Non-Uniform Sampling Super Resolution

Project Number: **22-1-1-2423**

Project Report

Student: Tomer Rajwan ID: 212290209

Student: Etay Ashkenazi ID: 324078500

Supervisor: Khen Cohen

Project Carried Out at: Tel Aviv University

Contents

List of figures

List of tables

List of equations

Abstract

In this project, we explore the use of non-uniform sampling methods for improving signal reconstruction. To do that we incorporate machine learning algorithms to identify the contexts where ML preforms better compared to other sampling methods.

We will look at cases where we don't attain Nyquist minimal sampling frequency and try to find the optimal sampling points. The algorithm will work iteratively, by choosing a sampling point based only on values sampled prior to that point (causality), one point at a time. We will assume limited memory, both in the ML, and in the interpolator.

Recurrent neural networks (RNN), such as LSTM and GRU, have had outstanding success in sequential modelling, and have been applied in many application fields, including signal processing. These models have become an obvious choice for time series modelling and a promising tool for handling irregular time series data.

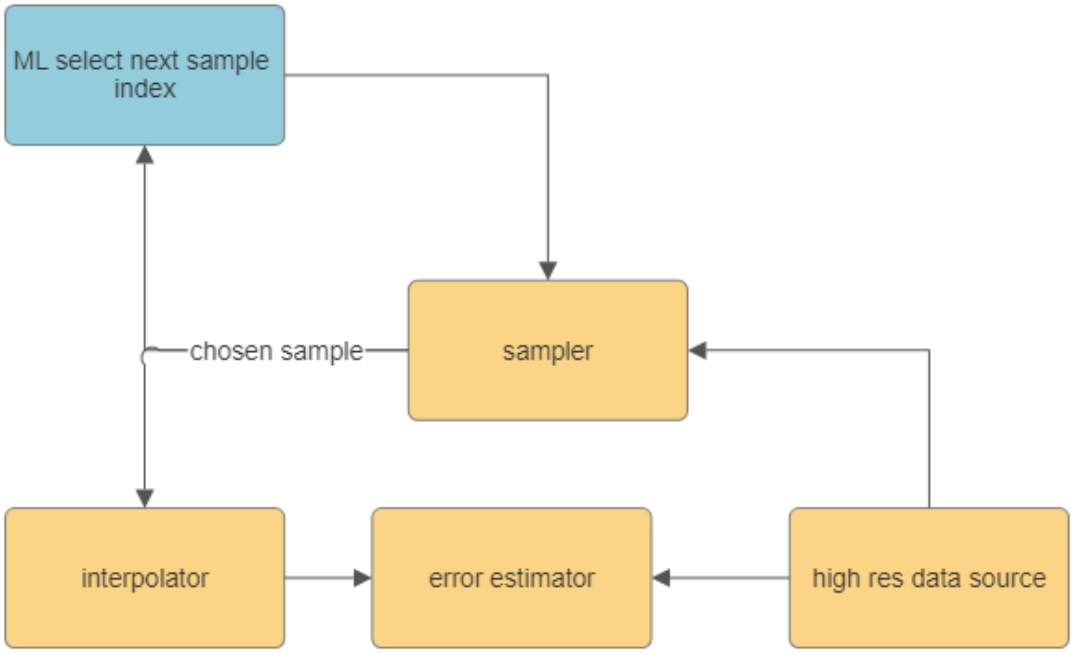


Figure 1: Block diagram

1 Introduction

Our project explores the possibility of improving reconstruction results through non-uniform sampling of a given signal. We aim to compare the performance of linear and cubic spline interpolation on signals with varying frequencies, both above and below the Nyquist conditions for uniform sampling. By incorporating machine learning algorithms, we aim to identify where does ML yield the most effective results in this context.

Goals

The goals we set for our project is to compare between different sampling methods. Training an ML model to sample as one of the methods. Because we weren't sure that an improvement over all other methods is possible just by choosing different slots to sample, we chose to set as a goal for the model to at least perform better than the best other method by 5%.

Motivation

Our project could have a range of potential applications, The ability to reconstruct signals from fewer samples could be particularly useful in applications where data acquisition is expensive or time-consuming, or where data storage is limited, such as MRI, where each sample is expensive and can be harmful.
Additionally, non-uniform sampling may enable the detection of features in signals that would be missed in uniform sampling, leading to improved resolution in various applications, such as audio and video processing. by taking the samples in regions of interest, we can achieve better reconstruction results.

The problem

We will take signals and divide them into frames in a rate of 30 fps, where each frame is divided into 10 slots. The challenge is to determine which slots should be sampled in order to achieve the most accurate reconstruction of the signal.
Machine learning algorithms will be used to determine the optimal sampling strategy for each frame, based on the samples up to this point.

Approaches to the problem

First is the naive approach: we will sample at a determined point;
we will compare different sampling policies, like Chebyshev or boosting sampling, that we will define later.
Second is ML approach, we will use different supervised ML networks to try and learn what are the optimal slots to choose.
the training will be done using pre-calculated right answers calculated with future knowledge of the signal.

Comparison against existing works

We couldn't find any other works concerning this exact problem. The closest we found was a work about using ML to predict the next **value** of series but the series has missing values and non-uniform intervals between samples. Or using ML to help processing the non-uniformly sampled data.

2 Theoretical background

Given 1D signal $y(t)$ and system sampling rate of T, if the spectral bandwidth of $y(t)$ is smaller than $\frac{1}{2T}$ the reconstruction of the signal will be unique, according to Nyquist theorem of sampling. Even though if the signal $y(t)$ contains higher spectral components, without any prior knowledge, the signal cannot be reconstructed uniquely.

We deal with systems that have small degree of freedom for sampling rate, so for each system sampling rate interval T ,we call a frame, we define points for sampling we will call a slot or sampling index. For example, if we set an up-sampling factor of 10, the available sampling points inside each frame, are:

$$[\, 0, 0.1T, 0.2T, 0.3T, \dots, 0.9T \,]$$

In our system, we define the problem by selecting one slot to sample at in each frame. The sampled values until this point is then used to determine (using the sampling method) the next index we want to sample at.

2.1 Sampling Methods

In the following examples we will look at deterministic sampling methods. In the following figures the blue lines represent the start/end of a frame and the red dot represent the sampled point the method picked.

2.1.1 Uniform

The uniform sampling method is sampling with constant time intervals:

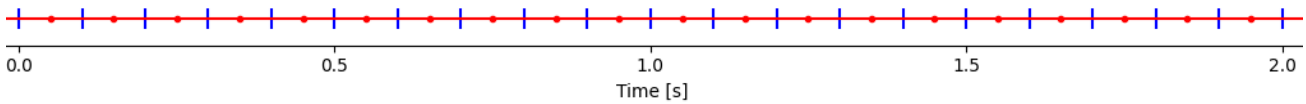


Figure 2: Uniform sampling

2.1.2 Boosting

The Boosting sampling method is sampling that tries to take sets of N samples and place the samples closer together. In the example below we take groups of 5 points and making it effetely increasing Nyquist by 25% for the interval of those 5 points.
It works by taking the outer most points of each group of N frames and choses the sampling indexes to be the closest to the center as possible (for up sampling of 10 it would be index 0 and 9) and the rest of the points are evenly spaced in that range. For example: for N = 5 and up sampling of 10:

$$index_N(n) = index_5(n\%5) = index_5(\tilde{n}) = \begin{cases} 9\,, & \tilde{n} = 0 \\ 7\,, & \tilde{n} = 1 \\ 5\,, & \tilde{n} = 2 \\ 3\,, & \tilde{n} = 3 \\ 0\,, & \tilde{n} = 4 \end{cases} \qquad (1)$$

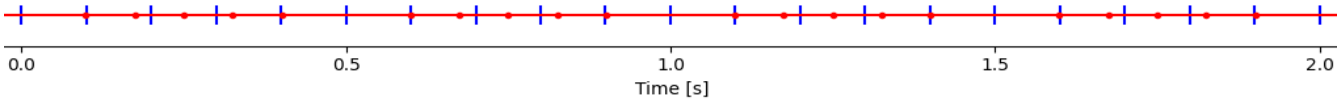


Figure 3: An image of Boosting sampling with N = 5

2.1.3 Chebyshev

The Chebyshev sampling method is sampling at points which are the roots of an order N Chebyshev polynomial.
The Chebyshev sampling points are often used as nodes in polynomial interpolation because the resulting interpolation polynomial minimizes max absolute error in an interpolation interval.

In the continues case the points are:

$$for\ each\ group:\ x_{\tilde{n}} = \cos\left(\frac{2\tilde{n}-1}{2N}\,\pi\right), \qquad \tilde{n} \in \{0, 1, 2 \dots N\} \qquad (2)$$

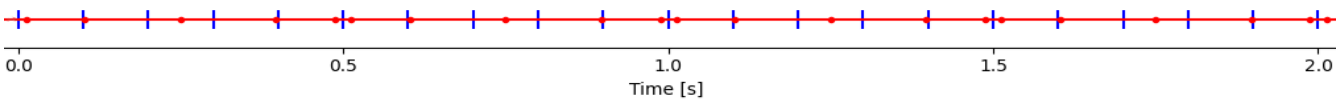


Figure 4: Chebyshev sampling with N = 5

2.1.4 Random

The Random sampling method is sampling at each frame at a random index.

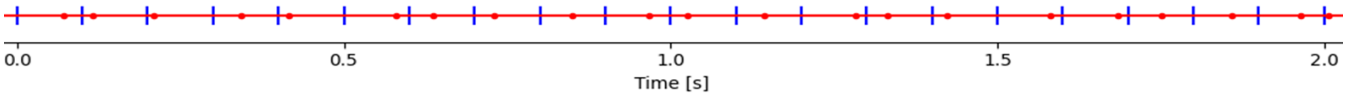


Figure 5: Random sampling

2.1.5 Conclusion

To understand the different methods better here is a graph of the time difference between each sample of the method.

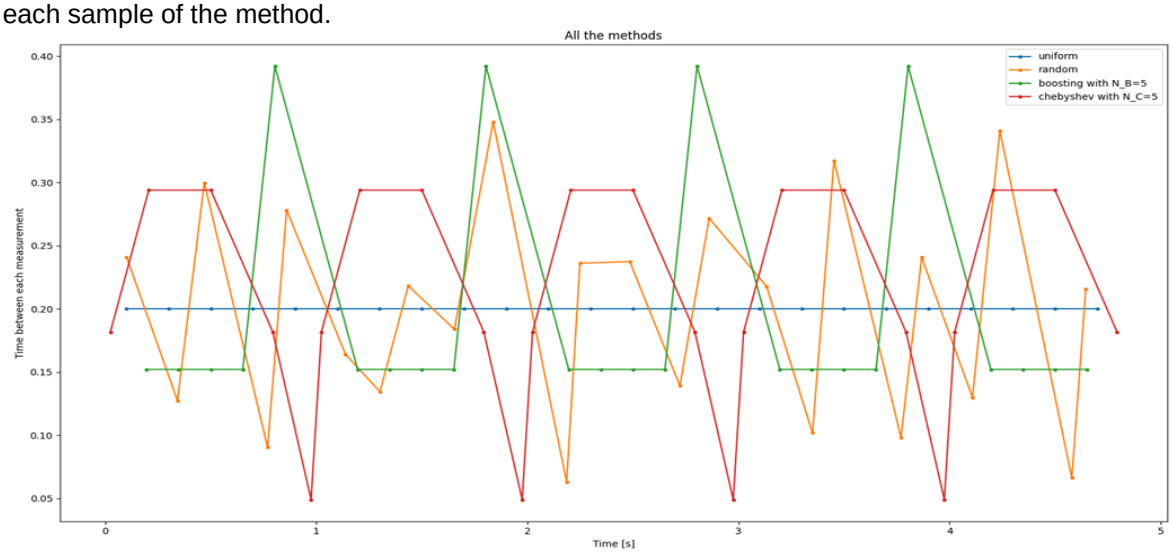


Figure 6: The time difference between samples

The uniform sampling (blue) is at a constant time between each sample.
The Chebyshev sampling method (red) is sampling at points which are the roots of an order N Chebyshev polynomial. The Chebyshev sampling method has the smallest difference between two points as seen in the graph as the lowest value in the y axis.
The Boosting sampling method (green) is sampling that tries to take sets of N samples and place the samples closer together. We can see from the graph that it has all its points at smaller intervals then uniform except from one point that we place as high as possible.
The random sampling method (orange) choose a random sampling point in the frame.

2.2 Interpolation methods

As the sampling rate of the signal doesn't assume Nyquist frequency and even if Nyquist frequency is not maintained, perfect interpolation using sinc waves is computationally expensive and if we don't stand in Nyquist then it is not perfect. For that we chose to look at other interpolation methods in our project to get a faster interpolation using less memory and see what the effects of different interpolation methods are.

In the project, two of the ways that we use to interpolate the signals to be able to calculate the error are first order linear interpolation and Cubic Spline interpolation.

