EE 113D: Digital Signal Processing Design

Mini-Project #2
Vowel Recognition

Names: Erik Hodges & Ivan Manan

Due Date: December 3, 2018

Objective

The purpose of this lab was to imitate an automatic speech recognition system by recognizing vowel sounds. For simplicity, an algorithm was implemented that successfully recognizes the spoken vowel as one of the four candidates: “ah” for father, “eh” for bed, “ee” for see, and “oo” for food.

Theory

The Mel-frequency cepstrum is a representation of the short-term power spectrum of a sound. The Mel-frequency cepstrum coefficients (MFCC) are coefficients that collectively make up the Mel-frequency cepstrum. The MFCC’s are derived commonly derived as:

1. Collect 1024 samples (64 ms) and Hamming window the samples
2. Take the FFT of the windowed signal.
3. Square the magnitude response.
4. Apply the filter bank onto the squared magnitude response. This also takes the logs of the powers at each of the mel frequencies, as well as multiply and accumulate.
5. Apply the discrete cosine transform to obtain the MFCC coefficients

The original signal is framed as 64 ms segments and multiplied by a Hamming window. The FFT was applied to every frame. The magnitude response was then squared and averaged. This gives an approximation of the power spectral density unscaled.

The next portion involves applying the filter bank onto the power spectral density. We obtained the filter peaks by using Mel frequency mapping. The mapping is defined below.

$$f_{mel}(f) = 2595 \log_{10}(1 + f/700)$$

Since the frequency range is 250 Hz to 8000 Hz, we can find the minimum and maximum Mel frequencies:

$$Min\, f_{mel} = f_{mel}(250) = 344$$
$$Max\, f_{mel} = f_{mel}(8000) = 2840$$

We find the real frequencies of the 26 filter peaks and 2 endpoints by mapping evenly spaced Mel frequencies. The Mel frequency spacing is found by

$$\frac{2840-344}{27} = 92.44444$$

We map the evenly spaced Mel frequencies back to real frequency by

$$f = 700 \times (10^{f_{mel}/2595} - 1)$$

Each filter is a triangle in the frequency domain that is equal to 1 at the center frequency point and equal to zero at the adjacent frequency points. The inner product of the 26 filters with the PSD of the signal gives 26 coefficients that we denote as $Y_1...Y_{26}$. By taking $\log_{10}$ of each Y, we obtain $X_1...X_{26}$. We perform the discrete cosine transform on the 26 X values to obtain the 13 MFCCs for the sample set.

After the 13 MFCCs are obtained, they can then be entered as inputs for a neural network. For simplicity, the biases and weight values were calculated via MATLAB to train an initial data set. The input to hidden layer of the neural network calculated the following functions:

$$h[x] = S(\sum_{k=1}^{m} w_{in}[k][x] \bullet i[k] + b_{in}[x]), \text{ for } x = 1, ..., n$$

$$S(x) = \frac{2}{1+e^{-2x}} - 1.$$

Likewise, the hidden to output layer of the neural network calculated the following functions:

$$o[x] = (\sum_{k=1}^{n} w_{out}[k][x] \bullet h[k] + b_{out}[x]), \text{ for } x = 1, ..., p$$

$$output[x] = \frac{e^{o[x]}}{\sum_{k=1}^{p} e^{o[x]}}, \text{ for } x = 1, ..., p.$$

The neural network outputs a 4-value vector. Each value is a percentage that all sums to 1. The position of the vector that corresponds to “ah”, “eh”, “ee”, and “oo” with the largest value denotes that vowel was recognized by the system.

