
Honors Program Thesis

Computer Engineering

# Heterogeneous Implementation of an Artificial Neural Network

Authors:
Anthony Carvino
Thomas Coppola

Advisor:
Yi-Chung Chen Ph.D.

May 16, 2018

**Abstract**

Reconfigurable logic devices, such as Field Programmable Gate Arrays (FPGA), offer ideal platforms for the dynamic implementation of embedded, low power, massively parallel neuromorphic computing systems. Though somewhat inferior to Application Specific Integrated Circuits (ASIC) with regard to performance and power consumption, FPGAs compensate for this small discrepancy by providing a versatile and reconfigurable fabric that is capable of implementing the logic of any valid digital system. Using the Xilinx ZYNQ 7 Series All Programmable System on Chip, as actuated and exposed by the PYNQ-Z1 Development Environment, the present work aims to provide a demonstration of the efficacy of the heterogeneous approach to neuromorphic computing. We expose a hardware implementation of a configurable neural layer to the processing system as a software module and handle its data and parameter flow at the productivity level using Python. Results indicate a nearly negligible increase (3%) in dynamic power consumption over that consumed by the processing system alone. Further, by specifically utilizing the embedded Digital Signal Processing (DSP) and memory blocks of the ZYNQ device, we employ a relatively large percentage of these resources (13% and 11%, respectively), but consume only 5% of the Lookup Table (LUT) fabric, preserving the vast majority of resources for the implementation of other, perhaps complementary systems. Although the successfully completed heterogeneous system demonstrates that it possesses the capacity to learn, the proper training of neuromorphic systems such as this Artificial Neural Network (ANN) is a project in and of itself, and so the focus herein is more on the heterogeneous system engineered than on the prototypical application selected, which is text-independent speaker verification using Mel Frequency Cepstral Coefficients (MFCC) and log-filterbank energies as features. Fast, low power, small footprint neuromorphic systems are desirable for embedded applications that might improve the state of their art by exploiting applied artificial intelligence. Systems such as the configurable neural layer developed herein – which make use of the naturally versatile, low power, and high-performance FPGA in conjunction with a microprocessor control system – seem not only technologically viable, but well suited for handling intelligent embedded applications.

**Keywords:** Computer Engineering, Neuromorphic Computing Systems, Heterogeneous Systems, Reconfigurable Computing, Low Power Embedded Systems, Speaker Verification, Speech Processing, Feature Extraction

# Contents

# 1 About this Project

As the Information Age transcends its adolescence and attains to maturity, embedded digital devices are finding their way into all areas and aspects of our lives. Small footprint, low power, embedded systems actuate hundreds of devices and appliances that humans interface with daily, and these are only becoming more ubiquitous. Though connection to the Internet is not always necessary (or even preferable) for embedded applications, the explosive emergence of embedded systems is epitomized by the notion of the Internet of Things (IoT), which aims to evolve the Internet from a network of servers and general purpose computers into a richly diverse fabric that incorporates all of these emerging embedded systems by connecting them to the existing internetwork structure [1].

This considered, it is perhaps reasonable to speculate that some portion of applications requiring small, low power embedded solutions may benefit from systems capable of operating at level of intelligence not found in typical embedded microcontroller systems. Such applications might include, for instance, real-time natural language translation, identification of a user by intelligent perception of vocal or facial patterns, highly adaptive modulation of controlled environments, compensation for disabilities or faculties lost due to injury, and countless others [2], [3], [4]. Some might argue that many of these applications are easily addressable with an IoT approach, where data is collected by the embedded system and then sent to a server where high order analyses are performed, the results then being returned to the embedded system for utilization, thus accomplishing the intelligent task and giving the device the appearance of intelligence without it actually possessing any. This may be true, and in some situations this approach may even be preferable to an intelligent system approach, but for many other situations dependence on the Internet is a massive liability: this might be due to security concerns; reduction in power efficiency, resource utilization, and general performance as a result of task offloading and required network interface hardware; or even something so general as a decrease in reliability owing to the unnecessary complexities and dependencies of an IoT approach.

Ultimately, for many situations, the offloading of intelligent tasks to nonlocal machines in order to meet the demands of intelligent embedded applications seems more like a work-around than a valid approach, with the nonlocal computation effectively compensating for the lack of intelligence in the embedded system itself. From the perspective of this team, there is a clear and present demand for embedded systems that possess some degree of local artificial intelligence. In response to this demand, we have engineered an intelligent, small footprint, heterogeneous, low power, and versatile embedded system suitable for tasks fit to benefit from Applied Artificial Intelligence (AAI). Our system is designed to emulate certain characteristics of biological nervous systems (neuromorphism) and would be classified as a feed-forward Artificial Neural Network (ANN). Although the architecture is general enough to be adapted and applied to a myriad of tasks, the prototypical application selected for this project is embedded, offline speaker identification based on text-independent voice patterns.

The remainder of this report is organized as follows: Section 2 gives a basic overview of Artificial Neural Networks, and discusses the various common implementations thereof; Section 3 describes the functionality of specific components of the NeuraLayer system; Section 4 discusses preliminary work done during Senior Design I; Section 5 specifies the details of the design, implementation, training, and testing of our final system; Section 6 gives suggestions for future work; Section 7 discusses how the knowledge acquired of various disciplines encountered during our undergraduate education contributed to the

design of the NeuraLayer system; Section 8 notes various constraints imposed on the design process; Section 9 elaborates on the strange, experimental, and mostly standard-less nature of work with neuromorphic technology; and Section 10 offers concluding remarks.

## 2 Artificial Neural Networks

When the neuromorphic approach to Applied Artificial Intelligence (AAI) first attained technological practicality, it quickly established a new state of the art for common machine learning tasks such as classification, regression, dimensionality reduction, time series modeling, and perception [5], [6], [7]. Since then, connectionist, neuromorphic models, particularly Artificial Neural Networks (ANN), have consistently advanced the state of these arts [8], especially as the rapid evolution of technology continues to yield exponentially more powerful devices and processes, while also drastically reducing the cost of computation and information storage [1].

An ideal Artificial Neural Network is a connectionist model consisting of individual information processing nodes that are connected to each other in various configurations [8]. The purpose of these networks typically involves learning an accurate approximation of a nonlinear function that maps from one informational dimension (numerical features extracted from some phenomenon) to another informational dimension of a different character (the correct class of the input phenomenon, for instance) [9]. Often, ANNs possess a fully connected feed-forward architecture, which denotes the organization of nodes into discrete layers and the weighted connection of every node in a given layer to every node in the next layer, but with no connections between nodes in the same layer [8]. ANNs may also possess more advanced architectures, such as in the case of Recurrent Neural Networks (RNN), where nodes gain a connection to themselves across time-steps [8]. The generic architecture of a feed-forward ANN is given in Figure 1.

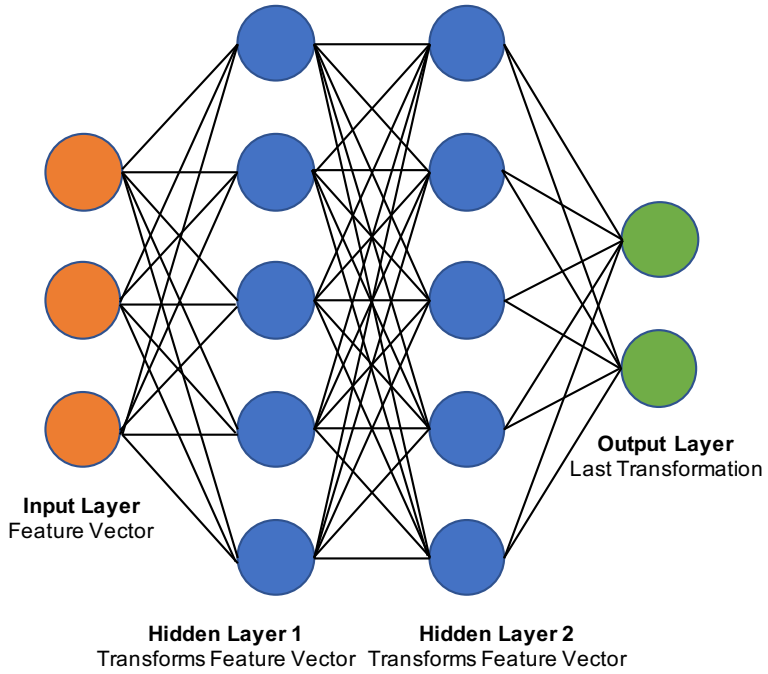


Figure 1: Generic Architecture of a Feed-Forward ANN

## 2.1 Implementing Artificial Neural Networks

Conventionally speaking, digital devices possess specific attributes that make them particularly well suited for performing certain tasks, the execution of which is intractable or impossible for humans [10]; and in a complementary fashion, the massive parallelism and complexity of biological nervous systems permits humans to easily and unconsciously perform tasks that would dumbfound a supercomputer. However, this division of capacity between human and machine is being rapidly eroded as neuromorphic technologies advance, slowly eliciting and manifesting the archetype of the silicon brain and elucidating its essential characteristics.

Without a doubt, one of the most significant of these characteristics is immense versatility. Humans, as adaptable as we may be, are more or less constrained to the form that we inherently possess. On the other hand, artificial neural systems are constrained only by the cutting edge of technological innovation. A whole host of devices possess the capacity to implement neuromorphic systems, each of them offering various advantages and disadvantages. Generally, there are four major categories of device that are currently used to implement ANNs: these are conventional multicore processors, general purpose Graphical Processing Units (GPGPU), Application Specific Integrated Circuits (ASIC), and reconfigurable logic devices such as Field Programmable Gate Arrays (FPGA) [2].

Before briefly discussing the merits and demerits of these devices, it is important to emphasize the massively parallel nature of neural networks. In effect, each layer of an ANN performs a nonlinear transformation on an input vector, and then passes the new vector forward as the input to the next layer; but as discussed, each of the nodes within a given layer performs a unique transformation on the same input vector, and are thus completely independent of each other's computations [8]. From this notion it is clear that, in an idea scenario, every single node in a given layer should perform its computations in parallel, as there is no need for any node to wait for any another node in the same layer to complete its operation. This observation should be specifically noted, as it logically follows that technologies that provide natural support for massive parallelism are likely to prove better suited for implementing neuromorphic systems than technologies that are primarily serial in their operation.

### 2.1.1 CPU Implementations of ANNs

Typical Central Processing Units (CPU) may be useful for quickly implementing or simulating small to medium sized neural networks, but the architecture of these devices is either serial or minimally parallel, and they are specifically designed to handle general applications [10], whereas ANNs require highly specific computations and are ideally, as discussed, massively parallel systems. As a result of these discrepancies, traditional CPUs are of very limited utility when it comes to implementing large networks or networks that must achieve high performance while consuming little power.

### 2.1.2 GPGPU Implementations of ANNs

An alternative hardware platform for implementing ANNs is the general purpose GPU. GPUs share many similarities with CPUs, but instead of having one or several full-feature processing cores, GPUs are composed of many (perhaps hundreds, or even thousands) of smaller, parallel, less complex processing cores [8]. Considering the parallel nature of ideal neural networks, the GPU sounds like a powerful, promising platform; and indeed, general

purpose GPUs are perhaps the most frequently used hardware for implementing ANNs for applications requiring high performance and extremely dense datasets [2]. However, all of those independent processors with floating-point capabilities constrains one nearly inevitable characteristic of GPUs: immense power consumption, often in the range of hundreds of Watts [8]. In a high performance computing (HPC) environment where power consumption is often regarded as less important than performance, this is not an intractable concern, and the raw power of GPGPUs often makes them the go-to technology for HPC Artificial Intelligence (AI) applications.

### 2.1.3 ASICs for Embedded Environments

Ultimately, though the high power consumption of GPGPUs is merely a small nuisance in an HPC environment, it is absolutely and unequivocally insuperable in an embedded environment, where the general engineering objectives are to achieve the lowest power consumption, highest performance, and lowest resource utilization possible.

Generally speaking, ASICs are considered the gold standard for embedded applications [11]. This is due to the fact that application specific circuits naturally include only the components that are absolutely necessary for the application, thus eliminating waste from unused and unnecessary components. Additionally, modern Complementary Metal Oxide Semiconductor (CMOS) manufacturing processes are ideal for producing extremely small, low power, and high performance integrated circuits [11]. Indeed, there is much speculation among engineers that ASICs will be the final form of the embedded neuromorphic system, as the advantages that they offer over GPUs and CPUs are so attractive and significant [2]. There is, however, one key attribute of ASICs that is simultaneously their greatest strength for general embedded applications as well as their greatest weakness for specifically implementing neuromorphic systems: that is, their static, hard-wired nature. By establishing a static circuit in the silicon with CMOS manufacturing techniques, the immense power, area, and performance benefits of ASICs are obtained; however, ANNs are, ideally, highly adaptable and reconfigurable systems, and this is where the ASIC for embedded artificial intelligence applications hits a major obstacle. How can a static device be ideal for implementing an extremely dynamic and adaptive system?

### 2.1.4 FPGAs for Embedded AAI

Field Programmable Gate Arrays (FPGA) are integrated circuits that effectively provide a reconfigurable hardware canvas that is capable of emulating the behavior of any valid digital system [12]. The primary resource of FPGAs is the configurable logic block, which uses lookup tables (LUT), flip-flops, multiplexers, and switch matrices in order to perform a particular logic function; these logic blocks are configured by generating a bitstream from a behavioral hardware description and then loading it into the fabric [12]. Additionally, most modern FPGAs also include high performance digital signal processing (DSP) and memory blocks (BRAM) which are embedded in the configurable fabric: this introduces an aspect of heterogeneity, and a significant increase in performance and versatility is achievable by exploiting these embedded resources for suitable applications [12]. The generic architecture of a modern FPGA is given in Figure 2.

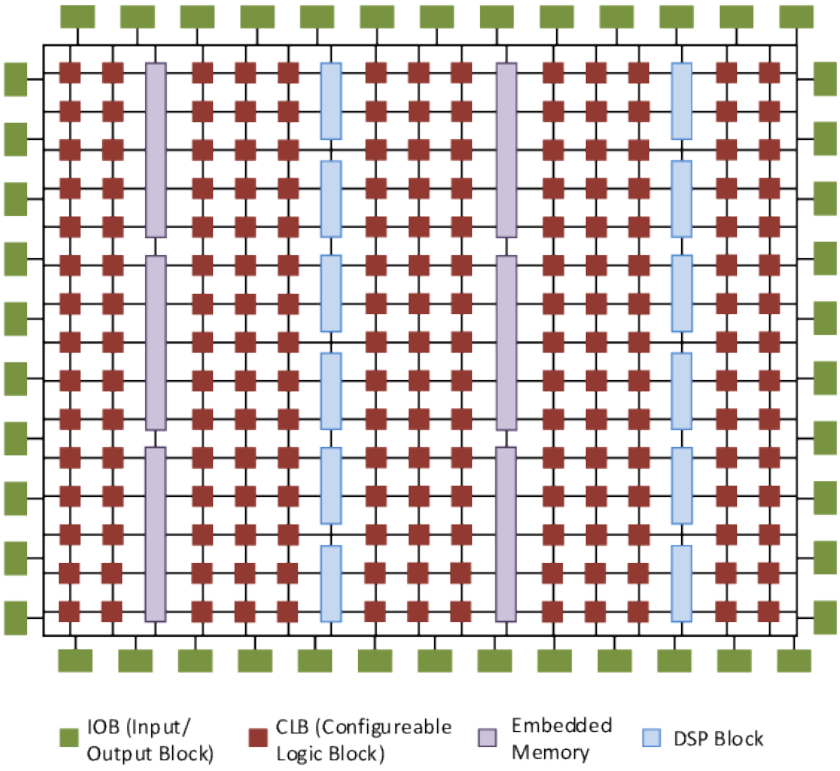


Figure 2: Generic Architecture of a Xilinx FPGA [13]

Interestingly, the class of tasks for which DSP blocks are most useful is that encompassing fast, parallel arithmetic operations on data held locally in BRAM or distributed LUT memory [14]; and coincidentally, this is the exact variety of operation that an artificial neuron must perform—specifically, the multiply-accumulate operation [8]. This makes the modern FPGA ideal for implementing the independent artificial neurons that compose ideal ANNs; and further, FPGAs, though less fast and power efficient than static ASICs, are still very fast and low power, all while offering a completely configurable and highly versatile hardware fabric, which gives users the capacity to load and update networks at will, and to grow and evolve an integrated neural network/peripheral system to meet dynamic demands. It seems reasonable to opine that this capacity, offered by FPGA devices, is so immensely desirable as to overcome the slight power reduction and performance gains of ASICs, and to render FPGAs a preferable technology for implementing dynamic, low power, high performance ANNs for embedded applications.

### 2.1.5 Heterogeneous Approach to Neuromorphic Computing

As discussed, programmable logic devices such as FPGAs offer ideal platforms for the implementation of fast, low power ANNs for embedded applications. However, for practical systems, it will be necessary to control the general operation of the ANN, provide a user interface, serve data to the ANN, and perform any necessary data preprocessing. Evidently, these are rather general operations - not specific like those required by the ANN hardware - and it is likely that they would be more effectively handled by a CPU than by systems implemented on the FPGA fabric. As such, it seems that an ideal hardware platform for the implementation and handling of an embedded ANN would tightly integrate a CPU with FPGA fabric. It was this hypothesis that led this design team to select a board featuring the Xilinx ZYNQ-7020 All Programmable System on Chip device

as our development platform. This device is one of a family of chips that integrates a dual core ARM Cortex-A9 processor with an FPGA on the same piece of silicon [14]; its block diagram is given in Figure 3. It was on this all programmable platform that this team engineered a hardware implementation of a Configurable Neural Layer (CNL, or NeuraLayer) capable of emulating the behavior of a multilayer ANN, and developed extensive Python software to control the hardware module, preprocess data, handle the data and parameter flow to and from the CNL, and to train and test the network. The NeuraLayer system provides a prototypical solution to the exigency of small, low power systems for embedded artificial intelligence applications.

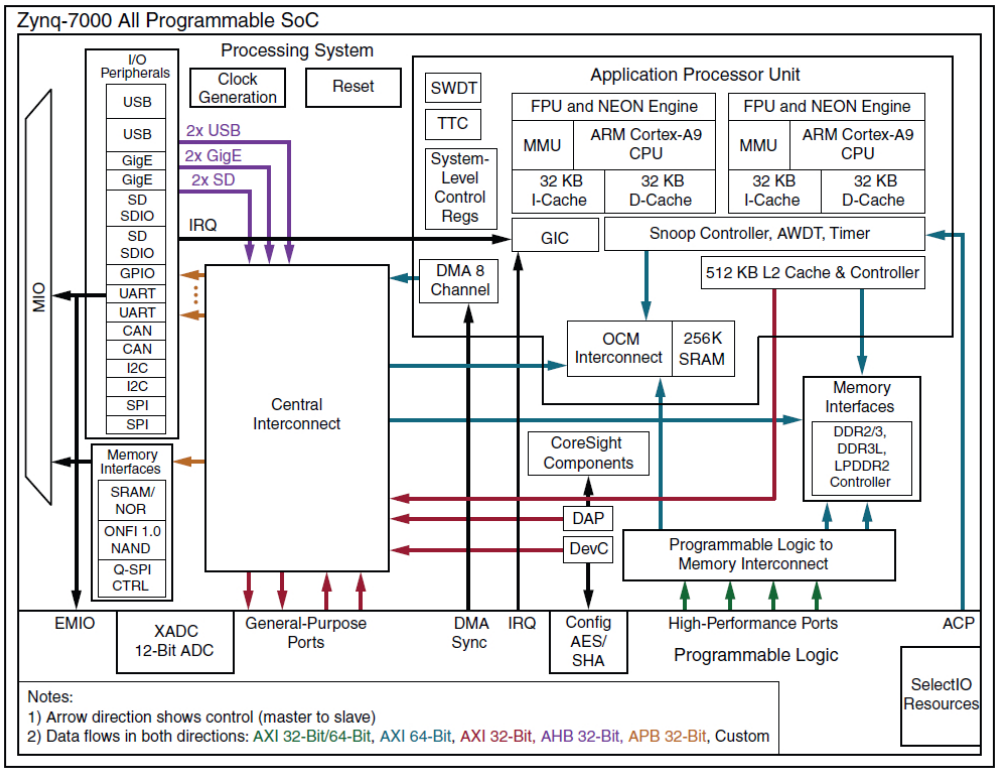


Figure 3: Block Diagram of the ZYNQ 7 Series Device [14]

# 3 Functional Description

Figure 4 gives a high level view of the primary components of our system. Highlighted in red is the Python Environment, which consists of the Processing System (PS) of the ZYNQ chip, the Dual Data Rate (DDR) memory, and the Jupyter Notebook Environment where all of our software is written and executed. This software serves the function of handling the data and parameter flow of the hardware. In blue is the Custom Hardware Environment, which consists of the ZYNQ Programmable Logic (PL), on which is implemented the NeuraLayer hierarchical module. The NeuraLayer module performs all of the feed-forward arithmetic calculations of the ANN. In between these two environments is the Advanced Extensible Interface (AXI), which is used to facilitate the interaction of the two distinct systems [15].