The **linear interpolation** is the weighted average of the two closest points:

$$y(x) = y_0 + \frac{(x - x_0)(y_1 - y_0)}{x_1 - x_0} \quad (3)$$

The **Cubic Spline interpolation** method was chosen also because of it is smoother and has smaller error than some other interpolating polynomials such as Lagrange polynomial. Also, we speculated that maybe the model will learn to use how it works to try and follow the signal more accurately.
we used SciPy Cubic Spline function. In cubic spline interpolation, the curve is constructed by connecting adjacent data points with a series of cubic polynomials. These polynomials are chosen in such a way that the resulting curve is continuous, and its first and second derivatives are also continuous at the data points.

2.3 Errors

There are 3 places in this project that uses error functions. First in the data creation, second in training the ML model and last at the result analysis. Here are the main error functions we used:

2.3.1 Mean Squared Error (MSE)

For most cases we used MSE as it is a good way to evaluate the differences between 2 vectors:

$$MSE = \frac{1}{n}\sum_{i=0}^{n-1}\left(Y_i - \hat{Y}_i\right)^2 \quad (4)$$

2.3.2 Mean Absolut Error (MAE)

also known as the absolute error, measures the absolute difference between the predicted and actual values. It is commonly used because it provides a robust metric that is less sensitive to outliers and yields a more interpretable measure of the prediction accuracy.

$$MAE = \frac{1}{n}\sum_{i=0}^{n-1}\left|Y_i - \hat{Y}_i\right| \quad (5)$$

2.3.3 Maximum Absolute Error

This error is the max error between 2 vectors.

$$Max\ Error = \max_{i\in[0,n-1]}\left|Y_i - \hat{Y}_i\right| \quad (6)$$

Where: Y is the original signal, and $\hat{Y}$ is the interpolated signal. both with n elements.

2.3.4 Cross Entropy Loss

Cross-entropy is a loss function commonly used in classification tasks. It measures the dissimilarity between predicted class probabilities and true class labels, making it suitable for training models in scenarios where the goal is to classify data into multiple classes accurately.

$$\mathrm{H(P,Q)} = -\sum_{x} \mathrm{P(x)} * \log(\mathrm{Q(x)}) \quad (7)$$

Where Q is the predicted probability of the class and P is a vector representing the right answer.

2.4 Derivatives and FFT

As an additional features to the model, we added an approximation of the Derivative and FFT from the last points sampled.

2.4.1 Derivative

We looked at different approximations. The first is simply learning approximation of the derivative between two points. We didn't choose this one because derivative approximations have high errors especially when the distance between two points is large we wanted an approximation using more than two points.
Second, we looked at Five-point stencil approximation but it assumes that the distances between the points are constant and we don't have that assumption here.
The approximation we chose used for the derivative at the end was taken from the paper: A simple finite-difference grid with non-constant intervals

$$f_i' = \frac{f_{i+1} - \left(\frac{h_i}{h_{i-1}}\right)^2 * f_{i-1} - \left(1 - \left(\frac{h_i}{h_{i-1}}\right)^2\right) * f_i}{h_i * \left(1 + \frac{h_i}{h_{i-1}}\right)} \quad (8)$$

Using three close points we approximate the value of the derivative at the middle point. This approximation is reached by writing down the Taylor series of f_{i-1}, ,f_{i+1}, f_i under assumption of grid with non-constant intervals.

2.4.2 FFT

For the FFT we calculate a DTFT of the last points sampled and add it as an extra feature. We didn't expect it to work as the points are not sampled at uniform intervals, so the DTFT we get is the same as the DTFT of the interpolated signal using zero-order interpolation.

2.5 Oscillating Chirp

A function we believe our algorithm can perform well is the "Oscillating Chirp", it is a sine wave that moves between two frequencies:

$$f(t) = \sin\left(t * \left(w_{base} + \Delta w * \sin\left(\frac{w_{base}}{\alpha} * t\right)\right)\right) \quad (9)$$

We expect to be able to get better interpolation using a non-uniform method in cases like this, as using better placed samples for the high frequency intervals that can help with the interpolation.
Example of the Oscillating Chirp and its STFT are on *Appendix F* .

3 Data Base

In order to train the supervised ML models (namely the LSTM), we've needed a collection of functions with the optimal slot in each frame to sample the function at. Optimal in the sense of MSE between the interpolated (in our data base the interpolation will be linear interpolation or cubic spline interpolation) samples and the high-resolution signal.
To achieve that, first we've generated a variety of signals, that we can sample, interpolate and estimate errors on them.
We've chosen to work with signal of 7 [sec], and the frame length is $\frac{1}{30} = 0.03333 \ldots$ $[sec]$ (fps = 30), and to have 10 slots at each frame.
To find the optimal sampling slots, across all the $30 \cdot 7 = 210$ frames, we will need to check all 10^{210} permutations of the slot indexes. Clearly brute force approach is infeasible.
In our project, because we work with linear interpolation and cubic spline interpolation, we are able to reduce the number of permutations to calculate.
In the linear interpolation choosing a different slot in a frame will affect the interpolated signal only in the previous and the following slots, so we can choose an optimal slot by looking at the effect of the change in a window of these frames, and reduce dramatically the computation time. The same applies to cubic spline interpolation, but with larger window of frames, each slot will only affect the surrounding four frames.
To reduce the calculation time even more we don't interpolate the whole signal and calculate that MSE, we interpolate only a small interval, surrounding the frame we are looking the optimal slots for.
Overall, our algorithm aims to find the "best" slots to sample at in order to minimize the MSE. For every frame it looks at the four frames ahead (for both the interpolation methods) and search for the slot indexes in the window, that minimize the MSE of the interpolated signal in a predefined interval.
For each frame we look at 10^4 permutations ($210 * 10^4$ total).

For our project we've wanted to train some ML models on variety of signals, so we've made 5 classes of databases:

- Sine and oscillating chirp signals in frequency between $1Hz$ to $20Hz$ with cubic spline interpolation.
- Sine, oscillating chirp and sawtooth signals in frequency between $1Hz$ to $20Hz$ with linear interpolation.
- Sum of Sine signals, from one Sines up to five sins, with linear interpolation.
- Gaussian filtered random signal with different gaussians, with linear interpolation.
- Pixel intensity from a real video, with linear interpolation.

Example of running the database creator:

```
Function number 3 of 100
unif mse: 0.003109924471750482
100%|████████████████████████████████████████████████████████████| 205/205 [02:42<00:00,  1.27it/s]
min mse (all): 0.0017571388278938974
x_slot_index 4: [6, 5, 3, 1, 0, 9, 8, 8, 8, 6, 4, 2, 0, 6, 4, 2, 0, 0, 0, 0, 0, 2, 1, 0, 7, 5, 3, 1, 0, 2,
 1, 0, 0, 0, 0, 0, 9, 7, 4, 2, 0, 0, 0, 0, 0, 4, 9, 9, 7, 6, 5, 9, 7, 5, 2, 0, 3, 2, 1, 0, 4, 4, 4, 3, 2,
1, 0, 0, 9, 9, 9, 7, 5, 3, 1, 0, 3, 7, 8, 6, 4, 2, 0, 4, 2, 1, 0, 4, 3, 1, 0, 8, 6, 4, 3, 2, 1, 1, 9, 7, 4
, 2, 9, 7, 5, 4, 3, 2, 1, 0, 4, 3, 1, 0, 6, 4, 2, 0, 9, 8, 7, 6, 5, 3, 1, 0, 0, 3, 9, 8, 6, 4, 2, 9, 7, 5,
 2, 0, 2, 1, 0, 0, 0, 0, 0, 7, 5, 3, 1, 0, 1, 1, 9, 7, 6, 5, 9, 7, 5, 2, 0, 4, 3, 3, 9, 7, 4, 1, 9, 7, 4,
2, 9, 7, 4, 1, 8, 5, 2, 0, 7, 5, 2, 0, 5, 3, 1, 0, 5, 3, 1, 0, 5, 3, 1, 0, 2, 2, 2, 9, 7, 5, 2, 0, 2]
```

Figure 7: Example of running the database creator

Here we see that the MSE of the uniform sampled of the signal is 0.0031
And the MSE after we choose the slot indexes (we see printed) is 0.0017 .

4 Architecture / Methods

4.1 DQN

At the start of the project, the first direction we tried to implement is a Deep Q Learning algorithm. The DQN is a non-supervised ML algorithm is designed to approximate and learn an optimal action-value function, also known as
Q-function.

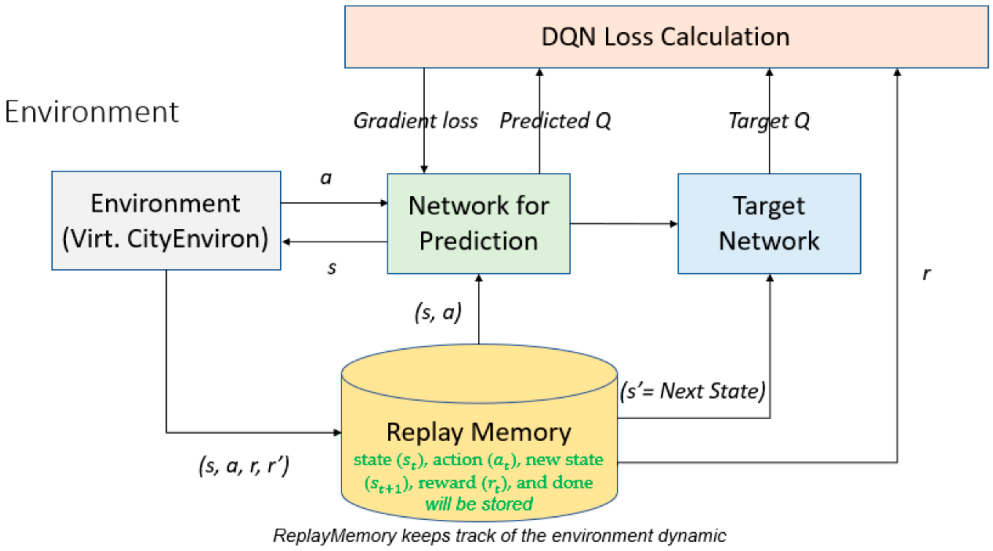


Figure 8: General DQN architecture

DQN seemed like a good fit to our problem, due to several reasons:

1. The number of possible states is very large, impossible to map.
2. The best decision at a single moment is not the greedy decision.
3. On the other hand, looking at the reward only at the end, there will be no effective gradient flow. The DQN allows to give a reward earlier.

Although we've implemented full DQN architecture, we've been unable to make it converge. Thus, we've changed course, and tried solving using Supervised architecture.

4.2 Supervised

Supervised architecture, in contrast to non-supervised architecture, is provided with the "correct" answers, and tries to learn them. In our case, we are using the database as explained in chapter *3 Data Base* , as the correct answers.

4.2.1 Inputs

We've defined the inputs as the last 20 samples, each contains the exact time and the value sampled. In addition, we can add feature vectors, such as derivative or FFT, calculated based on these 20 samples only, as explained in *2.4 Derivative and FFT*. Note all inputs are casual. (Input size = 20*(2 + number of features))

4.2.2 Outputs

The output of the ML architecture is a class-based choice representing the next index to be sampled. That is, 10 numbers holding soft weights, one per slot. During test, we used *argmax* to choose sample point.
During training, we look at the "correct" answer, and define the correct probability to be 1.0 at the "correct" answer, 0.0 elsewhere, and calculate cross-entropy loss between the output

distribution and this target. The final loss we used also contained a loss based on the weights of the model itself, this helps preventing it from exploding eigenvectors when training.
Note: we are not assuming that close answers (e.g. “correct” is 3 and predicted is 4) are better than other answers (e.g. 8). We’ve tested this assumption later on.

4.2.3 RNN / LSTM Architecture

LSTM, Long Short-Term Memory networks, is a variety of Recurrent Neural Networks (RNNs) that are capable of learning long-term dependencies. This kind of architecture performs well on sequence-to-sequence problems.

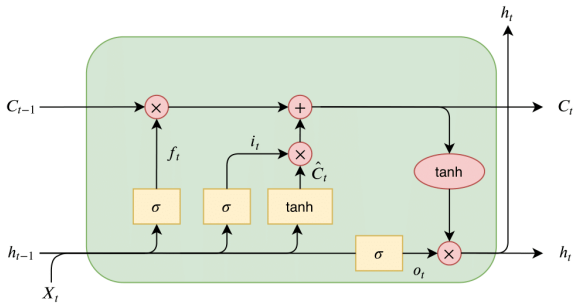


Figure 9: LSTM cell

The architecture we’ve ended is the LSTM cell Using PyTorch.nn we set it to have 7 layers of 70 weights for the hidden state and cell state. then we use a 70 weights linear layer to calculate the final output.
We use 40% to 50% drop out chance for each layer to help prevent overfit.
Tuning options are the drop out precent, the error function, the learning rate and adding or removing the feature vectors we described at chapter *4.2.1 Inputs* .

