phase we decided to test much more complex versions of our baseline model with many more data points of synthetic data to evaluate performance as we up scaled our model. Training on our CPU's was no longer appropriate as the more complex models would often take tens of minutes per epoch to train. To better experiment and work with our model we therefore decided to take advantage of the graphics cards we had at our disposal, specifically the NVIDIA GeForce RTX 2070 SUPER and the NVIDIA GeForce RTX 3060. In order to run our models with the graphics cards rather than the CPU it is necessary to install the appropriate GPU Tensor libraries which can be found here. These steps were all completed on machines running Windows 10 with WSL installed with the Ubuntu distro. Training on the GPU's drastically lowered our training time and allowed us to rapidly test ideas and new ways to improve our baseline model.

In regards to our multiple frequency model, we were not able to accurately predict our real data and did not have enough time to experiment and figure out why. However, we posit that it could do with the processing of our real data, or overfitting to the synthetic data.

X. Conclusion

This paper addresses the challenge of accurately extracting frequency components from signals in signal processing. Traditional methods, such as Fast Fourier Transform (FFT), are computationally expensive and may not meet the real-time processing demands. The paper proposes a novel solution using a Recurrent Neural Network (RNN)-based network, departing from traditional mathematical algorithms. The RNN model is trained on synthetic signals with single and combined periodic components, including noise and phase differences, to enhance robustness and generalization. The evaluation, primarily conducted on synthetic signals, demonstrates the model's significant predictive potential, even in the presence of noise and phase variations. The authors also present real data experiments using an STM32 microcontroller and an Analog-to-Digital Converter, showing the model's applicability to real-world scenarios. The paper introduces a custom loss function to address the issue of averaging behavior in predicting multiple frequency components. The proposed branching LSTM architecture proves effective in predicting mixed signals. The paper contributes to the field of signal processing by offering an alternative, data-driven approach to frequency component estimation.