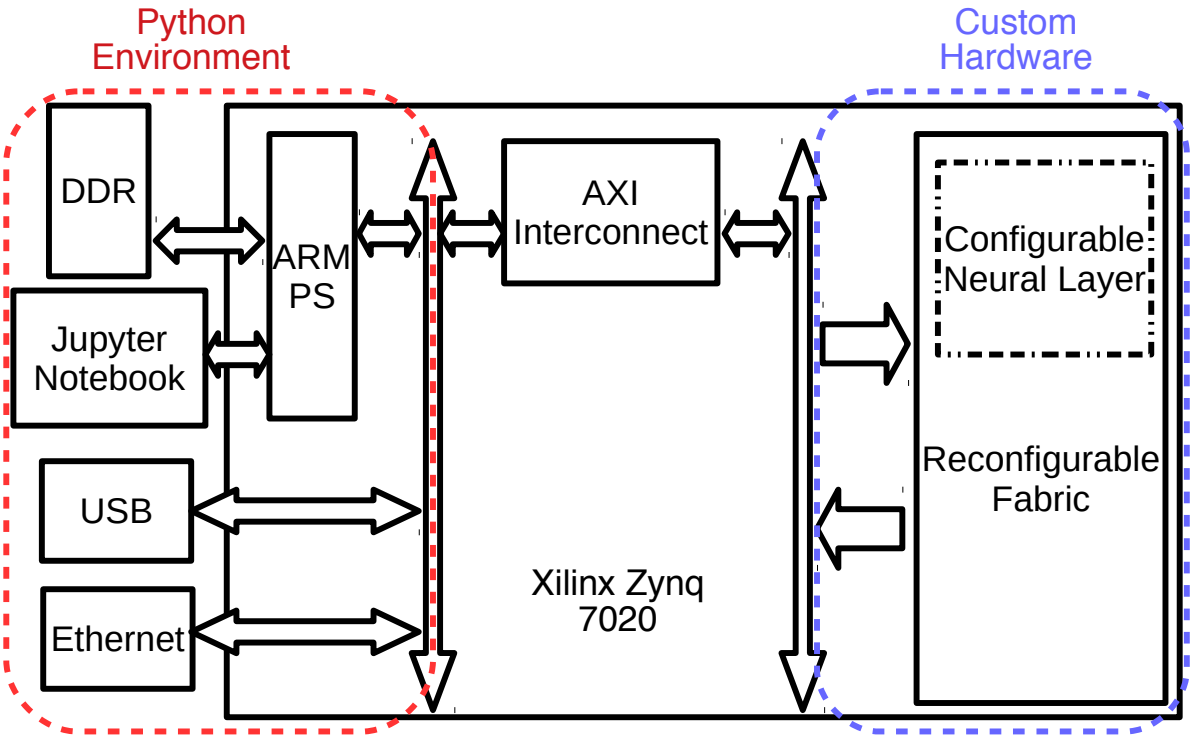


Figure 4: High Level View of Heterogeneous System

# 4 Preliminary Design

Due to the somewhat unique nature of our project, we spent our time in Senior Design I (SD1) not creating a prototype for the system discussed in this document, but rather we designed a precursor project. This project served as a learning experience where we became familiar with all of the tools that we would have to use in this project: that is, Jupyter Notebooks, Xilinx Vivado and the PYNQ Environment. The SD1 project helped us to get acquainted with these tools and made the design and implementation of this project much easier. The name of our Senior Design I project was: System on Chip Implementation of an Adaptive Linear Element for Linear Data Classification. In this project we designed a much simpler ANN which could classify the Iris Dataset, a linear dataset that contains numerical attributes of different types of flowers. To us, this project was very much so a success; we developed a very similar, much simpler system to help us on our path to design our final system. The paper written about that system can be found in Appendix 13.3.

# 5 The NeuraLayer System

As the purpose of our preliminary project was to build familiarity with the complicated tools, design flow, and environments required for completing our final project, the difference in depth and magnitude between our preliminary design and our final design is somewhat astounding, though the overall character of the two systems share core similarities, and this team considers both to be successfully implemented independent systems. As discussed earlier, the primary components of our final system are the software writ-

ten and executed in the Jupyter Notebook Environment that is served by the processor of the ZYNQ chip, the NeuraLayer hierarchical hardware module that resides on the ZYNQ FPGA, and the Advanced Extensible Interface (AXI) that is used to integrate and harmonize the two distinct environments. The design flow surrounding each of these components will be discussed in this section, but as the system design choices specific to this final prototype are derived from and dependent on its application and the data that it will need to process, it makes sense to address these aspects first.

## 5.1 Data

Given the nature of this type of technology, a vast amount of data is needed for our system to be able to learn and adjust its internal parameters. The data that we used comes from an open source dataset provided by Sphinx Speech Group at Carnegie Mellon University. Supplementary audio data of our voices was also recorded.

### 5.1.1 AN4 Dataset

The AN4 dataset is a collection of audio files that the Carnegie Mellon Robust Speech Recognition Group recorded in 1991 [16]. This was the main source of our audio data that was not recorded by us. The dataset includes both male and female speakers who recorded utterances, words, numbers and sentences. This dataset includes a total of 1,058 raw audio files, 941 for training and 117 for testing. Unfortunately the AN4 dataset was recorded in the *.raw* format which is an uncompressed audio file format which is not too different from the *.wav* format, aside from the fact that the Python library we were using called for us to .wav audio files. Due to this we had to convert each file to the *.wav* format. Instead of converting each one of the 1,058 files one at a time, we used a audio editing command line tool called Sound eXchange (SoX). This tool helped batch convert all of the files in the dataset so that we could work with them in Python.

### 5.1.2 Additional Voice Data

We recorded a total of eighty audio files for both Anthony and Tom, we recorded these samples using a Blue Yeti USB microphone and the open source audio editor Audacity. When creating these samples we used words, phrases and numbers taken mostly from the Dhammapada, or Sayings of the Buddha—this book was selected for its abundance of general, commonly used, and universal words and phrases.

## 5.2 Audio Data Preprocessing

Before we attempted to train our machine, it first had to be able to understand the data that we were giving it. We converted our audio data to a numerical representation that our system could interpret. For this, we used a procedure called Mel Frequency Cepstral Analysis (MFCA) which is a type of sound processing specifically used in speech/speaker recognition.

### 5.2.1 Mel Frequency Cepstral Analysis

The mel scale is a non-linear scale that, in short, is a representation of how the human cochlea (the part of the ear that responds to sound vibrations) understands sounds of a

different frequencies. Shown in Figure 5 is a visualization of what the mel scale looks like when applied to an audio sample [17].

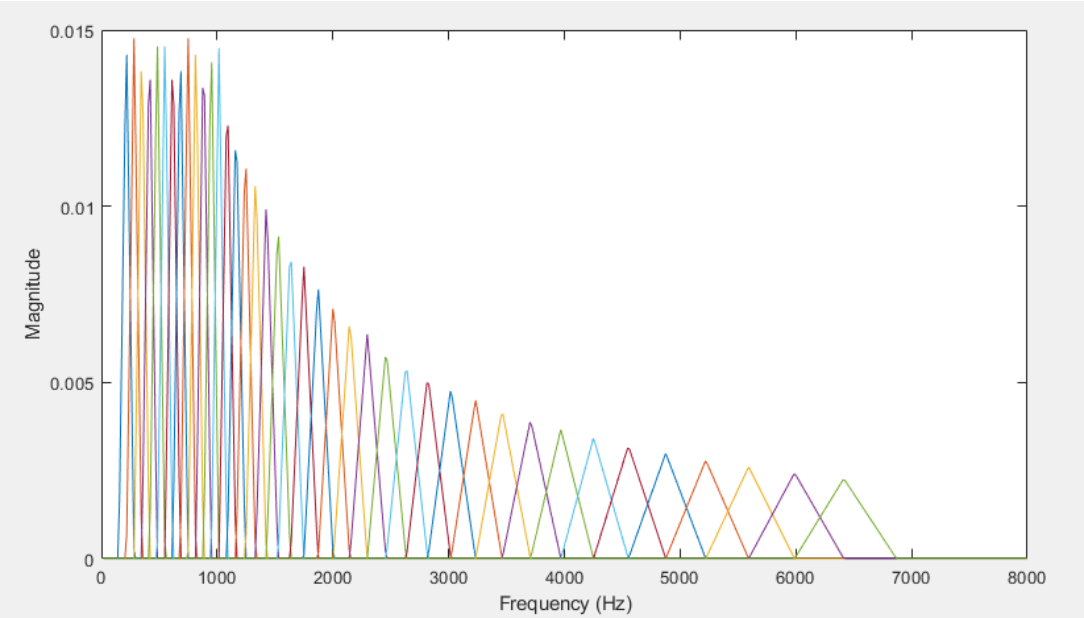


Figure 5: Visualization of a typical Mel Filter-bank

The scale makes the data in the lower frequency range, zero to about 10KHz, most important because it is where human vocal range resides, thus making the frequencies outside of this range less important. The MFCA of an audio file is a representation of the short term energy in a frame of the sample, it is based on the linear cosine transform on the non-linear mel scale. From MFCA we yield a small number of coefficients based on the cochlea, we call these coefficients Mel Frequency Cepstral Coefficients (MFCC). Although we do not compute MFCC's by hand, it is still important to understand the high-level sequence of steps [18]:

1. Compute the discrete Fourier transform for the selected window (small piece of audio signal in this case our window length is .03s)

2. Map the Fourier transform to the mel scale using the triangular overlapping windows shown in Figure 5.

3. Take the logarithm of each of the mel frequencies.

4. Compute the discrete cosine transform

5. MFCC's are the amplitudes of the resulting spectrum.

Luckily, we use a Python library to help us complete these computations, specifically Python Speech Features created by James Lyons [19]. To use this library we specify which parameters we want to change from the default settings. In our case we had to change both our window length and number of Fast Fourier Transforms (FFT), to .03 and 2048 respectively, in order to not clip our data.

### 5.2.2 Preprocessing System

In order to handle all the audio files and their processing, we designed a system to extract MFCC's for each audio file and output a spreadsheet. This system was designed using Python in the Jupyter notebook environment, some important libraries that we used are as follows: numPy, pandas, python-speech-features, scipy and scipy.io.wavfile. This system parses through each of audio samples and outputs a spreadsheet with columns for the filename, MFCCs one through thirteen, the total energy per frame which replaces the first MFCC, and the name of the speaker (ant, tom, or other). An example of this table can be seen in Figure 6.

| File Name | Total Energy | MFCC2 | MFCC3 | MFCC4 | MFCC5 | MFCC6 | MFCC7 | MFCC8 | MFCC9 | MFCC10 | MFCC11 | MFCC12 | MFCC13 | Speaker |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| an1-mblw- | 20.82238 | -42.6467 | 20.02282 | -9.21333 | 0.885417 | 0.528202 | -14.4078 | 9.103135 | -5.40649 | 5.902551 | -3.68764 | -6.18446 | 8.525695 | other |
| an1-mblw- | 16.46894 | 2.758835 | -2.08353 | 11.79165 | -4.30452 | -27.6926 | -20.0544 | -5.25717 | 14.03591 | -7.89584 | 1.53031 | 5.08427 | 4.600304 | other |
| an1-mblw- | 9.041134 | -14.5593 | -4.14765 | -7.2588 | -8.57559 | -6.5604 | -3.659 | -12.0858 | -7.50095 | 6.697695 | -1.33395 | 4.947655 | 1.888753 | other |
| an1-mblw- | 17.08024 | -2.6276 | -7.05019 | 27.49374 | -1.80174 | -35.2415 | -28.6433 | -12.3881 | 6.927579 | -26.4156 | -0.55801 | 7.728919 | -5.05616 | other |
| an1-mblw- | 11.01772 | -19.2932 | -3.45183 | -2.42721 | 3.55892 | -21.369 | -3.98693 | -0.53946 | 1.475134 | -20.1075 | 13.83166 | -21.3086 | 2.73024 | other |
| an1-mblw- | 16.13378 | 8.029366 | -21.8068 | 0.816774 | -21.5965 | -1.67458 | 19.55354 | -26.2389 | 11.59582 | 16.62449 | -1.78373 | -19.4648 | -20.8541 | other |
| an1-mblw- | 13.229 | 17.17257 | 19.07421 | 1.851618 | -21.402 | -29.8855 | 10.83285 | -16.7432 | 0.696185 | -19.5101 | -1.57297 | -8.66139 | 3.517758 | other |
| an1-mblw- | 13.40955 | -6.74218 | -14.8695 | 7.076794 | -18.034 | 15.82381 | -14.7146 | 1.07172 | 18.85441 | 16.79048 | -8.30441 | -9.09639 | 4.445482 | other |
| an1-mblw- | 8.643906 | -7.30776 | -4.29367 | -1.66255 | -2.76421 | 2.242433 | -5.78909 | -8.63187 | -2.14361 | -2.13766 | 8.78505 | 4.494359 | -6.37617 | other |
| an1-mblw- | 19.24683 | -55.1654 | 14.16709 | -3.66234 | -6.53821 | 12.8245 | -14.623 | 4.118575 | 6.33522 | -6.79228 | -15.0512 | -12.522 | 16.21918 | other |
| an1-mblw- | 10.35051 | -21.1879 | -0.19908 | -1.96485 | -7.37832 | -18.0247 | -5.09767 | -8.42723 | -11.8503 | -8.87147 | 0.862635 | -11.9021 | -7.0323 | other |
| an1-mblw- | 19.80475 | -40.1154 | 9.761283 | 0.920411 | -10.4302 | 19.65358 | -16.6577 | -3.28627 | 9.9692 | -4.23615 | -4.31907 | -12.2077 | 6.915585 | other |
| an1-mblw- | 10.68354 | -18.7943 | 4.356463 | -8.53892 | -5.61325 | -20.247 | -3.68485 | -17.6353 | -4.62319 | -17.1301 | 11.11638 | -14.4296 | 4.668842 | other |
| an1-mblw- | 19.21047 | 12.31273 | -35.7521 | -5.41797 | -17.35 | -10.9872 | 17.2383 | -18.2161 | 13.41917 | 24.05084 | -10.6551 | -10.3481 | -38.7411 | other |
| an1-mblw- | 13.27623 | 19.54221 | 15.97409 | 2.613934 | -31.9687 | -18.1012 | 6.075636 | -8.73551 | -10.7077 | -10.2956 | -12.271 | -1.49799 | 5.882976 | other |
| an1-mblw- | 17.40703 | 5.118079 | -12.4256 | 10.85678 | -7.96695 | -33.189 | -15.9083 | -11.7029 | 21.70128 | -5.83342 | -1.85767 | -3.53217 | -7.96934 | other |
| an1-mblw- | 19.84424 | -1.43094 | -26.5731 | 19.96121 | -18.7696 | -24.5766 | -18.5459 | 3.479123 | 30.04204 | -2.06831 | -2.62176 | 8.151311 | -12.627 | other |
| an1-mblw- | 15.12993 | -3.98891 | -3.13792 | 18.05972 | -12.6719 | -34.3441 | 7.796804 | -7.61478 | 8.591375 | 23.30969 | -3.33912 | -13.821 | -10.5138 | other |
| an1-mblw- | 16.11083 | -12.1059 | 13.99443 | 15.86226 | -8.79996 | 3.535462 | -12.019 | 1.537231 | -3.50078 | -18.0339 | 1.498148 | -1.76101 | 4.865851 | other |
| an1-mblw- | 17.85452 | -5.96773 | 5.658046 | -18.9261 | -4.18479 | -16.9576 | -12.9843 | -12.4901 | 8.059623 | 27.67717 | 9.528346 | -20.2574 | -1.90611 | other |
| an10-fjmd- | 16.78345 | 0.318749 | -8.76812 | 17.51515 | -18.1243 | -12.7235 | -22.8327 | 11.83852 | -9.10936 | -31.0698 | -14.3809 | -15.3687 | 6.803589 | other |
| an10-fjmd- | 19.83573 | -33.6154 | 2.399259 | -6.97864 | -24.5302 | -13.1652 | -1.42799 | -0.98166 | -5.6257 | -16.2683 | 14.18051 | -9.3726 | -8.41393 | other |
| an10-fjmd- | 9.500918 | -8.4994 | -7.12413 | -1.01836 | -8.14452 | -5.45661 | -2.20566 | -7.70099 | -4.43737 | -13.0355 | 3.408447 | -5.48169 | 2.942303 | other |
| an10-fjmd- | 15.15631 | 11.02307 | 0.681181 | 6.285424 | -31.2959 | 4.177349 | -15.2192 | -45.2479 | -1.62623 | -9.86123 | 30.55255 | -29.4103 | -3.05059 | other |
| an10-fjmd- | 9.49359 | -8.69116 | -6.94548 | -7.4668 | -10.3515 | -9.89837 | -10.984 | -16.8752 | -12.2656 | -4.83954 | 3.562218 | -13.934 | -8.04417 | other |
| an10-fjmd- | 17.27313 | 1.215754 | -11.5789 | 13.93617 | -15.0059 | -11.5378 | -17.1962 | 10.57699 | -5.68054 | -31.8416 | -15.0146 | -14.8358 | 12.75627 | other |
| an10-fjmd- | 16.54284 | 6.46137 | -2.23246 | 23.93432 | -16.1575 | -10.3007 | -27.8383 | -33.3794 | 17.12554 | -15.7404 | 5.790873 | -28.1465 | 20.23688 | other |
| an10-fjmd- | 9.312725 | -12.0771 | -7.60411 | -2.9195 | -8.2629 | -7.0749 | -3.11594 | -10.3729 | -10.1699 | 2.872699 | 7.454545 | -1.27246 | -2.63277 | other |
| an10-fjmd- | 12.99222 | 2.515369 | 15.6476 | 27.06987 | 3.018563 | -15.0237 | -21.2844 | 4.714658 | -36.4307 | -11.6216 | -0.14293 | -15.0444 | 6.649474 | other |
| an10-fjmd- | 14.99885 | 9.7567 | 14.19295 | 24.72931 | -8.92042 | -18.1057 | -9.62006 | -3.13539 | -36.7845 | 3.696912 | -3.86623 | -21.0278 | 16.73398 | other |
| an10-fjmd- | 15.5541 | 10.48791 | 4.411514 | 5.378914 | -23.5402 | 9.270791 | -11.7533 | -49.4526 | -0.21707 | -10.1071 | 33.32113 | -27.8443 | -4.08512 | other |
| an10-fjmd- | 15.18694 | 3.275364 | -8.07971 | 30.56681 | -19.6883 | -14.8875 | -19.1534 | -3.99779 | -17.9555 | -29.7105 | -14.445 | 0.018603 | -4.28559 | other |
| an10-fjmd- | 18.41356 | -11.1519 | 10.71091 | 42.58157 | -44.55 | -12.9615 | -11.0219 | -19.9776 | -20.9839 | -4.18692 | 10.59794 | -28.7282 | 11.59019 | other |
| an10-fjmd- | 18.11811 | -3.18021 | -3.11322 | 37.92821 | -44.7258 | -20.6955 | -15.2585 | -5.66296 | -14.5364 | -1.28586 | 17.05756 | -14.9904 | 11.47845 | other |
| an10-fjmd- | 17.61372 | -1.43912 | 10.70812 | 31.45733 | -45.8166 | -17.9097 | -17.087 | -15.3987 | -24.825 | 7.167729 | 10.05154 | -18.612 | 22.69133 | other |
| an10-fjmd- | 9.509293 | -8.58547 | -4.75704 | -3.73004 | -9.76829 | -9.17896 | -10.9225 | -22.1183 | 0.674936 | -5.75749 | 5.10514 | -9.06192 | 0.05631 | other |

Figure 6: An Example Output from our Pre-processing System

It's important to note that each file yields many MFCC vectors, as seen in Figure 6, one file (in this case an1-mblw.wav) yields around twenty vectors. Since this process would generate so many vectors per file, we had to find a minimize this number. We ended up selecting random mfcc vectors per audio files. We found out later that this was a bad approach and we decided to change it.

