**Intelligent Discrete Fourier Transform**

*Author:* Oisín Watkins

*Date:* 15/01/2021

*Abstract*

This project sets out to implement a new way for sensor / signal data to be read and understood by Neural Networks, of varying types, that replaces the common approaches of Recurrent Neural Networks (RNN’s) or their equivalents and Convolutional Neural Networks (CNN’s) with pre-processed data. The resulting algorithm amounts to an Intelligent Discrete Fourier Transform, which simultaneously transforms time-domain data to frequency-domain data and learns what content of the signal is meaningful to the given application. The final implementation was built using the TensorFlow API and is available for download from GitHub.

*Introduction*

At the time of writing this, the common practice for handling signal data using Neural Networks breaks down into one of two choices: Recurrent Neural Networks (RNN’s) and their equivalents, or pre-processing data into images to accommodate 2-D Convolutional Neural Networks (CNN’s). Neither approach is optimised to the task. RNN’s (or more recently 1-D CNN’s) handle series data, which in the context of signal processing is the equivalent of *only* working in the time domain. While it is possible, the amount of computation required to overcome the limitations of working in the time domain prohibit the use of RNN’s / 1-D CNN’s in most complex real-time applications. Likewise, pre-processing data into image format before using a conventional 2-D CNN is both time consuming and wasteful from a processing standpoint. The need to generate an image before being able to discern meaning from the data all but exclude this setup from being viable in edge-sensor networks or in real-time systems.

What both approaches lack is the ability to transform sensor data from time-domain to frequency-domain. Certainly, the sensor data – to – image approach attempts to overcome this hinderance, but the question remains: why perform unnecessary processing on a signal prior to discerning meaning from the signal. In this regard the RNN approach has the lead, as it does not pre-process the data at all, but being limited to the time-domain does hold this approach at a disadvantage. The goal of this project is to find a way to pull the best parts from both approaches to make one computationally efficient, real-time system capable approach of handling signal data in the frequency domain. The result was a Discrete Fourier Transform which learns what parts of the signal are meaningful for the given application (an Intelligent Discrete Fourier Transform, if you will).

After the initial tests and demonstrations using MATLAB, a DFT layer was written using the TensorFlow API in Python. As a test of efficacy, this new layer was used to build a model which categorised 1s audio clips of one-word voice commands (eg: “yes”, “no”, “up”, “down”, etc.). The effectiveness of this model was compared to that of a 1-D ConvNet used as a demonstration in numerous online guides for audio processing with Artificial Intelligence. In this paper, we will review the mathematics behind the Intelligent DFT, the resulting implementation in TensorFlow, and the limitations of this implementation. For reference, the entire body of code discussed in this paper is available at: <https://github.com/OisinWatkins/Intelligent_Signal_Reader>

All models were defined and trained on a Dell Precision 5530 Laptop, supporting an Intel Core i7-8850H CPU, running at 2.6GHz with on-board graphics.

*Overview*

The Fourier Transform is a well-documented variant of the Laplace Transform commonly used in the realm of signal processing. Formally it is defined as:

|  |  |  |
| --- | --- | --- |
|  |  | [1] |

However, implementing this equation on digital systems requires the use of Discrete Time, rather than the native Continuous Time equation shown in Equation 1. This Discrete Time Fourier Transform (more commonly referred to as the Discrete Fourier Transform, or DFT) is formally defined as:

|  |  |  |
| --- | --- | --- |
|  |  | [2] |

Where:

|  |  |  |
| --- | --- | --- |
|  |  | [3] |

As written by ~ the DFT can be implemented using Matrix Algebra, as shown in Equation 4.

|  |  |  |
| --- | --- | --- |
|  |  | [4] |

It is this matrix implementation of the DFT that forms the basis of this research. Using this approach, one could theoretically incorporate a DFT into a larger Deep Learning network. Given that backpropagation simply adjusts network parameters along the negative of the error gradient, and that the derivative of the error w.r.t. the values of the DFT matrix is linear, there should be few issues with incorporating this equation into larger networks.

Principally this does hold true, however as will be shown in the following sections, there are still some issues with implementation. Foremost among them is the inability of most Deep Learning frameworks to accommodate Imaginary Values in the Error Gradient. At the time of writing this it does not pose a major issue. With careful implementation it is possible to build a fully working Neural Network that uses Imaginary values in its layers. However, this limitation does have an impact on model performance. More on this later.

*Implementation*

Considering again Equation (4), building a rough proof of concept in a scripting language seems a reasonable place to start. The most prudent things to test here are:

1. Efficacy
2. Response to various Loss Functions
3. Stability

Under the “Demonstration” folder of the repository is just such a proof-of-concept stored as a .mlapp file. Viewing the underlying code will require a valid MATLAB license. This demonstration explores the behaviour of a simple model which only performs a DFT and trains using an error signal tied to white noise in the input signal. Figure (1) shows the application just after opening.