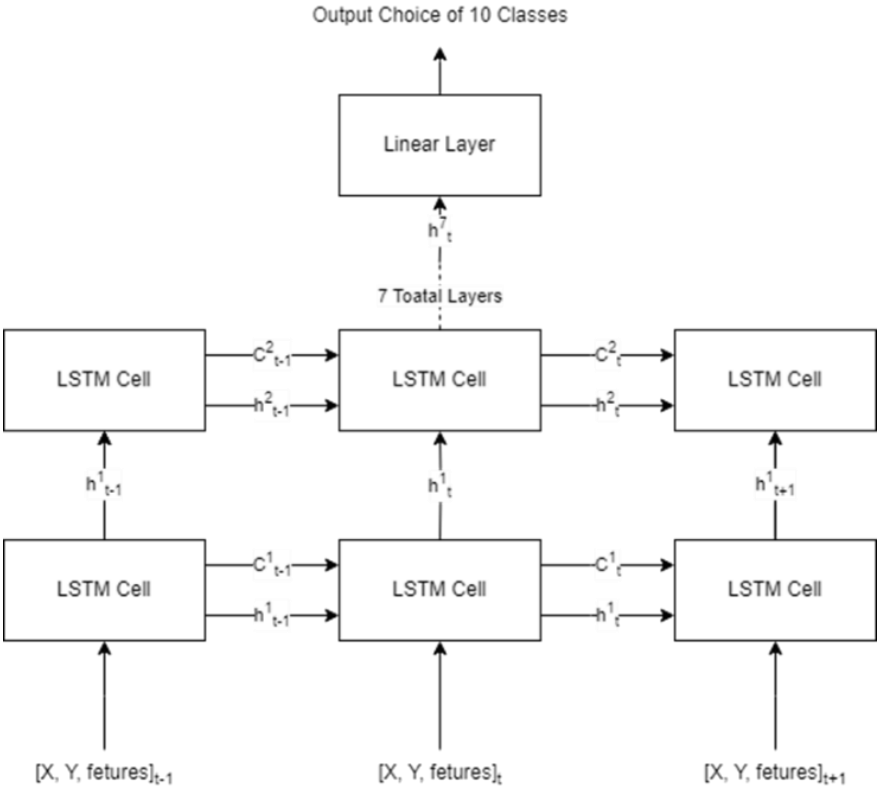


Figure 10: Multi-layered LSTM

Other alternatives investigated are RNN alone, GRU from PyTorch.nn. We tried to add a decoder encoder convolution network.

4.2.4 Training

Each model is trained only on database containing one type of functions, as defined in chapter *3 Data Base*. We use about 600 functions, each with 210 frames.
At the start of the training loop, we create a training matrix, containing the X/Y values of the optimal sampling points for each function. We also calculate the derivatives/FFT, if extra features are needed. In the first 20 samples, we sample uniform, i.e., setting all sample slots to zero.

We split the 600 functions to batch_size=20. And we update the model every 7 timestamps (sequence_size).
To get the input of the model, we process the training matrix which to a slice of:

[batch_size=20, 20 last samples * feature amount (2 to 5),
sequence_size=7]

In the training, we stop every training_interval=50 epochs, to evaluate the model on testing functions, which are part of the 600 functions, but ones that the model did not see during training. That way, we can know if the model is over-fitting the data or is learning.
The expected loss at epoch zero, with up sampling factor of 10, from the cross-entropy loss should be:

$$\ln(upsampling\ factor) = \ln(10) \approx 2.302 \quad (10)$$

4.2.5 Inference / evaluating the model

Inference is putting the trained ML model into production. In this mode, the ML inputs are only based on the (casual) samples chosen by it, and not the “correct” as in the training. The functions are generated using the same methods as the database, but ones that were never seen during training. And the error measure is MSE (MAE, or whatever defined) of the interpolated output vs the high-resolution input function, and not “correct” slots.

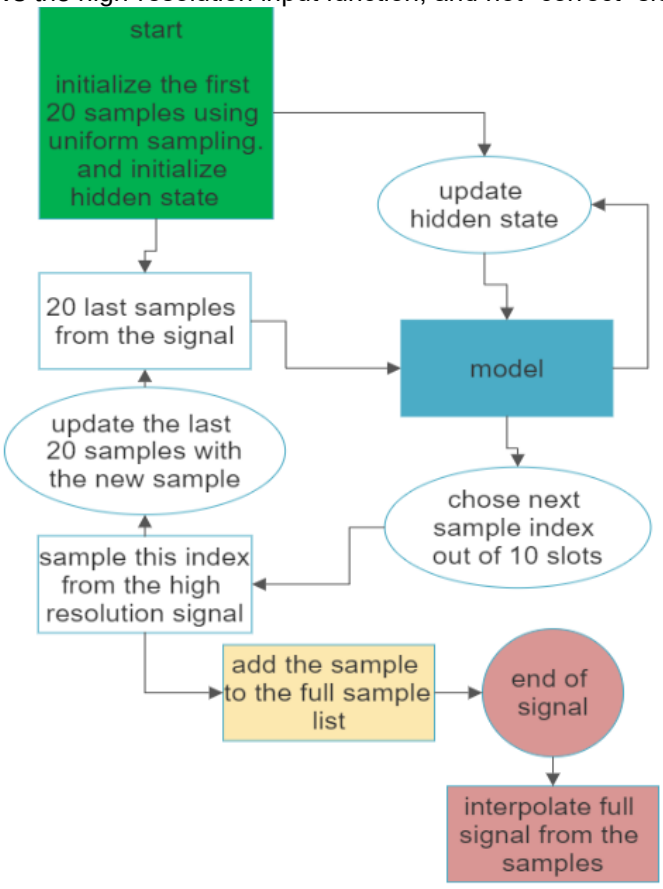


Figure 11: Inference the model

5 Analysis of results

5.1 Data Base

Here we will give an analysis of what we found while creating our databases.
For each database, that contains hundreds of functions, we will find what is the average improvement of the calculated best slot MSE over the uniformly sampled signal, and the maximum improvement in percentages: (The complete table at *Appendix E*)

Signal Type	Average Improvement	Maximum Improvement
Oscillating chirp signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{145}\%$	$\mathbf{3215}\%$
Sine signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to$\mathbf{10}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{8}\%$	$\mathbf{249}\%$
Sine signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{242}\%$	$\mathbf{1155}\%$
Sawtooth signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{238}\%$	$\mathbf{1270}\%$
Sum of Sine signals, from one Sines up to five sins, with linear interpolation	$\mathbf{299}\%$	$\mathbf{548}\%$
Pixel intensity from a real video, with linear interpolation	$\mathbf{272}\%$	$\mathbf{866}\%$

Table 1: The improvment of the MSE over uniformaly sampled

The formula we used to calculate the improvement:

$$Improvment = \frac{unif\ slots\ mse - best\ slots\ mse}{best\ slots\ mse} * 100\% \qquad (11)$$

Based on this table, we can anticipate the significant improvement that can be achieved using the ML model, in most signal types there. The results demonstrate that the potential gains are highly satisfactory and promising.
But, if we take the Sine signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with cubic spline interpolation we can see that the improvement is insignificant, so we cannot except our model to perform much better than uniform sampling.

Examples of the best slots over the uniformly sampling in Oscillating chirp.
As we can see our model (blue) is right on top of the signal also in cases where we are under Nyquist sampling rate but the uniform sampling (orange) is missing some edges.
And the MSE of the uniform is 0.1142 and for the best slots is 0.1076 .

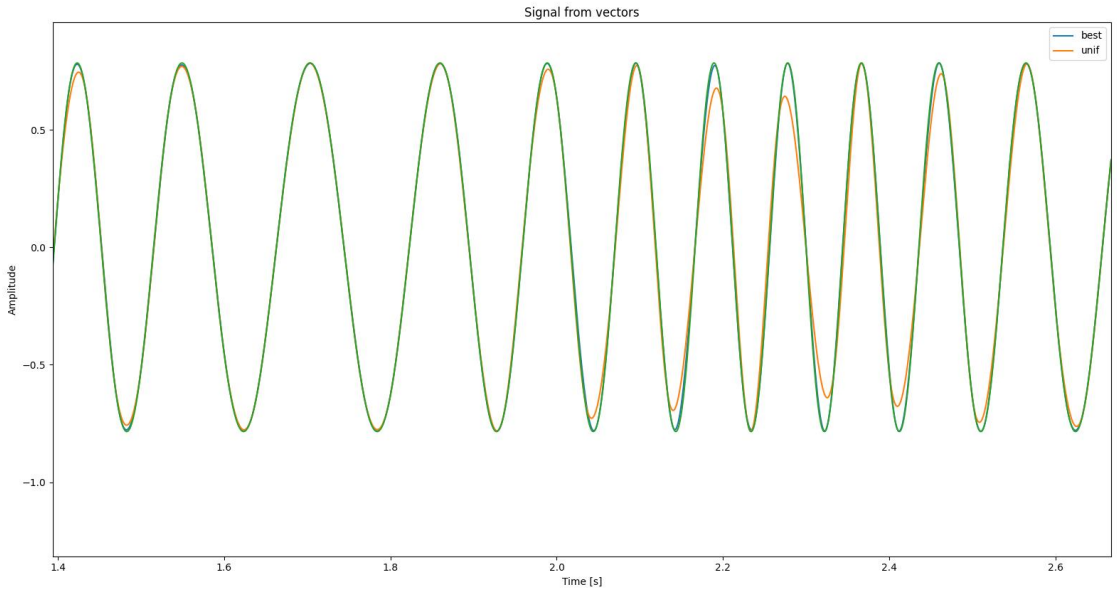


Figure 12: Example of best slots on an oscillating chirp with cubic spline interpolation

And some more examples in: *Appendix A* .

5.2 Supervised

Overall, after training on all our databases, with variety of architectures, we obtained some good results, in term of improving the MSE, in some of the signal types. But there were few signal types that our model was not able to converge and didn't gave good results.

After testing a lot of parameters of the model, we reached a conclusion that adding the FFT and the derivative as features are unnecessary and sometimes even harmful to performance of the model, so at the end we decided to train without them.

Some examples of training loss and testing loss of two of our models:

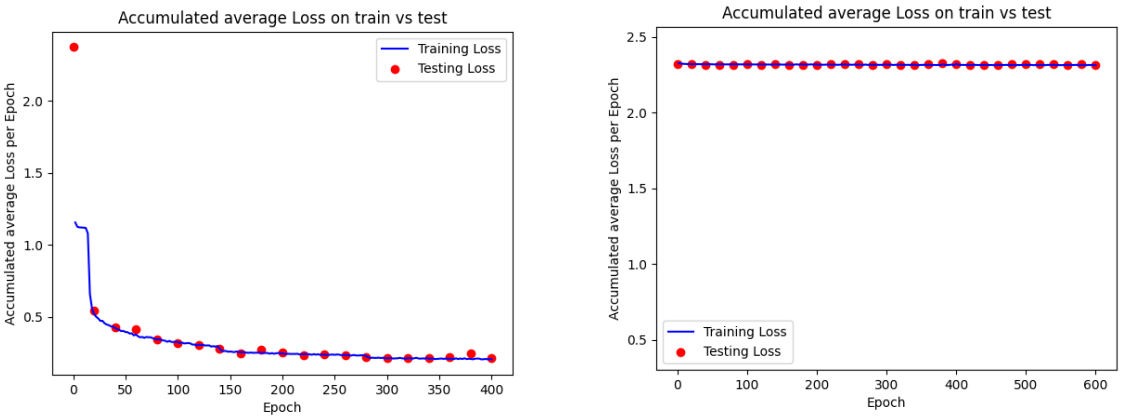


Figure 13: Two accumulated losses over epochs.
On the left train on sawtooth, on the right train on random function with $\sigma = 35$

we can see that for the sawtooth the model converged and reached low loss, on the other hand the training on the random signal didn't reduce the loss a lot from the starting point, this is expected as there is no connection from the data of the random signal to the "correct" sample points, as the data is too random.

Because every signal can be reconstructed using a sum of sin waves, we wanted to train models with sum of number of sine waves:

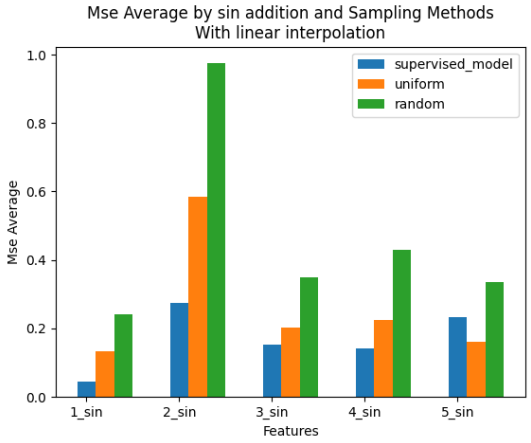


Figure 14: Performance of sampling methods over sum of number of sine waves

As we can see the supervised model preforms better then uniform in most cases but does get worse the more sine waves we add. This is something we expected, as before we run this test, we had models that were trained on more complex signals, and they didn't perform well.