### 5.2.3 Data Clipping

Data clipping is something that we realized was a big problem with our preprocessing system after we started training. We originally had created MFCC vectors for each of the audio samples, these vectors had a width of 13 and a very long length– equal to the amount of frames we had collected. Since, by default, a frame is analyzed every 25ms and each frame overlaps the next by 10ms, there was a lot of data for a just one two to five second audio clip. Naturally we thought that we had to limit the amount of data

we were producing from MFCA because our system takes time to analyze said data, and without any modification our preprocessing system yielding massive amounts of data. Our first MFCA spreadsheet had about 200,000 rows, a file like this would take about twenty hours to train our system on, an unacceptable amount to be able to test our systems functionality on. Since one audio file would yield many vectors we decided to take a different approach in preprocessing. Instead of generating MFCCs for each and every file we decided to minimize our dataset to be much smaller and instead keep all MFCC vectors instead of picking random ones. This had the effect of producing more accurate training and testing results.

## 5.3 Network Handling in Jupyter

One of the primary advantages offered by the PYNQ Environment and corresponding packages is the capacity to load hardware modules onto the ZYNQ Programmable Logic (PL) simply by specifying its bitstream and Tool Command Language (Tcl) block diagram file [20]. The Jupyter code cell used to load the NeuraLayer bitstream to the ZYNQ PL is given in Figure 7. Additionally, these hardware modules can be wrapped in Memory Mapped Input/Output (MMIO) objects using the PYNQ MMIO class; when MMIO address calls are wrapped in user generated driver functions, this has the effect of exposing the hardware module to the end user in a manner that is identical to how a software module is exposed [21]. Essentially, the PYNQ Environment permits the productivity level synthesis of both hardware and software, effectively dissolving the typically drastic distinctions between these two varieties of system. The entire Jupyter Notebook is given in Appendix 13.1.

```
In [2]: neuralNetwork = Overlay("/home/xilinx/jupyter_notebooks/NeuralNetwork/NeuraLayer/NeuraLayer.bit")
        neuralNetwork.download()
        net = MMIO(0x43C00000,0x10000)
```

Figure 7: Instantiation and MMIO Wrapping of NeuraLayer Module

### 5.3.1 Data Loading and Handling

The data preprocessing discussed in the previous section produced Excel spreadsheets containing thousands of standardized data samples: each of these samples consists of 12 numerical features, one class label, and one file name attribute. As such, the first step in either training or testing the NeuraLayer module after loading it onto the board is to load the data from the Excel sheet into columnar list objects: this is done using the Dataframe from the Pandas Python package. After loading the data, it is scaled by a factor of 100 and then rounded. This is done for two reasons: firstly, the hardware can only accept integer data, and scaling by 100 preserves several decimal places of precision and also serves to better fit the data to the 12 bit integer range selected for the hardware module (this will be discussed further in Section 4.4.4); secondly, by preserving the most important decimal places of the data and then truncating the remainder, we eliminate much noise and yield more general samples, which we hypothesize is good for reducing overfitting, a common problem during neural network training where the model becomes excellent at classifying training data but cannot generalize well to new data [6]. After this step, the data is ready to be randomly selected from the columnar lists by mutual indexing, compiled into a feature vector, and submitted to the hardware module for training or testing.

### 5.3.2 Software Handling of Hardware Module

The PYNQ Environment may provide the general means for connecting hardware to software, but in order to synthesize the hardware and software at the level of this particular application, it is necessary to develop functions capable of translating and formatting the various types of integer input data so that it fits the highly specific arrangement that the hardware module expects to receive its data in. Among these functions are some that return the properly formatted binary or two's complement representation of an integer, some that fuse together binary strings and convert them to an integer representation, and also a whole host of functions that expose various hardware ports through wrapping calls to the MMIO inside of typical Python functions. Many of these functions are simple, but some require a complex coordination of various sub functions.

### 5.3.3 Parameter Skeleton

One of the most important aspects of the software side of our heterogeneous system is the Parameter Skeleton: in effect, this is the software component of the ANN itself, and it possesses objects representing every node in every layer, and stores parameters for each of them. This allows parameters to be specified from a text file, arranged properly, and then quickly and efficiently loaded into the NeuraLayer hardware module via the facilitative functions. Additionally, the node and network objects of the Parameter Skeleton also possess functions for implementing the backpropagation training algorithm, which calculates and executes updates to various network weighting parameters [22].

## 5.4 NeuraLayer Hierarchical Hardware System

The heart of the entire heterogeneous ANN system is the NeuraLayer hardware module. NeuraLayer is a Configurable Neural Layer (CNL) that is capable of emulating the behavior of a multilayer ANN by adapting its internal parameters as directed by a Layer Select control signal on the top-level module. It is designed with a fixed network size, but this size can be altered by making minimal changes to the Verilog Hardware Description Language (HDL) files. Engineering this hierarchical module was certainly the most challenging and time consuming portion of this Senior Design project. For reference, the conceptual block diagram of the NeuraLayer system is given in Figure 8.

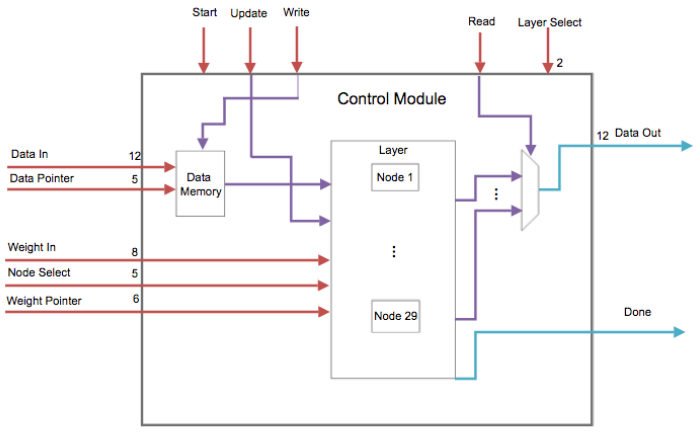


Figure 8: NeuraLayer Hierarchical Block Diagram

It is important to discuss the discrete ranges selected for this prototypical iteration of the NeuraLayer system. When designing the hardware, we had to assume a minimalist mindset, as there was no way to accurately predict how much of the ZYNQ PL the module would consume. In order to ensure that the design would fit on the chip, we chose to minimize the number of nodes in the widest layer (29, 30 with bias), and to constrain the data/weight ranges as much as possible. For data, we chose a 12-bit signed precision, giving us an integer range from -2047 to +2047, and for weighting parameters, we chose an 8-bit signed precision, yielding a range of -127 to +127.

### 5.4.1 Node Module

Our preliminary Senior Design I project required only one single artificial neuron, and one with a linear identify activation function at that; and further, as the hardware for this node was described using C in Vivado High Level Synthesis, there was significant abstraction from the Register Transfer Level (RTL) of the hardware's functionality. As such, the first step in the implementation of our final design was to design a single configurable neural node, but this time the design had to be done in Verilog ourselves, and since the node is the principal active component of the ANN, this is also the module that is most impactful on the performance of the full design. The primary component of the node module is a clock synchronous state machine that computes and accumulates a new weight scaled data value at each time step, and then performs the activation function on the net output once the calculation pointer reaches the maximum number of inputs. This final node output is stored in a data out register and exposed as a wire bus in the top-level module so that it can be read from MMIO by the processor. The activation function of the nodes in the NeuraLayer module is the Rectified Linear Unit (ReLU) activation function, which was chosen because it is extremely simple and easy to compute in hardware. The ReLU passes an activation equivalent to the net output if the net output exceeds 0, or passes 0 if the net output is negative [23]. However, as discussed previously, our data range is constrained to 12 bits, so the maximum value that can be passed from the output of one node to the input of a node in the next layer is +2047. As such, we amend the ReLU function to rectify in the positive direction as well as in the negative, behaving as an identify function if the net output is between 0 and +2047, but saturating to one of these extremes if the net output is larger than +2047 or less than 0.

### 5.4.2 Layer Module

The layer module is one level up on the hierarchy from the node module. All node modules are instantiated inside of the layer module, as are their respective weight memories; the layer module also stores new weights to the correct memory block and facilitates the flow of all data and weights to and from individual node modules. Memory updates are performed by selecting the proper memory block using a node addressable scheme, and the data flow is governed by a state machine.

### 5.4.3 Control Module

The control module is the top-level module of the NeuraLayer hardware system. It receives all of its inputs from the ZYNQ PS and returns all outputs. Its primary components are a machine to collect input data, a state machine to serve this data to the layer module

for dispersal to the nodes, and a state machine to cycle through the node output busses, exposing them sequentially to the data out port for extraction by the PS.

# 6 Results

The primary contribution of this project is theoretical and architectural, so in many respects the most significant result is the heterogeneous system itself. Nevertheless, this prototypical instance of the NeuraLayer system was designed for a specific task, and so we are able to report results for this particular instance of the design.

## 6.1 Resource Utilization

As discussed earlier, one of the more important characteristics of an ideal embedded system is low resource utilization: smaller devices are more versatile, typically consume less power, and leave more room for other system components [11], [1]. As such, a primary objective of this team was to minimize FPGA area consumption, as a small hardware footprint means that a smaller chip could be employed for the task. Additionally, at the beginning of the hardware design process, it became clear to us that it was impossible to predict how large the module would be when it was complete, so to ensure that it would fit onto the chip, we decided to use resources even more sparingly, minimizing whatever seemed like excess. For example, the decision to use only 29 nodes in the widest layer, and to use a rather narrow 12-bit and 8-bit data range (for inputs and weights, respectively) was motivated by a mindset of reduction and conservation. Ultimately, we more than met our conservative objective, with our design only occupying an extremely small footprint on the programmable logic: we consume 5-6% of the LUT fabric, 11% of the embedded memory blocks, and 13% of the embedded DSP slices. The precise resource utilization is given in Table 1.

Table 1: Resource Utilization

| Resource | Utilized | Total |
|---|---|---|
| LUT | 2,660 (5 %) | 53,200 |
| DSP | 29 (13.18 %) | 220 |
| BRAM | 15 (10.71 %) | 140 |

These results reveal several important observations: firstly, even a significantly expanded iteration of the NeuraLayer module would fit comfortably on the ZYNQ 7020 chip; secondly, with such little utilization, the vast majority of resources are preserved for the implementation of other, perhaps complementary/peripheral systems; and lastly, it is now clear that a single ZYNQ chip could support several different NeuraLayer modules simultaneously—this has significant theoretical implications for future work. For purposes of visualization, a labeled, post-implementation schematic view of the NeuraLayer module is given in Figure 9, and an internal chip view of the fully placed and routed hardware module is given in Figure 10, where the dark blue fabric is the programmable logic, the black rectangle is the processing system and other chip components, and the footprint of the NeuraLayer module is indicated by light blue highlighting.

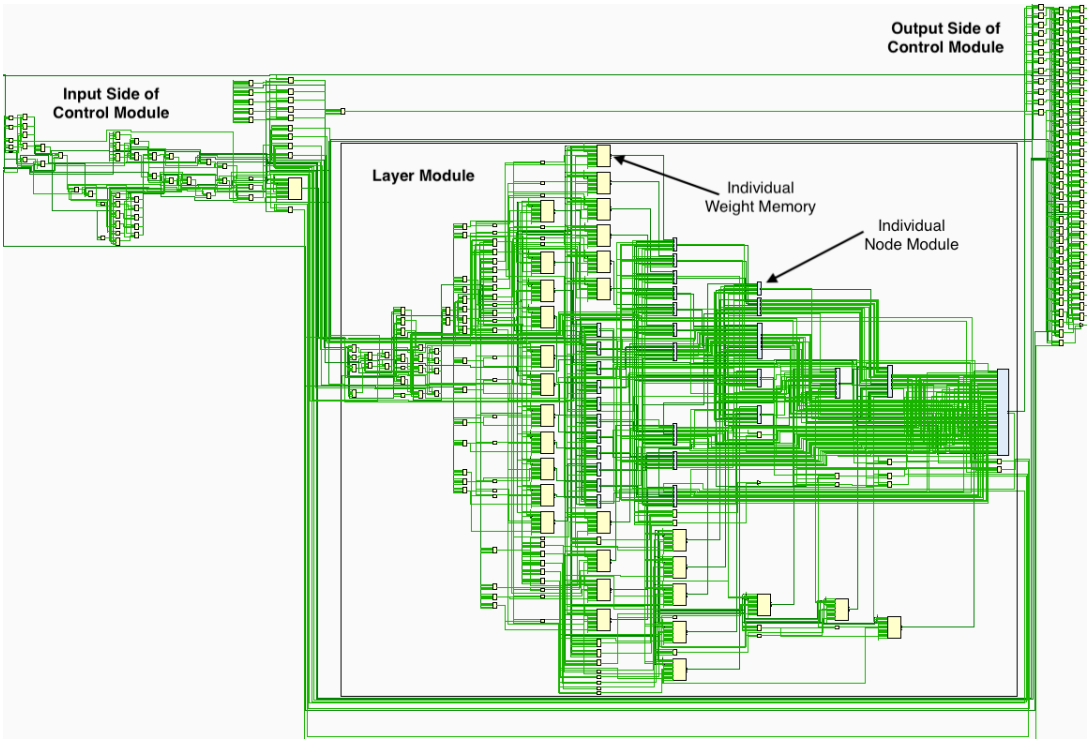


Figure 9: Schematic View of NeuraLayer Module

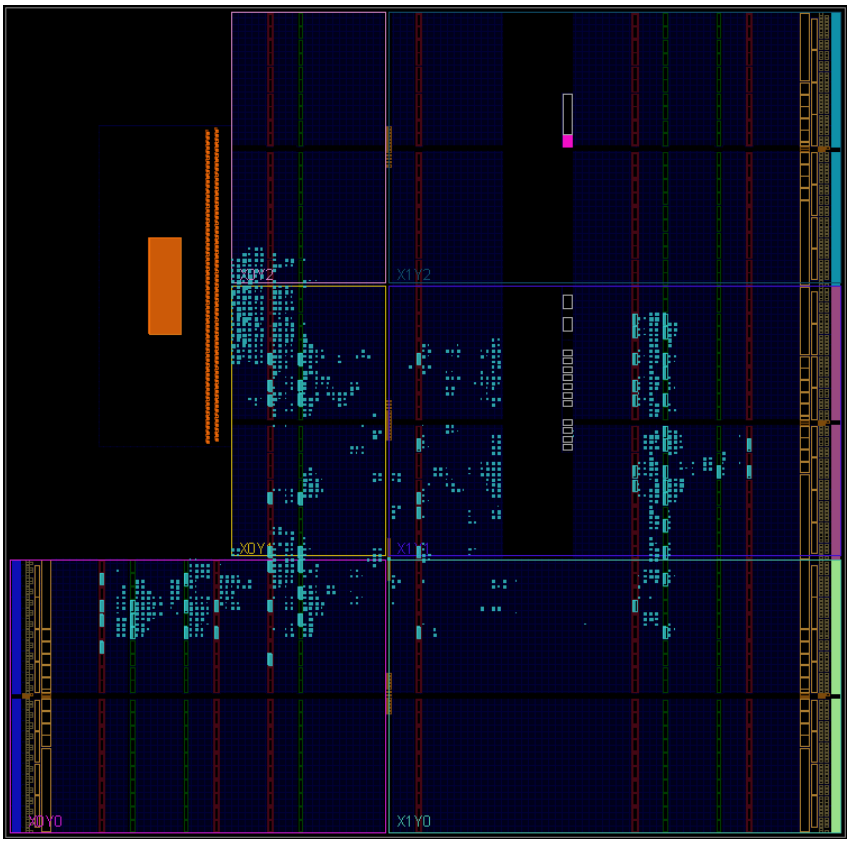


Figure 10: Internal View of NeuraLayer Module on ZYNQ Chip

## 6.2 Power Consumption

Another essential characteristic of systems intended to operate effectively in an embedded environment is low power consumption. As the NeuraLayer hardware system was engineered with conservation and efficiency in mind, it indeed consumes very little power, thus meeting another of our primary objectives. The entire system, including all on-chip components, consumes only 1.44 Watts, while the NeuraLayer hardware module itself accounts for only 0.036 Watts (2.5%) of this total power. The post-implementation power summary from Vivado is given in Figure 11. As can be seen, the vast majority of the total power is consumed by the ZYNQ Processing System (PS), the next largest portion is a result of static power consumption, and the smallest portion is from the NeuraLayer module. One important implication of this extremely uneven breakdown of power consumption is that, even if the NeuraLayer module was to be scaled up significantly, the only portion of the power breakdown being scaled would be the smallest portion; this means that the total on-chip power consumption of a NeuraLayer module two or three times as large as the prototype discussed herein would only consume marginally more power, despite being significantly larger and more capable.

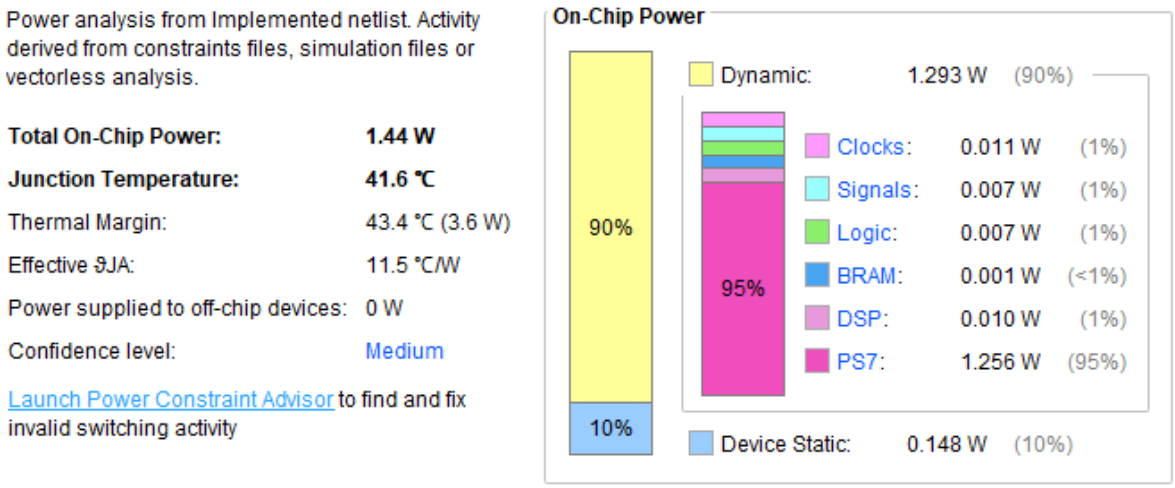


Figure 11: Vivado Power Summary

## 6.3 Training and Testing

The training of an ANN is a project in its own right, and as the primary focus of this team was the engineering of the system itself, there was a negatively disproportionate amount of time available to be given to the training of the final system on the example task of speaker identification. Nevertheless, the system was trained using the MFCC data discussed earlier, and we found that our system reached a point of diminishing returns (for the specific training vein) after 18 passes through the training dataset (epochs) at about 5 minutes/epoch. However, as each initial weight configuration and the final configuration evolved from it is unique, it is reasonable to speculate that, given more time, a more ideal final weight configuration than the one discussed here would have been obtained.

There is one very interesting outcome that deserves specific mention. The example task that this prototype is designed for is speaker identification: it should be able to learn to distinguish a given speaker and place them into one of three classes: Anthony, Thomas, or Other. However, as previously mentioned, during hardware design we had to minimize

the size and range of our network in order to ensure that it would fit on the chip. As a result of this, our network is actually rather small. So, during training, when we tasked the system with learning to separate MFCC data into one of three classes, we had a very interesting result. It almost seemed as if the NeuraLayer module comprehended its own limitations and recognized that it was not large enough to form complex enough of a model to correctly classify the three groups of data, so instead, the module learned a completely different – though very useful – task, but one that it was not explicitly trained to learn. The system learned to separate the two verified speakers (Anthony and Tom) from all other speakers, and it did so with 77% accuracy on a testing set of four thousand data samples that it had never encountered before; and further, this accuracy was achieved by the first successfully trained weight configuration of the system. The testing results for this configuration after being trained for 14, 15, 17, and 18 epochs is given in Figure 12. To us, this demonstrates the capacity of the NeuraLayer system to learn, but indicates that, in order to improve the performance of the system, its maximum layer width and also its data and weight ranges will need to be expanded, as these changes will give the system more breathing room and will allow it to form more complex representations.

```
test() #after 14 epochs
```

```
Total: 3961
Correct: 2804
Incorrect: 1157
Accuracy = 0.7079020449381469
```

```
test() #after 15 epochs
```

```
Total: 3961
Correct: 2822
Incorrect: 1139
Accuracy = 0.7124463519313304
```

```
test() #after 17 epochs
```

```
Total: 3961
Correct: 2883
Incorrect: 1078
Accuracy = 0.7278465034082302
```

```
test() #after 18 epochs
```

```
Total: 3961
Correct: 3056
Incorrect: 905
Accuracy = 0.7715223428427165
```

Figure 12: Test Accuracy Improvement over Several Epochs

# 7 Future Work

The instance of the NeuraLayer system discussed herein is the first functional prototype of a highly general and potentially very powerful neuromorphic architecture. As a result of this, there are many possible courses that this work could follow. Linearly speaking, further iterations could expand the size of the hardware system and increase performance for this same application of speaker identification. Tangentially, the system could be adapted for embedded applications completely different from the prototypical application. However, this team considers the most intriguing possibility to be not evolutionary in any specific direction, but revolutionary.