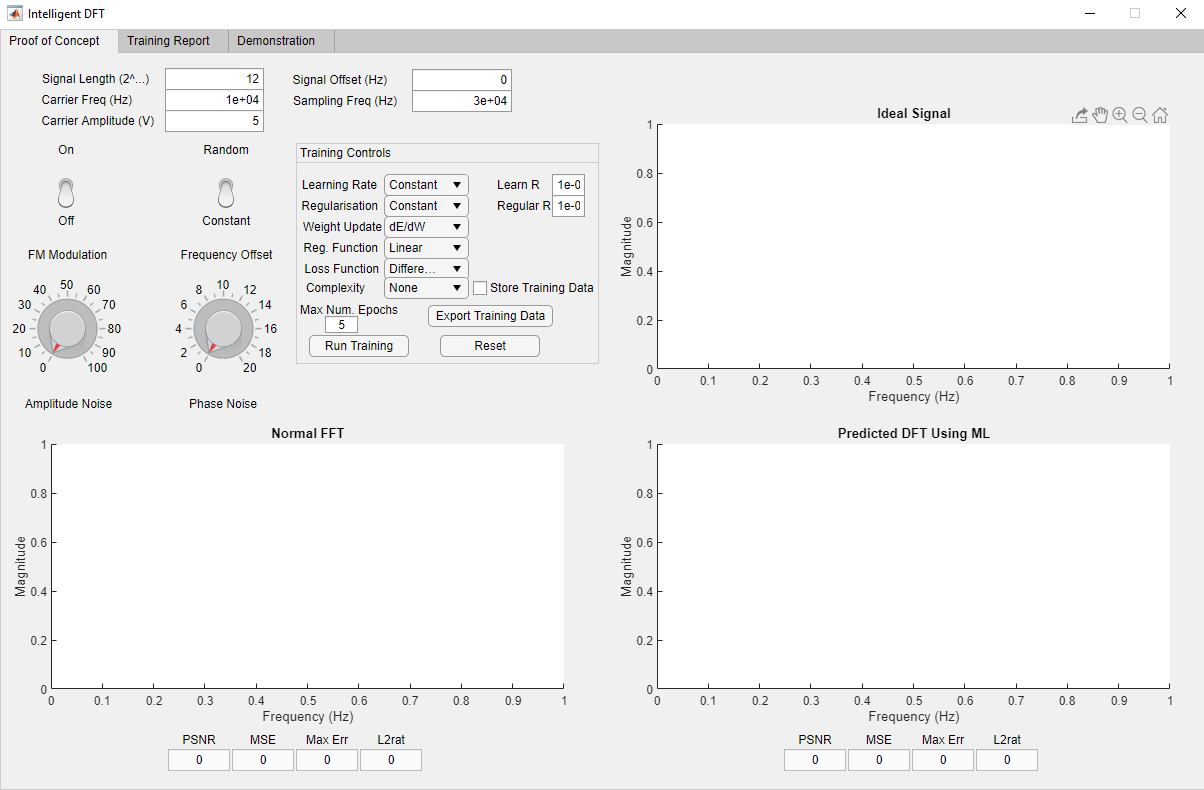


Figure : Intelligent DFT Proof-of-Concept GUI

As with any trained model, there is a wide range of possible Error Signals (and thus Error Gradients) and weight-update rules that can be employed. In this proof of concept, the following selection of Error Signals, Learning Rates, Weight-Updates and Regularisation Equations were used:

|  |  |  |
| --- | --- | --- |
| Weight Update Rules | Equation | Equation Number |
| 1) |  |  |

|  |  |  |
| --- | --- | --- |
| Error Gradients | Equation | Equation Number |
| 1) |  |  |
| 2) |  |  |
| 3) |  |  |
| 4) |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Learning Rate | Equation | | Equation Number |
| 1) |  |  | |
| 2) |  |  | |
| 3) |  |  | |
| 4) |  |  | |
| 5) |  |  | |
| 6) |  |  | |

|  |  |  |
| --- | --- | --- |
| Complexity Equation | Equation | Equation Number |
| 1) |  |  |
| 2) |  |  |
| 3) |  |  |
| 4) |  |  |

|  |  |  |
| --- | --- | --- |
| Regularisation Rate | Equation | Equation Number |
| 1) |  |  |
| 2) |  |  |
| 3) |  |  |
| 4) |  |  |
| 5) |  |  |
| 6) |  |  |

Each of the figures [2] – [5] shows a slightly modified version set of hyper parameters used to train the Intelligent DFT layer. The output of the Intelligent DFT layer is compared with the output from a conventional FFT algorithm based on the PSNR, MSE, Max Error and L2 norm ratio. Any value highlighted in green indicates that it is the lower of the two in each respective comparison.

