Machine learning in embedded systems project description

The goal of this project was to build an electronic signal shape classifier leveraging Machine learning technology and to port this model to c language, so that it could be deployed to ESP32 development board.

Neural Network

First using a neural network was considered. The idea was to make it so ESP32 would "see" the signal as a sort of image. 255 consecutive datapoints called input vector would be inputted into a basic neural network. This NN (neural network) would output 4 probabilities one for each signal type. If given datapoints would resemble training data for one of the signals, the NN would output something close to 1(very likely) and if the given datapoints would not resemble anything that the model had taken in during training, it would output for that signal something close to 0(improbable).

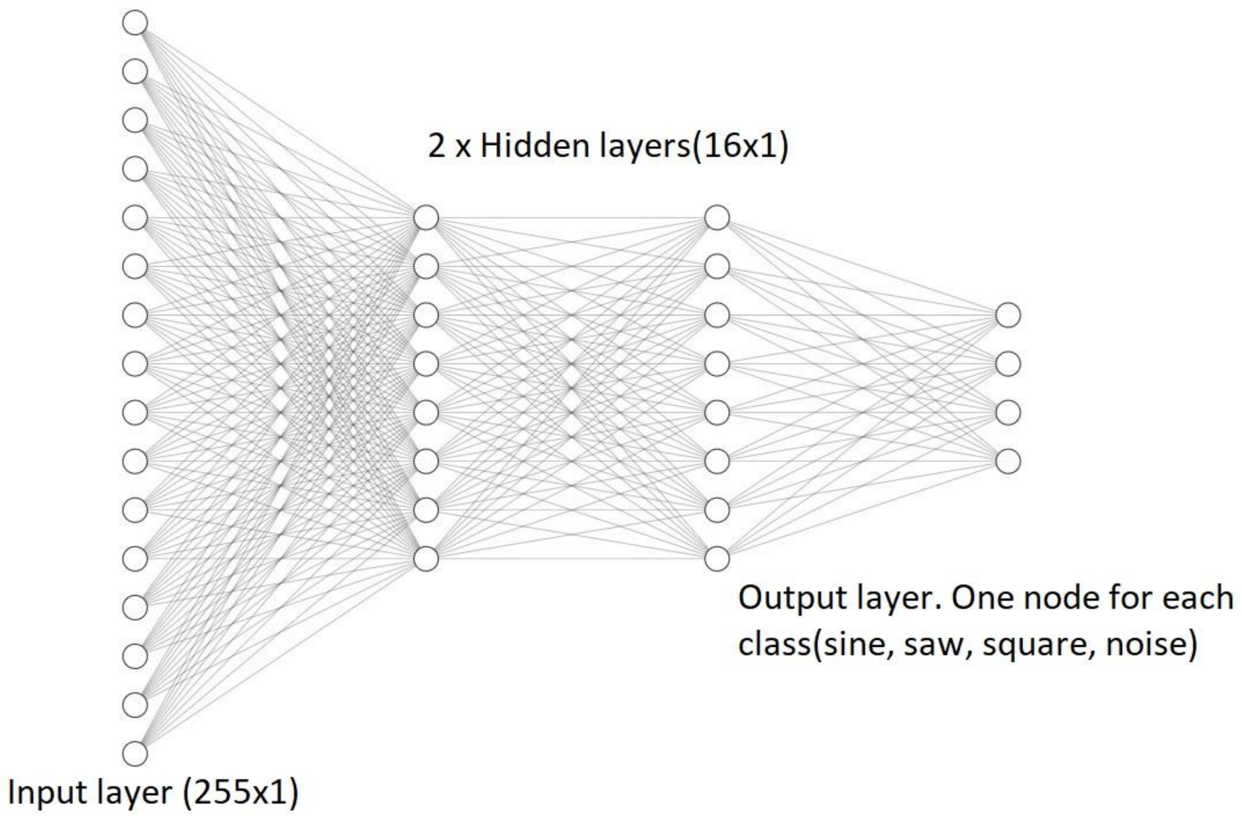
2 x Hidden layers(16x1)
Output layer. One node for each class(sine, saw, square, noise)
Input layer (255x1)

Figure 1. Neural network architecture (simplified)

Building this neural network model was done in python using tensorflow and keras packages. These packages have ready made tools for Neural network architecture defining, model training and testing. After training and testing had been done that model was ported with ewerywhereml package to C++.

Generating the training data was done with MATLAB. The MATLAB script included a lot of randomness in its design. That meant that all the signals had varying characteristics such as amplitudes, frequencies, phase shifts, dc components and some of the signals had white noise. An array with dimensions [10 000x255] was created and saved to a csv file.