As discussed earlier, even a markedly expanded iteration of the NeuraLayer hardware module would consume only a modest quantity of FPGA resources. This is certainly important for many conventional reasons: minimizing area, power consumption, and maximizing performance; but there is also an unrelated but tantalizing implication of NeuraLayer's small footprint. The ZYNQ 7020 chip that is used for this project is on the small/moderately sized end of the ZYNQ family spectrum, yet even with its modest resources (53,200 LUT, 220 DSP, 140 BRAM) [14] the 7020 could accommodate several NeuraLayer modules. With some ZYNQ devices featuring more than a quarter million LUT, upwards of 900 DSP slices, and over 500 BRAM [14] (and this is not to mention devices from vendors besides Xilinx), it would be more than feasible to implement several extremely expanded NeuraLayer modules simultaneously on the same chip, potentially employing them for distinct but perhaps complementary tasks; and further, the heterogeneous nature of the ZYNQ device would permit the productivity level integration and synthesis of these independent networks with each other, as well as with any conventional peripheral systems or components. Ultimately, this high-level synthesis would effect something new: a higher order neural network system capable of operating at a level of complexity unobtainable by a single ANN.

This has serious implications. Drawing again from observations of biological nervous systems, it is beyond apparent that the nervous systems of all intelligent entities are not single neural networks, but are in fact higher order integrated neural network systems. From this observation, we speculate that, if real progress is to be made toward the engineering of truly intelligent systems, it will be necessary to integrate the efforts of many independent ANNs. From the experience of engineering the NeuraLayer system, this team feels that the productivity level synthesis of such high order network systems is a revolutionary but logical progression from this work.

# 8 Academic Disciplines Utilized

Table 2 shows the disciplines used in our project from our program. We used knowledge from all of our Computer Engineering course that we have taken here at New Paltz.

We used all of the courses in the software discipline to assist us in creating logical, maintainable and readable code. We used all of the courses in the Comptuer Systems discipline to construct an understanding of how computer systems are built and utilized for specific tasks. The courses under the Digital Systems discipline at New Paltz imparted to us the knowledge necessary to develop efficient Verilog HDL.

Table 2: Computer Engineering Disciplines and Courses within the New Paltz program.

| Disciplines | Course Name |
|---|---|
| Software | EGC251 C/C++ Programming<br>CPS310 Computer Science II (Data Structures)<br>CPS210 Computer Science<br>CPS353 Software Engineering |
| Computer Systems | EGC331 Microprocessors<br>EGC332 Microprocessors Lab<br>EGC433 Embedded Systems<br>EGC442 Computer Architecture<br>EGC451 Real-Time Systems |
| Digital Systems | EGC220 Digital Logic Fundamentals<br>EGC221 Digital Logic Lab<br>EGC320 Digital System Design<br>EGC441 System On Chip<br>EGC445 VLSI Design<br>EGC446 VLSI Lab<br>EGC447 Functional Verification |

# 9 Design Constraints

Design constraints are helpful in the early stages of development in any type of project. They keep the project steered in the correct direction throughout the entirety of the projects development. Our design constraints fall into two different sections: computing system constraints and auxiliary constraints.

## 9.1 Computing System Constraints

When designing any type of computing system there are certain constraints that need to be taken in to account at the projects inception. Some of these constraints include: speed of the hardware system, the amount of digital resources used on our development board and power consumption.

### 9.1.1 Speed

An important part of our design is the speed of the system, specifically the speed in which it can train. Machine learning applications with large datasets inherently take very long periods of time. It was important to minimize the training time for our own productivity so we could rapidly test and change our system.

### 9.1.2 Resource Utilization

In order for a system to operate effectively in an embedded environment, it is typically required that it consume minimal resources: this helps with speed and power consumption, but also makes the embedded system less intrusive and allows it to blend into its environment more easily. Since we designed our project with this in mind it means we could use a smaller, cheaper chip in the future.

### 9.1.3 Power Consumption

With the embedded environment in mind it was also necessary to keep the power consumption of the system to a minimum. This insures that the system has the smallest parasitic impact on the devices that it interacts with.

# 10 Engineering Standards

Being derived from observations of the perceived functionality of biological nervous systems, neuromorphic systems are extremely experimental, and it seems that all 'standards' are up for debate and interpretation. A search of the IEEE Standards for Neuromorphic Engineering returned no results. There are, however, several aspects of neuromorphic design that approach the level of being a standard. Among these is the perceptron model of the artificial neuron, and the backpropagation training algorithm.

## 10.1 Perceptron Model

The Perceptron model of the artificial neuron sums a set of weighted inputs and applies that sum to an activation function, thus producing the final output of the neuron. This functionality is based on the perceived gross functioning of the nervous cell. The Perceptron is very versatile, and is still the most common model for implementing artificial neurons [25]. A generic representation of the Perceptron is given in Figure 13.

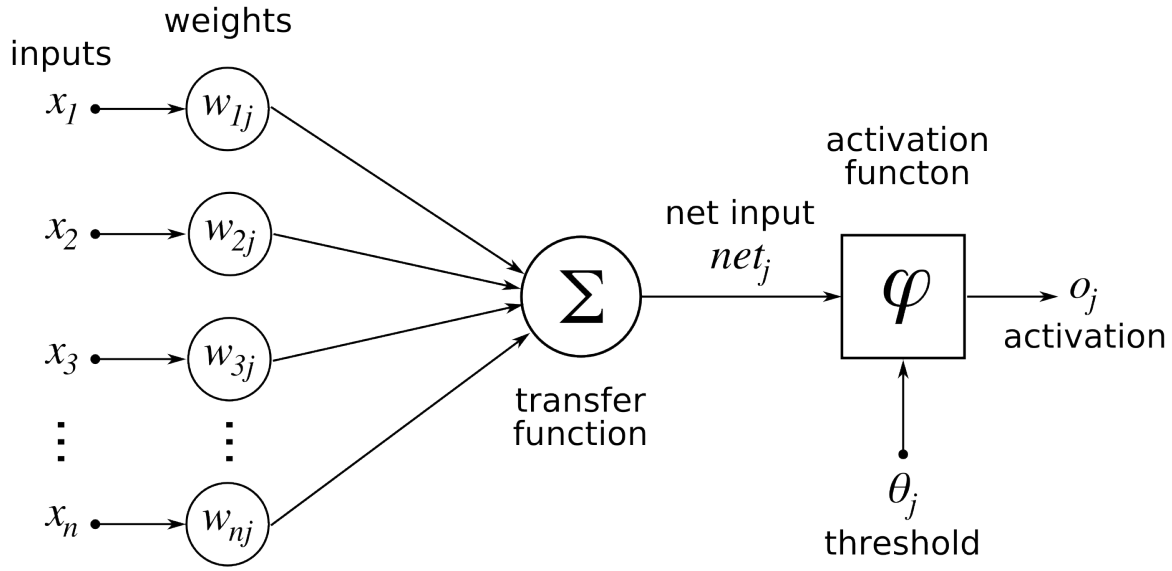


Figure 13: Perceptron Model of the Artificial Neuron [24]

## 10.2 Backpropagation

The backpropagation algorithm is by far the most frequently used method of training ANNs [22]. The algorithm works as follows:

1. Calculate error at each output node by comparing the output value with the target value

2. Find the derivative of the error with respect to each weighting parameter of the current node

3. Accumulate error signals for each node in the output layer and propagate to the previous layer

4. Calculate weight updates for nodes in the previous layer as in the output layer, but with a different formula

5. Accumulate error signals as before

6. Repeat until input layer is reached

For reference, a full derivation of the backpropagation algorithm for the sigmoid activation function is given in [26]. However, it was necessary that we derive our own version of the equations in order to fit our specific design. The first step was to derive an equation for calculating a weight update for a connection from hidden layer 2 to the output layer:

$$\Delta W_{kj} = \varepsilon[\overbrace{t_k - a_k}^{\delta_k}]a_j$$

Where $k$ denotes a node in the output layer, $j$ denotes a node in the second hidden layer, $i$ indicates a node in the first hidden layer, and $h$ signifies an input node. $\varepsilon$ is the learning rate, which tapers or exacerbates the magnitude of weight updates. $a$ is the activation of a node in the subscripted layer, and $t$ is the correct, or target output of a node in the subscripted layer. The $\delta_k$ value is obtained for each output, and the sum of these values is used in the update equation for weights between hidden layer 1 and 2:

$$\Delta W_{ji} = \varepsilon\Big[\overbrace{\sum_k (t_k - a_k)W_{kj}}^{\delta_j}\Big]a_j$$

Lastly, the weight update for a connection between the input layer and the first hidden layer is calculated using the sum of the $\delta_j$ values from the hidden layer 1 to 2 updates:

$$\Delta W_{ih} = \varepsilon\Bigg[\sum_j \Big[\sum_k (t_k - a_k)W_{kj}\Big]W_{ji}\Bigg]a_h$$

# 11 Conclusion

In this capstone project, our design team engineered a prototype of a heterogeneous Artificial Neural Network (ANN) and applied it to the task of embedded, offline speaker identification. The primary fruit of our efforts is the NeuraLayer hierarchical hardware module, which is a fast, small footprint, and low power configurable neural layer capable of implementing the functionality of an entire multilayer ANN. The entire System on Chip (SoC) consumes 1.44 Watts, while the proprietary hardware module itself consumes only 0.036 Watts, and utilizes only $\approx 5\%$ of the FPGA LUT fabric, 11% of the embedded memory blocks, and 13% of the embedded DSP blocks. Ultimately, the small network size resultant from a conservationist engineering mindset rendered this prototypical iteration of our system too small to accommodate our initial task of separating speech data into

three classes, two verified and one unverified; but despite being too small, instead of learning nothing, the small, overly constrained system autonomously arrived at a model that unites the two verified classes and differentiates them from the unverified class with $\approx 77\%$ accuracy. This outcome was unexpected and surprising, but useful nonetheless, and the arrival at this solution by the system without explicit training is intriguing. This team feels that the architecture of the NeuraLayer system is general enough to be adapted and applied to many different situations requiring small, intelligent, and low power embedded systems.

# 12 References


[1] M. Otis, "Embedded Systems" at State University of New York at New Paltz.

[2] T. Potok, C. Schuman, R. Patton, T. Hylton, H. Li, and R. Pino, "Neuromorphic Computing, Architectures, Models, and Applications. A Beyond-CMOS Approach to Future Computing, June 29-July 1, 2016, Oak Ridge, TN," 2016.

[3] E. Variani, X. Lei, E. Mcdermott, I. L. Moreno, and J. Gonzalez-Dominguez, "Deep neural networks for small footprint text-dependent speaker verification," 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2014.

[4] C. E. Vivaracho, J. Ortega-Garcia, L. Alonso, and Q. I. Moro, "A Comparative Study of MLP-based Artificial Neural Networks in Text-Independent Speaker Verification against GMM-based Systems," Eurospeech 2001, 2001.

[5] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: an overview," 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," Neural Networks, vol. 61, pp. 85–117, 2015.

[7] L. can der Maaten and E. Postma, "Dimensionality Reduction: A Comparative Review," Tilburg centre for Creative Computing, Oct. 2009.

[8] Y. Chen, "Advanced Computing Systems" at State University of New York at New Paltz.

[9] L. Bako, "Real-Time Clustering of Datasets with Hardware Embedded Neuromorphic Neural Networks," 2009 International Workshop on High Performance Computational Systems Biology, 2009.

[10] B. Izadi, "Computer Architecture" at State University of New York at New Paltz.

[11] D. Radhakrishnan, "VLSI Design" at State University of New York at New Paltz.

[12] Y. Chen, "System on Chip" at State University of New York at New Paltz.

[13] Bajaj, Ronak & Fahmy, Suhaib. (2015). Mapping for maximum performance on FPGA DSP blocks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 35. 1-1. 10.1109/TCAD.2015.2474363.

[14] Zynq-7000 All Programmable SoC Data Sheet: Overview. (2017). 1st ed. Xilinx Inc., p.17.

[15] Xilinx Inc., (2017, July). Vivado Design Suite: AXI Reference Guide. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[16] Speech.cs.cmu.edu. (1991). CMU Robust Speech Recognition Group: Census Database. [online] Available at: http://www.speech.cs.cmu.edu/databases/an4/ [Accessed 6 May 2018].

[17] "Speaker Identification Using Pitch and MFCC," MATLAB & Simulink. [Online]. Available: https://www.mathworks.com/help/audio/examples/speaker-identification-using-pitch-and-mfcc.html. [Accessed: 12-May-2018].

[18] En.wikipedia.org. (2018). Mel-frequency cepstrum. [online] Available at: https://en.wikipedia.org/wiki/Mel-frequency_cepstrum [Accessed 6 May 2018].

[19] J. Lyons, "Python Speech Features," GitHub, 31-Aug-2017. [Online]. Available: https://github.com/jameslyons/python_speech_features. [Accessed: 12-May-2018].

[20] Xilinx Inc., Python Productivity for ZYNQ Documentation: Loading Overlays. [Online]. Available: http://pynq.readthedocs.io/en/latest/pynq_ overlays/loading_an_overlay.html

[21] Xilinx Inc., Python Productivity for ZYNQ Documentation: MMIO Package. [Online]. Available: http://pynq.readthedocs.io/en/latest/pynq_package/pynq.mmio.html

[22] S. L. Pinjare and A. K. M, "Implementation of Neural Network Back Propagation Training Algorithm on FPGA," International Journal of Computer Applications, vol. 52, no. 6, pp. 1–7, 2012.

[23] Rectifier (neural networks), Wikipedia, 14-May-2018. [Online]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks). [Accessed: 15-May-2018].

[24] Artificial Neural Networks/Activation Functions, Wikibooks. [Online]. Available: https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Activation_Functions#/media/File:Artifi
[Accessed: 10-May-2018].

[25] Artificial neuron, Wikipedia, 04-May-2018. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neuron. [Accessed: 15-May-2018].

[26] L. Meeden, "Derivation of Backpropagation," Derivation of Backpropagation. [Online]. Available: https://www.cs.swarthmore.edu/ meeden/cs81/s10/BackPropDeriv.pdf. [Accessed: 06-May-2018].

# 13 Appendices

## 13.1 Jupyter Notebook for Handling NeuraLayer

# NeuraLayer

May 15, 2018

## 1 Productivity Level Environment of NeuraLayer System

This Notebook is a record of the code necessary to instantiate, handle, train, and test an instance of a Configurable Neural Layer (CNL) Hierarchical Hardware Module for the application of offline, embedded, and low-power Speaker Verification.

### 1.1 Import Packages

```
In [ ]: import math
        import time
        import random
        import numpy as np
        from pynq import Overlay
        from pynq import MMIO
        from pandas import DataFrame, read_csv
        import matplotlib.pyplot as plt
        import pandas as pd
```

### 1.2 Download Hardware to ZYNQ Programmable Logic

```
In [ ]: neuralNetwork = Overlay("/home/xilinx/jupyter_notebooks/NeuraLayer/NeuraLayer.bit")
        neuralNetwork.download()
        net = MMIO(0x43C00000,0x10000)
```

### 1.3 Load Data from .csv File Columns into Lists and Scale

```
In [ ]: global dataDepth
        #read in data from csv file and save to columnar list objects
        df = pd.read_csv('mfccTrainingStd.csv', names = ['file', 'energy', 'f1', 'f2', 'f3',
                                                         'f4', 'f5', 'f6', 'f7', 'f8', 'f9',
                                                         'f10', 'f11', 'speaker'],
                                                              index_col = False)

        energy = list(df.energy)
        f1 = list(df.f1)
        f2 = list(df.f2)
        f3 = list(df.f3)
```

```
f4 = list(df.f4)
f5 = list(df.f5)
f6 = list(df.f6)
f7 = list(df.f7)
f8 = list(df.f8)
f9 = list(df.f9)
f10 = list(df.f10)
f11 = list(df.f11)
speaker = list(df.speaker)

dataDepth = len(f1)

#scale all features
scalar = 100
energy = [x*scalar for x in energy]
f1 = [x*scalar for x in f1]
f2 = [x*scalar for x in f2]
f3 = [x*scalar for x in f3]
f4 = [x*scalar for x in f4]
f5 = [x*scalar for x in f5]
f6 = [x*scalar for x in f6]
f7 = [x*scalar for x in f7]
f8 = [x*scalar for x in f8]
f9 = [x*scalar for x in f9]
f10 = [x*scalar for x in f10]
f11 = [x*scalar for x in f11]

#round all features to integer
energy = [round(x) for x in energy]
f1 = [round(x) for x in f1]
f2 = [round(x) for x in f2]
f3 = [round(x) for x in f3]
f4 = [round(x) for x in f4]
f5 = [round(x) for x in f5]
f6 = [round(x) for x in f6]
f7 = [round(x) for x in f7]
f8 = [round(x) for x in f8]
f9 = [round(x) for x in f9]
f10 = [round(x) for x in f10]
f11 = [round(x) for x in f11]
```

### 1.3.1 Retrieve Basic Data Info

```
In [ ]: print('Max of each feature list')
        print(max(energy))
        print(max(f1))
        print(max(f2))
        print(max(f3))
```

```
print(max(f4))
print(max(f5))
print(max(f6))
print(max(f7))
print(max(f8))
print(max(f9))
print(max(f10))
print(max(f11))
print('')
print('Min of each feature list')
print(min(energy))
print(min(f1))
print(min(f2))
print(min(f3))
print(min(f4))
print(min(f5))
print(min(f6))
print(min(f7))
print(min(f8))
print(min(f9))
print(min(f10))
print(min(f11))
```