Random with gaussian filter with different sigma values: (some examples of those signals in *Appendix D*)

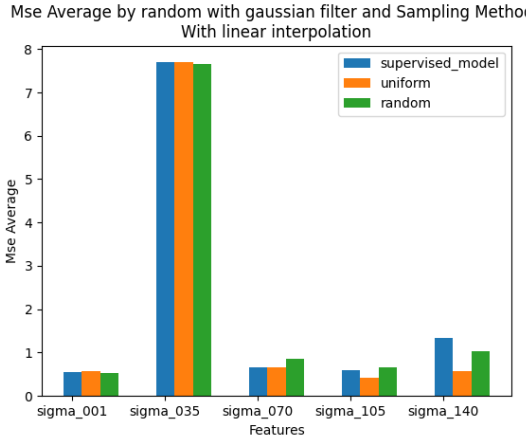


Figure 15: Performance of sampling methods over random signal with gaussian filter with different sigma values

As we can see the model didn't perform so well on this task, this is expected, as it's not possible to know what the best sampling indexes from the data itself is as it is random (there are no connections between the best sampling indexes). There are even cases where random sampling performed better than the other methods as can be seen above.

Another kind of testing that we perform is a sweep on frequencies of created signals. For each frequency we sampled, interpolated and checked the error in relation to the high-resolution "real" signal.
Here is the sweep for sawtooth signal with linear interpolation:

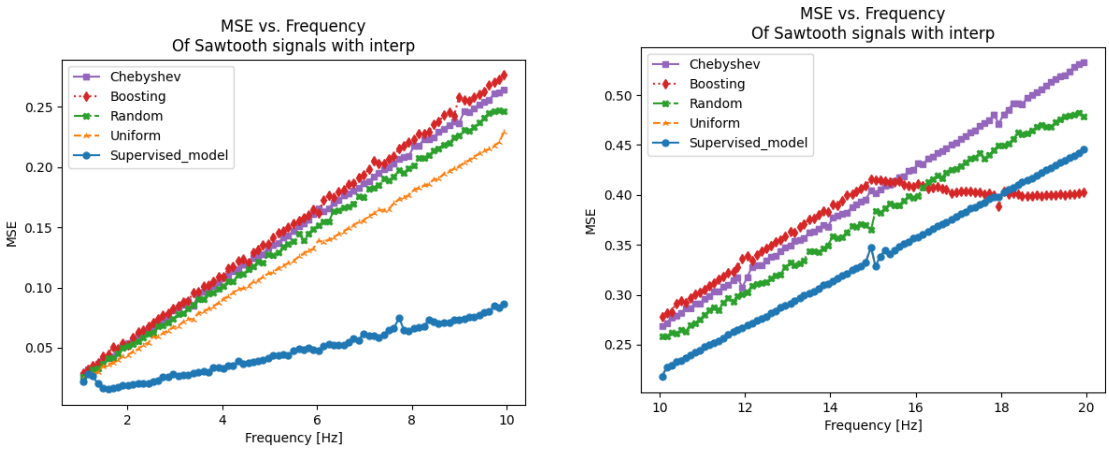


Figure 16: MSE vs. Frequency of sawtooth waves using linear interpolation

We can see that in the range of $1Hz - 10\ Hz$ our model performs better than all the other sampling methods. But in the $10Hz - 20\ Hz$ our model chooses to sample uniformly, and up to $18Hz$ did better than the other, until boosting overtook him (we'll explain this later).
Note: in this project we trained a model for each range of frequencies described above. So, there are two different models creating those sweeps.

Here is the sweep for sine wave signal with linear interpolation:

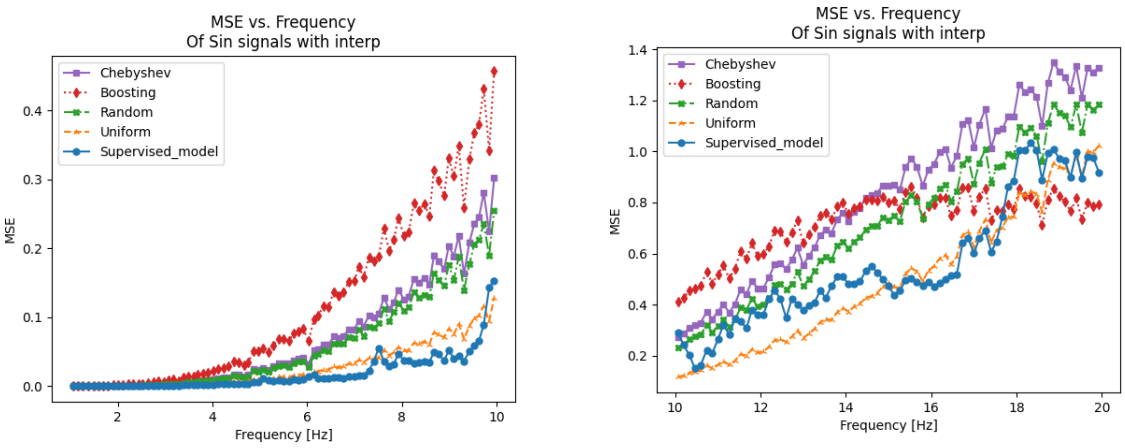


Figure 17: MSE vs. Frequency of sine waves using linear interpolation

We can see that our model, in the low range of frequencies, performs better at most of the points than the rest of the sampling methods.

But in the higher frequencies the model starts to struggle and doesn't perform as well in most frequencies.
also, in this case the boosting start to takeover the other sampling methods in higher frequencies, and there are two models for each frequency range.

In addition to MSE we also checked for MAE and MAX errors,
The sweep plot of the MAE looked similar to the MSE plots (as we can see from *figure 25* in *Appendix B*).
But for the Max Error, most sampling methods at some point happen to sample at a bad point causing it to get close to the max error possible on sawtooth with that frequency.

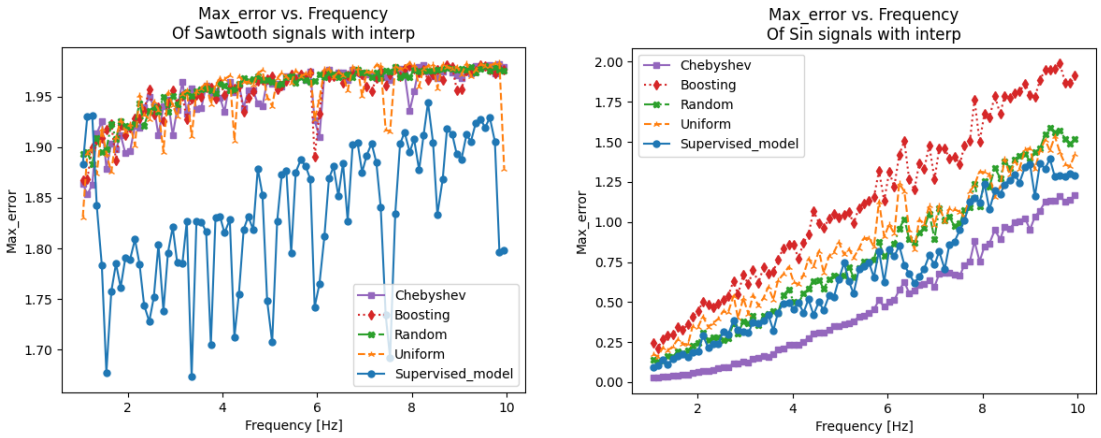


Figure 18: Max Error vs. Frequency.
On the left for sawtooth waves, on the right for sine waves

Another point we can look at from the Max Error is that the Chebyshev sampling method preformed the best as we expected (from *2.1.3 Chebyshev*) as this is what it was designed to do.

For cubic spline interpolation the results weren't so good compared to the linear interpolation, see sweep examples at *Appendix C* .

About why we think the boosting take over in high frequency?

As we can see in *figure 18* and *17* at high frequency the boosting starts to perform better than other sampling methods. At those high frequencies the samples can't follow the signal. Interpolating to a constant 0 may even get better results than trying to use the samples, as the frequency is high such that between each two samples on average there is more than half a cycle of the signal such that if we sample a high point and then another high point then we will interpolate a line between high points but at that time the signal went down and back up and it was mostly low as he has more than half a cycle. Boosting get less hurt from that as busting specifically has intervals of higher frequency such that this doesn't happen. And between those intervals it doesn't preform worse than other sampling methods. We can see it clearly at *figure 17* as you can see this happen exactly at 15Hz that the boosting starts to overtake.

More sweep plots in: *Appendix B* .

Here are some examples for cases that we can see how our models preforms better than the uniform sampling (and better than any other deterministic sampling regime we've discussed): Here, for the sine wave at $10.5Hz$ the MSE of the uniform sampling is 0.018 and for our model is 0.034

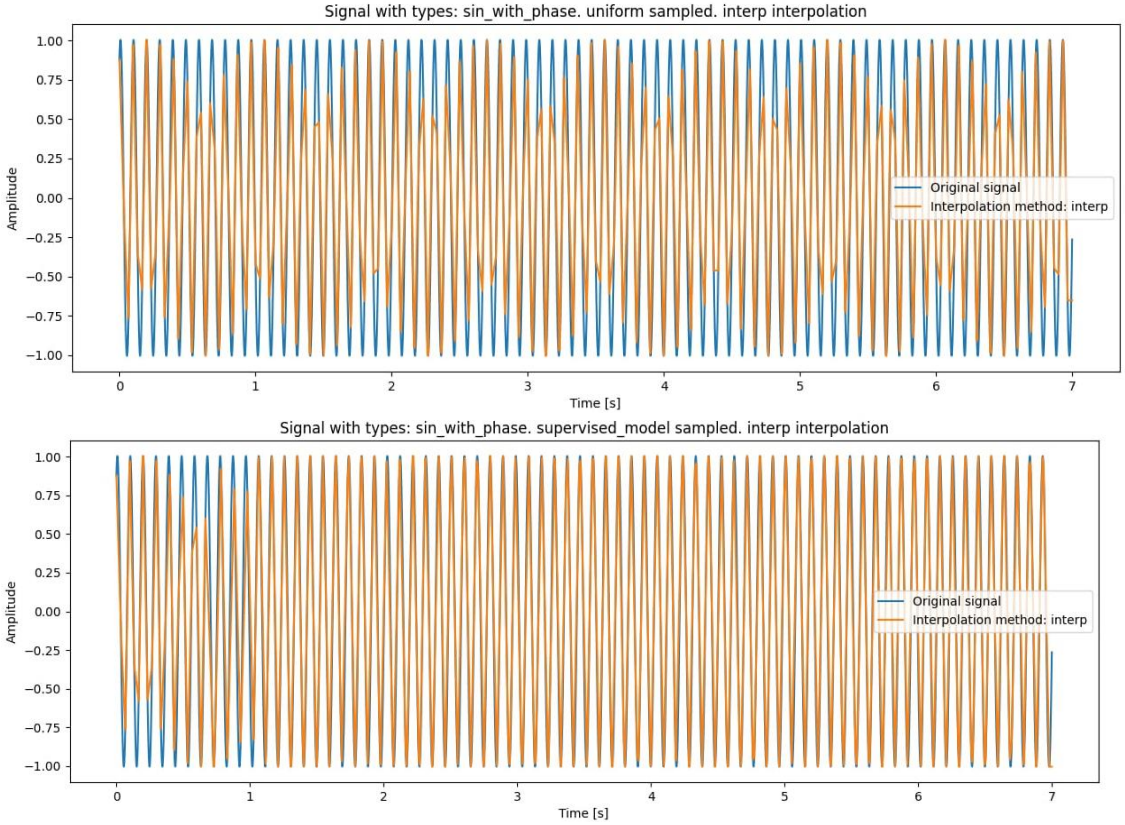


Figure 19: Example of linear interpolation of sin wave using uniform sampling (upper) and using a trained model to sample (lower)

We can see how the model manages to pick sampling points on the high points of the sin wave and by that making the linear interpolation MSE a lot better.