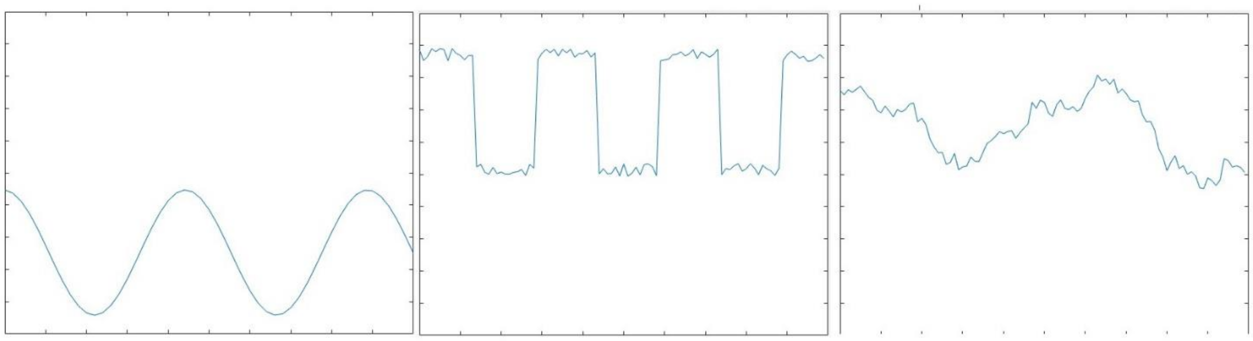
Figure 2. Examples of generated data

The csv file would be red in python and this data would be used for training and testing the model. During training and testing the model in python seemed to be working adequately. The model did work when it was tested on ESP32. The model was tested by sending samples of the original generated data. When testing analog signals that were red with an analog pin of the ESP32, the model rarely classified the signal correctly. This is when the troubleshooting process began. Firstly, it was suspected that the differing sources of data was the problems. IE the training data and the actual data inputted to the model during usage was too different. Decision of gathering training data with the ESP32 was made, instead of generating it synthetically with MATLAB. Data was gathered and it still had the same amount of variation as the prior data. After training and deploying the new model, it appeared that the model was still not working as wanted.

Why didn’t it work?

Although the model seemed to be performing sufficiently well during initial testing it didn’t perform well when it was given new unseen data. This was because the model was overfitting. The model couldn’t generalize well and few datapoints had too much meaning for the outcome of the outputs.

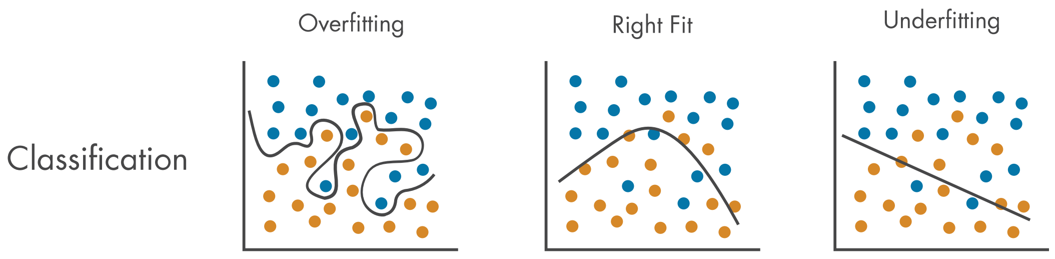

Figure 3. Overfitting visualized

In short, the signals had too much variation for such a simple Neural network. Very similar input signals with little difference could have largely different outputs. The data should be normalized so that for every datapoint vector the maximum value in that vector would be 1 and lowest value 0. (Actually, they could be 1 and -1 or any number just as long they are for any given input vector the same). Even bigger problem is the variation in phase shift and frequency.

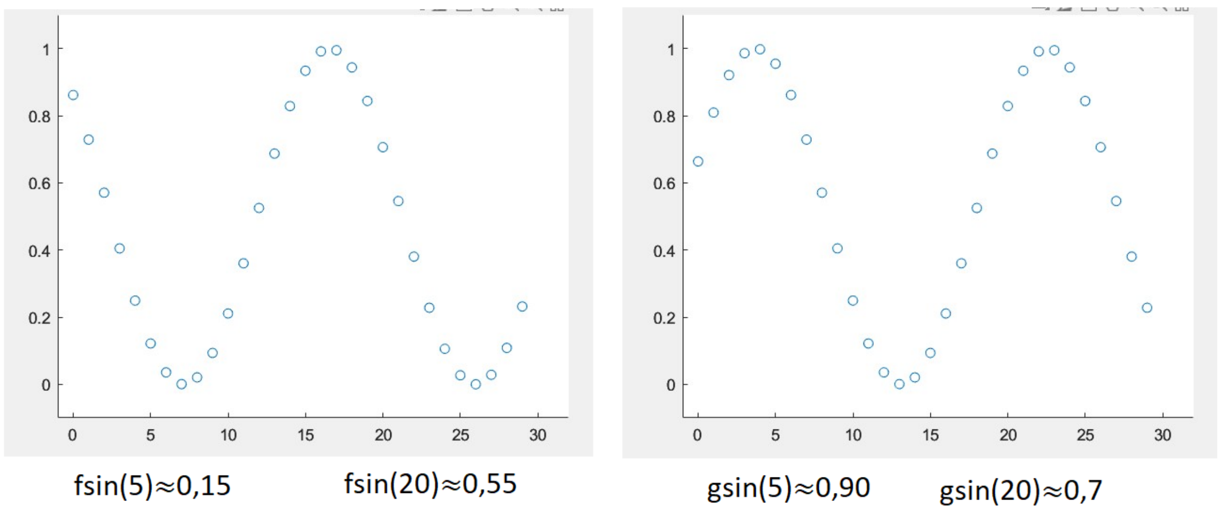

Figure 4. Same signal with two differing phase Shifts

The signals in Figure 4. look like each other for human eyes but for this specific neural network it is impossible to perceive any "similarity" due to the difference in phase because it leads to a large difference for any given input datapoint. This same effect would be caused by variation in frequency.