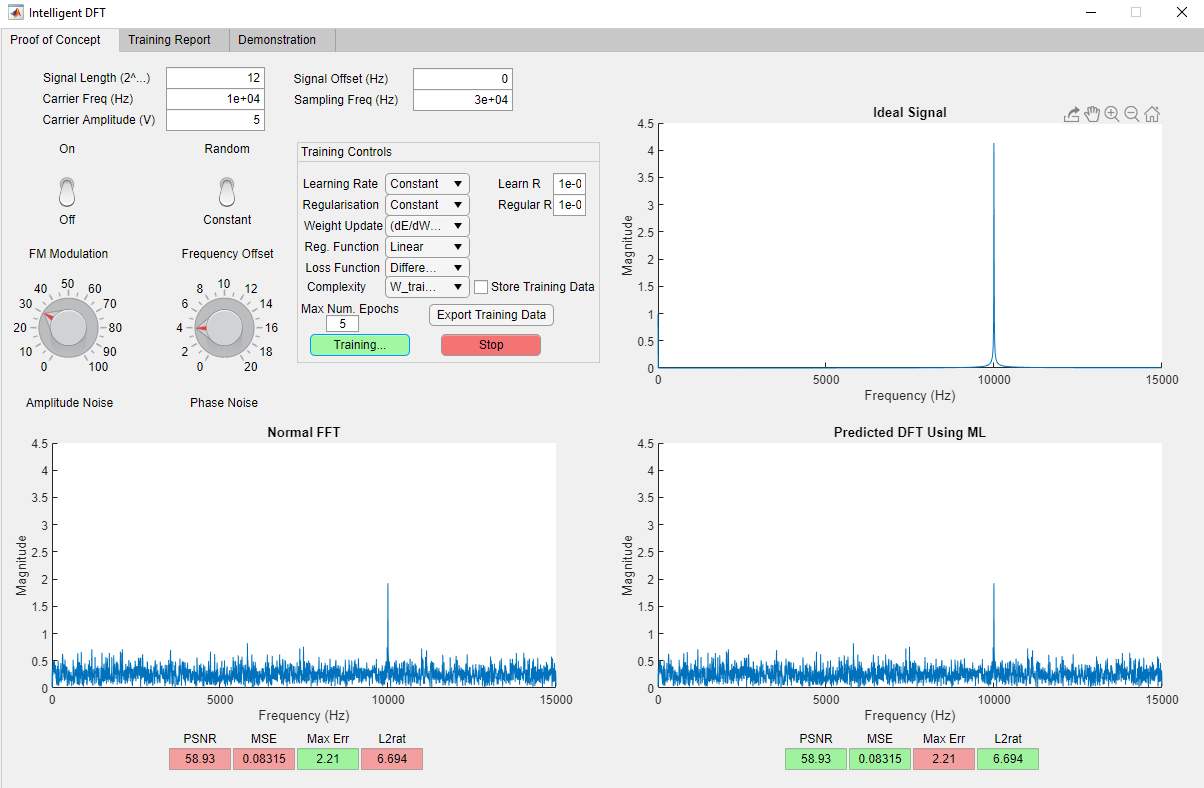


Figure : Proof-of-Concept -> Capture 1

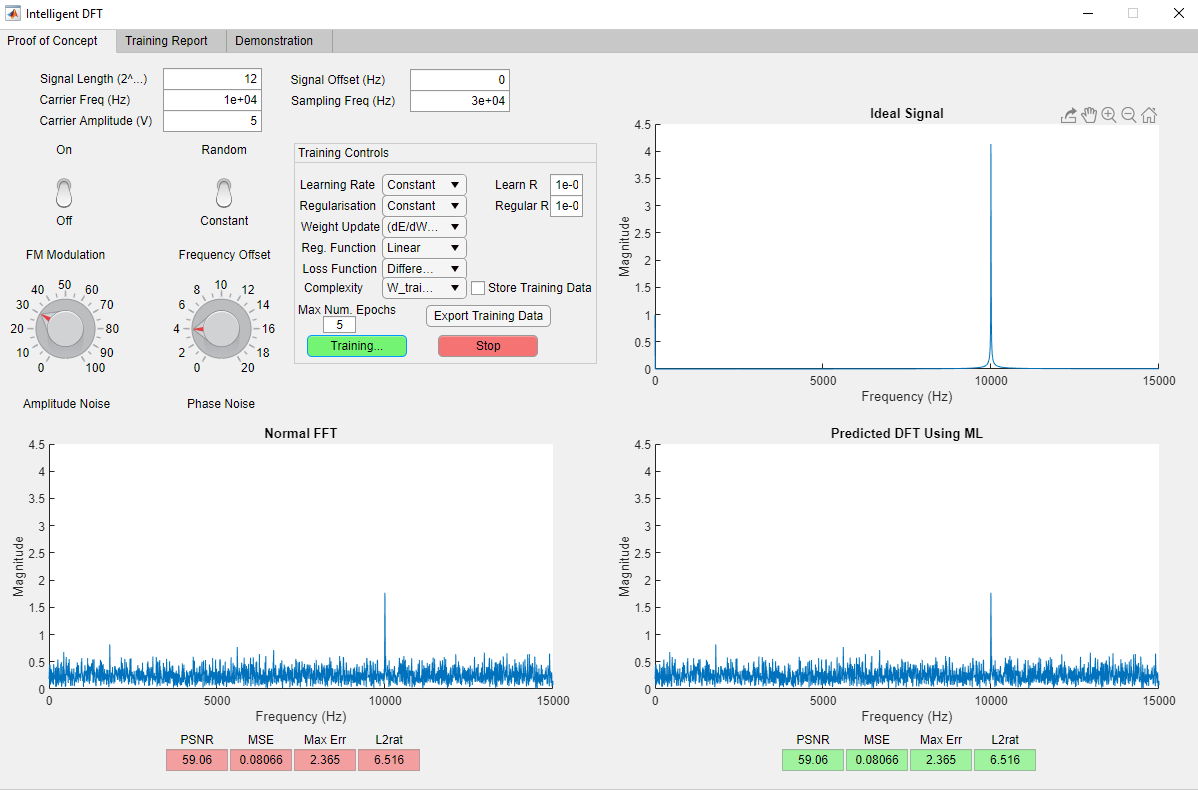


Figure : Proof-of-Concept -> Capture 2

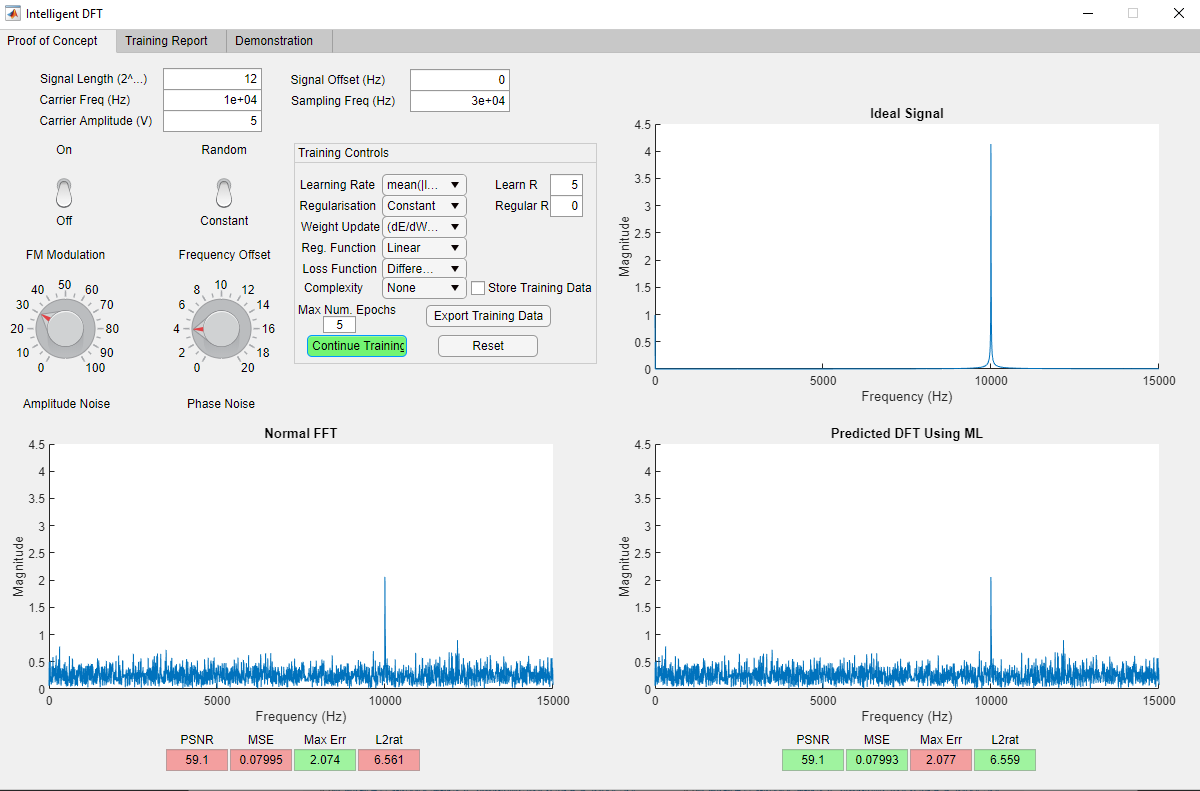


Figure : Proof-of-Concept -> Capture 3

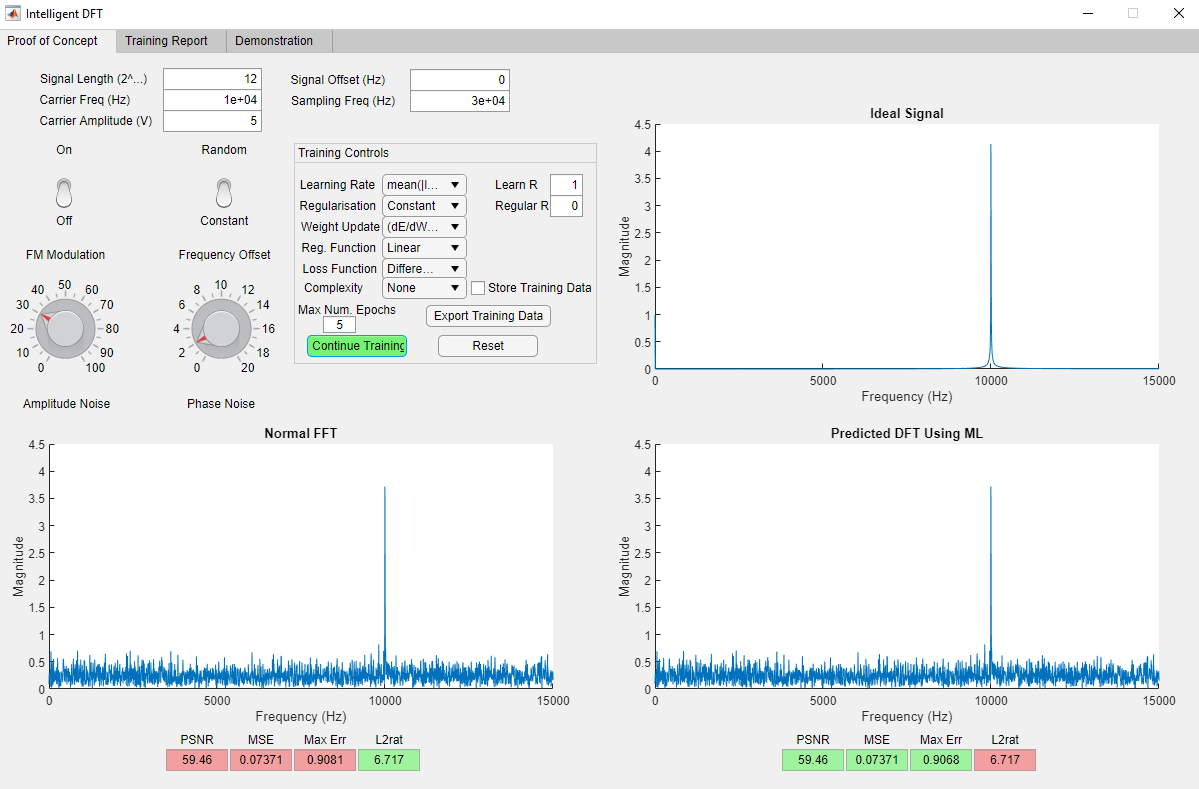


Figure : Proof-of-Concept -> Capture 4

This very low-level implementation certainly proves that this concept at least has potential to provide a benefit to signal-processing applications that benefit from having intelligence in their systems. In this proof-of-concept, I used noise to provide an error signal to train against. Certainly, with the correct configuration of all training hyperparameters, it is possible to develop a model which mitigates noise in systems such as these. However, there is a real balance to strike between noise mitigation and stability. It may be the case that systems with more complex error signals could see greater stability in training, such a classification models or audio-generation models.

With the assurance that this concept has potential, the higher-level implementation using the TensorFlow API was built. The goal here was to build a Layer object that had less customisation options as the MATLAB proof-of-concept but was capable of being integrated to a larger Neural Network. The resulting DFT Layer (shown in Figure (6)) can be found in the Fourier\_Transform.py file:

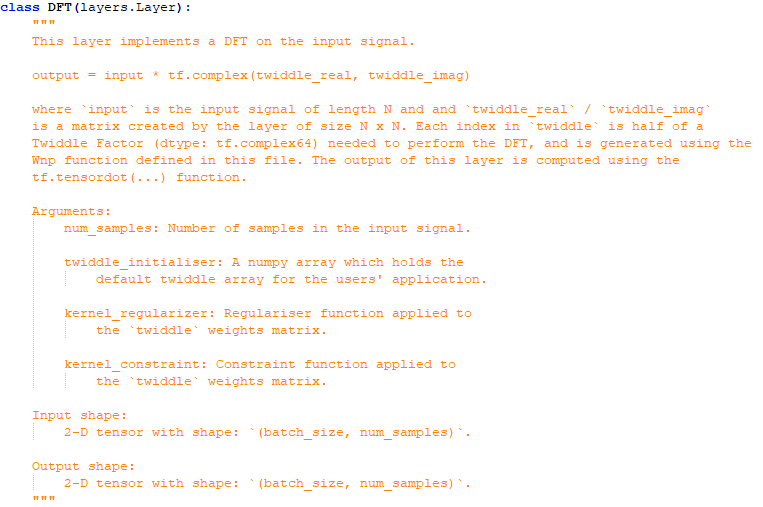


Figure : Description and Definition of the DFT Layer.

The Layer weights are stored as Float32 values and conjoined into the necessary Complex64 values using the tf.complex(…) function. This is done to mitigate gradient issues, which will be detailed in full in the Discussion section of this paper. Crucially, this implementation is focused on 2-D input tensors. While it may be possible to perform a 3-D DFT using equation (4), I have not attempted it in this iteration.

Note also, there is room provided to give this Layer a kernel regulariser and a kernel constraint, howeverthis was not the focus of this early implementation. The proof-of-concept seemed to indicate that adding such functionality to the Layer could provide some benefit to the training stability and operating efficacy, so it is worth returning to this implementation and fleshing it out further.

Many applications will require signals with a great number of samples be processed in near real-time. An example of this would be analysing snippets of voice. The minimum sampling rate accepted for voice recording is 8kHz, meaning that a 1s audio clip requires a vector with shape: (1, 8000). For this layer, that would mean it would need 2 \* 80002 parameters, a total of 128,000,000 Float32 parameters. Irrespective of the performance benefit this layer could provide, the number of parameters it needs is simply prohibitive. Figure (7) shows the un-spliced model layout as a block diagram.