Design Rationale

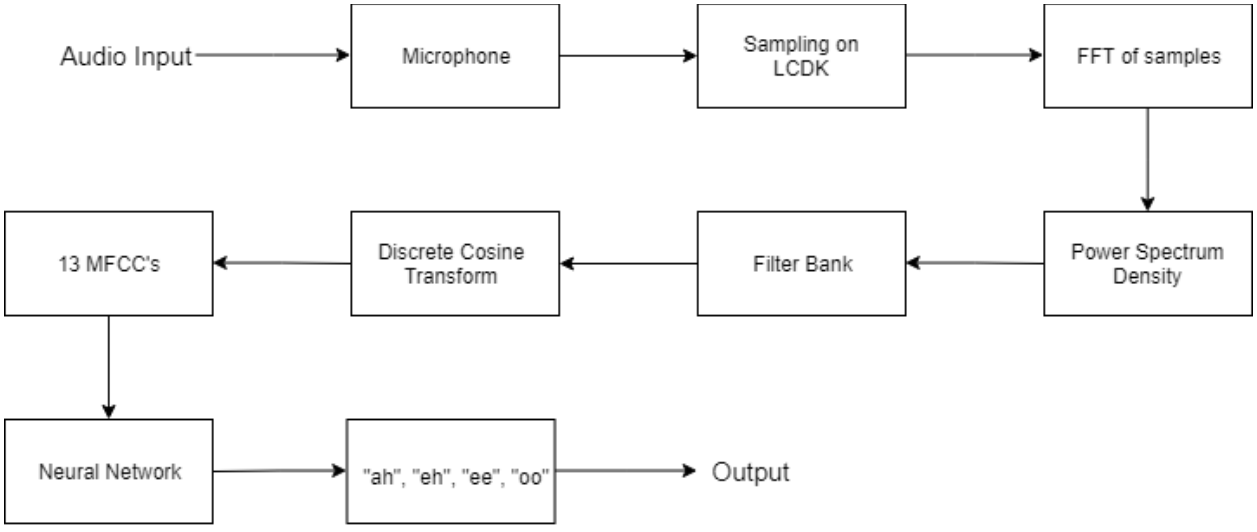

Figure 1. System Diagram of the Speech Recognition System

The procedure for this lab was divided up into five major components:

1. Sampling audio as the input signal.
2. Take the FFT of the input, and square the magnitude response.
3. Apply the filter bank onto the squared magnitude response.
4. Apply the discrete cosine transform. This gives the 13-element feature Mel frequency cepstrum coefficients.
5. Do machine learning via neural networks.
    a. Generate training on the LCDK.
    b. Train neural networks in MATLAB.
    c. Transfer neural network coefficients to the LCDK.
    d. Create a neural network in the LCDK.
    e. Run trials of the neural network on the LCDK. This outputs one of the four vowel sounds: “ah”, “eh”, “ee”, and “oo”.

When sampling the audio as the input signal, there were factors to consider. For instance, we had to code a procedure that cues when the audio started recording. This cue was done by flashing all the available LED’s on the LCDK. When the audio recording is over, the LED’s would turn off. This process would repeat for each vowel.

Data Acquisition Code Operation

We implemented counters to track both time and the current data set. The samp_ctr is incremented on every clock cycle of the LCDK and keeps track of time for recording samples and for flashing the LEDs. The data_ctr counts through the 40 data sets which correspond to the 40 rows of MFCC_arr. The first 10 are for “ah”, the second 10 are for “eh”, the third 10 are for “ee”, and the fourth 10 are for “oo”.

The data acquisition code is a state machine that allows the entire process of acquiring samples to be automated. There are 7 states which are described below.

Kill: Prevents the interrupt function from doing anything and resets all counters. This is used before the user initiates sample collection and after collection is complete. The state machine is put into the Kill state upon startup and whenever switch 5 is moved from the on position to the off position. The state machine is moved to the Countdown4 state when switch 5 is moved from the off position to the on position.

Countdown4, Countdown5, Countdown6, Countdown7: Before data collection for each vowel, a countdown occurs on LEDs 4, 5, 6, and 7. Each LED turns on for a half second with no delay between them. Once LED7 turns off, the machine transfers to the Recording state.