And here we have sum of four sine waves with different frequencies, amplitudes, and phases. The MSE of the uniform sampling is 0.692 and for our model is 0.139

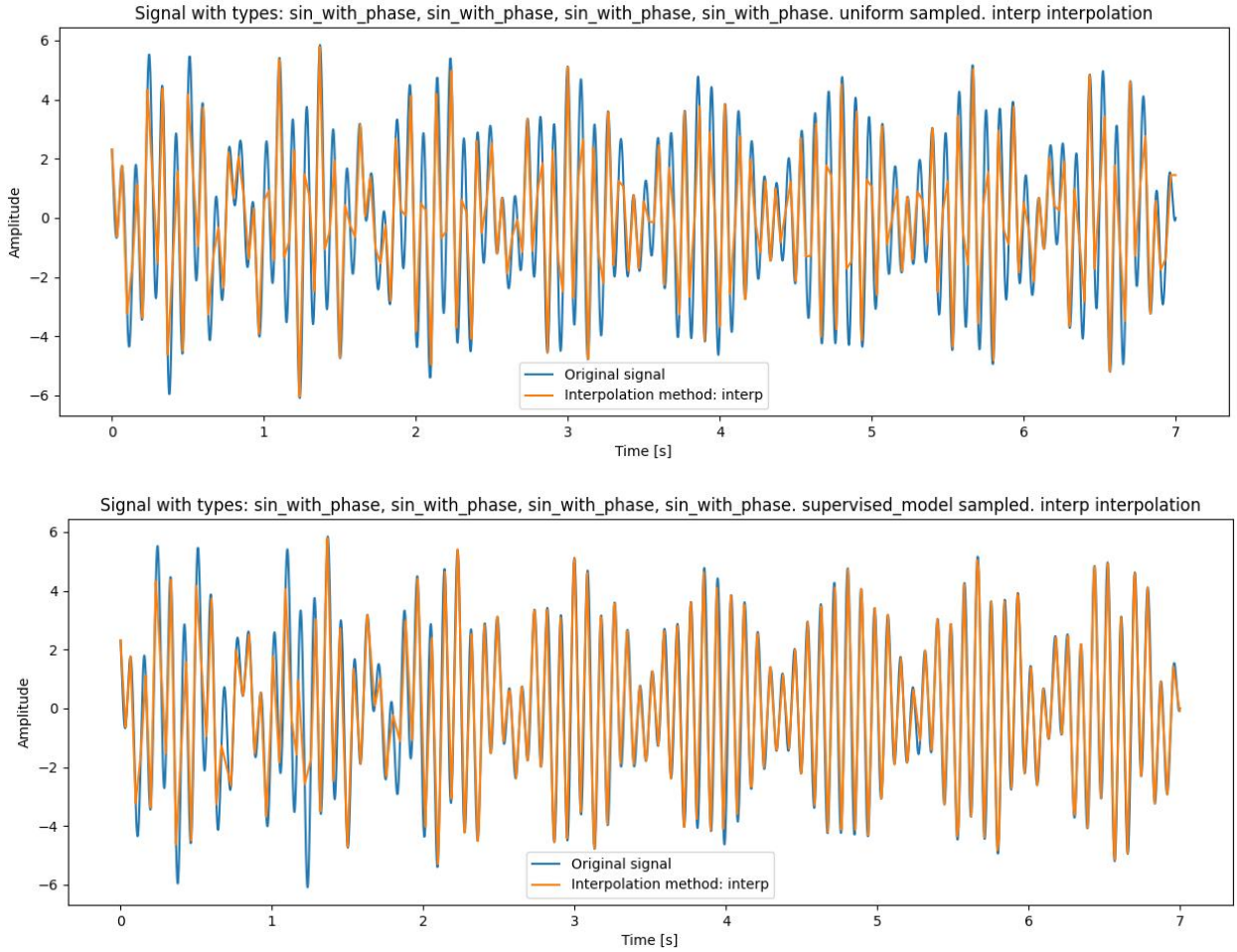


Figure 20: Example of linear interpolation of addition of four sine waves using uniform samples (upper) and our model (lower)

And some more examples in: *Appendix C* .

We wanted to explored signals that are more "real" in our project. We chose to examine a video and look at the intensity levels of a randomly selected pixel over time, using this as our signals database. Through training our model on this type of data, we achieved improvements in the results over uniform sampling. However, it is worth noting that currently we didn't try it on a lot of different videos. On some of the videos we tried it on, it didn't show improvement. The training and testing were done on a signal video at a time in order to perform well.

A statistical analysis of the improvements we observed can be seen in *table 2* .
Some examples of the model sampling the pixel signal are in: *Appendix C* .

Summary of all the models performances on all our data bases:

Signal Type	MSE Improvement ratio over uniform	Correct precent
Oscillating chirp signals in frequency between $\mathbf{1Hz}$ to $\mathbf{10Hz}$ with cubic spline interpolation	$\mathbf{0}\%$	$\mathbf{21.5}\%$
Oscillating chirp signals in frequency between $\mathbf{10Hz}$ to $\mathbf{20Hz}$ with cubic spline interpolation	$\mathbf{-18}\%$	$\mathbf{21.2}\%$
Sine signals in frequency between $\mathbf{1Hz}$ to$\mathbf{10Hz}$ with cubic spline interpolation	$\mathbf{0}\%$	$\mathbf{21.3}\%$
Sine signals in frequency between $\mathbf{10Hz}$ to $\mathbf{20Hz}$ with cubic spline interpolation	$\mathbf{-3}\%$	$\mathbf{16.2}\%$
Sine signals in frequency between $\mathbf{1Hz}$ to $\mathbf{10Hz}$ with linear interpolation	$\mathbf{74}\%$	$\mathbf{56.8}\%$
Sine signals in frequency between $\mathbf{10Hz}$ to $\mathbf{20Hz}$ with linear interpolation	$\mathbf{0}\%$	$\mathbf{17.2}\%$
Sawtooth signals in frequency between $\mathbf{1Hz}$ to $\mathbf{10Hz}$ with linear interpolation	$\mathbf{172}\%$	$\mathbf{86.0}\%$
Sawtooth signals in frequency between $\mathbf{10Hz}$ to $\mathbf{20Hz}$ with linear interpolation	$\mathbf{0}\%$	$\mathbf{32.1}\%$
Oscillating chirp signals in frequency between $\mathbf{1Hz}$ to $\mathbf{10Hz}$ with linear interpolation	$\mathbf{-2}\%$	$\mathbf{35.8}\%$
Oscillating chirp signals in frequency between $\mathbf{10Hz}$ to $\mathbf{20Hz}$ with linear interpolation	$\mathbf{-1}\%$	$\mathbf{21.7}\%$
Sum of Sine signals, from one Sines up to five sins, with linear interpolation	$\mathbf{80\%/18\%/-6\%/-23\%/-35\%}$	$\mathbf{50\%/31.6\%/25.9\%/24.6\%/23.2\%}$
Gaussian filtered random signal with different gaussians, with linear interpolation	$\mathbf{0\%/0\%/0\%/-26\%/-50\%}$	$\mathbf{10.3\%/19.4\%/19\%/15.2\%/15.7\%}$
Pixel intensity from a real video, with linear interpolation	$\mathbf{3.5}\%$	$\mathbf{34}\%$

Table 2: Models performances on different databases

The formula we used to calculate the improvement:

$$Improvment = \frac{1}{N}\sum_{i=0}^{N-1}\frac{(unif\ mse)_i - (model\ mse)_i}{(model\ mse)_i} * 100\% \qquad (12)$$

Where N is the number of signals in the test dataset.
Correct percent is the precent of times the model chose the slot that we computed as optimal.

We also tested what is the improvement when the model chose the “correct” choice and when it didn’t. We wanted to see if we could assume that close slots are correlated as described in *4.2.2 Outputs*. What we saw is that close answers where better, In sense of MSE, than far away on average.

6 Conclusions and further work

At the start of the project, we wanted to use non-supervised ML architecture, so we expected the model to learn to be what is the best sampling method. The goal we set for the model was to perform at least as good as the best other sampling methods, on a training data set and get at least 5% MSE improvement in a given interpolation method. For most cases we succeeded in getting a better MSE then all other sampling methods (as seen in *Table 2*).
The assumptions we made worked and we think there is future of the methods we used.

On the other hand, the model didn't learn to be the best method for all our databases as can be seen on *figure 17*, in which, for example, the boosting method perform better but the supervised model couldn't learn it as it wasn't what was shown in the "correct" answers.

We have had many ideas over the project for improvements or changes. We didn't do them because of time constraints, some because it was too late to change at that time and some are just taking the project in a different direction.

One big point is the fact that we didn't use an interpolation network or any interpolation method that is tailored for interpolate from non-uniform samples.
Maybe with an interpolation network that know how to work with the sampling model, better results can be reached.

Another idea was to expand the model architecture as seen in other papers that made use of a network that had different parts with specific task in mind to make the network more focused and by that helping it learn more complex signals.

Another approach could be to have the training process of the model include some of its own outputs, such that it will know how to work with the values it gives itself. (Right now, the training uses "teacher forcing" where at each time step the correct answer is inputted instead of the answer the model chose before)

Furthermore we could check what we can get if the system is not causal. Meaning that we can get samples in any order we want. This case can be imagined as a big data base that it is expensive to ask for data and we want to interpolate a function from it using the least amount of data accesses.

A further suggestion for a different direction of the project was also working on a problem that the samples that we can take from the signals are not confined to a frame, meaning for example: we can sample 100 times at any point the model choses.

It is worth considering a different direction of the project is we thought about at the start was a no limit mode where you are not confined in a frame and can chose where the next sample will be at any time from now. So, in this case we are bounded only by the number of samples allowed.

Furthermore, we could explore the idea of the project to be more about audio signals like music or recordings. When you hear sound, the ear doesn't care about the phase of the signal, it only cares about the frequency. So, we had the idea to work with an error function that does STFT over the high-resolution signal and the interpolated signal and gives loss only on wrong frequency.

7 Project Documentation

Our project is done by software only, so in this chapter we'll document the project code, and give a brief explanation of how to run the main parts of our project.
The GitHub link : github.com/tomerraj/NUSTSR

Firstly, we've created a signal class, that let us create, sample, interpolate and show the signal.
We can create a signals of a variety of types and sum them up (or to apply other function on them). Examples of types are:
oscillating_chirp_with_phase, *sin_with_phase*, *chirp_with_amplitude*, *square_changing_height*, *triangular_wave*, *sawtooth_with_phase*, *chirp*, *pisewise_linear_periodic*, *pixel*, *rand_with_gaussian_sigma_20* and more.
We can sample the signal with those sampling polices:
Uniform, *Random*, *Boosting* and *Chebyshev*.
And after that we can interpolate those sample with those interpolation methods: *LaGrange*, *Cubic Spline*, *zero order*, *second order* and *first order*.
After that we can estimate the error with error types: *MSE*, *MAE* and *MAX*.

Then, we've created the DQN model, train and sample functions.

After we've created the Database creator and other file that help shows and calculate the created data.
And the supervised ML model functions, that let you create various of different RNN based architectures like GRU or LSTM and we can easily add and remove feature from the architecture like the size of the input (based on the features we give the network), dropout layer and the initialize of the hidden state. After the creation of the architecture, we have function that train the network on a database with various of architecture and configuration and compare their results. And function that sample with the model.

Lastly, we've created statistics functions that create functions and perform the different sampling polices on them and displays the results in plots and tables.
To run our project, like the block diagram explain,
first you need to open: Data_Base/Data Saver.py and set these variables with the parameters of the database we want to create:

```
interpolation_method = 'CubicSpline'  # interp , CubicSpline
database_name = '_both_freq_sin_with_phase_1-10'
simulation_time = 7
factor = 2
fps = 30
number_of_slots = 10
freqs = (1, 10)
all_type_list = ['sin_with_phase']
num_of_funcs_to_save = 400
```

and run this code to create the database.

After that you need to open: supervised/model_run.py and set these variables with the parameters related the model we want to create,
Here we can select which features to send to the model (FFT, derivative, both or none), we can choose the range of signal from the database to train and test on, we can choose the architecture and its shape and we can choose the loos function and the learning rate function (linear combination of them):

```
parameters = {
    'PATH': 'models',
    'MODEL_NAME': 'cubic_osci_chirp',
    'data_path':
'Data_Base_CubicSpline_both_freq_oscillating_chirp_with_phase_1-
10.npy',
```

```
    'start_from_func': 0,
    'end_in_func': -1,  # -1 for all function in the data_path

    'fps': 30,
    'number_of_slots': 10,
    'simulation_time': 7,

    'state_sample_length': 20,
    'output_size': 10,  # same as number of slots
    'batch_size': 20,
    'seq_length': 7,
    'feature_dict': {'derivative': 1, 'FFT': 0},

    'hidden_dim': 70,
    'n_layers': 6,
    'architecture': {'RNN': 0, 'GRU': 0, 'LSTM': 1, 'TGLSTM': 0},
    # only one should be on, if not the first one would be chosen
    'extra_to_model': {'dropout': 0.3, 'encoder_channels': 0,
'decoder_channels': 0, 'hidden': 'rand_gauss'},
    # the dropout value is the drop_prob
    #  rand_gauss ,   rand_unif ,   zeros
    'n_epochs': 0,
    'lr_start': 0.0005,
    'evaluation_interval': 20,
    'gamma': 280,
    'lr_func': {'step_decay': 0, 'cosine_annealing': 0,
'poly_decay': 0, 'exp_decay': 0, 'exp_log_decay': 0,
                'linear_decay': 1},  # the linear is more like a log
    'loss_funcs': {'CrossEntropyLoss': 1, 'L1_loss_on_weghits': 0.8,
'L2_loss_on_weghits': 0, 'L1Loss': 0.1, 'MSELoss': 0}
}
```

Also, choose the databases to train on and compare their model performances:

```
PATHs = ['Data_Base_interp_sin_with_phase_1-10.npy',
         'Data_Base_interp_sin_with_phase_10-20.npy']
```

After that run this fie.

