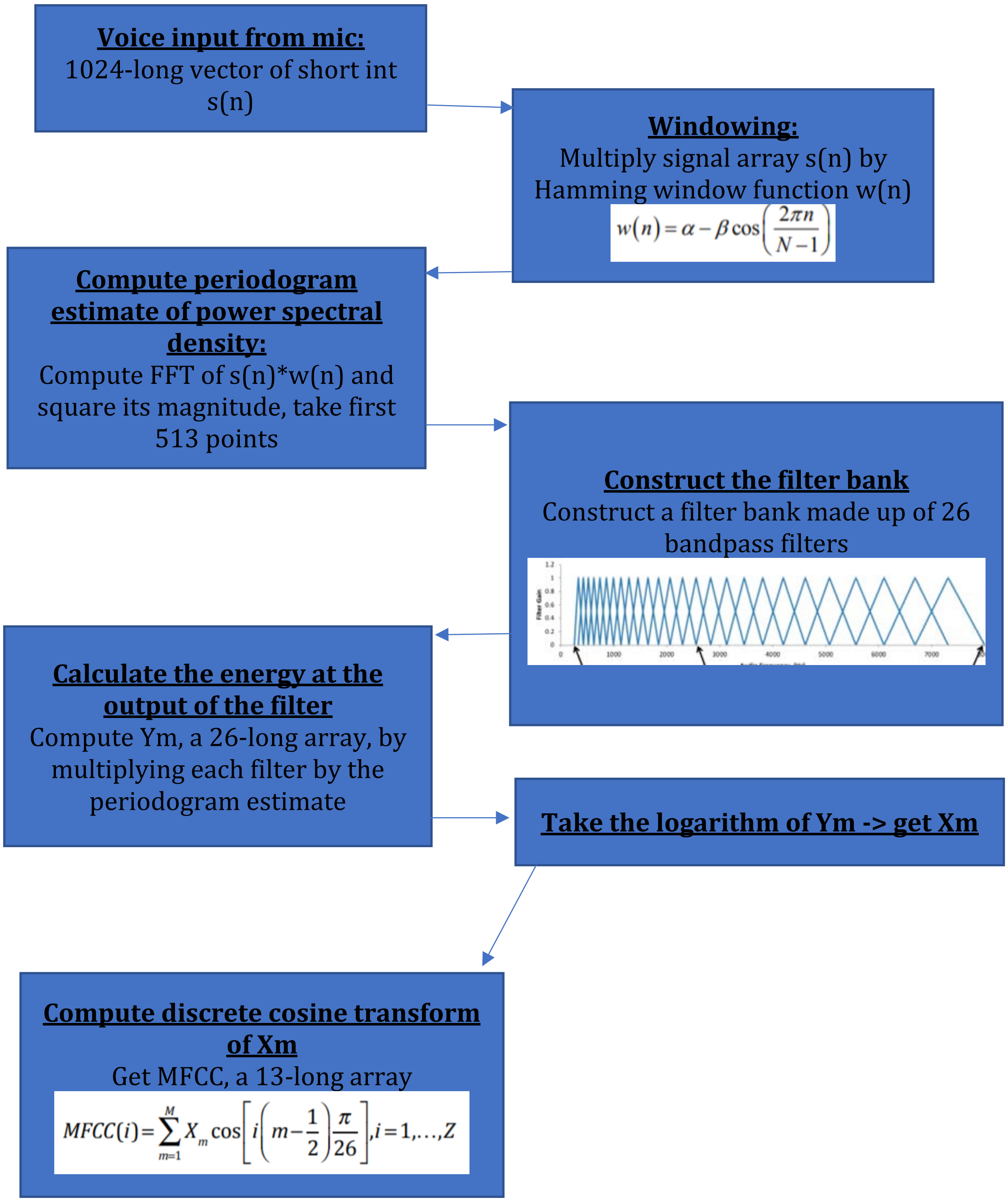
Mini-Project 2: Vowel Recognition

Objective:

The objective of this mini-project is to devise a neural network based on a sample set which can help identify vowel sounds. More specifically the sounds "ah, oo, ee, eh" are detected.