Recording: LCDK takes samples from the microphone and stores them in a 1024 long array. The first 100 samples are set to zero due to problems with the LCDK. The samples are also Hamming windowed as they are recorded. When the array is full, transfers to the Wait state.

Wait: This state forces the interrupt function to wait for data processing to be completed. This is necessary because the processing step takes too long to complete in one interrupt cycle. The main function contains an infinite while loop that continuously checks if the state machine is in the wait state. If the wait state is detected, then main calls the compute() function that processes the 1024 long array of samples to obtain 13 MFCC coefficients. Once computation is done, the machine transfers back into the recording state with two exceptions. If the number of sample sets collected is a multiple of 10, then it is time to start recording the next vowel sound so the machine transfers into the done state. If all 40 sample sets have been collected, then data collection is finished so the machine transfers to the Kill state.

Done: Notifies the user that collection for a given vowel sound is finished by flashing all 4 LEDs for a half second and then waits one second before transitioning into the Countdown4 state.

Functions

compute(): The compute function does all of the data processing on the inputted samples. First it takes the FFT by using the dsplib function and squares the magnitude to obtain the PSD. Next, it performs the inner product (correlation) of each filter with the PSD and stores the 26 values in the array Xm. It then takes the log of Xm to obtain Ym. Then, it computes the MFCCs by taking the discrete cosine transform and stores them in the corresponding row of MFCC_arr. Lastly, it updates counters and the current state.

magnitude_square(): Computes the magnitude square of the FFT output array to get the PSD.

fill_hamming(): Generates a 1024 long Hamming window. This runs once during startup to reduce computation time.

get_filter_peaks(): Uses inverse Mel frequency mapping to obtain the PSD array indices of the filter peaks. These indices are important when applying the filters in the compute() function. This runs once during startup to reduce computation time.

gen_cos_table(): Generates a 26x13 array that serves as a lookup table for the DCT. This runs once during startup and greatly reduces computation time.

all_on(): Turns on all LEDs.

all_off(): Turns off all LEDs.

neural_network(): Executes the neural network trial.

preprocessing(): Does all the initial calculations for the MFCC coefficients in order to convert them to proper inputs for the neural network

Neural Network

input_to_hidden_layer(): This does the computation of the inputs using the bias input and weight input values from MATLAB.

hidden_to_output_layer(): Computes the softmax function from the outputs of the hidden layer using bias output and weight output values from MATLAB. This calculates the output vector to determine which vowel sound was recognized.

Data

The MFCC coefficients were obtained using the procedure under the Design Rationale section. Each vowel produced a distinct spectrum of values for the MFCC coefficients array. The MFCC coefficients were then used as inputs for neural network training as well as building the neural network by taking into account the weights and bias values.

Table 2 below displays the accuracy of the neural network on recognizing the speech. It was compared to the actual vowel sounds pronounced into the microphone.

	Prediction					
	“ah”	**“eh”**	**“ee”**	**“oo”**	**Total**	**Error Rate**
“ah”	32	3	1	4	40	20%
“eh”	0	35	1	4	40	12.5%
“ee”	0	0	40	0	40	0%
“oo”	0	0	0	40	40	0%

Table 2. Performance of Neural Network; The leftmost column denotes the expected vowel, while the top row indicates the predicted vowel from the neural network.

Notice that the “ah” vowel had the largest error rate. This was due to the frequency content of “ah”. This error rate is explained in detail in the Challenges section below.

Challenges

Automating the data collection process introduced a lot of extra work into the design process. There were a lot of bugs where there was a strange case that caused an incorrect state transition. It took several hours to get these bugs fixed such that the state machine worked correctly every time. The complexity of the mathematical operations involved caused a lot of bugs as well. One especially challenging part was implementing the filter bank by calculating the filter values using slopes and the peak sample indices. Another challenge was finding out there was a typo in the summation for the softmax function, which explained why the initial implementation of the neural network was not working.