At last, to compare the results of the model to other sampling methods.
Here we have two ways visualizing the data the first is a sweep based (e.g. the frequency of the signal Vs. the MSE) or feature based (e.g. the amount of added sines signal Vs. MAE).

```
func_type = 'sawtooth_with_phase'
sampling_methods = ["supervised_model", "uniform", "random",
"boosting", "chebyshev"]
interpolation_method = "interp"  # interp, CubicSpline

supervised_parameters = {
    'model_path_low_freq' : 'supervised/models/…/',
    'model_path_high_freq': 'supervised/models/…/',
    ...
}
num_repetitions = 2  # for averaging
signals_count = 12

error_type = "Max_error"  # Max_error ,  MSE ,  L1_error
```

and run.

8 References

Papers:

[1] H. Sundqvist and G. Veronis, “A simple finite-difference grid with non-constant intervals”, Tellus A: Dynamic Meteorology and Oceanography, vol. 22, no. 1, p. 26-31, 1970.DOI, https://doi.org/10.3402/tellusa.v22i1.10155

[2] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning, https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[3] Philip B. Weerakody, Kok Wai Wong, Guanjin Wang, Wendell Ela, A review of irregular time series data handling with gated recurrent neural networks, Neurocomputing, Volume 441, 2021, Pages 161-178, https://www.sciencedirect.com/science/article/abs/pii/S0925231221003003

Links:

[1] Time Gated LSTM for irregular time series, https://github.com/FedericOldani/TGLSTM

[2] A Beginner’s Guide on Recurrent Neural Networks with PyTorch, https://blog.floydhub.com/a-beginners-guide-on-recurrent-neural-networks-with-pytorch/

[3] Long Short-Term Memory: From Zero to Hero with PyTorch, https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/

[4] PyTorch Loss Functions: The Ultimate Guide, *https://neptune.ai/blog/pytorch-loss-functions*

[5] Double Dueling Deep Q-Learn with PyTorch, https://github.com/Arrabonae/openai_DDDQN

[6] REINFORCEMENT LEARNING (DQN) TUTORIAL, https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Figures:

[1] LSTM cell, https://thorirmar.com/post/insight_into_lstm/

[2] DQN data flow diagram, https://www.mdpi.com/2076-3417/12/7/3220

Appendix A.

Some examples of the calculated best slots:
here the MSE of the uniform sampling is 17.086 and for the best slots is 2.772

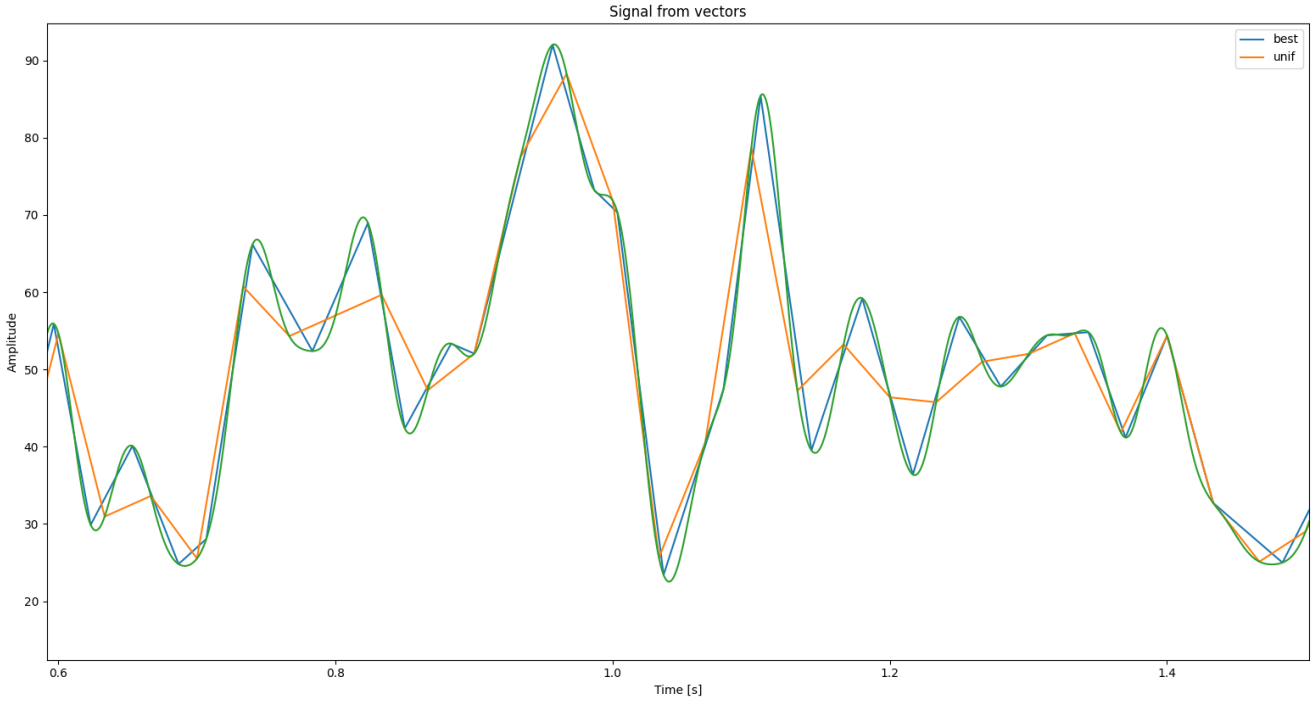


Figure 21: Example of best slots on an intesity of a pixel from a video with linear interpolation

And here the MSE of the uniform sampling is 0.143 and for the best slots is 0.041

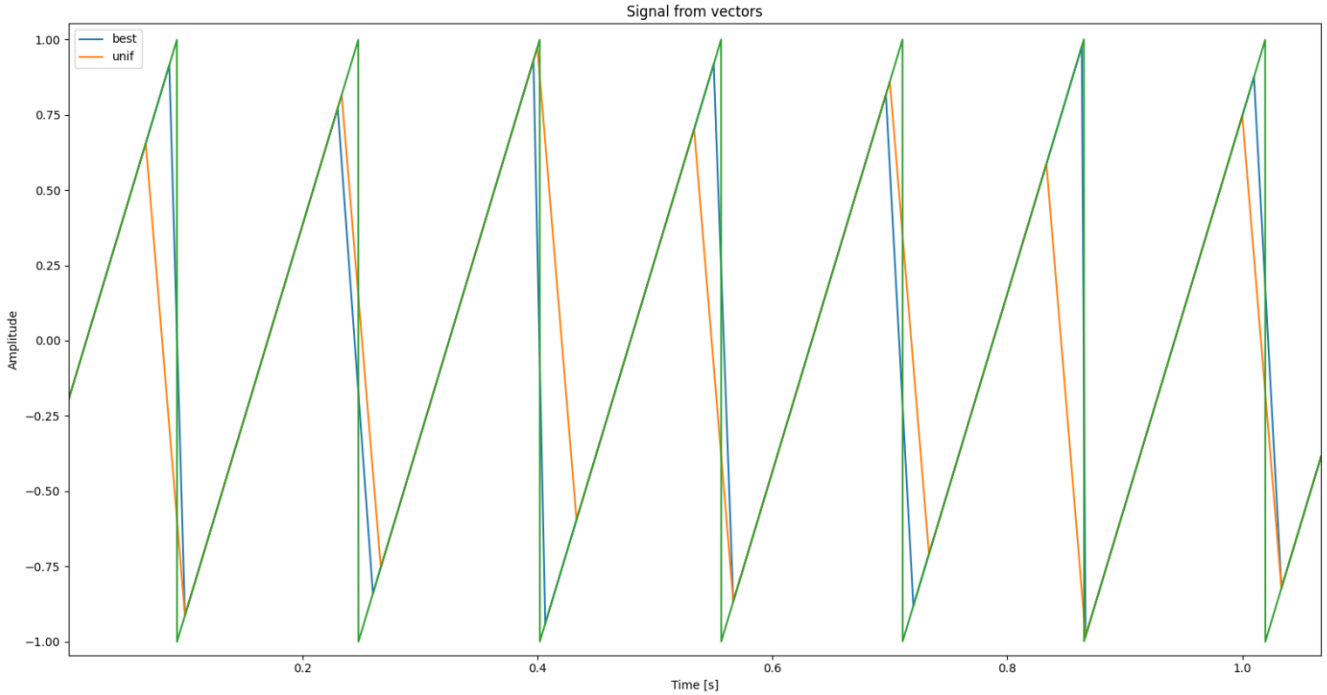


Figure 22: Example of best slots on a sawtooth signal with linear interpolation

Appendix B.

More sweep plots:
Here we have the sweep of the oscillating chirp, in which our model doesn't perform as well

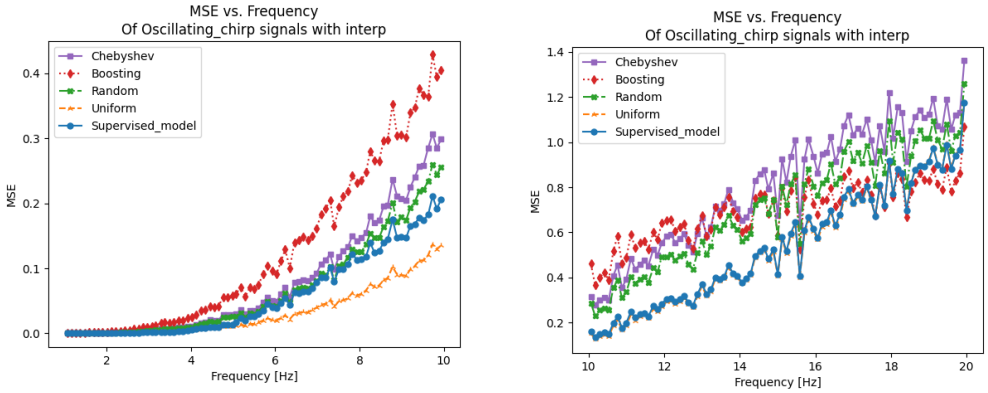


Figure 23: MSE vs. Frequency of oscillating chirp waves using linear interpolation

Here is a sweep plot of sawtooth signals but with MAE error:

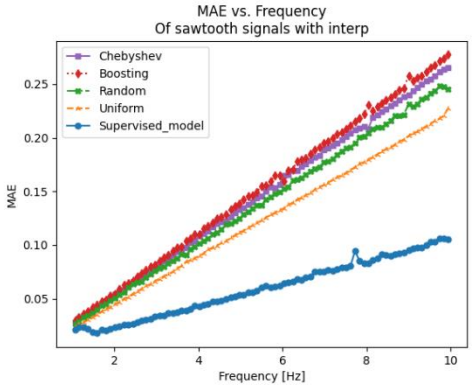


Figure 24: MAE vs. Frequency of sawtooth waves using linear interpolation

Here is a sweep plot of: Oscillating chirp interpolated with Cubic Spline, as we see on the left the model learned to be exactly the same as uniform (which is not optimal) and on the right we can see that the model didn't perform better than other methods consistently.

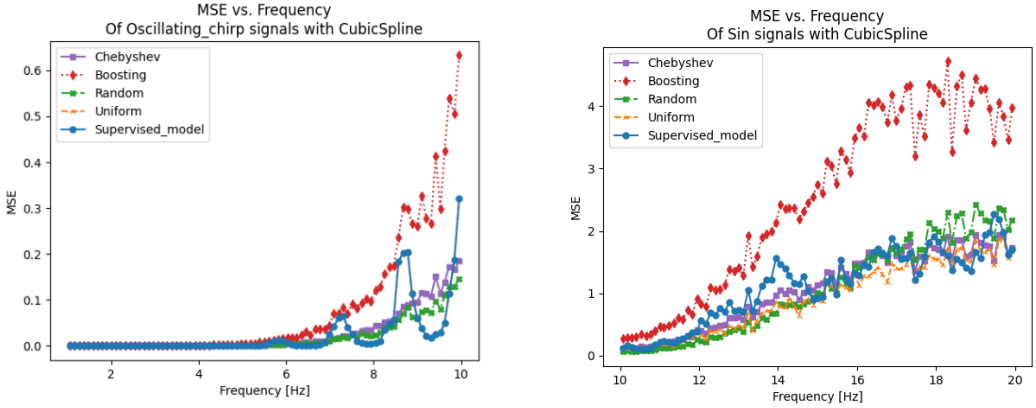


Figure 25: MSE vs. Frequency oscillating chirp waves using Cubic Spline interpolation

Appendix C.