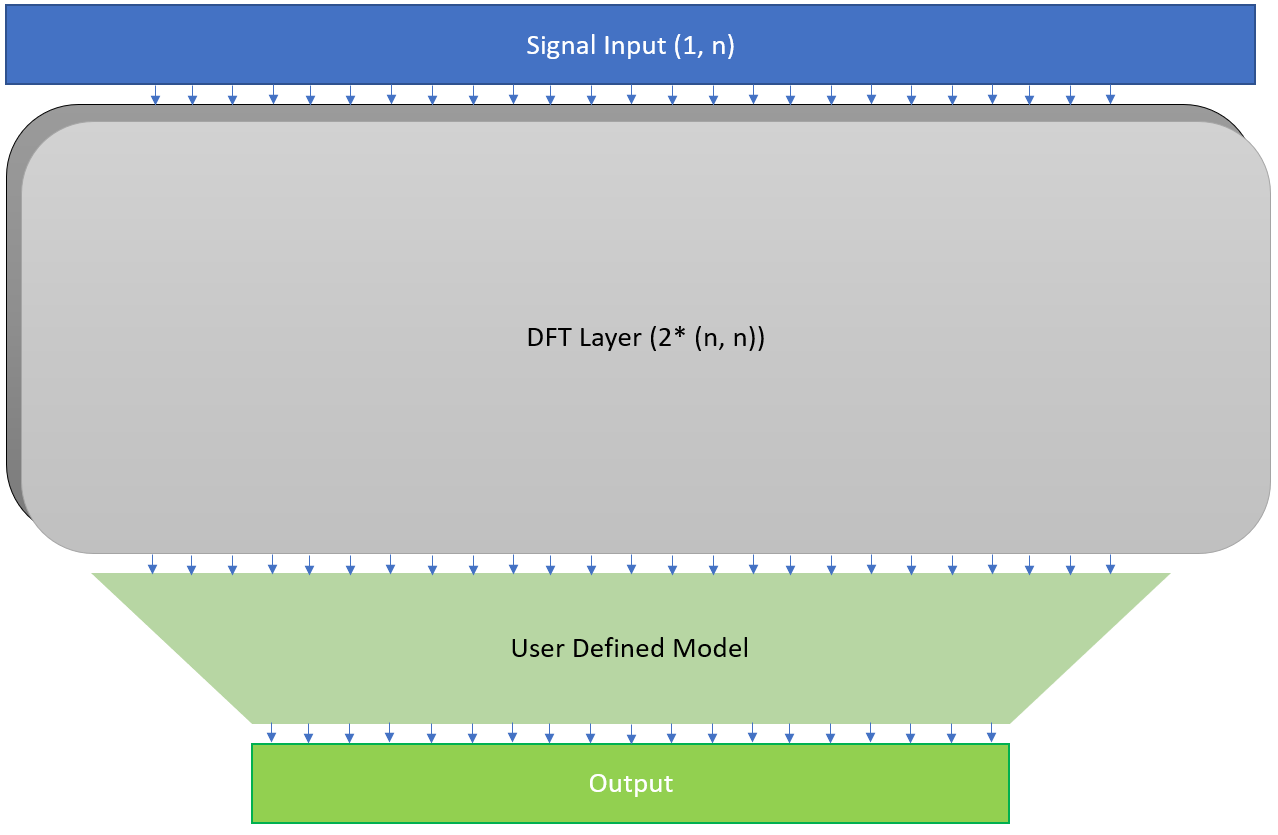


Figure : Model using DFT Layer without Input Splicing.

Depending on the application, however, it may be possible, even beneficial, to splice the input into N-even slices and compute each DFT independently. Aside from providing more of a “Spectrogram” instead of just a DFT, this approach also allows for significant memory savings. Using the algorithm shown in figure (8), a user can determine the number of splices of the input that will yield the minimum number of total parameters (written in MATLAB):

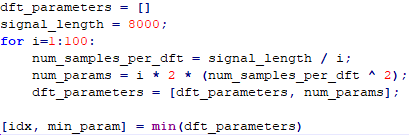


Figure : Algorithm to Find No. Splices to achieve minimum No. Parameters.

Depending on the specific use-case, it may also be worthwhile extending the for-loop to run to a higher maximum value. There is also a decision to be made as to input padding. In its current form, this algorithm does not account for the values of ‘i’ which do not divide the signal length evenly. The two choices a user has is to either use as many full DFT layers as possible and 1 DFT layer that is a fraction of the size, or to pad the input and have an exact number of equally sized DFT layers. Either approach will work, but no experimentation has been done to determine if there is any performance difference between these approaches. Figure (9) shows a block diagram of this splicing methodology.

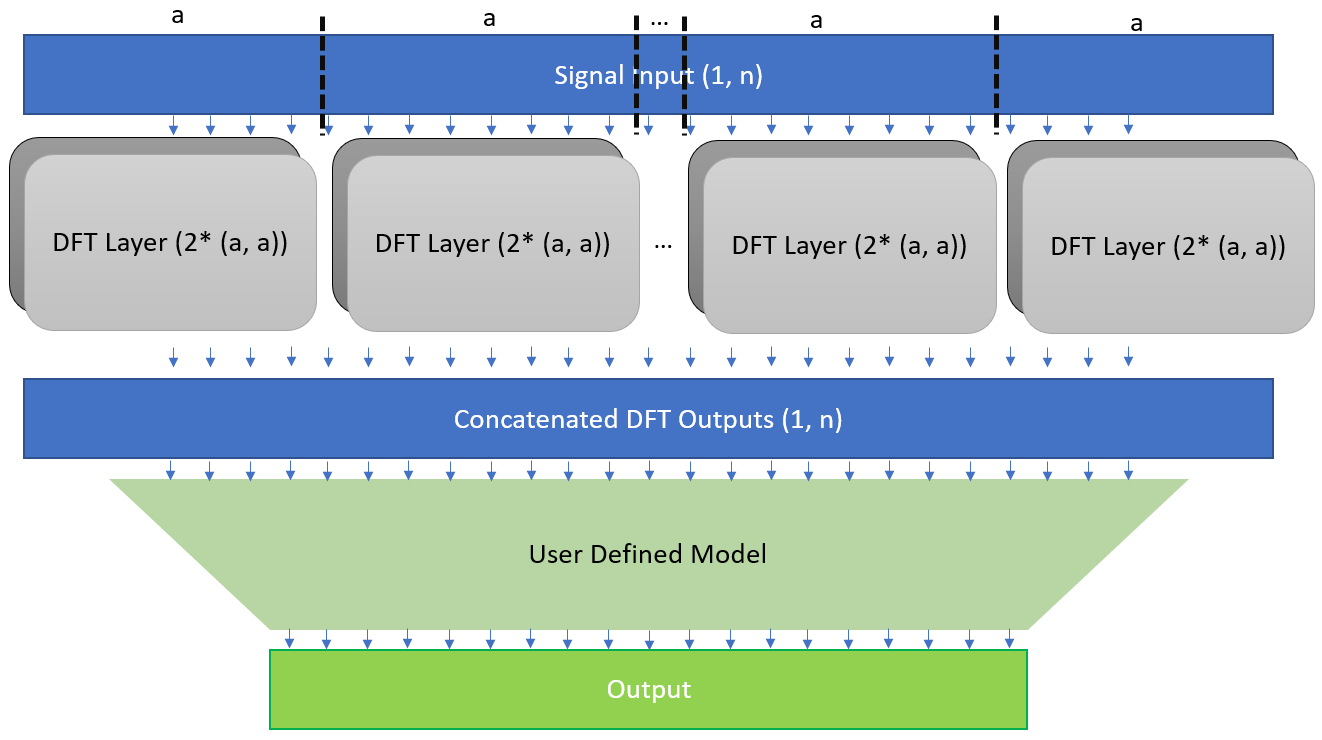


Figure : Model using DFT Layer without Input Splicing.

Taking this input-splicing approach, with input-padding, this Layer was then incorporated into a model which classifies 1s voice clips into 1-word commands. The sound clips can be downloaded for free at the following link:

<https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data>

The list of words used in training is shown in Figure (10):



Figure : List of all words the upcoming models will be trained to identify.

For reference, a more standard model which would commonly be used to tackle this task was also downloaded. It is available at the following link:

<https://github.com/aravindpai/Speech-Recognition/blob/master/Speech%20Recognition.ipynb>

This same model was defined and trained on the same laptop as the DFT model (defined in full in the Results section). Their training and validation performance were compared, as well as their behaviour in training noted.