The last challenge was that “ah” and “eh” were less easily identified than “ee” and “oo”. This was because the people providing the speech input have low voices. This combined with the fact that “ah” and “eh” have lower frequency content than “ee” and “oo” resulted in some of the peaks in the FFT occuring at frequencies below 250 Hz, the minimum frequency for MFCC calculation. As a result, the algorithm was poorly trained to recognize these sounds from people with higher voices.

Certificate of Completion

Name Ivan Manan

Name Erik Hodges

Successfully completed Mini-Project 2.

Comments

Signed

Code for Mini-Project 2

```
#include "L138_LCDK_aic3106_init.h"
#include "L138_LCDK_switch_led.h"
#include "evmomapl138_gpio.h"
#include <stdint.h>
#include <math.h>
#include <ti/dsplib/dsplib.h>



#define PI 3.14159265358979323
#define numData 40
#define numMFCCs 13
#define frameSize 1024 //number of samples 64ms (real+imag)
#define halfSec 8000 //number of clock cycles for half sec
#define freq_inc 15.6403 // increment between adjacent FFT frequencies
#define mel_inc 92.4444 //increment between adjacent mel frequencies

/*
 * the code in the interrupt function is a state machine that handles the
 * timing for collecting audio samples and the LED cues for the user
 * Below, we define the states for the state machine
 */
#define countdown4 0 //led4 on during countdown
#define countdown5 1 //led5 on during countdown
#define countdown6 2 //led6 on during countdown
#define countdown7 3 //led7 on during countdown
#define recording 4 //recording in progress
#define wait 5 //wait for main to finish computations
#define done 6 //flash all leds to show that a given sound is done
#define kill 7 //prevents any further recording
int current_state = kill;

float corr = 0;
float slope;
int16_t left_sample;

float filter_bank[1024];

float PSD_arr [frameSize];
float hamming_window [frameSize];
float Xm[26]; //input to DCT
float Ym[26];
float MFCC_arr[numData][numMFCCs];
float cos_table[26][13]; //DCT lookup table; mrow kcol
int filter_peaks[28]; //contains PSD array sample indices of the filter peaks
int data_ctr = 0; //iterates through 40 sample sets
int MFCC_ctr = 0; //iterates through 13 MFCCs
int samp_ctr = 0; //iterates through clock cycles

int idx = 0;

/* Align the tables that we have to use */

// The DATA_ALIGN pragma aligns the symbol in C, or the next symbol declared in C++, to an alignment boundary.
// The alignment boundary is the maximum of the symbol's default alignment value or the value of the constant in
bytes.
// The constant must be a power of 2. The maximum alignment is 32768.
// The DATA_ALIGN pragma cannot be used to reduce an object's natural alignment.

//The following code will locate mybyte at an even address.
//#pragma DATA_ALIGN(mybyte, 2)
//char mybyte;

//The following code will locate mybuffer at an address that is evenly divisible by 1024.
//#pragma DATA_ALIGN(mybuffer, 1024)
//char mybuffer[256];
#pragma DATA_ALIGN(x_in,8);
int16_t x_in[2*frameSize];

#pragma DATA_ALIGN(x_sp,8);
float   x_sp [2*frameSize]; //input samples array
#pragma DATA_ALIGN(y_sp,8);
float   y_sp [2*frameSize]; //fft output array
```

```
#pragma DATA_ALIGN(w_sp,8);
float   w_sp [2*frameSize]; //twiddle factor

// brev routine called by FFT routine
unsigned char brev[64] = {
    0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
    0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
    0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
    0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
    0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
    0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