Some more examples of sampling with our model over uniformly sampling:
We can see here that the model takes the correctly last slot in the last sample, here the MSE of the uniform sampling is 0.0005 and for our model is 0.00008

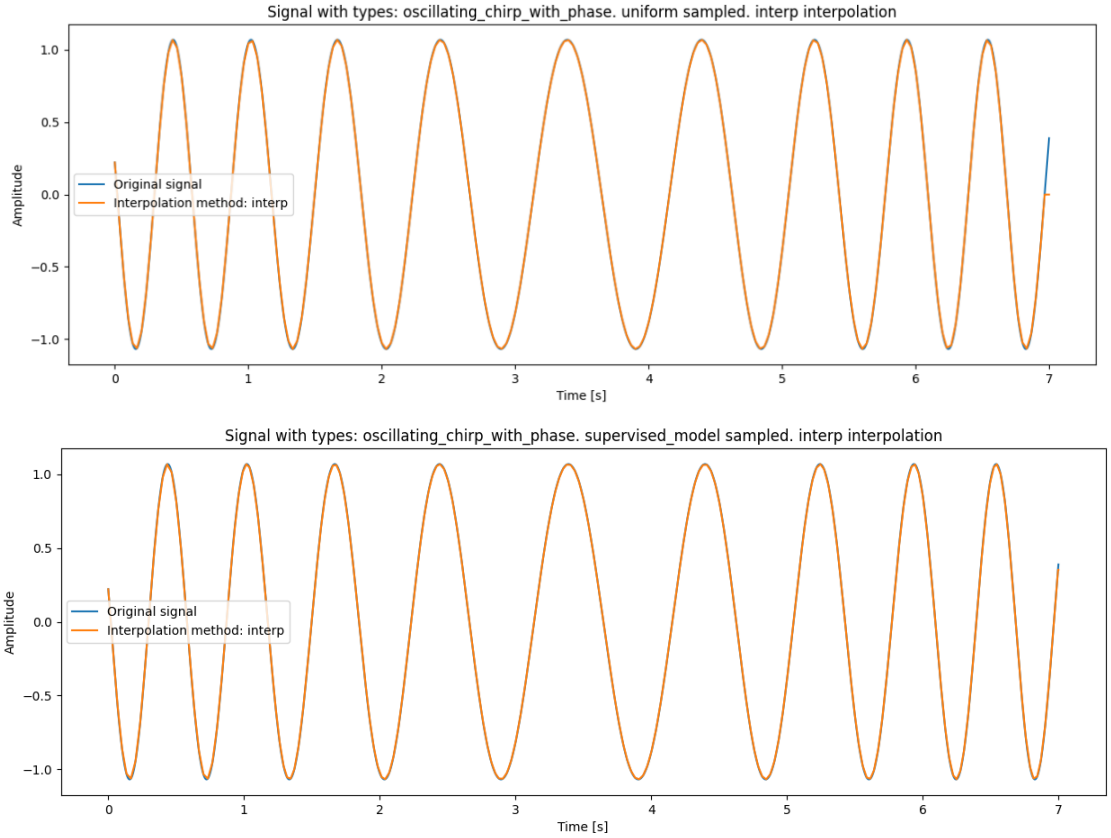
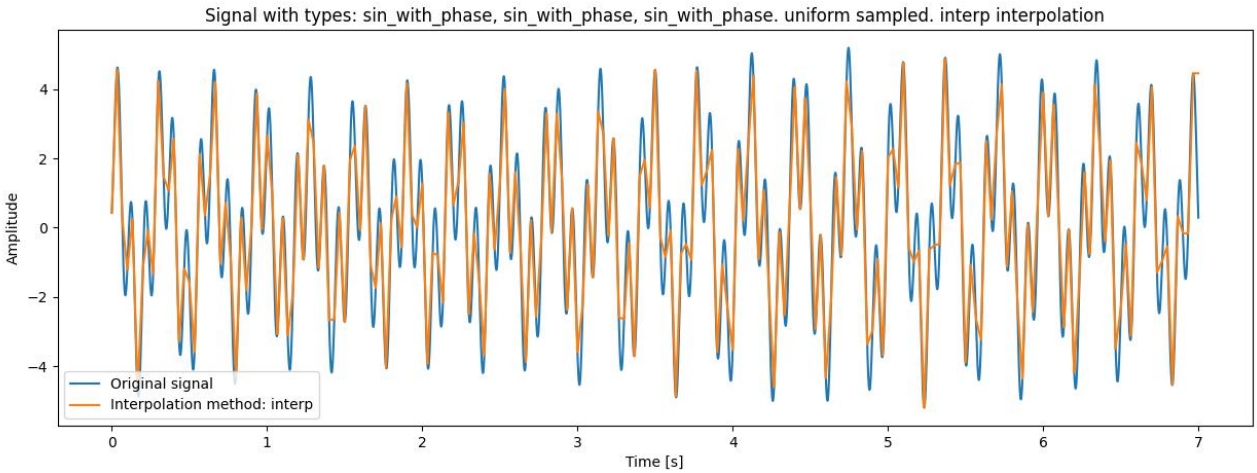


Figure 26: Example of linear interpolation of oscillating chirp using uniform samples (upper) and our model (lower)

Here our model sample a signal that is an addition of number sine waves with different frequencies, amplitudes, and phases, here the MSE of the uniform sampling is 0.565 and for our model is 0.328



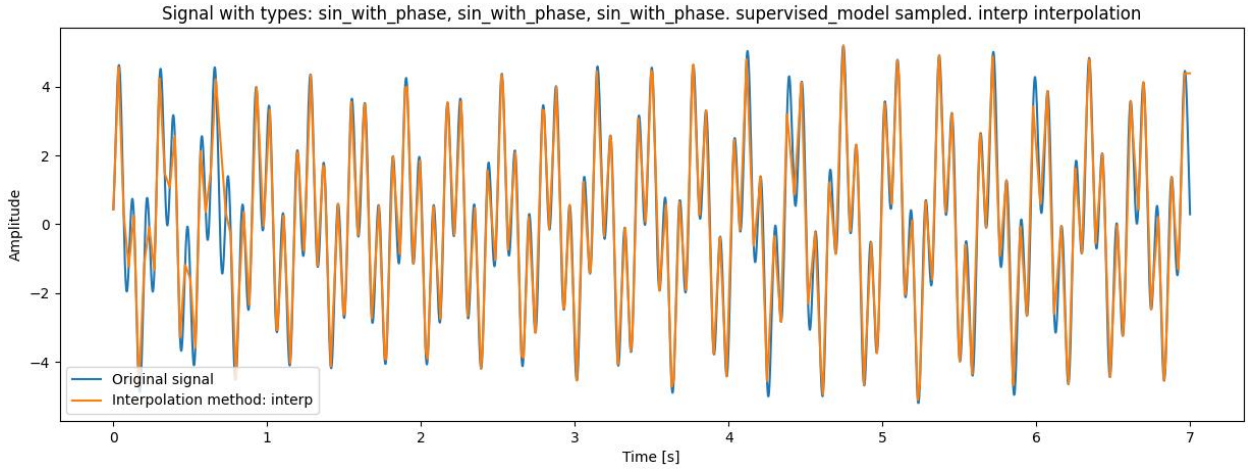


Figure 27: Example of linear interpolation of addition of three sine waves using uniform samples (upper) and our model (lower)

And here, for the sawtooth wave at $1.5Hz$ the MSE of the uniform sampling is 0.04 and for our model is 0.013

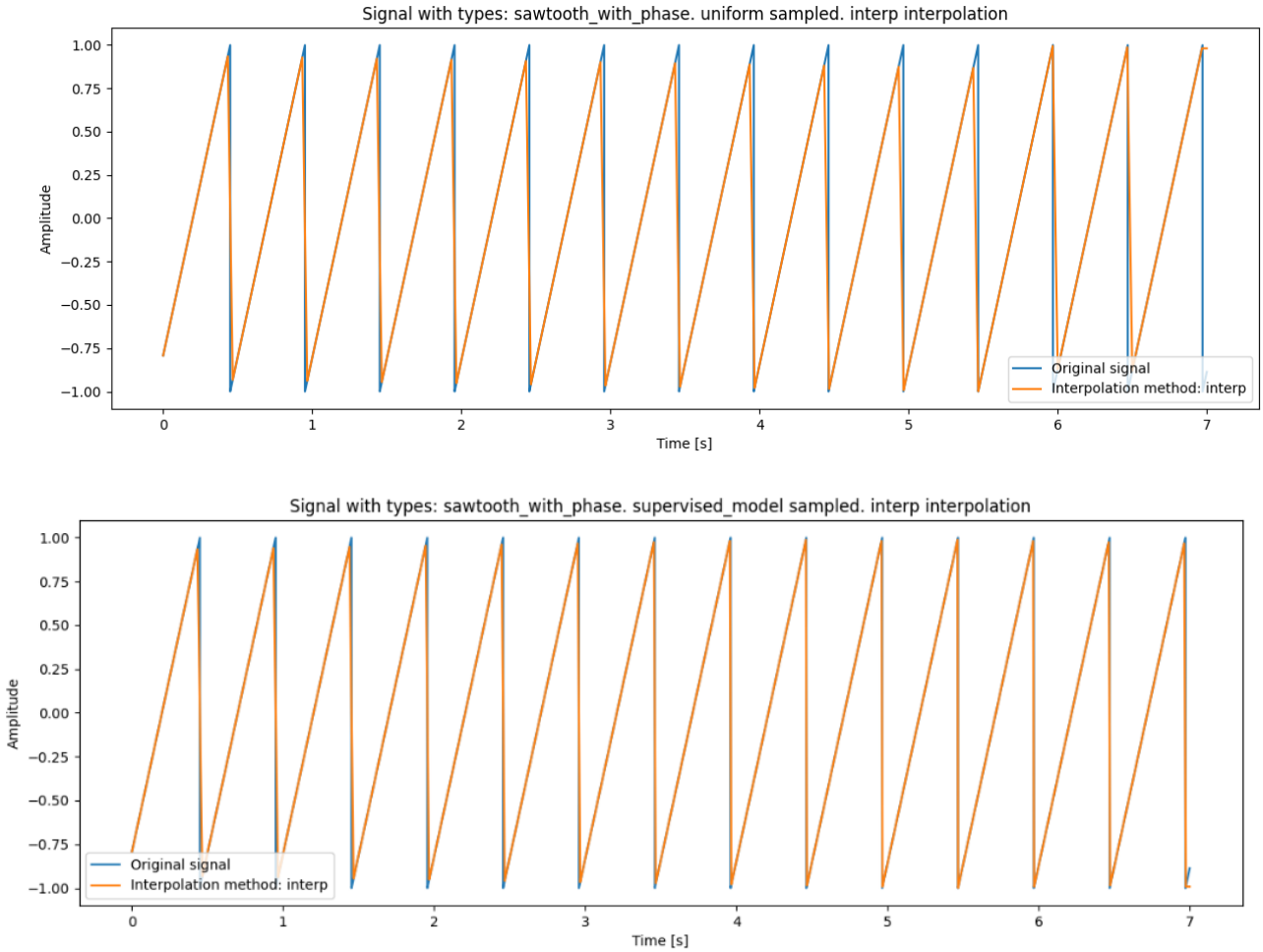


Figure 28: Example of linear interpolation of sawtooth using uniform samples (upper) and our model (lower)

And here, for an intensity of a pixel from a video. The MSE of the uniform sampling is 0.939 and for our model is 0.902