### 1.4 Multilayer Perceptron Parameter Skeleton

**The following parameter skeleton is the software aspect of the NeuraLayer Heterogeneous Artificial Neural Network** It consists of an object oriented network skeleton that possesses objects for each layer and node in a variably specified network: each node object has attributes and functions to facilitate temporary information storage, as well as updates to the node parameters by implementing backpropagation.

```
In [ ]: #Class definition for Network object
        class Network:
            #Initialize network
            #hidden nodes and output nodes are created and appended to their respective lists
            def __init__(self, inputLayerSize, hiddenLayer1Size,
                         hiddenLayer2Size, outputLayerSize):
                self.ILsize = inputLayerSize
                self.HL1size = hiddenLayer1Size
                self.HL2size = hiddenLayer2Size
                self.OLsize = outputLayerSize
                self.NetworkOutput = []
                self.hiddenLayer1 = [HiddenNode(self.ILsize + 1) for i in range(self.HL1size)]
                self.hiddenLayer1.append(BiasNode()) #Add bias node
                self.hiddenLayer2 = [HiddenNode(self.HL1size + 1) for i in range(self.HL2size)]
                self.hiddenLayer2.append(BiasNode()) #Add bias node
                self.outputLayer = [OutputNode(self.HL2size + 1) for i in range(self.OLsize)]
```

```
#Receives input vector from training function
#Output from one layer is collected and sumbitted to the next layer
#Output from final layer is stored in class attribute NetworkOutput
def storeInput(self, inputVec, layer):
    if layer == 1:
        for i in range(self.HL1size):
            self.hiddenLayer1[i].storeInputs(inputVec)
    elif layer == 2:
        for i in range(self.HL2size):
            self.hiddenLayer2[i].storeInputs(inputVec)
    elif layer == 3:
        for i in range(self.OLsize):
            self.outputLayer[i].storeInputs(inputVec)

def storeOutput(self, outputs, layer):
    if layer == 1:
        for i in range(self.HL1size):
            self.hiddenLayer1[i].storeOutput(outputs[i])
    elif layer == 2:
        for i in range(self.HL2size):
            self.hiddenLayer2[i].storeOutput(outputs[i])
    elif layer == 3:
        for i in range(self.OLsize):
            self.outputLayer[i].storeOutput(outputs[i])

#Method that directs the computation and backpropagation of error signals
#Receives vector of target outputs from training function
def updateWeights(self, targetVec, mbi):
    sumOfErrors = 0
    deltaK = [] #List of error signals from update function of output nodes
    deltaJ = [] #List of error signals from update function of HL2 nodes
    #update weights from output to hidden layer 2
    for i in range(self.OLsize):
        deltaK.append(self.outputLayer[i].update(targetVec[i], mbi))
        sumOfErrors += self.outputLayer[i].error
    #print ("Sum of Squared Errors: ", sumOfErrors) #report SSE
    #update weights from hidden layer 2 to hidden layer 1
    for i in range(self.HL2size):
        deltaJ.append(self.hiddenLayer2[i].update(deltaK, i, 2, mbi))
    #update weights from hidden layer 1 to input layer
    for i in range(self.HL1size):
        self.hiddenLayer1[i].update(deltaJ, i, 1, mbi)
    return sumOfErrors

def transferGradient(rawGradient):
    grad = rawGradient
    if grad < 500 and grad > 30:
        grad = 1
```

```
        elif grad >= 500 and grad < 3000:
            grad = 2
        elif grad >= 3000 and grad < 5000:
            grad = 3
        elif grad >= 5000 and grad < 7000:
            grad = 4
        elif grad >= 7000:
            grad = 5
        elif grad < -30 and grad > -500:
            grad = -1
        elif grad <= -500 and grad > -3000:
            grad = -2
        elif grad <= -3000 and grad > -5000:
            grad = -3
        elif grad <= -5000 and grad > -7000:
            grad = -4
        elif grad <= -7000:
            grad = -5
        return grad

#Hidden node class definition
class HiddenNode:
    #Initialize node attributes and weights based on number of inputs
    def __init__(self, numInputs):
        self.numInputs = numInputs
        self.inputs =[]
        self.nodeOutput = 0
        self.weights = []
        self.gradient = []
        for i in range(numInputs): #Initialize weights to random values
            self.weights.append(round(7*np.random.normal(0, math.sqrt(2/numInputs))))
            self.gradient.append(0)

    def storeInputs(self, inputs):
        self.inputs = inputs

    def storeOutput(self, output):
        self.nodeOutput = output


    #Computes weight updates based on backpropagated error signal (delta)
    def update(self, delta, index, layer, mbi):
        interimSum = 0 #Variable to hold sum of earlier delta signal/weight products
        if layer == 2:
            for i in range(len(delta)):
                interimSum += delta[i] * network.outputLayer[i].weights[index]
            rateSum = lr * interimSum
            if mbi == False:
```

```
                if self.nodeOutput > 0 and self.nodeOutput < 2047:
                    for i in range(self.numInputs):
                        self.gradient[i] += rateSum * network.hiddenLayer1[i].nodeOutput
                    deltaJ = interimSum
                else:
                    deltaJ = 0
            else:
                if self.nodeOutput > 0 and self.nodeOutput < 2047:
                    for i in range(self.numInputs):
                        self.gradient[i] += rateSum * network.hiddenLayer1[i].nodeOutput
                    deltaJ = interimSum
                else:
                    deltaJ = 0
                for i in range(self.numInputs):
                    grad = round(self.gradient[i])
                    grad = transferGradient(grad)
                    self.weights[i] += grad
                    if self.weights[i] > 127:
                        self.weights[i] = 127
                    elif self.weights[i] < -127:
                        self.weights[i] = -127
            return deltaJ #deltaJ is then backpropagated to HL1
        elif layer == 1:
            for i in range(len(delta)):
                interimSum += delta[i] * network.hiddenLayer2[i].weights[index]
            rateSum = lr * interimSum
            if mbi == False:
                if self.nodeOutput > 0 and self.nodeOutput < 2047:
                    for i in range(self.numInputs):
                        self.gradient[i] += rateSum * self.inputs[i]
            else:
                if self.nodeOutput > 0 and self.nodeOutput < 2047:
                    for i in range(self.numInputs):
                        self.gradient[i] += rateSum * self.inputs[i]
                for i in range(self.numInputs):
                    grad = round(self.gradient[i])
                    grad = transferGradient(grad)
                    self.weights[i] += grad
                    if self.weights[i] > 127:
                        self.weights[i] = 127
                    elif self.weights[i] < -127:
                        self.weights[i] = -127

#Class definition of output node
class OutputNode:
    def __init__(self, numInputs):
        self.numInputs = numInputs
        self.nodeOutput = 0
```

```
            self.weights = []
            self.gradient = []
            self.error = 0 #Error at node; used to compute sum of squared errors (SSE)
            for i in range(numInputs):
                self.weights.append(round(7*np.random.normal(0, math.sqrt(2/numInputs))))
                self.gradient.append(0)

        def storeOutput(self, output):
            self.nodeOutput = output

        #Update weights from HL2 to output node based on target output
        def update(self, targetOutput, mbi):
            deltaK = targetOutput - self.nodeOutput #deltaK will be backpropagated to HL2
            self.error = .5 * deltaK * deltaK
            if mbi == False:
                for i in range(self.numInputs):
                    self.gradient[i] += lr * deltaK * network.hiddenLayer2[i].nodeOutput
            else:
                for i in range(self.numInputs):
                    grad = round(self.gradient[i] + lr * deltaK
                                 * network.hiddenLayer2[i].nodeOutput)
                    grad = transferGradient(grad)
                    self .weights[i] += grad
                    if self.weights[i] > 127:
                        self.weights[i] = 127
                    elif self.weights[i] < -127:
                        self.weights[i] = -127
            return deltaK

    #Class definition of bias node
    #Bias nodes are appended to the layer previous to the layer they affect
    class BiasNode:
        def __init__(self):
            self.nodeOutput = 1
```

### 1.4.1 Generate an Instance of the Parameter Skeleton

```
In [ ]: #Network(inputLayerSize, hiddenLayer1Size, hiddenLayer2Size, outputLayerSize)
        global lr
        network = Network(12, 29, 19, 3)
        lr = .0000001
```

## 1.5 Functions for Exposing the Hardware to the Python Environment

Special functions are necessary to facilitate control of the NeuraLayer hardware module from the Python Environment.

```
In [ ]: #Compute network outputs
        def computeOutput(inputs):
```

```
    selectLayer(1)
    out1 = []
    out2 = []
    out3 = []
    inputs.append(1) #layer 1 bias
    loadData(inputs)
    start(1)
    network.storeInput(inputs, 1)
    for i in range(network.HL1size):
        out1.append(read())
    network.storeOutput(out1, 1)
    out1.append(1) #layer 2 bias
    selectLayer(2)
    reset()
    loadData(out1)
    start(1)
    for i in range(network.HL2size):
        out2.append(read())
    network.storeOutput(out2, 2)
    out2.append(1) #layer 3 bias
    selectLayer(3)
    reset()
    loadData(out2)
    start(1)
    for i in range(network.OLsize):
        outD = read()
        if outD > 2047:
            t = tC(outD, 1)
            outD = int(t[2:],2)*-1
        out3.append(outD)
    network.storeOutput(out3, 3)
    network.NetworkOutput = out3

#Compute network outputs and print dataflow
def computeOutputP(inputs):
    selectLayer(1)
    out1 = []
    out2 = []
    out3 = []
    inputs.append(1) #layer 1 bias
    print(inputs)
    loadData(inputs)
    start(1)
    print('')
    network.storeInput(inputs, 1)
    for i in range(network.HL1size):
        out1.append(read())
    print(out1)
```

```
    network.storeOutput(out1, 1)
    out1.append(1) #layer 2 bias
    selectLayer(2)
    reset()
    loadData(out1)
    start(1)
    print('')
    for i in range(network.HL2size):
        out2.append(read())
    print(out2)
    network.storeOutput(out2, 2)
    out2.append(1) #layer 3 bias
    selectLayer(3)
    reset()
    loadData(out2)
    start(1)
    print('')
    for i in range(network.OLsize):
        outD = read()
        if outD > 2047:
            t = tC(outD, 1)
            outD = int(t[2:],2)*-1
        out3.append(outD)
    print(out3)
    network.storeOutput(out3, 3)
    network.NetworkOutput = out3

#Load weights from skeleton into hardware module
def loadWeights():
    numInputsHL1 = network.ILsize + 1
    numInputsHL2 = network.HL1size + 1
    numInputsOL = network.HL2size + 1
    olOffset = numInputsHL1 + numInputsHL2
    for i in range(1, 30):
        if i == 1:
            writeWeight(network.hiddenLayer1[i-1].weights[0], i, 0)
            update(1)
            for j in range(1, numInputsHL1):
                writeWeight(network.hiddenLayer1[i-1].weights[j], i, j)
            for j in range(numInputsHL2):
                writeWeight(network.hiddenLayer2[i-1].weights[j], i, j + numInputsHL1)
            for j in range(numInputsOL):
                writeWeight(network.outputLayer[i-1].weights[j], i, j + olOffset)
        elif i > 1 and i <= 3:
            for j in range(numInputsHL1):
                writeWeight(network.hiddenLayer1[i-1].weights[j], i, j)
            for j in range(numInputsHL2):
                writeWeight(network.hiddenLayer2[i-1].weights[j], i, j + numInputsHL1)
```

```
            for j in range(numInputsOL):
                writeWeight(network.outputLayer[i-1].weights[j], i, j + olOffset)
        elif i > 3 and i <= 19:
            for j in range(numInputsHL1):
                writeWeight(network.hiddenLayer1[i-1].weights[j], i, j)
            for j in range(numInputsHL2):
                writeWeight(network.hiddenLayer2[i-1].weights[j], i, j + numInputsHL1)
        elif i > 19 and i <= 29:
            for j in range(numInputsHL1):
                writeWeight(network.hiddenLayer1[i-1].weights[j], i, j)
    update(0)

#Write a specific weight to a specific node in the module
def writeWeight(weight, nodeSel, weightPtr):
    ptr = toBin(weightPtr, 1)
    node = toBin(nodeSel, 4)
    if weight >= 0:
        wt = toBin(weight, 0)
    elif weight < 0:
        wt = tC(weight, 0)
    wrData = fuse3(wt, node, ptr)
    net.write(0x08, wrData)

#Load dataStream into the module
def loadData(data):
    data.append(1) #dummy data to ensure ptr reaches max
    writeData(data[0], 0)
    write(1)
    for i in range(1, len(data)):
        writeData(data[i], i)

#Write specific datum to module data memory
def writeData(data, dataPtr):
    ptr = toBin(dataPtr, 3)
    if data >= 0:
        dt = toBin(data, 2)
    elif data < 0:
        dt = tC(data, 1)
    wrData = fuse2(dt, ptr)
    net.write(0x04, wrData)

#Reset the module
def reset():
    net.write(0x18, 1)
    net.write(0x18, 0)

#Toggle update control bit
def update(sel):
```

```
        if sel == 1:
            net.write(0x00, 2)
        elif sel == 0:
            net.write(0x00, 0)

    #Toggle write control bit
    def write(sel):
        if sel == 1:
            net.write(0x00, 4)
        elif sel == 0:
            net.write(0x00, 0)

    #Sequentially extract output data
    def read():
        net.write(0x00, 8)
        data = net.read(0xc)
        net.write(0x00, 0)
        return(data)

    #Toggle start control bit
    def start(sel):
        if sel == 1:
            net.write(0x00, 1)
        elif sel == 0:
            net.write(0x00, 0)

    #Set layerSelect control signal
    def selectLayer(layer):
        if layer == 1:
            net.write(0x14, 1)
        elif layer == 2:
            net.write(0x14, 2)
        elif layer == 3:
            net.write(0x14, 3)

    #Check if module is done computing
    def done():
        isDone = net.read(0x10)
        if isDone == 1:
            return(True)
        elif isDone == 0:
            return(False)
```

### 1.6 Miscellaneous Facilitative Functions

```
In [ ]: #obtain properly formatted two's complement string representation of a negative integer
    def tC(num, ind):
        s = bin(num)
```

```
    s = s.replace('0','2')
    s = s.replace('1','0')
    s = s.replace('2','1')
    ma = len(s)-1
    b = list(s)
    b[2] = '1'
    b = b[2:ma+1]
    ma = len(b)-1
    if b[ma] == '0':
        b[ma] = '1'
    else:
        b[ma] = '0'
        for i in range(ma-1,-1,-1):
            if b[i] == '1':
                b[i] = '0'
            else:
                b[i] = '1'
                break
    if ind == 0: #weight
        x = 8
    elif ind == 1: #data
        x = 12
    for i in range(x-(ma+1)):
        b.insert(0,'1')
    tc = ''.join(b)
    return(tc)

#Obtain properly formatted binary string representation of a positive integer
def toBin(num, ind):
    s = bin(num)
    b = list(s)
    if ind == 0: #weight
        while len(b) < 10:
            b.insert(2, '0')
        b = b[2:len(b)]
    elif ind == 1: #weight ptr
        while len(b) < 8:
            b.insert(2, '0')
        b = b[2:len(b)]
    elif ind == 2: #data
        while len(b) < 14:
            b.insert(2, '0')
        b = b[2:len(b)]
    elif ind == 3: #data ptr
        while(len(b) < 7):
            b.insert(2, '0')
        b = b[2:len(b)]
    elif ind == 4: #nodeSel
```

```
        while(len(b) < 7):
            b.insert(2, '0')
        b = b[2:len(b)]
    binRep = ''.join(b)
    return(binRep)

#Fuse binary strings together and convert to integer representation
def fuse2(lower, higher):
    fused = higher + lower
    fused = int(fused,2)
    return(fused)

def fuse3(lower, middle, higher):
    fused = higher + middle + lower
    fused = int(fused,2)
    return(fused)

#Generate weight stream by parsing parameter skeleton
def getWeightStream():
    weightStream = []
    for i in range(29):
        if i < 3:
            L1 = [network.hiddenLayer1[i].weights[j]
                  for j in range(network.hiddenLayer1[i].numInputs)]
            L2 = [network.hiddenLayer2[i].weights[j]
                  for j in range(network.hiddenLayer2[i].numInputs)]
            L3 = [network.outputLayer[i].weights[j]
                  for j in range(network.outputLayer[i].numInputs)]
            weightStream += L1 + L2 + L3
        elif i >= 3 and i < 19:
            L1 = [network.hiddenLayer1[i].weights[j]
                  for j in range(network.hiddenLayer1[i].numInputs)]
            L2 = [network.hiddenLayer2[i].weights[j]
                  for j in range(network.hiddenLayer2[i].numInputs)]
            weightStream += L1 + L2
        elif i >= 19 and i < 29:
            L1 = [network.hiddenLayer1[i].weights[j]
                  for j in range(network.hiddenLayer1[i].numInputs)]
            weightStream += L1
    return weightStream

#Initialize parameter skeleton from a saved weighStream
def initFromStream(stream):
    numInputsHL1 = network.ILsize + 1
    numInputsHL2 = network.HL1size + 1
    numInputsOL = network.HL2size + 1
    HL1and2 = numInputsHL1 + numInputsHL2
    for i in range(29):
```

```
            offset = getOffset(i)
            if i < 3:
                for j in range(numInputsHL1):
                    network.hiddenLayer1[i].weights[j] = stream[offset + j]
                for j in range(numInputsHL2):
                    network.hiddenLayer2[i].weights[j] = stream[offset + j + numInputsHL1]
                for j in range(numInputsOL):
                    network.outputLayer[i].weights[j] = stream[offset + j + HL1and2]
            elif i >= 3 and i < 19:
                for j in range(numInputsHL1):
                    network.hiddenLayer1[i].weights[j] = stream[offset + j]
                for j in range(numInputsHL2):
                    network.hiddenLayer2[i].weights[j] = stream[offset + j + numInputsHL1]
            elif i >= 19 and i < 29:
                for j in range(numInputsHL1):
                    network.hiddenLayer1[i].weights[j] = stream[offset + j]

    def getOffset(node):
        if node < 3:
            offset = node * 63
        elif node >= 3 and node < 19:
            offset = ((node - 3) * 43) + (3 * 63)
        elif node >= 19 and node< 29:
            offset = ((node - 19) * 13) + (16 * 43) + (3 * 63)
        return offset



    #Write current weight stream to a text file
    def saveWeightStream(weightStream, fileNum):
        fileName = 'weightStream' + str(fileNum) + '.txt'
        f = open(fileName, 'w')
        for x in range(len(weightStream)):
            f.write(str(weightStream[x]) + '\n')
        f.close()

    #Read weight stream from a text file
    def loadWeightStream(fileNum):
        fileName = 'weightStream' + str(fileNum) + '.txt'
        stream = []
        f = open(fileName, 'r')
        for line in f:
            y = int(line)
            stream.append(y)
        return(stream)
```

### 1.7 Initialize Parameter Skeleton from Saved WeightStream

```
In [ ]: initFromStream(loadWeightStream(1))
```

### 1.8 Training Function

```
In [ ]: BatchSize = 7475
        high = 2047
        low = -2047
        def train(x):
            indexList = random.sample(range(dataDepth), dataDepth)
            errorPlotVec = []
            mbi = False #minibatch indicator
            reset()
            loadWeights()
            for i in range(dataDepth):
                reset()
                index = indexList[i]
                #generate input vector
                inVec = [energy[index], f1[index], f2[index], f3[index], f4[index], f5[index],
                            f6[index], f7[index], f8[index], f9[index], f10[index],
                            f11[index]]
                #select appropriate target vector
                if speaker[index] == 'ant':
                    targetVec = [high, low, low]
                elif speaker[index] == 'tom':
                    targetVec = [low, high, low]
                else:
                    targetVec = [low, low, high]
                #set minibatch indicator
                if i%BatchSize == 0 and i != 0:
                    mbi = True
                else: mbi = False
                #operate NeuraLayer
                if i%BatchSize == 1 and i != 1:
                    loadWeights()
                    print('Epoch:', x)
                computeOutput(inVec)
                errorPlotVec.append(network.updateWeights(targetVec, mbi))
            return errorPlotVec
```

#### 1.8.1 Train for a given number of epochs and plot error

```
In [ ]: errorPlotList = []
        for x in range(15):
            errorPlotList += (train(x))
        plt.plot(errorPlotList)
        plt.show()
```

### 1.9 Save WeightStream to .txt File

```
In [ ]: saveWeightStream(getWeightStream(), 2)
```

### 1.10 Data Flow Visualization Function

```
In [ ]: high = 2047
        low = -2047
        index = random.randint(1, 14950)
        print(index)
        #generate input vector
        inVec = [energy[index], f1[index], f2[index], f3[index], f4[index], f5[index],
                            f6[index], f7[index], f8[index], f9[index], f10[index],
                            f11[index]]
        if speaker[index] == 'ant':
            targetVec = [high, low, low]
        elif speaker[index] == 'tom':
            targetVec = [low, high, low]
        else:
            targetVec = [low, low, high]
        reset()
        #loadWeights()
        computeOutputP(inVec)
        #network.updateWeights(targetVec)
        print ("Prediction: ", network.NetworkOutput)
        print('')
        print ("Actual: ", targetVec)
```

### 1.11 Testing Function

```
In [ ]: def test():
            indexList = random.sample(range(dataDepth), dataDepth)
            reset()
            loadWeights()
            correct = 0
            incorrect = 0
            total = 0
            for i in range(dataDepth):
                reset()
                index = indexList[i]
                inVec = [energy[index], f1[index], f2[index], f3[index], f4[index], f5[index],
                        f6[index], f7[index], f8[index], f9[index], f10[index], f11[index]]
                if speaker[index] == 'ant' or speaker[index] == 'tom':
                    target = 'Anthony or Tom'
                else:
                    target = 'Impostor'
                computeOutput(inVec)
                output = network.NetworkOutput
```

```
            if output == [2047, -2047, -2047] or output == [-2047, 2047, -2047]:
                prediction = 'Anthony or Tom'
            else:
                prediction = 'Impostor'

            if target == prediction:
                correct += 1
            else:
                incorrect += 1
            total += 1
        print('Total:', total)
        print('Correct:', correct)
        print('Incorrect:', incorrect)
        print('Accuracy =', correct/total)
```

```
In [ ]: test()
```

### 1.12 Op Tests

```
In [ ]: #Test functionality of network IP core
        def opTest0(layer):
            if layer == 1:
                dc = 14
                rc = 29
                print('testing layer 1...')
            elif layer == 2:
                dc = 31
                rc = 19
                print('testing layer 2...')
            elif layer == 3:
                dc = 21
                rc = 3
                print('testing layer 3...')
            #reset network IP
            reset()
            #update weights
            weights = []
            firstWeight = random.randint(-127,128)
            print('loading weights...')
            weights.append(firstWeight)
            writeWeight(firstWeight, 0)
            update(1)
            for i in range(1, 1007):
                weight = random.randint(-127,128)
                writeWeight(weight, i)
                weights.append(weight)
            update(0)
            print('done loading weights...')
```

```
    #Input data to network
    selectLayer(layer)
    inputData = []
    firstData = random.randint(-2047, 2048)
    inputData.append(firstData)
    print('loading data...')
    writeData(firstData, 0)
    write(1)
    for i in range(1, dc):
        data = random.randint(-2047, 2048)
        writeData(data, i)
        inputData.append(data)
    print('inputs: ', inputData)
    print('done loading data...')
    #begin computation
    print('beginning computation...')
    start(1)
    time.sleep(.3)
    #collect output data
    outData = []
    for i in range(rc):
        outD = read()
        outData.append(outD)
    print('output: ', outData)

    #Test functionality of network IP core
def opTest(layer):
    if layer == 1:
        dc = 14
        rc = 29
        print('testing layer 1...')
    elif layer == 2:
        dc = 31
        rc = 19
        print('testing layer 2...')
    elif layer == 3:
        dc = 21
        rc = 3
        print('testing layer 3...')
    #reset network IP
    reset()
    #update weights
    weights = getWeightStream()
    print('loading weights...')
    writeWeight(weights[0], 0)
    update(1)
    for i in range(1, len(weights)):
        writeWeight(weights[i], i)
```

```
    update(0)
    print('done loading weights...')
    #Input data to network
    selectLayer(layer)
    inputData = []
    firstData = random.randint(-800, 800)
    inputData.append(firstData)
    print('loading data...')
    writeData(firstData, 0)
    write(1)
    for i in range(1, dc):
        data = random.randint(-800, 800)
        writeData(data, i)
        inputData.append(data)
    print('inputs: ', inputData)
    print('done loading data...')
    #begin computation
    print('beginning computation...')
    start(1)
    time.sleep(.3)
    #collect output data
    outData = []
    for i in range(rc):
        outD = read()
        outData.append(outD)
    print('output: ', outData)


#Secondary operational test function: uniform inputs and weights
def opTest1():
    #reset network IP
    reset()
    #update weights
    #loadWeights()
    selectLayer(1)
    time.sleep(.1)
    writeData(1, 0)
    write(1)
    for i in range(1, 14):
        writeData(1, i)
    #begin computation
    start(1)
    time.sleep(.3)
    if done() == True:
        #collect output data
        for i in range(29):
            outD = read()
            print(outD)
```