Block Diagram:

Part 1: Feature extraction



Voice input from mic:
1024-long vector of short int
s(n)

Windowing:
Multiply signal array s(n) by Hamming window function w(n)

$$w(n)=\alpha-\beta\cos\left(\frac{2\pi n}{N-1}\right)$$

Compute periodogram estimate of power spectral density:
Compute FFT of s(n)*w(n) and square its magnitude, take first 513 points

Construct the filter bank
Construct a filter bank made up of 26 bandpass filters

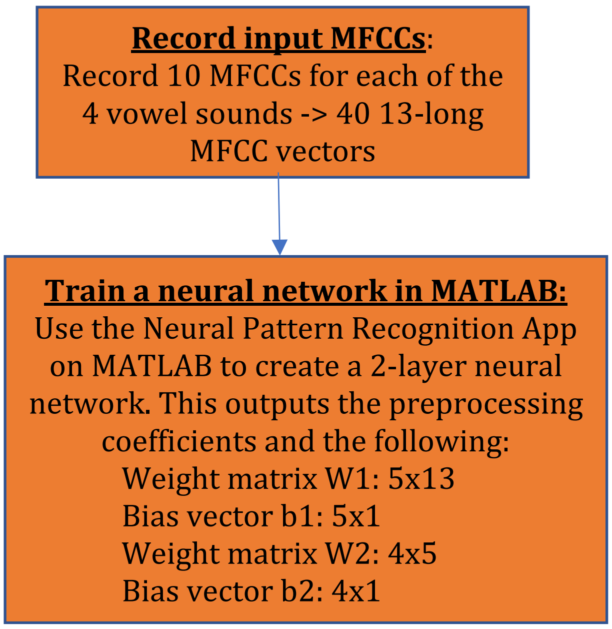
Calculate the energy at the output of the filter
Compute Ym, a 26-long array, by multiplying each filter by the periodogram estimate

Take the logarithm of Ym -> get Xm

Compute discrete cosine transform of Xm
Get MFCC, a 13-long array

$$MFCC(i)=\sum_{m=1}^{M} X_m \cos\left[i\left(m-\frac{1}{2}\right)\frac{\pi}{26}\right], i=1,\ldots,Z$$

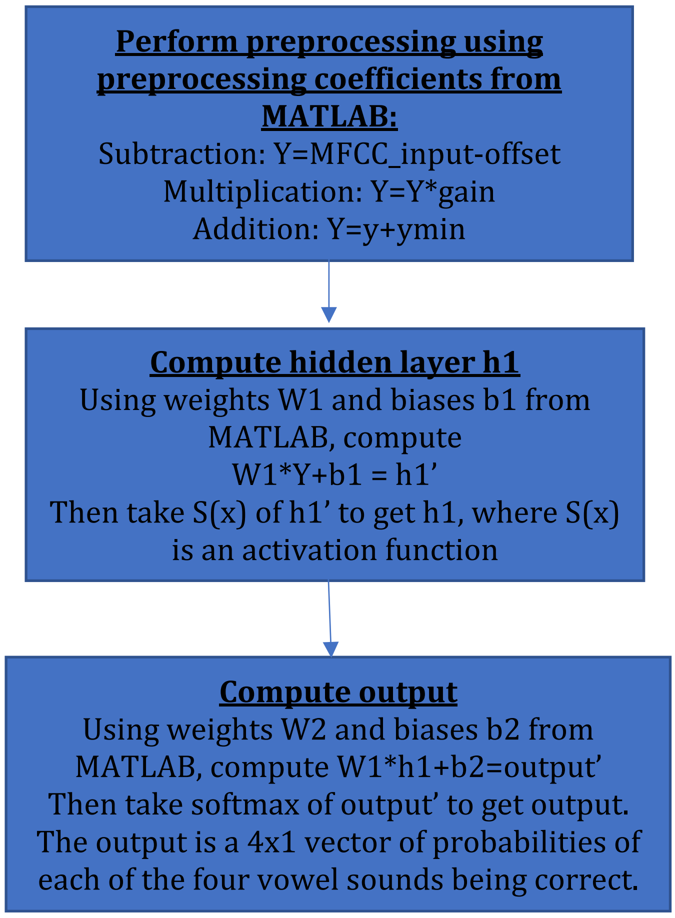
Part 2: Training and using a neural network