    0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
    0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};

// The seperateRealImg function separates the real and imaginary data
// of the FFT output. This is needed so that the data can be plotted
// using the CCS graph feature
float y_real_sp [frameSize];
float y_imag_sp [frameSize];

void export_mfcc();
void neural_networks();

separateRealImg () {
    int i, j;

    for (i = 0, j = 0; j < frameSize; i+=2, j++) {
        y_real_sp[j] = y_sp[i];
        y_imag_sp[j] = y_sp[i + 1];
    }
}

// Function for generating sequence of twiddle factors
void gen_twiddle_fft_sp (float *w, int n)
{
    int i, j, k;
    double x_t, y_t, theta1, theta2, theta3;

    for (j = 1, k = 0; j <= n >> 2; j = j << 2)
    {
        for (i = 0; i < n >> 2; i += j)
        {
            theta1 = 2 * PI * i / n;
            x_t = cos (theta1);
            y_t = sin (theta1);
            w[k] = (float) x_t;
            w[k + 1] = (float) y_t;

            theta2 = 4 * PI * i / n;
            x_t = cos (theta2);
            y_t = sin (theta2);
            w[k + 2] = (float) x_t;
            w[k + 3] = (float) y_t;

            theta3 = 6 * PI * i / n;
            x_t = cos (theta3);
            y_t = sin (theta3);
            w[k + 4] = (float) x_t;
            w[k + 5] = (float) y_t;
            k += 6;
        }
    }
}


void all_on() {   //turn on all leds
    LCDK_LED_on(4);
    LCDK_LED_on(5);
    LCDK_LED_on(6);
    LCDK_LED_on(7);
    return;
}

void all_off() {   //turn off all leds
    LCDK_LED_off(4);
    LCDK_LED_off(5);
```

```
    LCDK_LED_off(6);
    LCDK_LED_off(7);
    return;
}

void get_filter_peaks(){ // get indices of filter peaks and end points
    int i;
    for(i = 0; i < 28; ++i){
        //convert each mel frequency into real frequency and divide by freq_inc to get sample index
        //filter_peaks[0,27] are the indices of the zeros at the far ends
        filter_peaks[i] = (int)round(700 * (powf(10,((344 + i * mel_inc)/2595)) - 1) / freq_inc);
    }
}

float filter[1024];

void gen_filter(){

    int i;
    int j;

    for (i = 0; i < 1024; ++i){
        filter[i] = 0;
    }

    for(i = 1; i < 27; i += 2){ //iterate through filter peaks
        corr = 0;
        for(j = filter_peaks[i-1]; j < filter_peaks[i]; ++j){ //upslope
            slope = 1 / (float)(filter_peaks[i] - filter_peaks[i-1]);
            filter[j] = (j - filter_peaks[i-1]) * slope;
        }
        for(j = filter_peaks[i]; j <= filter_peaks[i+1]; ++j){ //downslope
            slope = 1 / (float)(filter_peaks[i+1] - filter_peaks[i]);
            filter[j] = (filter_peaks[i+1] - j) * slope;
        }
    }
}

void magnitude_square(){ //take magnitude squared of FFT to get PSD
    int i = 0;
    for(i=0; i < frameSize; ++i){
        PSD_arr[i] = sqrtf(y_sp[2*i]*y_sp[2*i] + y_sp[2*i+1]*y_sp[2*i+1]);
    }
    return;
}

void fill_hamming(){ //generate 1024 samples hamming window
    int i;
    for(i = 0; i < frameSize; ++i){
        hamming_window[i] = 0.54 - 0.46 * cosf((2 * PI * i) / (frameSize - 1));
    }
}

void gen_cos_table(){
    int m;
    int k;

    for(m = 1; m < 27; ++m){
        for (k = 1; k < 14; ++k){
            cos_table[m-1][k-1] = cosf((m-0.5) * ((float)k*PI/26.0));
        }
    }
}

void compute(){
    //store fft of input samples in y_sp
    DSPF_sp_fftSPxSP(frameSize,x_sp,w_sp,y_sp,brev,4,0,frameSize);
    magnitude_square(); // store PSD of input in PSD_arr

    int i;
    int j;

    //inner product of 26 filters with PSD
    for(i = 1; i < 27; ++i){ //iterate through filter peaks
        corr = 0;