## 13.2 Data Preprocesing

# Data Processing

May 15, 2018

### 0.1 Imports & Function Declaration

In [2]:
```
import pip
import os
import numpy as np
import scipy
import matplotlib.pyplot as plt; plt.rcdefaults()
from python_speech_features import mfcc
from python_speech_features import logfbank
from python_speech_features import fbank
import scipy.io.wavfile as wav
import pandas
from pandas import DataFrame
import time
import random
import numpy
import fnmatch

def plotMFCC(mfccArray):
    data = (mfccArray)
    newdata = np.squeeze(data) # Shape is now: (10, 80)
    plt.plot(data) # plotting by columns
    plt.ylabel('')
    #plt.suptitle('')
    plt.show()

def getListOfRandNums(inMfcc, n):
    #indexList = []
    #for i in range(n):
    randList = random.sample(range(len(mfcc_data1)), n)
    return(randList)

def selectRandomFromArray(inMfcc, indexList, numFilters):
    #print(indexList(0))
    L = len(indexList)
    testArray = numpy.zeros(shape=(L ,numFilters))
    for i in range(L):
```

```
            testArray[i] = inMfcc[indexList[i]]
        return(testArray)
    print("Done")
```

```
Done
```

## 0.2 Generate MFCCs for Training

```
In [3]: start_time = time.clock()
        print('Working...')
        # Initialize Vectors.
        l1,m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, l2 =
        [], [], [], [], [], [], [], [], [], [], [], [], [], []
        # Set directory.
        directory_str = "tomAnt_DataSet - Copy"
        directory = os.fsencode(directory_str)
        # Loop for each file in the directory:
        for file in os.listdir(directory):
            # Get file name.
            filename = os.fsdecode(file)
            # Look for .wav files
            if filename.endswith(".wav"):
                # Create file path with directory string and filename.
                path = (os.path.join(directory_str, filename))
                # Create MFCC.
                (rate,sig) = wav.read(path)
                mfcc_data1 = mfcc(sig,rate, nfft=2048, winlen=.03)
                if fnmatch.fnmatch(filename, 'tom (*).wav') or fnmatch.fnmatch
                (filename, 'ant (*).wav'):
                    pass
                    # Select 20 random MFCCs arrays.
                    #randomList = getListOfRandNums(mfcc_data1, 38)
                    #mfcc_data1 = selectRandomFromArray(mfcc_data1, randomList, 13)
                else:
                    pass
                    # Select 20 random MFCCs arrays.
                    #randomList = getListOfRandNums(mfcc_data1, 5)
                    #mfcc_data1 = selectRandomFromArray(mfcc_data1, randomList, 13)
                # Populate the local lists and create data frame.
                for i in range(len(mfcc_data1)):
                    l1.append(filename)
                    m1.append(mfcc_data1[i][0])
                    m2.append(mfcc_data1[i][1])
                    m3.append(mfcc_data1[i][2])
                    m4.append(mfcc_data1[i][3])
                    m5.append(mfcc_data1[i][4])
                    m6.append(mfcc_data1[i][5])
```

```
                    m7.append(mfcc_data1[i][6])
                    m8.append(mfcc_data1[i][7])
                    m9.append(mfcc_data1[i][8])
                    m10.append(mfcc_data1[i][9])
                    m11.append(mfcc_data1[i][10])
                    m12.append(mfcc_data1[i][11])
                    m13.append(mfcc_data1[i][12])
                    if fnmatch.fnmatch(filename, 'tom (*).wav'):
                        l2.append("tom")
                    elif fnmatch.fnmatch(filename, 'ant (*).wav'):
                        l2.append("ant")
                    else:
                        l2.append("other")
                #df = DataFrame({'MFCC': l2})
                df = DataFrame({'File Name': l1, 'Total Energy': m1, 'MFCC2': m2,
                                'MFCC3': m3, 'MFCC4': m4, 'MFCC5': m5, 'MFCC6': m6,
                                'MFCC7': m7, 'MFCC8': m8, 'MFCC9': m9, 'MFCC10': m10,
                                'MFCC11': m11, 'MFCC12': m12, 'MFCC13': m13,
                                'Speaker': l2})
                df = df[['File Name', 'Total Energy', 'MFCC2', 'MFCC3', 'MFCC4',
                         'MFCC5', 'MFCC6','MFCC7', 'MFCC8', 'MFCC9', 'MFCC10',
                         'MFCC11', 'MFCC12', 'MFCC13', 'Speaker']]
                #display(df)
                continue
            else:
                continue
        #display(df)
        # Convert data frame to spreadsheet. Print time elapsed and shape.
        df.to_excel('mfcc_training.xlsx', sheet_name='sheet1', index=False)
        print("--- %s seconds ---" % (time.clock() - start_time))
        print("MFCC Output of shape:", df.shape)
```

```
Working...
--- 4.748258039568876 seconds ---
MFCC Output of shape: (14956, 15)
```

### 0.3 Generate MFCCs for Testing

```
In [4]: start_time = time.clock()
        print('Working...')
        l1,m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, l2 =
        [], [], [], [], [], [], [], [], [], [], [], [], [], [], []
        # Set directory.
        directory_str = "Testing_DataSet"
        directory = os.fsencode(directory_str)
        # Loop for each file in the directory:
        for file in os.listdir(directory):
```

```
# Get file name.
filename = os.fsdecode(file)
# Look for .wav files
if filename.endswith(".wav"):
    # Create file path with directory string and filename.
    path = (os.path.join(directory_str, filename))
    # Create MFCC.
    (rate,sig) = wav.read(path)
    mfcc_data1 = mfcc(sig,rate, nfft=1535)

    if fnmatch.fnmatch(filename, 'tom (*).wav') or fnmatch.fnmatch
    (filename, 'ant (*).wav'):
        pass
    else:
        # Select 20 random MFCCs arrays.
        #randomList = getListOfRandNums(mfcc_data1, 20)
        #mfcc_data1 = selectRandomFromArray(mfcc_data1, randomList, 13)
    # Populate the local lists and create data frame.
    for i in range(len(mfcc_data1)):
        l1.append(filename)
        m1.append(mfcc_data1[i][0])
        m2.append(mfcc_data1[i][1])
        m3.append(mfcc_data1[i][2])
        m4.append(mfcc_data1[i][3])
        m5.append(mfcc_data1[i][4])
        m6.append(mfcc_data1[i][5])
        m7.append(mfcc_data1[i][6])
        m8.append(mfcc_data1[i][7])
        m9.append(mfcc_data1[i][8])
        m10.append(mfcc_data1[i][9])
        m11.append(mfcc_data1[i][10])
        m12.append(mfcc_data1[i][11])
        m13.append(mfcc_data1[i][12])
        if fnmatch.fnmatch(filename, 'tom (*).wav'):
            l2.append("tom")
        elif fnmatch.fnmatch(filename, 'ant (*).wav'):
            l2.append("ant")
        else:
            l2.append("other")
    #df = DataFrame({'MFCC': l2})
    df = DataFrame({'File Name': l1, 'Total Energy': m1, 'MFCC2': m2,
                    'MFCC3': m3, 'MFCC4': m4, 'MFCC5': m5, 'MFCC6': m6,
                    'MFCC7': m7, 'MFCC8': m8, 'MFCC9': m9, 'MFCC10': m10,
                    'MFCC11': m11, 'MFCC12': m12, 'MFCC13': m13,
                    'Speaker': l2})
    df = df[['File Name', 'Total Energy', 'MFCC2', 'MFCC3', 'MFCC4',
             'MFCC5', 'MFCC6','MFCC7', 'MFCC8', 'MFCC9', 'MFCC10',
             'MFCC11', 'MFCC12', 'MFCC13', 'Speaker']]
```

```
                #display(df)
                continue
            else:
                continue
        #display(df)
        # Convert data frame to spreadsheet. Print time elapsed and shape.
        df.to_excel('mfcc_testing.xlsx', sheet_name='sheet1', index=False)
        print("--- %s seconds ---" % (time.clock() - start_time))
        print("MFCC Output of shape:", df.shape)
```

```
Working...
--- 6.217475005247991 seconds ---
MFCC Output of shape: (3253, 15)
```

### 0.4 Generate Filter Bank Output Excel Sheet

In [ ]:
```
#Test
(rate,sig) = wav.read("tom4sec.wav")
fbank_data = logfbank(sig,rate, nfft=1535,nfilt=40)
numpy.set_printoptions(threshold=numpy.nan)
print(fbank_data)
print(fbank_data.shape)
```

In [ ]:
```
start_time = time.clock()
n0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13 =
[], [], [], [], [], [], [], [], [], [], [], [], [], []
m14, m15, m16, m17, m18, m19, m20, m21, m22, m23, m24, m25, m26 =
[], [], [], [], [], [], [], [], [], [], [], [], []
m27, m28, m29, m30, m31, m32, m33, m34, m35, m36, m37, m38, m39, m40 =
[], [], [], [], [], [], [], [], [], [], [], [], [], []


directory_str = "an4_all"
directory = os.fsencode(directory_str)
# Loop for each file in the directory:
for file in os.listdir(directory):
    # Get file name.
    filename = os.fsdecode(file)
    # Look for .wav files
    if filename.endswith(".wav"):
        # Create file path with directory string and filename.
        path = (os.path.join(directory_str, filename))
        # Create MFCC.
        (rate,sig) = wav.read(path)
        fbank_data = logfbank(sig,rate,nfft=1536,nfilt=40)

        randomList = getListOfRandNums(fbank_data, 20)
```

```
fbank_data = selectRandomFromArray(fbank_data, randomList, 40)

for i in range(len(fbank_data)):
    n0.append(filename)
    m1.append(fbank_data[i][0])
    m2.append(fbank_data[i][1])
    m3.append(fbank_data[i][2])
    m4.append(fbank_data[i][3])
    m5.append(fbank_data[i][4])
    m6.append(fbank_data[i][5])
    m7.append(fbank_data[i][6])
    m8.append(fbank_data[i][7])
    m9.append(fbank_data[i][8])
    m10.append(fbank_data[i][9])
    m11.append(fbank_data[i][10])
    m12.append(fbank_data[i][11])
    m13.append(fbank_data[i][12])
    m14.append(fbank_data[i][13])
    m15.append(fbank_data[i][14])
    m16.append(fbank_data[i][15])
    m17.append(fbank_data[i][16])
    m18.append(fbank_data[i][17])
    m19.append(fbank_data[i][18])
    m20.append(fbank_data[i][19])
    m21.append(fbank_data[i][20])
    m22.append(fbank_data[i][21])
    m23.append(fbank_data[i][22])
    m24.append(fbank_data[i][23])
    m25.append(fbank_data[i][24])
    m26.append(fbank_data[i][25])
    m27.append(fbank_data[i][26])
    m28.append(fbank_data[i][27])
    m29.append(fbank_data[i][28])
    m30.append(fbank_data[i][29])
    m31.append(fbank_data[i][30])
    m32.append(fbank_data[i][31])
    m33.append(fbank_data[i][32])
    m34.append(fbank_data[i][33])
    m35.append(fbank_data[i][34])
    m36.append(fbank_data[i][35])
    m37.append(fbank_data[i][36])
    m38.append(fbank_data[i][37])
    m39.append(fbank_data[i][38])
    m40.append(fbank_data[i][39])

df = DataFrame({'File Name': n0, 'FB1': m1, 'FB2': m2, 'FB3': m3,
                'FB4': m4, 'FB5': m5, 'FB6': m6, 'FB7': m7,
                'FB8': m8, 'FB9': m9, 'FB10': m10, 'FB11': m11,
```

```
                                'FB12': m12, 'FB13': m13, 'FB14': m14, 'FB15': m15,
                                'FB16': m16, 'FB17': m17, 'FB18': m18, 'FB19': m19,
                                'FB20': m20, 'FB21': m21, 'FB22': m22, 'FB23': m23,
                                'FB24': m24, 'FB25': m25, 'FB26': m26,'FB27': m27,
                                'FB28': m28,'FB29': m29,'FB30': m30,'FB31': m31,
                                'FB32': m32,'FB33': m33,'FB34': m34,'FB35': m35,
                                'FB36': m36,'FB37': m37,'FB38': m38, 'FB39': m39,
                                'FB40': m40 })
                    df = df[['File Name','FB1','FB2','FB3','FB4','FB5','FB6','FB7',
                             'FB8','FB9','FB10','FB11','FB12','FB13','FB14','FB15',
                             'FB16','FB17','FB18','FB19','FB20','FB21','FB22','FB23',
                             'FB24','FB25','FB26','FB27','FB28','FB29','FB30','FB31',
                             'FB32','FB33','FB34','FB35','FB36','FB37','FB38','FB39',
                             'FB40']]
                    #display(df)
                    continue
                else:
                    continue

        display(df)
        df.to_excel('filterBankLOutput.xlsx', sheet_name='sheet1', index=False)
        print("--- %s seconds ---" % (time.clock() - start_time))
        print("MFCC Output of shape:", df.shape)

In [ ]: for i in range(len(fbank_data)):
                    l1.append(filename)
                    m1.append(fbank_data[i][0])
                    m2.append(fbank_data[i][1])
                    m3.append(fbank_data[i][2])
                    m4.append(fbank_data[i][3])
                    m5.append(fbank_data[i][4])
                    m6.append(fbank_data[i][5])
                    m7.append(fbank_data[i][6])
                    m8.append(fbank_data[i][7])
                    m9.append(fbank_data[i][8])
                    m10.append(fbank_data[i][9])
                    m11.append(fbank_data[i][10])
                    m12.append(fbank_data[i][11])
                    m13.append(fbank_data[i][12])
                #df = DataFrame({'MFCC': l2})
                df = DataFrame({'File Name': l1, 'Total Energy': m1, 'MFCC2': m2,
                                'MFCC3': m3, 'MFCC4': m4, 'MFCC5': m5, 'MFCC6': m6,
                                'MFCC7': m7, 'MFCC8': m8, 'MFCC9': m9, 'MFCC10': m10,
                                'MFCC11': m11, 'MFCC12': m12, 'MFCC13': m13})
                df = df[['File Name', 'Total Energy', 'MFCC2', 'MFCC3', 'MFCC4',
                         'MFCC5', 'MFCC6','MFCC7', 'MFCC8', 'MFCC9', 'MFCC10',
                         'MFCC11', 'MFCC12', 'MFCC13']]
                #display(df)
```

```
            continue
        else:
            continue

    #display(df)
    df.to_excel('mfccOutput.xlsx', sheet_name='sheet1', index=False)
    print("--- %s seconds ---" % (time.clock() - start_time))
    print("MFCC Output of shape:", df.shape)
```

In [ ]:
```
import numpy
#Input: A 2x numpy array, amount of random samples to select from the array.
def getListOfRandNums(inMfcc, n):
    #indexList = []
    #for i in range(n):
    randList = random.sample(range(len(mfcc_data1)), n)
    return(randList)

def selectRandomFromArray(inMfcc, indexList):
    #print(indexList(0))
    L = len(indexList)
    testArray = numpy.zeros(shape=(L ,13))
    for i in range(L):
        testArray[i] = inMfcc[indexList[i]]
    return(testArray)


randomList = getListOfRandNums(mfcc_data1, 20)
newMfcc = selectRandomFromArray(mfcc_data1, randomList)
```

In [ ]:
```
#REDACTED CODE

#try:
#    import python_speech_features
#except:
#    import pip
#    pip.main(['install','python_speech_features'])
#    import python_speech_features

#print(fbank_feat[1:3,:])

#fig, ax = plt.subplots()
#mfcc_data= np.swapaxes(mfcc_data, 0 ,1)
#cax = ax.imshow(mfcc_data, interpolation='nearest', cmap=cm.coolwarm,
#origin='lower')
#ax.set_title('MFCC')

#plt.show()
```

```
#(rate,sig) = wav.read("ant1.wav")
#mfcc_data = mfcc(sig,rate, nfft=1536)
#mfcc_feat = mfcc(sig,rate,nfft=1536)
##fbank_feat = logfbank(sig,rate)

#print(fbank_feat[1:3,:])

#print(mfcc_feat[1:13])#807
#plotMFCC(mfcc_feat[1:13])




#directory_str = "an4_clstk/ffmm/"
#directory = os.fsencode(directory_str)
#
#for file in os.listdir(directory):
#    filename = os.fsdecode(file)
#    if filename.endswith(".wav"):
#        #fileNameList.append(filename)
#        path = (os.path.join(directory_str, filename))
#        #print(path)
#
#        # Create MFCC, Print & Plot
#        (rate,sig) = wav.read(path)
#        mfcc_data1 = mfcc(sig,rate, nfft=1536)
#        #plotMFCC(mfcc_data1)
#        table= mfccToTable(filename, mfcc_data1)
#        continue
#    else:
#        continue
#    #result = pd.concat(table)
#    table.to_excel('test.xlsx', sheet_name='sheet1', index=False)
```

```
#def mfccToTable(inFileName, mfcc_data1):
#    m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13 =
# [], [], [], [], [], [], [], [], [], [], [], [], []
#
#    for i in range(len(mfcc_data1)):
#        l1 = inFileName
#        m1.append(mfcc_data1[i][0])
#        m2.append(mfcc_data1[i][1])
#        m3.append(mfcc_data1[i][2])
#        m4.append(mfcc_data1[i][3])
#        m5.append(mfcc_data1[i][4])
#        m6.append(mfcc_data1[i][5])
#        m7.append(mfcc_data1[i][6])
#        m8.append(mfcc_data1[i][7])
#        m9.append(mfcc_data1[i][8])
#        m10.append(mfcc_data1[i][9])
#        m11.append(mfcc_data1[i][10])
#        m12.append(mfcc_data1[i][11])
#        m13.append(mfcc_data1[i][12])
#
#    #df = DataFrame({'MFCC': l2})
#    df = DataFrame({'File Name': l1, 'MFCC1': m1, 'MFCC2': m2, 'MFCC3': m3,
#    'MFCC4': m4, 'MFCC5': m5, 'MFCC6': m6, 'MFCC7': m7, 'MFCC8': m8,
#    'MFCC9': m9, 'MFCC10': m10, 'MFCC11': m11, 'MFCC12': m12, 'MFCC13': m13})
#    df = df[['File Name', 'MFCC1', 'MFCC2', 'MFCC3', 'MFCC4', 'MFCC5',
#    'MFCC6','MFCC7', 'MFCC8', 'MFCC9', 'MFCC10', 'MFCC11', 'MFCC12',
#    'MFCC13']]
#    #display(df)
#
#    return df




#def averageMFCC(inMFCC):
#    avgMFCC = []
#
#    for k in range(14):
#        avgMFCC.append(0)
#
#    for i in range(13):
#        for j in range(13):
#            avgMFCC[i] += inMFCC[i][j]
#    for i in range(13):
#            avgMFCC[i] = avgMFCC[i] / len(avgMFCC)
#    print(avgMFCC)
#
```

### 13.3 Preliminary Project Report


# Adaptive Linear Element

| Authors: | Professor: |
| --- | --- |
| Anthony Carvino | Dr. Yi-Chung Chen |
| Thomas Coppola | |

22 November 2017

**Abstract**

The sector of engineering pertaining to Artificial Neural Networks (ANN) stands to benefit greatly from System on Chip (SoC) technology. Large ANNs require the execution of an immense number of computations involving many parameters: ideally, these parameters would be widely distributed throughout the system in order to facilitate maximal parallelism and performance; however, ANNs are typically implemented using conventional programming languages on traditional processing systems, thus imposing a serial nature upon a variety of system that is particularly suited to benefit from parallel implementation—as a result of this imposition, performance suffers. SoCs featuring an integrated Field Programmable Gate Array (FPGA) are ideal systems for the implementation and utilization of high performance, high throughput, low latency, low power ANNs. The network can be developed as a hardware system, downloaded to the programmable logic of the SoC, and then interfaced with using a high-level language such as C, C++, or Python. This project utilized the PYNQ-Z1 development board (which features a ZYNQ-7020 SoC), in combination with the Vivado Design Suite, in the design and hardware implementation of a simple Adaptive Linear Element (Adaline). Also developed was Python code to interact with the module, train it on the task of classifying the popular Iris Dataset, and test its learned capacity to generalize to new data. The Jupyter Notebook Python environment served by the processing system of the PYNQ-Z1 was particularly well suited to our application, and our system learned to classify the isolated testing dataset with an accuracy of 100% in as few as 5 training epochs (using Stochastic Gradient Descent); however, the system became significantly more robust as the number of training epochs was increased. Though we present only a simple hardware system applied to a trivial task, our team feels that the success of this introductory project demonstrates the potential of FPGA SoC technology to revolutionize the implementation of ANNs.