Record input MFCCs:
Record 10 MFCCs for each of the 4 vowel sounds -> 40 13-long MFCC vectors

Train a neural network in MATLAB:
Use the Neural Pattern Recognition App on MATLAB to create a 2-layer neural network. This outputs the preprocessing coefficients and the following:
Weight matrix W1: 5x13
Bias vector b1: 5x1
Weight matrix W2: 4x5
Bias vector b2: 4x1

Part 3: Implementing the neural net on the LCDK



Perform preprocessing using preprocessing coefficients from MATLAB:
Subtraction: Y=MFCC_input-offset
Multiplication: Y=Y*gain
Addition: Y=y+ymin

Compute hidden layer h1
Using weights W1 and biases b1 from MATLAB, compute
W1*Y+b1 = h1’
Then take S(x) of h1’ to get h1, where S(x) is an activation function

Compute output
Using weights W2 and biases b2 from MATLAB, compute W1*h1+b2=output’
Then take softmax of output’ to get output.
The output is a 4x1 vector of probabilities of each of the four vowel sounds being correct.

Design Rationale:

In order to identify a vowel sound, we must first extract the features from the sound. For this project we make use of the Mel Frequency Cepstral Coefficients (MFCCs). Then we must build a data base for the vowels and using a machine learning approach with the help of MATLAB, recognize incoming vowel sounds with a certain probability of its accuracy.

Framing

The first step to make the speech recognition system is to frame the input sound. This makes the input signal statistically stationary over a short time interval in which the mean and variance of the signal remain constant.

To begin, we set the sample rate at 16KHz. The length of the frame is set at 64ms which yields a 1024 long input vector. Due to the pure nature of the vowel sound, without consonant or diphthong, we know that the vector will not be compromised.

In addition, due to a certain peculiarity in the LCDK we add an extra buffer of 100 samples in our input vector which is ignored and the 64ms sample and taken after the 100 samples.

Feature Extraction

Once the input sample is stored, we must perform feature extraction in order to identify the vowel sounds. For this process we use the MFCC approach. This involves devising a 13 elements feature vector for each frame is based on the DCT of the energy at the output of 26 bandpass filter banks. This is done in the following ways:

1. ***Hamming Window***

   First we must take the input sample and apply a Hamming Window. This window is applied so that there are no unreasonable discontinuities in the signal and is smooth everywhere. The Hamming window is applied using the given formula:

$$w(n) = \alpha - \beta \cos\left(\frac{2\pi n}{N-1}\right),$$

   where $\alpha = 0.54$ and $\beta = 0.46$.

```
sound_array[index]=left_sample;
windowed_sound_array[index] = sound_array[index] * (0.54-
(0.56*cos((2.0*PI*index)/1023))) ;
```

   The code above shows how the input sound_array[] is instantly windowed and stored in windowed_sound_array[] using the above formula. This is done in the interrupt handler section.