        for(j = filter_peaks[i-1]; j < filter_peaks[i]; ++j){ //upslope
            slope = 1 / (float)(filter_peaks[i] - filter_peaks[i-1]);
```

```
            corr += PSD_arr[j] * (j - filter_peaks[i-1]) * slope;
        }
        for(j = filter_peaks[i]; j <= filter_peaks[i+1]; ++j){ //downslope
            slope = 1 / (float)(filter_peaks[i+1] - filter_peaks[i]);
            corr += PSD_arr[j] * (filter_peaks[i+1] - j) * slope;
        }
        Ym[i-1] = corr; //this Xm = Ym
    }

    float logVal;
    //take log of Ym to get Xm
    for(i = 0; i < 26; ++i){
        logVal = log10f(Ym[i]);
        Xm[i] = logVal;
    }

    //DCT
    for(i = 0; i < numMFCCs; ++i){ // i=k
        MFCC_arr[data_ctr][i] = 0;

        for(j = 0; j < 26; ++j){ // j=m
            MFCC_arr[data_ctr][i] += Xm[j] * cos_table[j][i];
        }
    }
    ++data_ctr;
    samp_ctr = 0;

    if (data_ctr == 40) {
        export_mfcc();
    }

    if(data_ctr % 10 == 0){ //start next vowel after 10 cycles
        current_state = done;
    }
    else
        current_state = recording;
}

interrupt void interrupt4(void) // interrupt service routine
{
    if(LCDK_SWITCH_state(5) == 1){ //start recording

        switch (current_state){

        case countdown4 :
            LCDK_LED_on(4);
            ++samp_ctr;
            if(samp_ctr >= halfSec){
                LCDK_LED_off(4);
                current_state = countdown5;
                samp_ctr = 0;
            }
            break;

        case countdown5 :
            LCDK_LED_on(5);
            ++samp_ctr;
            if(samp_ctr >= halfSec){
                LCDK_LED_off(5);
                current_state = countdown6;
                samp_ctr = 0;
            }
            break;

        case countdown6 :
            LCDK_LED_on(6);
            ++samp_ctr;
            if(samp_ctr >= halfSec){
                LCDK_LED_off(6);
                current_state = countdown7;
                samp_ctr = 0;
            }
            break;

        case countdown7 :
            LCDK_LED_on(7);
            ++samp_ctr;
```

```
            if(samp_ctr >= halfSec){
                LCDK_LED_off(7);
                current_state = recording;
                samp_ctr = 0;
            }
            break;

        case recording : //store audio input in x_sp
            if(samp_ctr < 100){ //set first 100 samples = 0
                left_sample = input_left_sample();
                x_sp[2*samp_ctr] = (float)left_sample;
                x_sp[2*samp_ctr] = 0; // even index = real
                x_sp[2*samp_ctr + 1] = 0; //odd index = imag
                ++samp_ctr;
            }
            else if(samp_ctr < frameSize){ //store windowed samples in x_sp
                left_sample = input_left_sample();
                x_sp[2*samp_ctr] = (float)left_sample * hamming_window[samp_ctr]; // even index = real
                x_sp[2*samp_ctr + 1] = 0; //odd index = imag
                ++samp_ctr;
            }
            else {
                samp_ctr = 0;
                current_state = wait;
            }
            break;

        case wait : //do nothing while main processes data

            break;

        case done : //notify user to switch to next sound

            if (samp_ctr < halfSec){
                all_on();
                ++samp_ctr;
            }
            else if (samp_ctr < 2 * halfSec){
                all_off();
                ++samp_ctr;
            }
            else if (data_ctr == 40){ //check if done
                current_state = kill;
            }
            else {
                samp_ctr = 0;
                current_state = countdown4;
            }

            break;