# Contents

# 1 The Adaptive Linear Element

The Adaptive Linear Element, or Adaline, developed by Widrow and Hoff in 1960, is an improvement on the original Perceptron invented by Frank Rosenblatt in 1957. The difference between the Perceptron and the Adaline is that the Perceptron utilizes a unit-step transfer function so that its output is binary (-1 or 1, indicating one of two possible classes), whereas the Adaline utilizes a linear identity transfer function: that is, its output is a real number, and is thus differentiable, enabling the use of the gradient descent training algorithm. This difference is visualized in Figure 1, below [1].

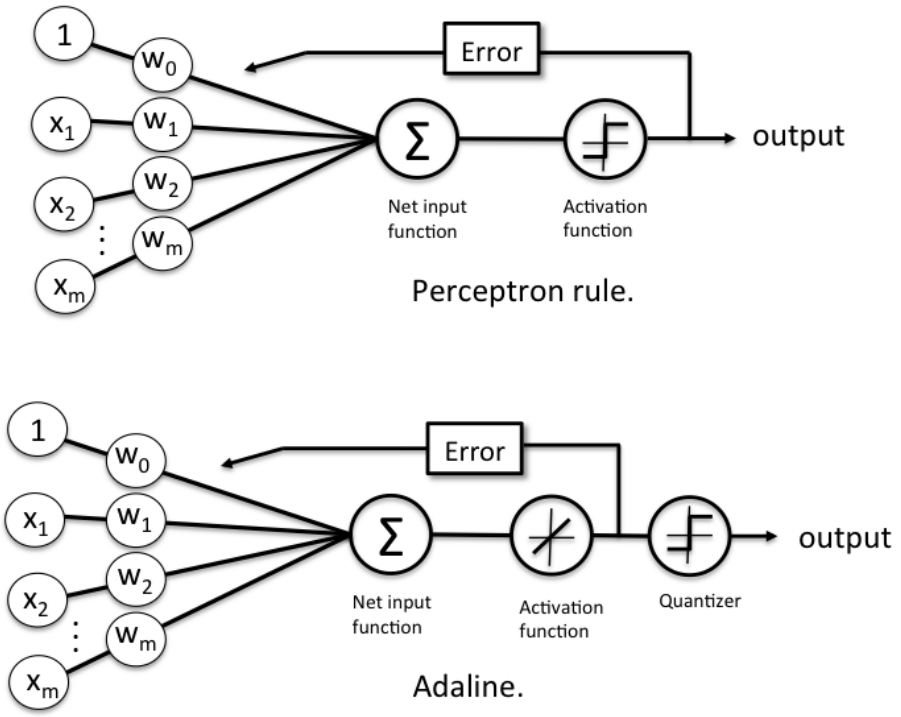


Figure 1: Perceptron vs Adaptive Linear Element

# 2 The Iris Dataset

The Isis Dataset is a popular machine learning dataset that contains 150 data samples, with 50 samples for each of three species of Iris flower. Each sample consists of four numerical features (sepal length, sepal width, petal length, and petal width) as well as a class label (Iris setosa, versicolor, or virginica). The data corresponding to Iris setosa is linearly separable from the versicolor and virginica data, but the versicolor samples are not linearly separable from the virginica samples. Since the Adaline is a linear classifier, we decided to exclude Iris virginica from our experiment. Further, we also decided to exclude the sepal width and petal length features, as the sepal width feature has a very poor, in fact negative class correlation (-0.4194), while the petal length feature is less correlated (albeit marginally so) than the petal width feature(0.9490 vs 0.9565); we selected the sepal length because of its good but not excellent class correlation (0.7826). The complete Iris Dataset (using the two features we selected) is graphed in Figure 2, below [2].

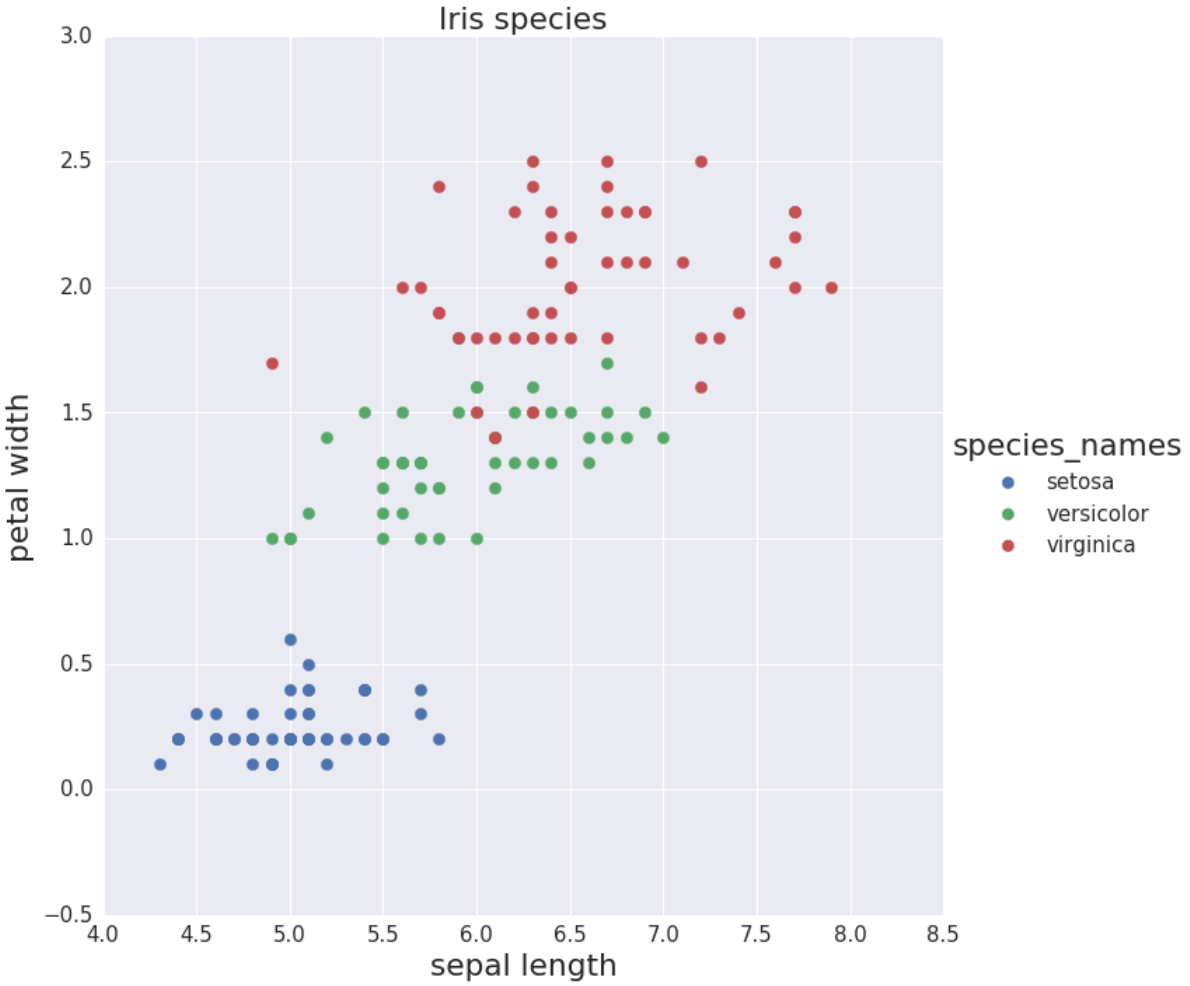


Figure 2: Iris Dataset; petal width vs sepal length; data in cm [3].

# 3 The Hardware Design Process

In order to classify the Iris Dataset in accordance with our approach, we require a single hardware node that is capable of interfacing with a software environment. This node was created and packaged using the Xilinx Vivado Design Suite 2017.2 Edition. The concern might be raised that implementing the Adaline in hardware is unnecessary and, indeed, counterintuitive, as the computational unit of the Adaline could be easily implemented in software with several lines of code directing simple arithmetic; however, the point of this project is not to classify the dataset (which could be done with a simple if-else statement, or, as discussed, with a software implemented Adaline), but rather our purpose is to demonstrate the potential of FPGA SoC technology for implementing efficient ANNs, and to act as a stepping stone for the implementation of larger hardware ANNs.

## 3.1 Xilinx Vivado High Level Synthesis

The behavior of the hardware module was specified in C++ using Vivado HLS. It is a simple multiply-add operation:

$$w\_sum = sl(sl\_w) + pw(pw\_w)$$

Where *w_sum* is the weighted sum, and *sl, sl_w, pw,* and *pw_w* are the sepal length, sepal length weight, petal width, and petal width weight, respectively.

Vivado HLS synthesizes the high-level description into RTL, both Verilog and VHDL. HLS also permits the specification of an AXI-4 slave interface, so that the inputs and outputs can be memory mapped and written to/read from later on. Once everything has been specified and the active solution is synthesized, the RTL is ready to be packaged as an IP and exported to Vivado. Figure 3, below, shows the C++ source as well as its directives; the testbench code as well as the HLS utilization estimate can be seen in Figures 4 and 5, respectively.

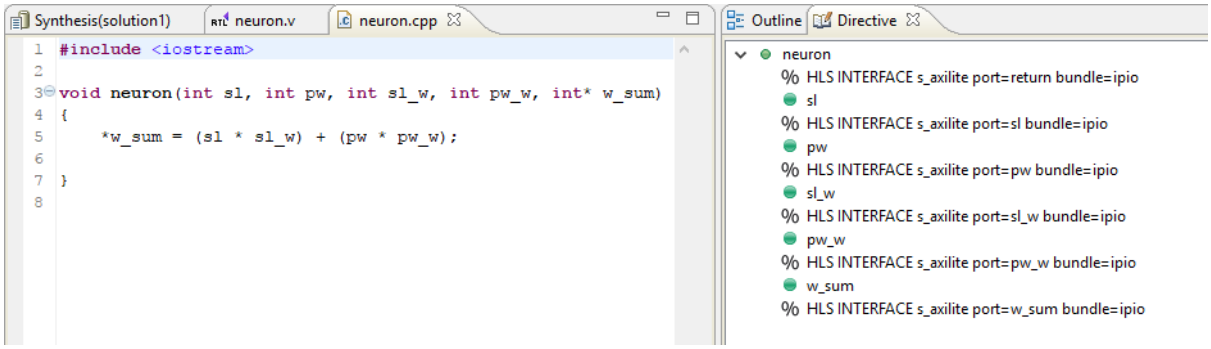


Figure 3: Adaline Source Code and Interface Directives

```
1  #include <iostream>
2
3  using namespace std;
4
5  void neuron(int sl, int pw, int sl_w, int pw_w, int* w_sum);
6
7  int main()
8  {
9      int sl = 51;
10     int pw = 2;
11     int sl_w = 1;
12     int pw_w = 1;
13     int w_sum;
14     neuron(sl, pw, sl_w, pw_w, &w_sum);
15     cout << w_sum << endl;
16     return 0;
17 }
18
```

Figure 4: Adaline Testbench Code

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 101 | 37 |
| FIFO | - | - | - | - |
| Instance | 0 | 8 | 556 | 460 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 21 |
| Register | - | - | 67 | - |
| Total | 0 | 8 | 724 | 518 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 3 | ~0 | ~0 |

Figure 5: Vivado HLS Utilization Estimate

## 3.2 Vivado and IP Integrator

Once the IP has been exported from HLS, it can be added to the IP Integrator catalog and instantiated on the block diagram canvas. Once this has been done, the ZYNQ processing system is likewise instantiated, and block/connection automation is executed in order to prepare and link all IPs together properly. In order to proceed to the PYNQ-Z1 environment, two files and two pieces of information are required: the bitstream file, the Tcl file, the offset address and range of the IP in memory, and the offsets for the inputs, output, and the ap_control bits. The system block diagram can be seen in Figure 6, and the address information can be seen in Figures 7 and 8. Once the files and address information are obtained, it is time to transition to the PYNQ environment.

In order to generate the bitstream, the design must first be synthesized and implemented. Once implementation has been performed, we can examine power, timing, and utilization reports. These reports can be seen in Figures 9, 10, and 11, respectively.

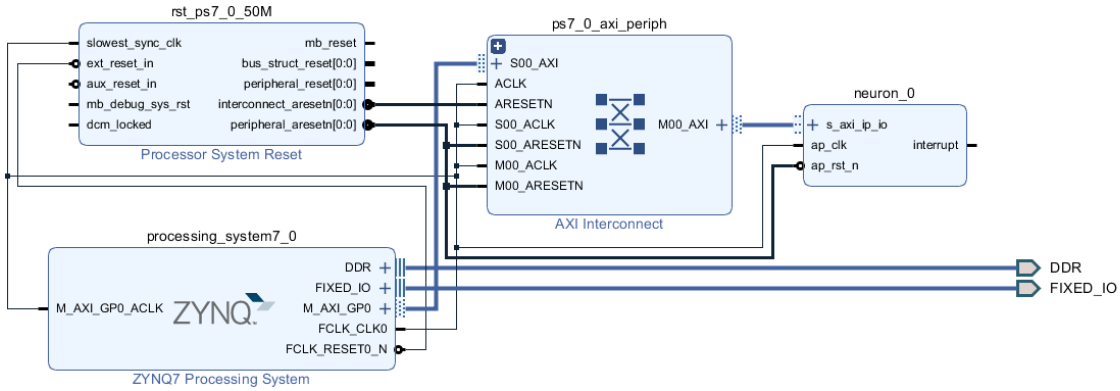


Figure 6: Vivado IP Integrator Block Diagram of our System

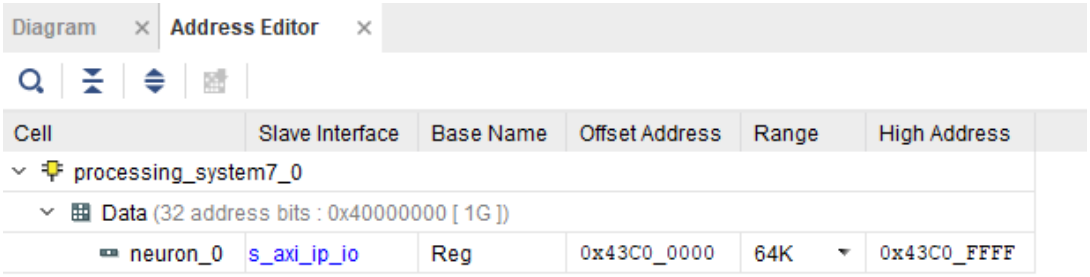


Figure 7: Vivado Address Editor

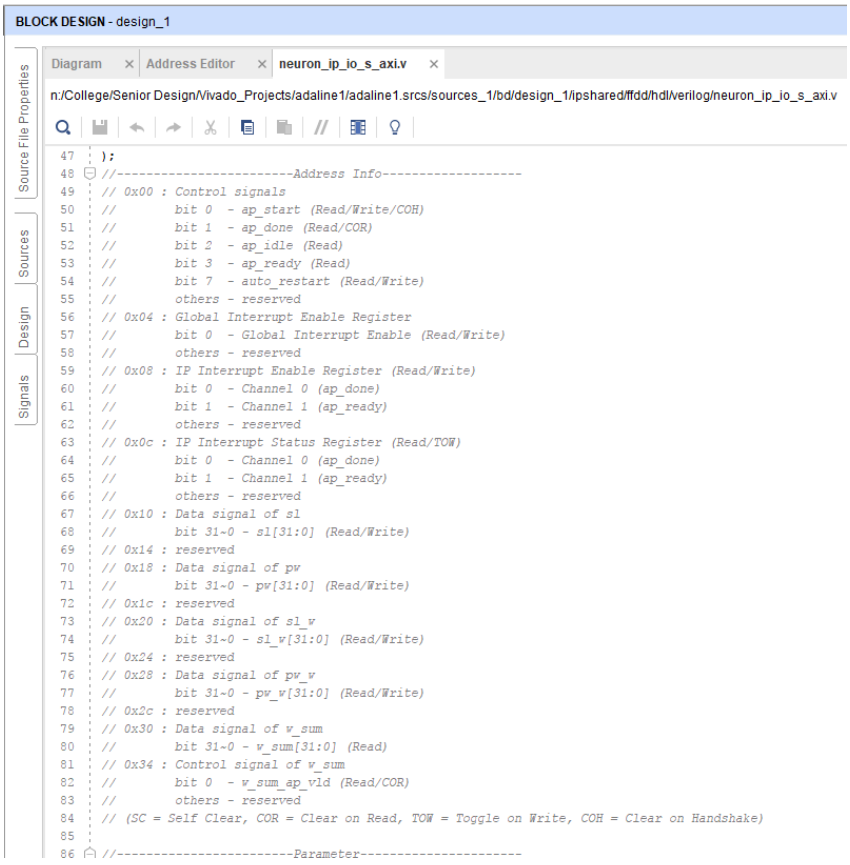


Figure 8: Address Information for Memory Mapped IO

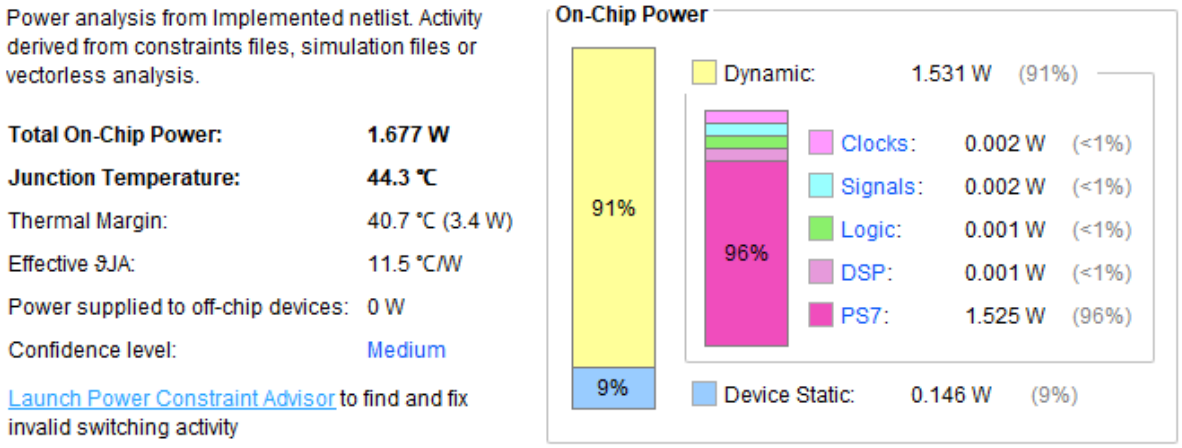


Figure 9: Vivado Power Report

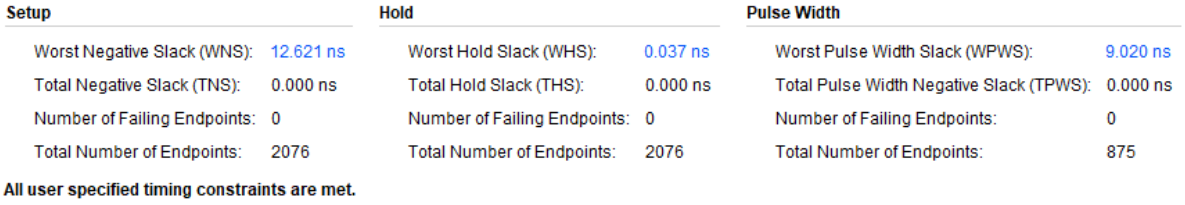


Figure 10: Vivado Timing Report

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 596 | 53200 | 1.12 |
| LUTRAM | 62 | 17400 | 0.36 |
| FF | 804 | 106400 | 0.76 |
| DSP | 6 | 220 | 2.73 |
| BUFG | 1 | 32 | 3.13 |

Figure 11: ZYNQ PL Resource Utilization

# 4 The PYNQ-Z1 Development Environment

The PYNQ-Z1 Development Board is a project from Xilinx, and is built around the ZYNQ-7020 System on Chip, which features a dual core ARM Cortex-A9 Processing System (PS), and a Programmable Logic (PL) system with 85k logic cells, 140 embedded 36Kb block RAMs, and 220 embedded DSP48 slices [4]. The PYNQ serves a Python Jupyter Notebook environment from the ZYNQ PS, and provides resources, documentation, and software libraries designed to support productivity. Within the Jupyter Notebook framework of the PYNQ environment, we created Python code to control, interact with, train, and test the Adaline.