This problem could perhaps be solved by somehow normalizing the measured data so for example in the input vector the first data point would be equal the maximum measured value and last would be the minimum value. This would mean that the NN would only be considering one predetermined (from sin(90°) to sin(270°)) part of the signal. Other solution could be to add a convolution layer. However, neither of these solutions would not be pursued.

At this point of the project a new machine learning technique would arise to be possibly a better way of achieving the wanted result. The model would make a simple statistical analysis of the measured data and then deduct from this information which class it is most likely to belong. This kind of architecture can be made with a Random Forest Classifier.

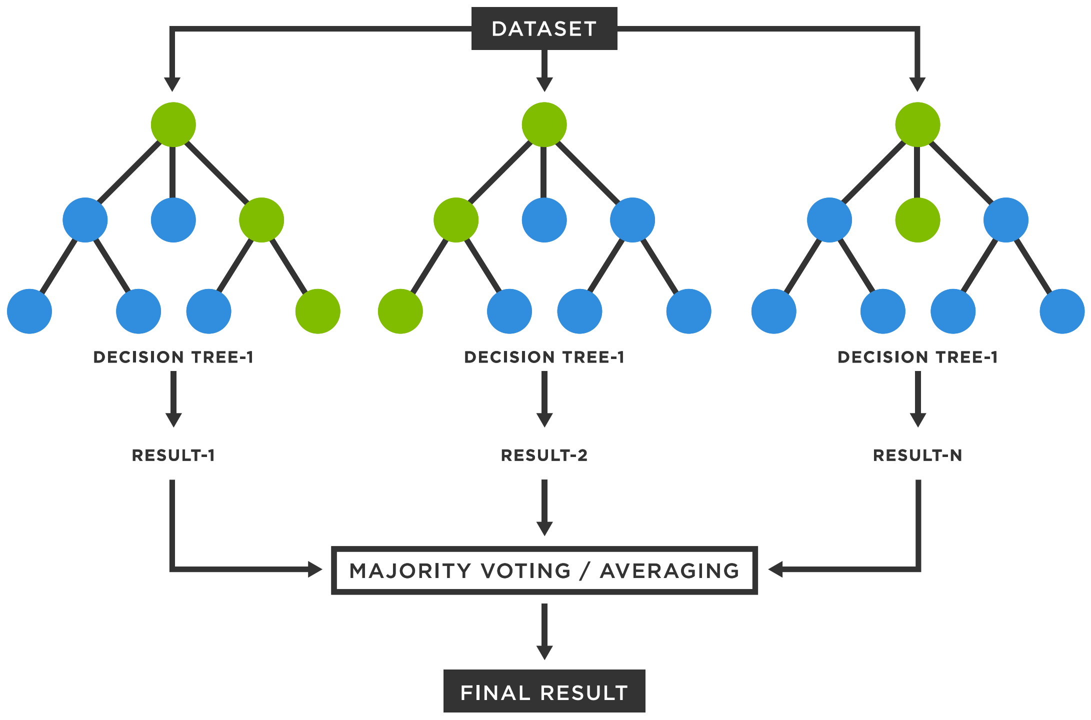

Figure 5. Random Forest Classifier

The whole python script needed to be rewritten when changing the architecture. The statistical analysis part required some trial and error as in which of the analysis methods were useful for classification. For example, the usage of mean was given up because it was almost always close to 0.5 no matter the signal shape so it would not offer any useful information. In the end 7 categories were acknowledged to be functional. First was the average delta between datapoints. This would be calculated with the formula:

$$\Delta_{avg} = \frac{|m_1 - m_2| + |m_2 - m_3| \ldots |m_{n-1} - m_n|}{n}$$

Where m1, m2 etc.. are the datapoints and n is the number of datapoint.

Second and third inputs to the model were Variance and Standard deviation. Fourth was Sum of squares which is calculated with the formula:

$$sqsum = \sqrt{m_1^2 + m_2^2 + m_3^2 \ldots m_n^2}$$

The last 3 methods work in the same principle as the first average delta, but instead of taking the next datapoint it would jump a determined interval. For example, Method 5 would take the average delta with datapoints that had 4 datapoints between them.

$$\Delta_{avg} = \frac{|m_1 - m_5| + |m_5 - m_{10}| \ldots |m_{n-5} - m_n|}{n/5}$$

Sixth method would have the interval of 10 and Seventh 50.

After training and testing the model with measured data it was ported to a c++ library with everywhereml. The library source code was placed in the same repository as the Arduino sketch so that it would be compiled and linked properly. The code that made the statistical analysis could not be ported automatically so it needed to be written in c manually.