        case kill : //stop until switch 5 is turned off

            break;
        }

    }

    else { // reset
        all_off();
        data_ctr = 0;
        MFCC_ctr = 0;
        samp_ctr = 0;
        current_state = countdown4;
    }

    output_left_sample(0);
    return;
}

void export_mfcc() {

    // Create file that exports array
    FILE * fp;
    fp = fopen("training_samples.txt", "w+");

    int row, col;
```

```
    for (row = 0; row < numData; row++) {
        for (col = 0; col < numMFCCs; col++) {
            fprintf(fp, "%f ", MFCC_arr[row][col]);
        }
        fprintf(fp, "\n");
    }

    fclose(fp);

    // Run neural networks trial with previous MFCC coefficients
    neural_networks();
}

////////////////////////////////////////////////////////////////////////////////////////////////////
// Neural Networks Portion

float Y[numData][numMFCCs];
#define HIDDEN_LIMIT 5
#define OUTPUT_LIMIT 4

// Hidden layer changes for every Y
float hidden[numData][HIDDEN_LIMIT];

// Output does all 40 elements in Y
float output[numData][OUTPUT_LIMIT];
float final_output[OUTPUT_LIMIT];

void preprocessing() {
    int r, c;

    // Hard-coded values from MATLAB
    float x1_step1_xoffset[13] =
{-5.780084,-3.740172,0.916667,-2.875884,-3.011461,-0.518285,-2.057891,-2.431821,-1.241548,-1.229252,-1.638786,-0
.610258,0.134162};
    float x1_step1_gain[13] =
{0.158651487738461,0.504327762612922,0.299783870818334,0.318338223539011,0.618532852444658,0.630574674229359,0.4
58762419558877,0.637804727153516,0.78466390098797,1.61359811402652,0.611393065213019,0.615608696950305,1.1613053
0716525};
    float x1_step1_ymin = -1;
    float temp;


    for(r = 0; r < 40; r++) {
        for(c = 0; c < 13; c++) {
            temp = MFCC_arr[r][c] - x1_step1_xoffset[c];
            temp = temp * x1_step1_gain[c];
            Y[r][c] = temp + x1_step1_ymin;
        }
    }
}

// Outputs hidden layer
void input_to_hidden_layer() {

    // 13 rows, 5 columns
    // w_in[m][n]
    float bias1[5] =
{-0.0024919283011837455,-0.56051485255633615,-0.40267961963214766,-0.98695806588698598,-1.2669879631757537};
    float w_in[13][5] = {{-0.456589034092960,   -0.714096036815131, 0.270735858494048,  -0.432451225595637,
-1.99074862358156},
    {0.833917049213021, 0.401926173889747,  0.102585920928219,  -0.670271149276768, 0.317366263808549},
    {0.628499460901589, -0.382644053799426, 0.0787853387765209, 0.0948079362789648, 1.74185989185266},
    {-0.405095151114448,    -0.144141949361290, -1.66608035624465,  -0.137981318459344, 0.948702596124123},
    {-1.01850701381652, 0.299121229571484,  -1.36631241044219,  1.20101974186178,   -0.181798599439349},
    {-0.185489171457087,    1.55528904546201,   0.266170848145575, -0.449766807681975, -0.176166860228918},
    {1.04297252941079, 1.25301204909685,    -0.343219319120604, -1.10849143188953,  1.02908595062549},
    {-0.214190952724444,    0.698688276775753,  -0.0479497120567590,    -0.954069097219034, 0.576361004154070},
    {0.208216272765312, 0.401125335297835,  -0.583057007280083, 0.729042458132281,  -0.261482705972536},
    {0.463654514935408, -0.231754216985730, -0.468844492943491, -0.168287582849335, 0.352188604492435},
    {1.06859439127621,  -0.652455491949002, 0.116622785622047,  -0.0301344004567368,    -0.144638477811155},
    {0.450597743638169, -0.104560004101484, 0.951840459660890,  -1.50707384779489,  0.909729237318600},
    {-1.29297996480711, 0.407057596774009,  -0.121205130067250, -0.114474538114915, -0.571154431891727}};

    const int m = numMFCCs;
    const int n = HIDDEN_LIMIT;

    int x;
```

```
    int k;
    int iteration;

    float S;