## 4.1 Instantiation and Customization

The first step is to download an instance of the hardware module onto the PL of the ZYNQ chip by specifying the bitstream. When the bitstream is indicated, the built in

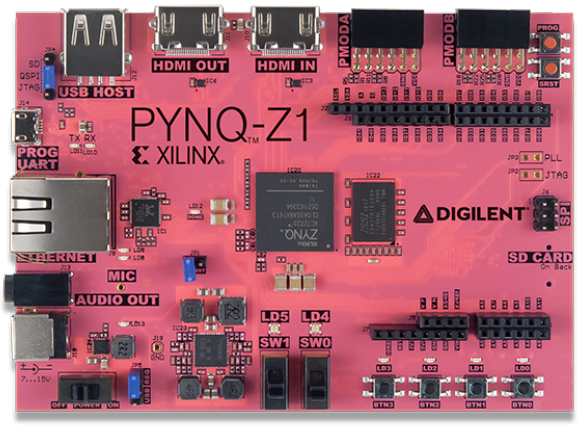


Figure 12: The PYNQ-Z1 Board

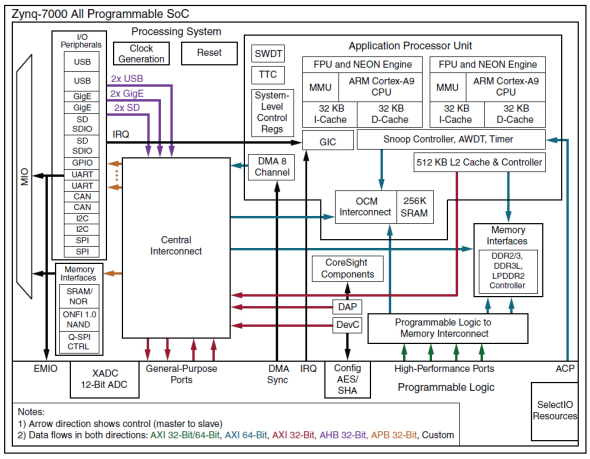


Figure 13: ZYNQ-7000 Diagram [4]

loading functions will find and parse the Tcl file—it is necessary for the .bit and .tcl files to have the same name. The code to specify and load the bitstream onto the board can be seen in Figure 14, below.

```
In [3]: from pynq import Overlay
        from pynq import MMIO
        ol = Overlay("/home/xilinx/jupyter_notebooks/AdaptiveLinearNeuron/adaline1.bit")
        ol.download()
        ip = MMIO(0x43C00000,0x10000)
```

Figure 14: Downloading the Bitstream to the PL

An Overlay object, *ol* is created and downloaded to the board using built in functions. A MMIO object called *ip* is created, and the address offset and range from Figure 7 are passed as arguments to the constructor.

After the node has been instantiated, it can be interacted with by writing bit or integer values to the address offset of the inputs and reading from the output offset address, as given in Figure 8. This method of interface is not very conducive to usability and versatility, so the next step is to create a custom driver to expose the interface as a typical Python function: this is done by adding the driver to the DefaultIP class and binding the function to the hardware node, as seen in Figure 15.

## 4.2 Training

Once the functionality of the hardware node is demonstrated and the *compute* function is defined, it is time to specify the dataset and the main code for the training phase. Three lists are defined, one containing the sepal length data, one holding the petal width data, and another containing true class labels to enable the utilization of the gradient descent algorithm. Also specified in this code block are the global variables *lr*, or learning rate, a value used to taper or exacerbate the magnitude of weight updates; *slWeight*, the sepal length weighting value; and *pwWeight*, the petal width weight. Typically the use of global variables is discouraged, but the unconventionality of neural elements and networks calls for the versatility of globally accessible and adjustable variables. Next, a number of training epochs is specified and the weighting values are initialized to a small, arbitrary

```
In [2]: from pynq import DefaultIP

        class AddDriver(DefaultIP):
            def __init__(self, description):
                super().__init__(description=description)

            bindto = ['xilinx.com:hls:neuron:1.0']

            def compute(self, sl, pw, sl_w, pw_w):
                self.write(0x10,sl)
                self.write(0x18,pw)
                self.write(0x20,sl_w)
                self.write(0x28,pw_w)
                self.write(0x00,1)
                return self.read(0x30)
```

Figure 15: Custom Driver Specification

value. An epoch, within the scope of this stochastic scheme, is defined as a period during which the entire training dataset is presented to the system in a random order, and the weighting values are updated slightly after each sample: thus, the weights change 80 times per epoch. Lastly, we iterate through a range up to the number of epochs in which we call the *train* function, to be discussed next, and print out status updates to the console.

```
In [8]: # Datasets for the training phase.
        # sl = sepal length, pw = petal width; data is in millimeters
        # iris type: 0 corresponds to iris versicolor, 100000 corresponds to iris setosa
        slTrain = [51,49,47,46,50,54,46,50,44,49,54,48,48,43,58,57,54,51,57,51,54,51,46,
                   51,48,50,50,52,52,47,48,54,52,55,49,50,55,49,44,51,70,64,69,55,65,57,
                   63,49,66,52,50,59,60,61,56,67,56,58,62,56,59,61,63,61,64,66,68,67,60,
                   57,55,55,58,60,54,60,67,63,56,55]

        pwTrain = [2,2,2,2,2,4,3,2,2,1,2,2,1,1,2,4,4,3,3,3,2,4,2,5,2,2,4,2,2,2,2,4,1,2,1,
                   2,2,1,2,2,14,15,15,13,15,13,16,10,13,14,10,15,10,14,13,14,15,10,15,11,
                   18,13,15,12,13,14,14,17,15,10,11,10,12,16,15,16,15,13,13,13]

        irisTypeTrain = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                         0,0,0,0,0,0,0,0,0,100000,100000,100000,100000,100000,100000,100000,
                         100000,100000,100000,100000,100000,100000,100000,100000,100000,
                         100000,100000,100000,100000,100000,100000,100000,100000,100000,
                         100000,100000,100000,100000,100000,100000,100000,100000,100000,
                         100000,100000,100000,100000,100000,100000]

        #training parameters and initial network parameters
        # lr = learning rate: used to taper or exacerbate the magnitude of weight updates
        # one epoch is one randomized pass through the training dataset
        global lr, slWeight, pwWeight

        lr = .00001
        epochs = 75
        slWeight = 100
        pwWeight = 100

        for i in range(0, epochs):
            print ("Epoch: ", i)
            train()
            print ("slWeight: ", slWeight)
            print ("pwWeight: ", pwWeight)
            print ("")
```

Figure 16: Main Code for the Training Phase

## 4.3 Function Definitions

The *train* function is called once per epoch, and performs the task of actually submitting each data sample to the hardware module: it randomly selects a data index, and then calls the *compute* function, passing the data corresponding to that index and the current weighting values as arguments. The output of the hardware element is stored in a variable, and this variable, as well as the randomly selected index, are passed to the *updateWeights* function.

The *updateWeights* function employs Stochastic Gradient Descent (SGD) by comparing the actual output to the target output, and calculating a change in weight value for each weight; the function then augments the global weight values by the calculated change in weight value.

Lastly, the processor returns from the *updateWeights* function to the *train* function, and is then directed to the *reportError* function, which calculates the sum of squared errors (SSE) and prints it to the console.

```
In [4]: import random
        #training function (one epoch)
        def train():
            outputList = []
            indexList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
                         15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
                         27, 28, 29,30, 31, 32, 33, 34, 35, 36, 37, 38,
                         39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                         51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
                         63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
                         75, 76, 77, 78, 79]
            random.shuffle(indexList)
            for i in range(0, 79):
                x = ol.neuron_0.compute(slTrain[indexList[i]], pwTrain[indexList[i]],
                                        slWeight, pwWeight)
                outputList.append(x)
                updateWeights(x, indexList[i])
            reportError(outputList, indexList)
            return

        #weight update function
        def updateWeights(outputVal, index):
            deltaSlWeight = lr * (irisTypeTrain[index] - outputVal) * slTrain[index]
            deltaPwWeight = lr * (irisTypeTrain[index] - outputVal) * pwTrain[index]
            global slWeight
            global pwWeight
            slWeight = slWeight + round(deltaSlWeight)
            pwWeight = pwWeight + round(deltaPwWeight)
            return

        #report error function
        def reportError(outputList, indexList):
            tempSum = 0
            for j in range(0, 79):
                x = (irisTypeTrain[indexList[j]] - outputList[j])
                tempSum = tempSum + (x * x)
            sumSquaredError = .5 * tempSum
            print ("SSE: ", sumSquaredError)
            return
```

Figure 17: Definition of Training, Weight Update, and Report Error Functions

## 4.4 Training Results

When the training code is executed, the *train* function runs for – in the most complete and convergent case – 75 epochs, reporting weights and SSE after each epoch. The SSE is, initially, incomprehensibly large, but drops dramatically as more epochs are completed. The weighting values for the sepal length and petal width begin to change drastically as well: the importance of the sepal length feature, being the less class correlated feature, is minimized over the 75 epochs, and the petal width, being highly correlated, is maximized. Data from several epochs is presented in Figure 18, and charts corresponding to the SSE, sepal length weight, and petal width weight are presented in Figures 19, 20, and 21, respectively.

```
Epoch:  0              Epoch:  35             Epoch:  72
SSE:  95201562375.0    SSE:  25433393937.5    SSE:  10767934986.5
slWeight:  873         slWeight:  336         slWeight:  -68
pwWeight:  383         pwWeight:  4435        pwWeight:  6791

Epoch:  1              Epoch:  36             Epoch:  73
SSE:  76053170582.0    SSE:  25190759217.5    SSE:  10296983119.5
slWeight:  908         slWeight:  286         slWeight:  -52
pwWeight:  553         pwWeight:  4513        pwWeight:  6835

Epoch:  2              Epoch:  37             Epoch:  74
SSE:  73735225682.0    SSE:  24421614126.0    SSE:  10818205579.0
slWeight:  867         slWeight:  279         slWeight:  -83
pwWeight:  712         pwWeight:  4604        pwWeight:  6868
```

Figure 18: Data from Selected Training Epochs

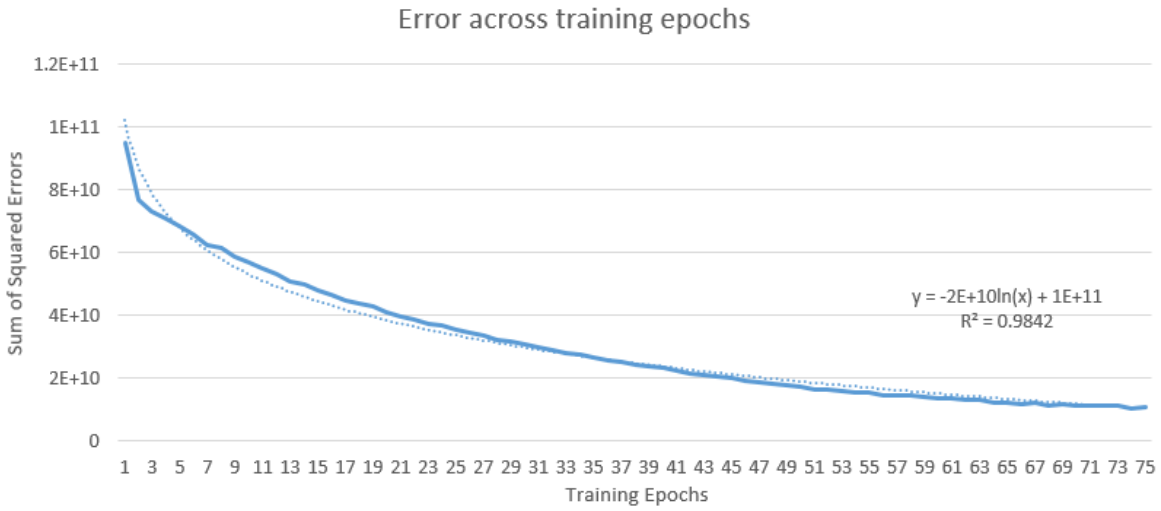


Figure 19: Sum of Squared Error Across 75 Epochs

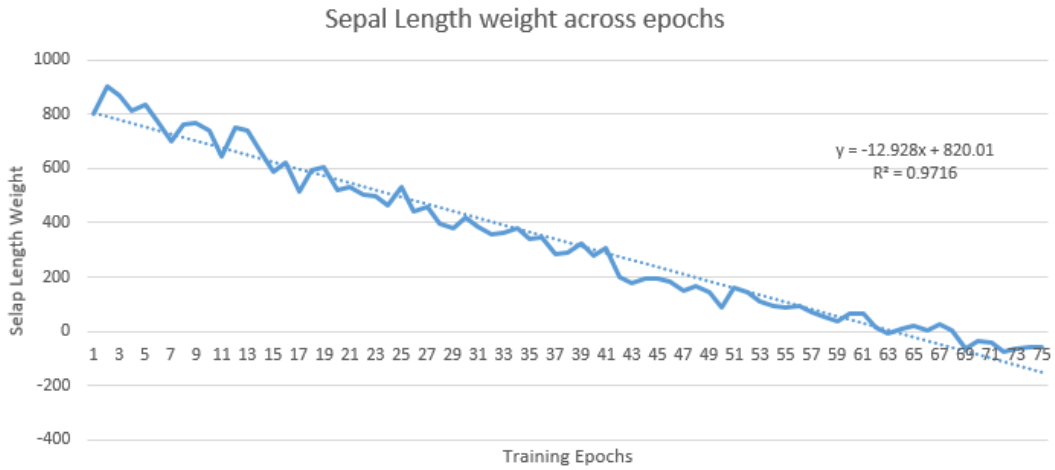


Figure 20: Sepal Length Weight over 75 Epochs

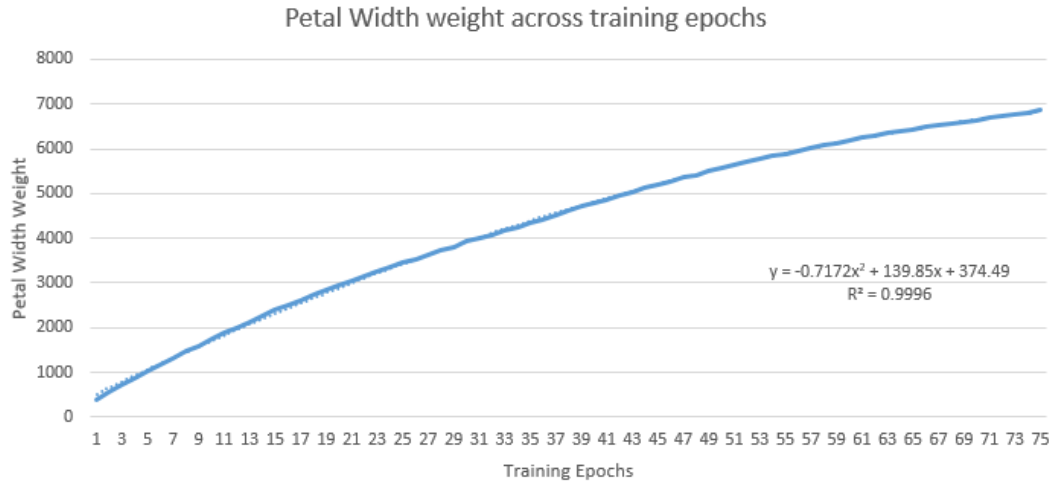


Figure 21: Petal Width Weight over 75 Epochs

# 5 Testing

Once the Adaline has been trained, it is ready to be tested for generalization: has the node only learned to classify data from the training set, or can it generalize, and classify data samples that it has never seen before? At the beginning of the coding process, we segmented the Iris dataset into a training set, containing 80 samples, and a testing dataset – to be isolated completely from the training data – consisting of 20 samples. For the testing phase, the testing dataset is randomized and submitted one-by-one to the Adaline, which outputs a value between 0 and 100,000: if the value is below 50,000, the system predicts Iris setosa; if the output is above 50,000, the system predicts versicolor. As can be seen in Figure 23, the Adaline predicts the correct Iris class with 100% accuracy. With 75 training epochs, the element becomes extremely robust and certain: that is, the output value is never anywhere close to the decision boundary of 50,000; however, our system has achieved 100% classification accuracy with as few as 5 training epochs, though its classification was often close to being incorrect.

```
In [11]: # Datasets for the testing phase.
         slTest = [50,45,44,50,51,48,51,46,53,50,55,61,58,50,56,57,57,62,51,57]
         pwTest = [3,3,2,6,4,3,2,2,2,2,12,14,12,10,13,12,13,13,11,13]
         irisTypeTest = [0,0,0,0,0,0,0,0,0,0,100000,100000,100000,100000,100000,
                         100000,100000,100000,100000,100000]

         indexList = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
         random.shuffle(indexList)
         for i in range(0, 19):
             x = ol.neuron_0.compute(slTest[indexList[i]], pwTest[indexList[i]],
                                     slWeight, pwWeight)
             if x > 50000:
                 prediction = "Iris Versicolor"
             elif x < 50000:
                 prediction = "Iris Setosa"
             irisType = irisTypeTest[indexList[i]]
             if irisType == 0:
                 trueClass = "Iris Setosa"
             elif irisType == 100000:
                 trueClass = "Iris Versicolor"
             print ("Data Index: ", indexList[i])
             print ("Prediction: ", prediction)
             print ("True Class: ", trueClass)
             print ("")
```

Figure 22: Main Code for the Testing Phase

```
Data Index:  3                Data Index:  12
Prediction:  Iris Setosa      Prediction:  Iris Versicolor
True Class:  Iris Setosa      True Class:  Iris Versicolor

Data Index:  9                Data Index:  16
Prediction:  Iris Setosa      Prediction:  Iris Versicolor
True Class:  Iris Setosa      True Class:  Iris Versicolor

Data Index:  17               Data Index:  2
Prediction:  Iris Versicolor  Prediction:  Iris Setosa
True Class:  Iris Versicolor  True Class:  Iris Setosa

Data Index:  18               Data Index:  14
Prediction:  Iris Versicolor  Prediction:  Iris Versicolor
True Class:  Iris Versicolor  True Class:  Iris Versicolor

Data Index:  11               Data Index:  19
Prediction:  Iris Versicolor  Prediction:  Iris Versicolor
True Class:  Iris Versicolor  True Class:  Iris Versicolor

Data Index:  4                Data Index:  5
Prediction:  Iris Setosa      Prediction:  Iris Setosa
True Class:  Iris Setosa      True Class:  Iris Setosa

Data Index:  13               Data Index:  15
Prediction:  Iris Versicolor  Prediction:  Iris Versicolor
True Class:  Iris Versicolor  True Class:  Iris Versicolor

Data Index:  1                Data Index:  6
Prediction:  Iris Setosa      Prediction:  Iris Setosa
True Class:  Iris Setosa      True Class:  Iris Setosa
```

Figure 23: Some Predictions from the Testing Phase

# 6 Discussion

Herein we have demonstrated a successful System-on-Chip implementation of an Adaptive Linear Element, utilizing a dedicated hardware node to perform the computation of the Adaline, and Python code running on the ZYNQ PS for training and testing. Our design is straightforward and simple, yet achieves 100% classification accuracy when tested, and can be trained to maximum performance in a matter of seconds. Most importantly, we have gained experience and familiarity with the hardware design process as well as with the PYNQ platform, and, having developed a functional system, we fully intend to implement more complex, nonlinear systems that will truly benefit from parallel implementation, and more completely showcase the efficacy of SoC technology for the implementation of Artificial Neural Networks. [2]

# References


[1] Single-layer Networks S. Raschka, "Single-Layer Neural Networks and Gradient Descent", Sebastian Raschka's Website, 2017. [Online]. Available: http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html. [Accessed: 25- Sep- 2017].

[2] Xilinx. "Zynq-7000 All Programmable SoC Data Sheet: Overview." Xilinx, Xilinx, 7 June 2017, www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[3] Fisher, R.A. "Iris Plants Database." Archive.ics.uci.edu, archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.names.

[4] http://gavinln.github.io/blog-site/img/irises/iris_species-sepal_length-petal_width.png