*Results*

*Standard Model:*

The Standard model follows the well documented approach of using Conv1D, Dropout and Maxpooling Layers to handle time-domain signals, before using a classifier built from a Flatten Layer and some Dense layers. A summary of this model architecture is provided in Figure (11):

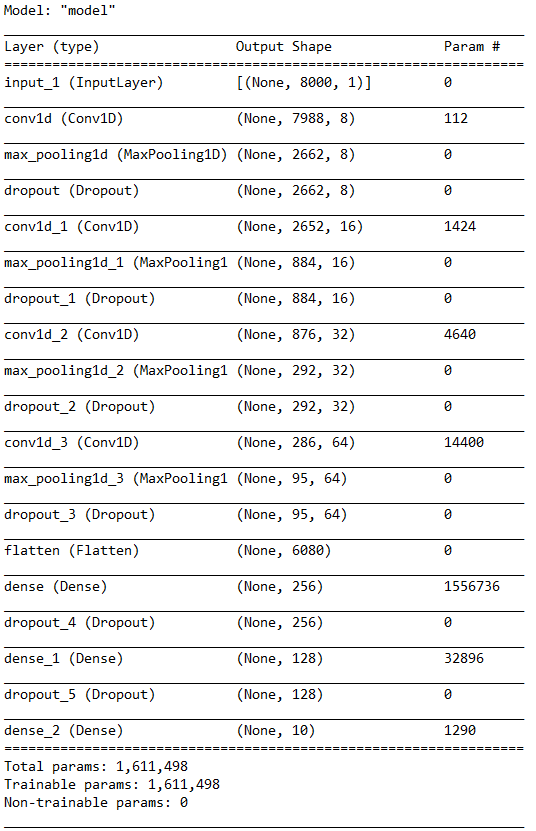
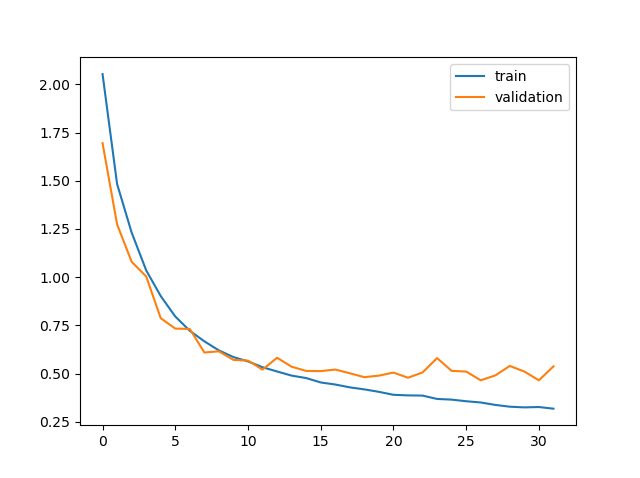


Figure : Standard Model Summary

A few comments on this model layout:

1. The simplicity of the layout will likely lead to a very repeatable and stable training process.
2. Of the ~1.6 million parameters in this model, 1.5 million of them are in the first Dense Layer. This is a common occurrence when using a Flatten Layer on a relatively highly dimensional input. The result is that most of the heavy lifting of this model will be handled by this one Layer.
3. The strictly linear format of this model means it will not benefit strongly from multi-threading. As shown in Figure (13), this impacts presentation processing times (8-9ms / sample) leading to a longer training cycle and more difficulties when applying this solution to real-time tasks.

Figure (12) shows this model’s Loss value plotted over Epoch for both training and validation. As can be seen, this model first starts to overfit at epoch 12, with training continuing until epoch 32 when the Early Stopping callback is invoked. Figure (13) details the training report for this model. Here we can see the processing time per sample, as well as all Loss and Accuracy scores during training and validation.



Epoch

Loss Value

Figure : Standard Model Loss during Training and Validation

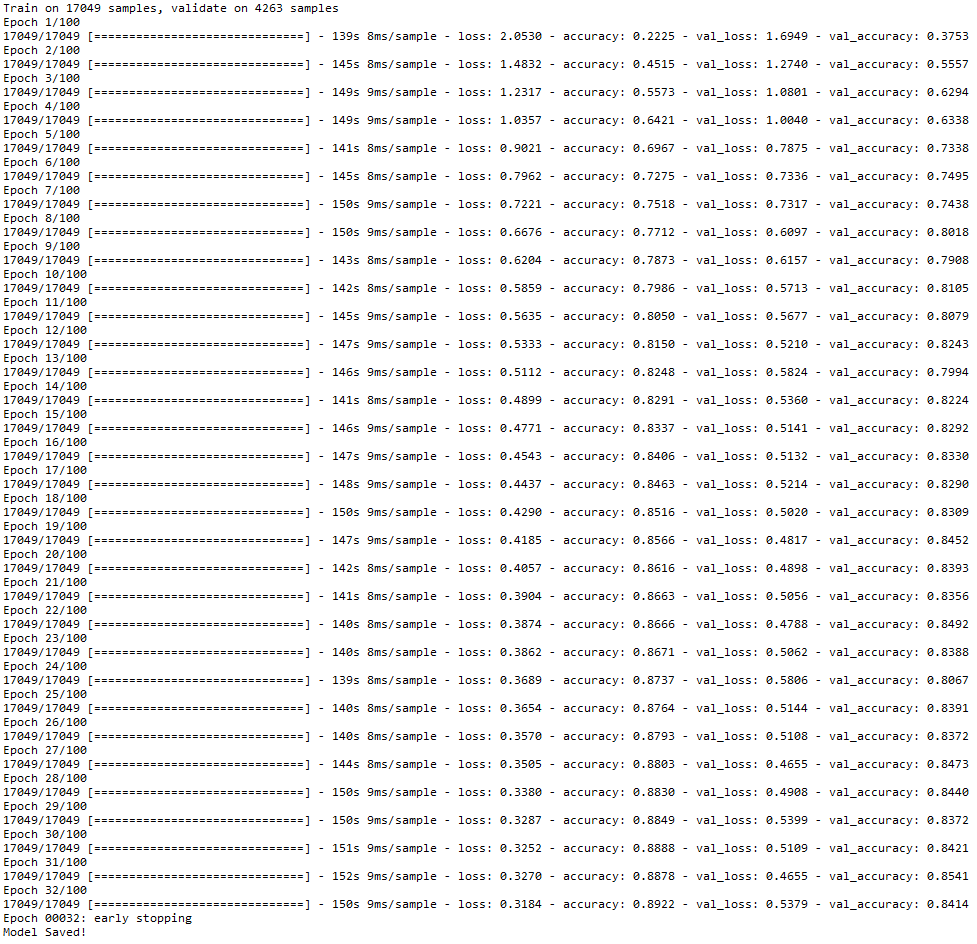


Figure : Standard Model Training Report

*DFT Model [Layout #2]:*

Multiple layouts for the DFT model were attempted, all are available on GitHub with their respective training reports. In this document we will inspect Layout #2. The model initially pads and splits the input into 63 splices, each one 128 samples long. Each splice is then fed into a dedicated DFT Layer, whose outputs are then concatenated and transposed to maintain the correct data shape (batches stored on the first dimension). The corrected “Spectrogram” is then classified using a collection of Separable Conv1D, Maxpooling, Dropout, Flatten and Dense layers.