    for(iteration = 0; iteration < numData; iteration++) {

    // For each MFCC coefficient
        for (x = 0; x < n; x++) {
            S = 0;

            // Take summation first
            for (k = 0; k < m; k++) {
                S += w_in[k][x] * Y[iteration][k];
            }
            // Take bias
            hidden[iteration][x] = (2 / (1 + expf(-2 * (S + bias1[x])))) - 1;
        }
    }
}

// Manipulates hidden layer
// Outputs output layer
void hidden_to_output_layer() {

    // 5 rows, 4 columns
    // w_out[n][p]
    float bias2[5] = {0.4358718506518805,0.75655223469057087,0.35622964910884081,-1.8567859135428995};
    float w_out[5][4] = {{0.590969672657138,   -1.39652063048604,   1.90678324638783,    -1.72748309192920},
    {-2.51468147761556,-3.50400932696632, 1.36854558138174 ,   3.87304375506483},
    {4.54061400666149, -2.37288653096860,   -1.78855982647102,   -1.22103842988618},
    {-1.41966807952103,4.48902036926536,    -0.871207794237570,  -1.45040496372696},
    {-1.97554523847240,-0.600140699054205,  5.40011532756768,    -4.04698775429002}};


    const int n = HIDDEN_LIMIT;
    const int p = OUTPUT_LIMIT;

    float denominator;
    float O;
    float temp;
    int iteration;

    int k, x, ite;

    for(iteration = 0; iteration < numData; iteration++) {

        for(x = 0; x < p; x++) {
            O = 0;

            for(k = 0; k < n; k++) {
                O += w_out[k][x] * hidden[iteration][k];
            }
            O += bias2[x];

            denominator = 0;
            for (k = 0; k < p; k++) {
                temp = 0;
                for(ite = 0; ite < n; ite++) {
                    temp += w_out[ite][k] * hidden[iteration][ite];
                }
                temp += bias2[k];
                denominator += expf(temp);
            }
            output[iteration][x] = expf(O) / denominator;
        }
    }
}

// See MiniProject 2 directions on implementing neural network
void neural_networks() {

    // Step 1 - pre-processing
    preprocessing();

    // Step 2 - input layer to hidden layer
    input_to_hidden_layer();
```

```
    // Step 3 - hidden layer to output layer
    hidden_to_output_layer();

    // Calculate average of entire output array to determine what vowel was pronounced
    // ah = output[0]
    // eh = output[1]
    // ee = output[2]
    // oo = output[3]

    int j;
    int i;

    for (j = 0; j < 4; j++) {
        final_output[j] = 0;
    }
    for (i = 0; i < 40; i++) {
        for (j = 0; j < 4; j++) {
            final_output[j] += output[i][j];
        }
    }

    // Obtain max
    float max = 0;
    int output_index = 0;
    for (j = 0; j < 4; j++) {
        if (final_output[j] > max) {
            max = final_output[j];
            output_index = j;
        }
    }

    // NOTE: Must say a single vowel for the entire duration of LED flashing
    switch(output_index) {
        case 0:
            printf("\n ah was pronounced.\n\n");
            break;
        case 1:
            printf("\n eh was pronounced.\n\n");
            break;
        case 2:
            printf("\n ee was pronounced.\n\n");
            break;
        case 3:
            printf("\n oo was pronounced.\n\n");
            break;
    }
}

int main(void) {
    L138_initialise_intr(FS_16000_HZ,ADC_GAIN_21DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);
    LCDK_GPIO_init();
    LCDK_SWITCH_init();
    LCDK_LED_init();

    current_state = kill;
    all_off();
    get_filter_peaks();
    fill_hamming();
    gen_cos_table();
    gen_twiddle_fft_sp(w_sp, frameSize);
    gen_filter();


    while (1){
        if(current_state == wait){
            compute();
        }
    }
}
```