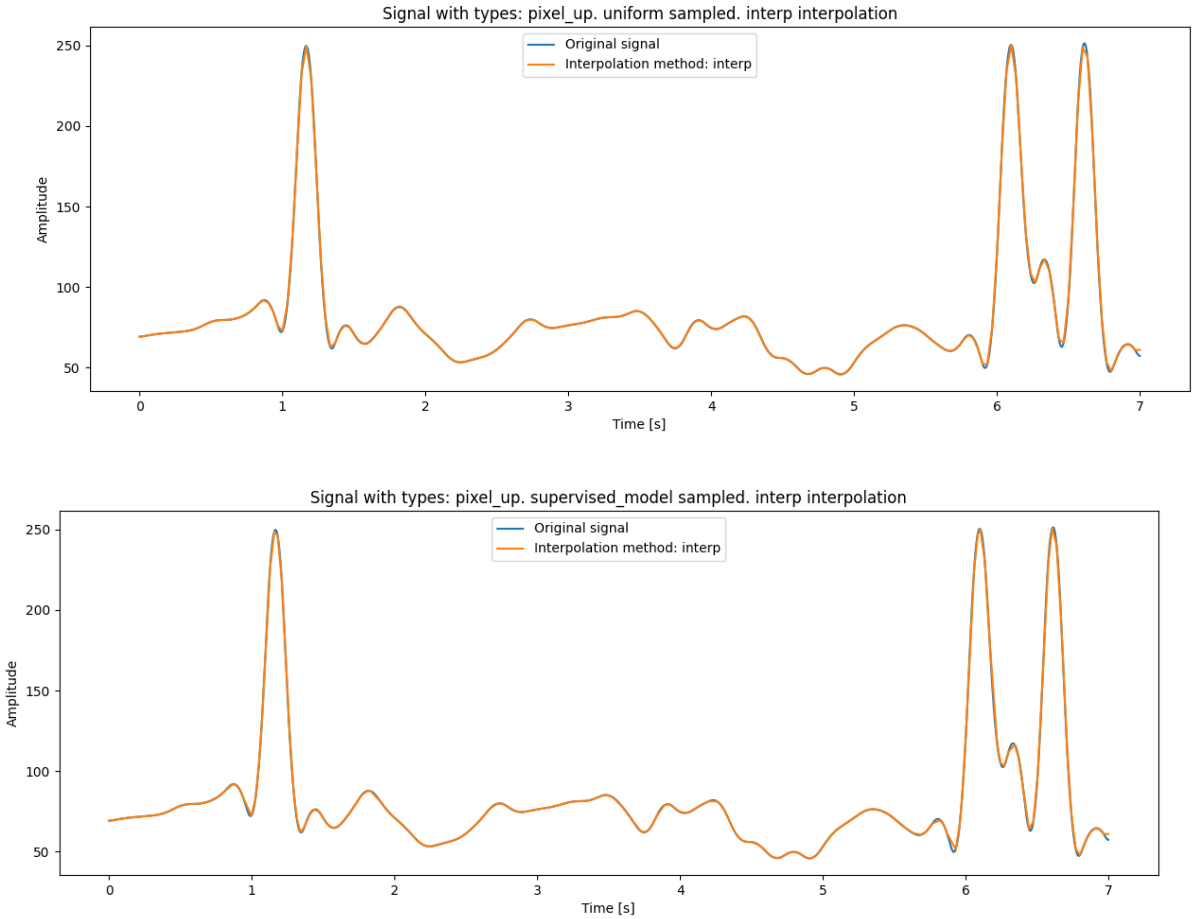


Figure 29: Example of linear interpolation of pixel intensity values using uniform samples (upper) and our model (lower)

Appendix D.

Some rando signals with gaussian filter with different sigmas:

A random signal with gaussian filter with sigma=1, Sampled using the model.

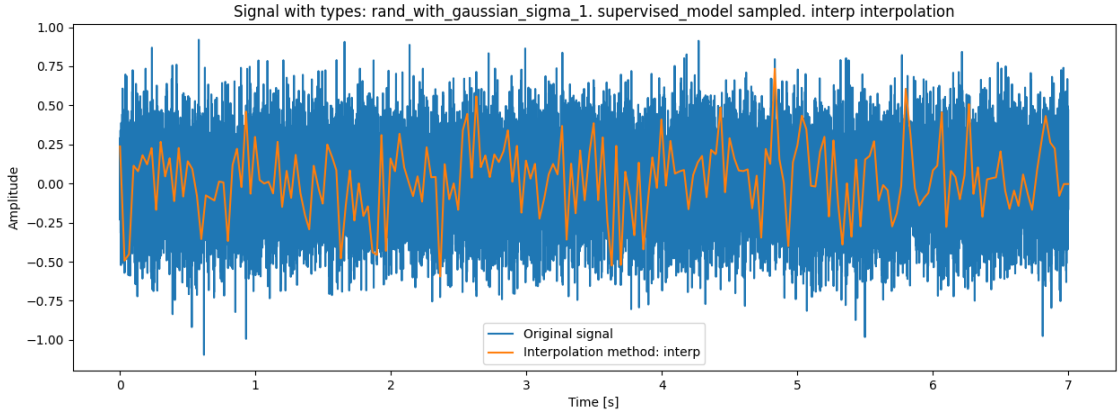


Figure 30: Example of random signal with gaussian filter with sigma=1 with the model samples

And random signal with gaussian filter with sigma=35, Sampled using the model.

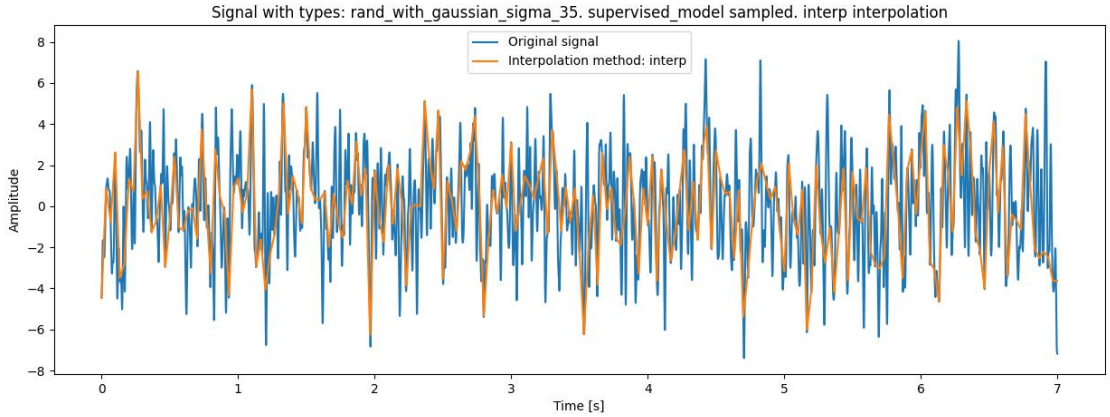


Figure 31: Example of random signal with gaussian filter with sigma=35 with the model samples

Another random signal with gaussian filter with sigma=105, Sampled using the model.

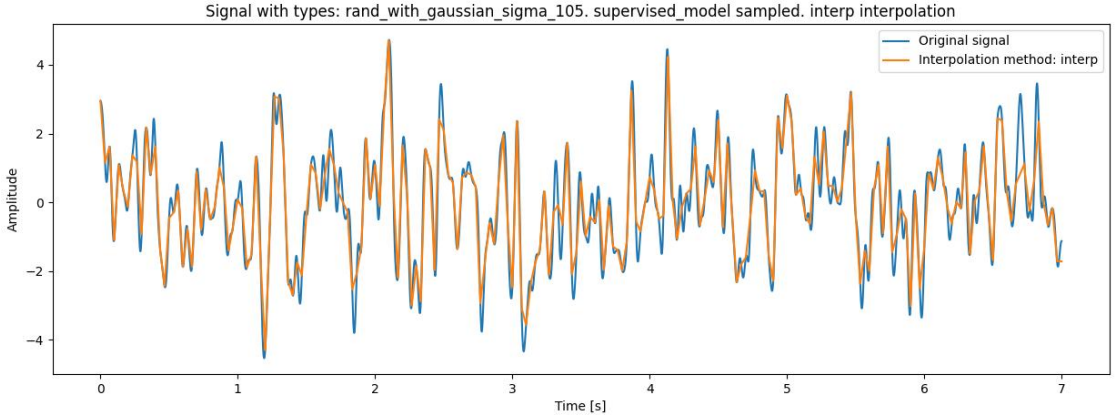


Figure 32: Example of random signal with gaussian filter with sigma=105 with the model samples

Appendix E.

Signal Type	Average Improvement	Maximum Improvement
Oscillating chirp signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{41}\%$	$\mathbf{1034}\%$
Oscillating chirp signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{145}\%$	$\mathbf{3215}\%$
Sine signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to$\mathbf{10}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{8}\%$	$\mathbf{249}\%$
Sine signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with cubic spline interpolation	$\mathbf{82}\%$	$\mathbf{1833}\%$
Sine signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{117}\%$	$\mathbf{822}\%$
Sine signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{242}\%$	$\mathbf{1155}\%$
Sawtooth signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{238}\%$	$\mathbf{1270}\%$
Sawtooth signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{168}\%$	$\mathbf{1794}\%$
Oscillating chirp signals in frequency between $\mathbf{1}\boldsymbol{Hz}$ to $\mathbf{10}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{140}\%$	$\mathbf{470}\%$
Oscillating chirp signals in frequency between $\mathbf{10}\boldsymbol{Hz}$ to $\mathbf{20}\boldsymbol{Hz}$ with linear interpolation	$\mathbf{191}\%$	$\mathbf{648}\%$
Sum of Sine signals, from one Sines up to five sins, with linear interpolation	$\mathbf{299}\%$	$\mathbf{548}\%$
Gaussian filtered random signal with different gaussians, with linear interpolation	$\mathbf{351}\%$	$\mathbf{806}\%$
Pixel intensity from a real video, with linear interpolation	$\mathbf{272}\%$	$\mathbf{866}\%$

Table 3: The complete table of improvment of the MSE over uniformaly sampled

Appendix F.

We can see this is a sine wave with an oscillating changing frequency:

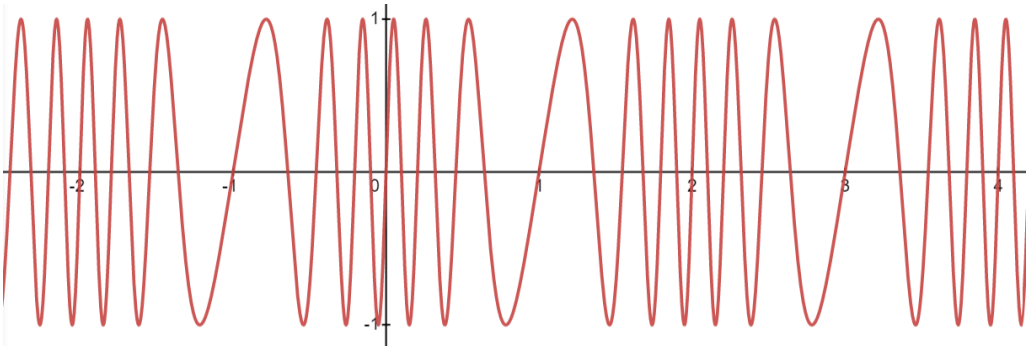


Figure 33: Oscillating Chirp

Here is a Short Time FT of the signal:

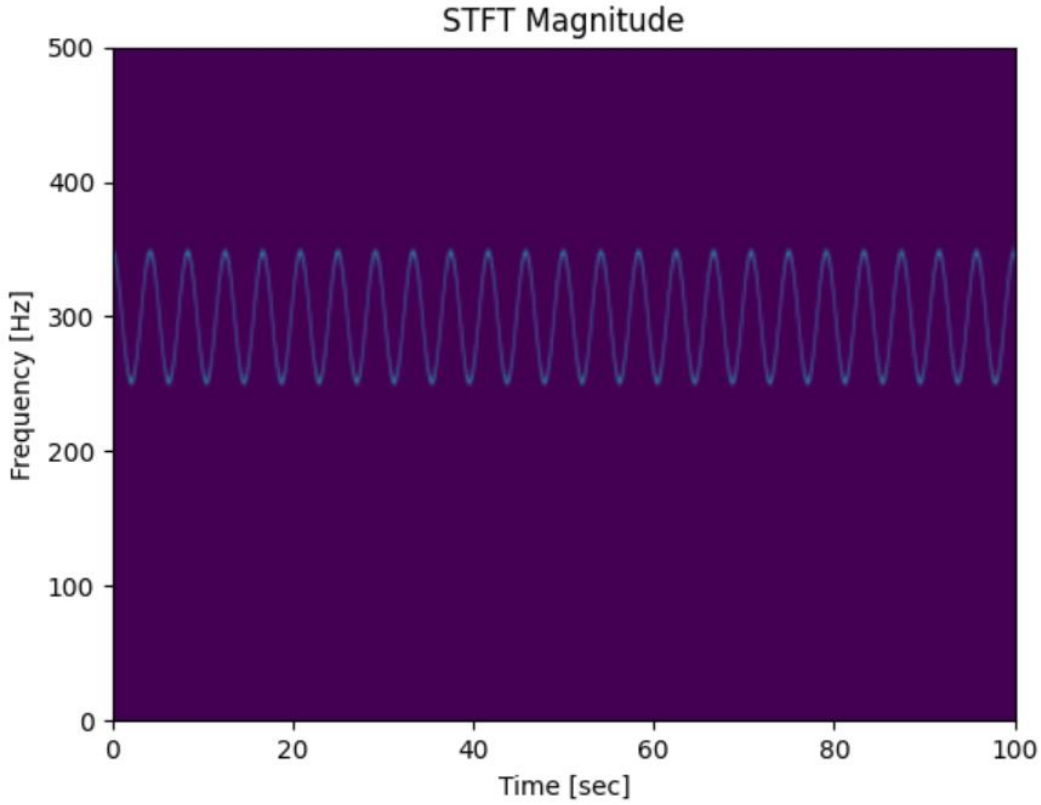


Figure 34: STFT of the Oscillating chirp

We can see that the frequency changes like a sine wave.