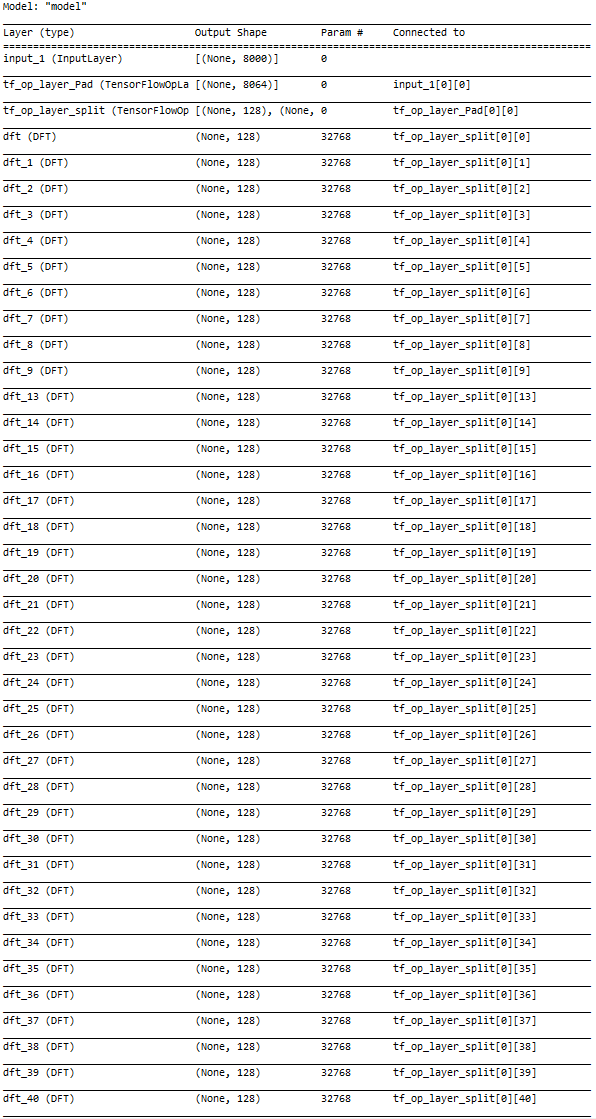


Figure : DFT Model [part 1]

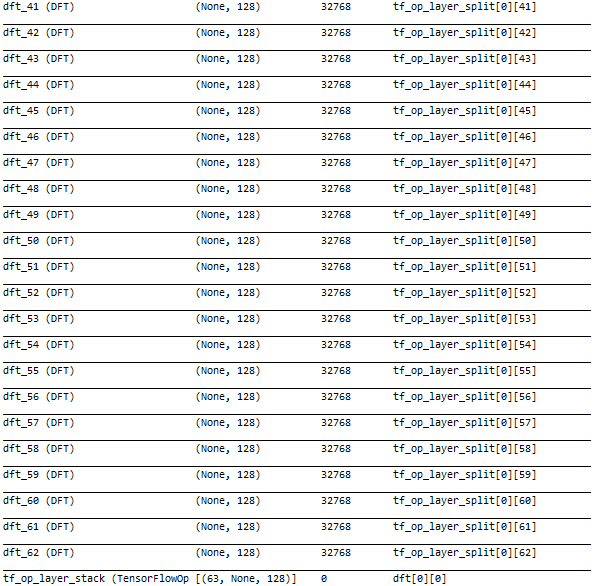


Figure : DFT Model [part 2]

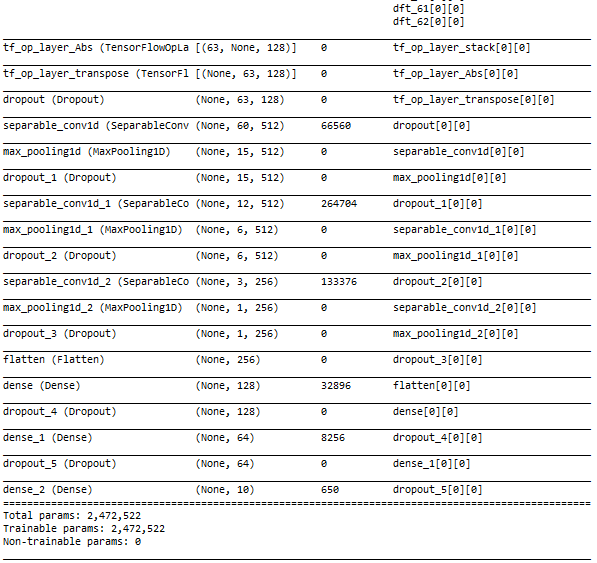
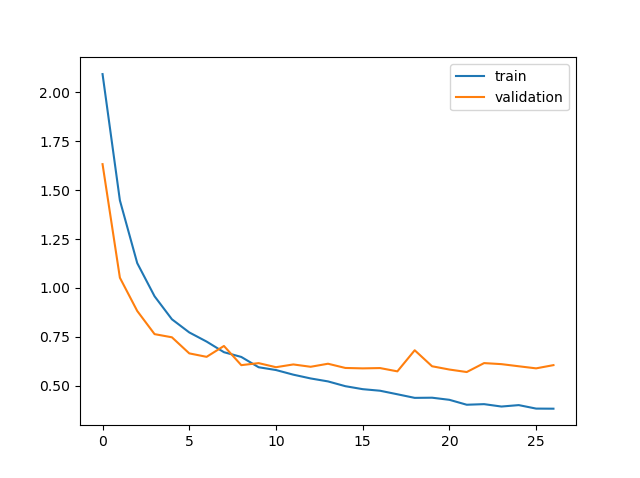


Figure : DFT Model [part 3]

Comments on this model:

1. The parallel nature of this model allows it to benefit strongly from multi-threaded operation. This model takes 3-4ms / sample, which is 2-3 times faster than the Standard model could achieve on the same computer.
2. Even with this “minimum-parameter” approach, the resulting network is comprised of ~2.5 million parameters, an increase of roughly 50% over the standard model.
3. The number of weights stored in each layer is now much more evenly spread throughout the model. No one layer takes most of the work. Each DFT layer has 32,768 parameters, meaning that roughly 2,000,000 parameters are dedicated to producing the “Spectrogram”.

Figure (17) shows the DFT model’s Loss value plotted of Epoch for both training and validation. As can be seen, this model first starts to overfit at epoch 11, with training continuing until epoch 27 when the Early Stopping callback is invoked. Figure (18) details the training report for this model. Here we can see the processing time per sample, as well as all Loss and Accuracy scores during training and validation.



Epoch

Loss Value

Figure : DFT Model Loss Value during Training and Validation

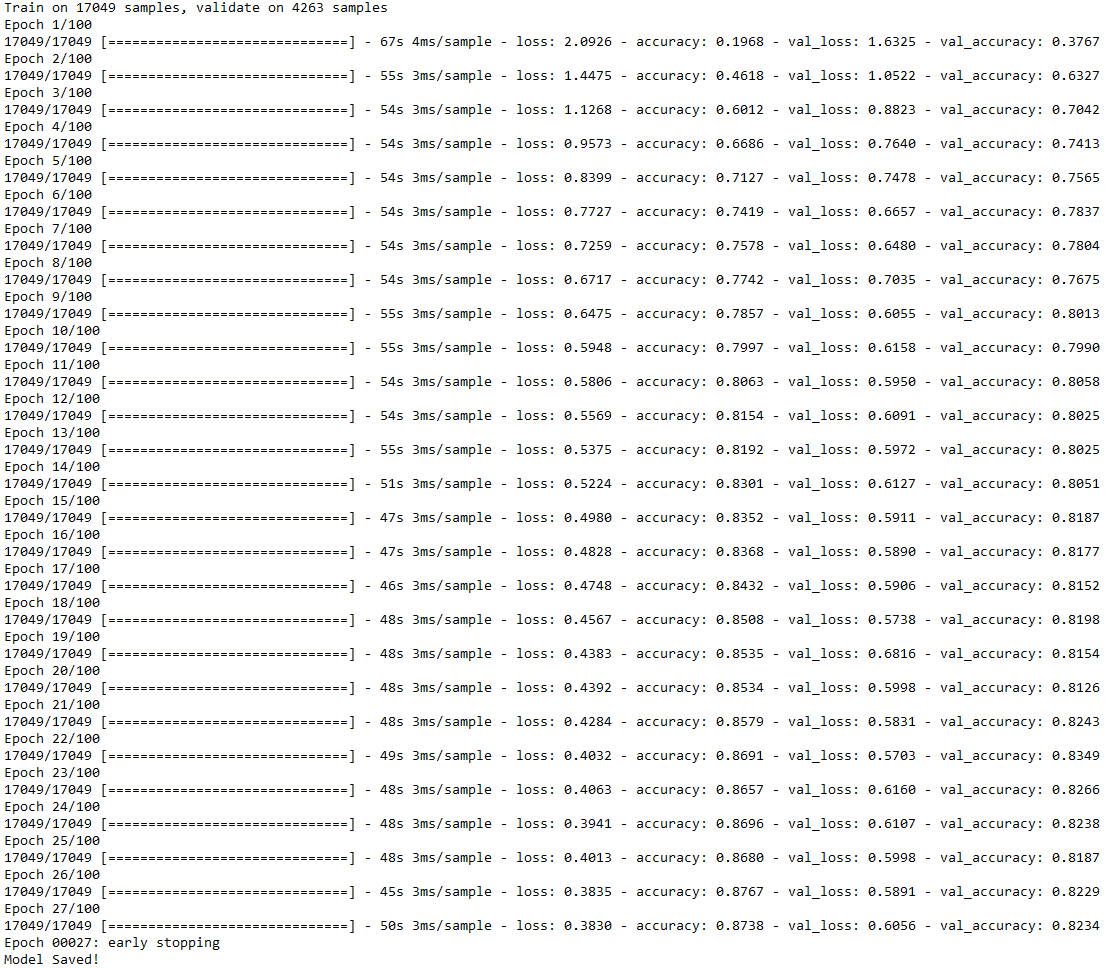


Figure : DFT Model Training Report

*Discussion*

Let’s begin by discussing the standard model. In summary, the final validation cycle saw Accuracy of 84.14% and a Loss of 0.5379. Given that this is the “go-to” approach and little thought was given to it other than to do what worked, these results are as good as we could expect. Adding more layers or adding more neurons to layers isn’t likely to improve the final performance very much. Achieving any higher accuracy or lower loss would likely require a more substantial amount of effort. Given all of this, taking 84.14% accuracy and 0.5379 loss as a baseline makes a lot of sense.

Now let’s consider the hardware. The Intel Core i7-8850H is a very decent laptop processor with respectable specifications. Running 6-cores, 12-threads at 2.6GHz, it’s a competent machine. Accompanying this processor is 16GB DDR4 RAM, which is more than enough to handle these applications. Given that this hardware setup readily supports multi-threaded operations and isn’t tight on memory, it’s fair to say that the machine of choice isn’t a bottleneck to be accounted for. Certainly, faster throughput times could be achieved for either model with faster hardware, however this experiment was not limited by hardware performance in the slightest.

With all of that in mind, the processing times for the standard model do highlight themselves as that model’s weakness. While 8-9ms may be fast enough for some real-time applications, adding more processing to mitigate overtraining or making the classifier any bit more complicated is likely to result in increasing these processing times noticeably. This is due in part to the linear nature of the model, and in part to the relative size of the classifier block in comparison to the rest of the model. This type of a bottom-heavy design will likely mean that scaling this model upwards will result in the addition of a great many parameters.

Now the DFT model. Achieving a final validation Accuracy of 82.34% and validation Loss of 0.6056, this model’s performance is comparable, if slightly worse than that of the standard model. The additional 50% in model size is nothing to be ignored. It will vary from application to application whether this extra size is an acceptable price to pay for the additional throughput speed. This increase in speed is due largely to the bulk of processing that is handled in the parallel DFT layers. This parallel architecture does allow the multi-threading capabilities of the laptop to provide a great speed boost to the model.

One of the issues with this model is how quickly it begins to overfit the training data. At this point insufficient testing has been done to determine the cause of this tendency to overfit so quickly. Two initial thoughts would be the abundance of trainable parameters so close to the input (2 million of the 2.5 million in the model) and that the “Spectrogram” representation is exceptionally effective for discerning voice commands. One helpful attribute of this layout, however, is the fact that the lower parts of the model (the Conv1D and Dense layers) don’t tend to be very large by comparison. This does offer decent scalability to the lower part of the model. In subsequent tests, it was found that expanding this lower section with more complex classifiers and greater overfitting-mitigation capability made a very large difference to training and validation performance. Inspect Layout #4 under the DFT Models folder in the GitHub repo for more details.

By far the biggest drawback to the DFT model is the lack of support for imaginary gradients in native TensorFlow. This model does get around the bulk of the issues posed by having imaginary gradients in TensorFlow models by using the tf.Complex(…) command as specified earlier, and by using a tf.abs(…) function call directly after the DFT layers. Mathematically speaking, backpropagation through an absolute value function will at best lead to training being limited to half its potential, and at worst lead to backpropagation failing completely. Early versions of this model could not account for the imaginary gradient at all, and while the forward pass seemed to work, no backward pass ever would. So, despite the performance of this model being very reasonable given the application, it could be the case that better performances can be achieved if imaginary gradients were supported in TensorFlow’s gradient implementation.

However, in a custom application (for example, an edge sensor network), it’s likely that the network would be built with some C-based backbone. If this DFT model were to be built from scratch, more like the MATLAB proof-of-concept, then the imaginary gradient issue would largely disappear. This does open the door to more investigation and, potentially, greater appeal to the layer in edge-sensor applications.

*Future Research*

The future research that this project warrants is easily defined:

1. Complete the DFT Layer implementation, providing support for kernel constraints and regularisers.
2. Characterise the Layer’s performance fully, using a broad range of applications and data types. Compare the performance in each circumstance to an equivalent model defined using more typical signal-processing methodology for Artificial Intelligence models.
3. Build a DFT model for an edge-sensor network using native C or another bare-metal language. This step will also play a big part in proving whether the lack of support for imaginary gradients in the TensorFlow API plays a part in limiting the training performance of the layer.

*References*