2. ***Periodogram estimate of power spectral density of input***
   First we need to compute the FFT of the input waveform. We use the following DSP function:

```
DSPF_sp_fftSPxSP(N,x_sp,w_sp,y_sp,brev,4,0,N);
```

   Once the FFT is determined we call the routine to separate the real and imaginary parts of the data using the following function:

```
separateRealImg ();
```

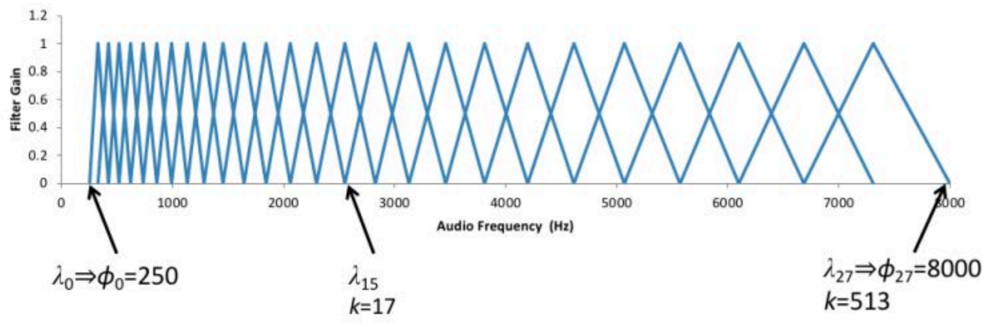
Next we must square the magnitude and take the first K (513) points of the result using the following code:

```
for (i = 0; i < 513; i++){
            y_mag_firstK[i] = y_real_sp[i]*y_real_sp[i] +
y_imag_sp[i]*y_imag_sp[i]; // first K points of SQUARED MAGNITUDE
}
```

3. ***Constructing the filter bank***
   Next we make a filter bank with M = 26 bandpass filters each of which has a triangular shaped frequency response $H_m(k)$ defined by the following parameters:

$$H_k = \begin{cases} 0 & k < \lambda_{m-1} \\ \dfrac{k-\lambda_{m-1}}{\lambda_m-\lambda_{m-1}} & \lambda_{m-1} \le k \le \lambda_m \\ \dfrac{\lambda_{m+1}-k}{\lambda_{m+1}-\lambda_m} & \lambda_m \le k \le \lambda_{m+1} \\ 0 & k > \lambda_{m+1} \end{cases}, k = 1 \dots K; n = 0 \dots M+1; \lambda \in (1, K); \lambda_n = int\left[\frac{K\phi_n}{f_{nyq}}\right]$$

The above define frequency response should have a structure similar to the following diagram:



Since we already have the audio freqnecy values, we can rearrange the formulas and determine the corresponding Mel frequencies using the following formula:

$$fmel_m = 2595 \log_{10}\left(1 + \frac{\phi_m}{700}\right), m = 1 \ldots M$$

The following 2-D array was created, audio_mel[][], which contains the audio frequency values in the first column and Mel frequency values in the second column. Since we know certain values already we are able to hard code them and use them to determine other values.

```
audio_mel[0][0]= 250;    audio_mel[27][0]= 8000;
audio_mel[0][1]= 344.16;    audio_mel[27][1]= 2840.02;
```

The other values to populate the audio frequency and Mel frequency array uses the following code:

```
for(i = 1; i<27; i++)    {
        audio_mel[i][1] = audio_mel[i-1][1]+92.44666;
        audio_mel[i][0] = 700*(pow(10, audio_mel[i][1]/2595)-1);
      }
```

Next we can determine the lambda bank using the following code:

```
for(i=0; i<28 ;i++){
         lambda_n[i] =  (int) audio_mel[i][0]*(0.064125);    }
```

Finally we can determine the filter bank values using the parameters above and using the following code version of the same restrictions:

```
for (m = 1; m < 27; m++){ // i is filter index (same as m in
notes)
        for (n = 1; n <= 513; n++){ // n is frequency index (same
as k in notes)
               if (n < lambda_n[m-1]){
                       filter_bank[m-1][n-1]=0;               }

               if (n >= lambda_n[m-1] && n <= lambda_n[m]){
                         filter_bank[m-1][n-1] = ((float)(n-
lambda_n[m1]))/((float)(lambda_n[m]-lambda_n[m-1]));
}
               if (n <= lambda_n[m+1] && n >= lambda_n[m]){
                                         filter_bank[m-1][n-1] =
((float)(lambda_n[m+1]-n))/((float)(lambda_n[m+1]-lambda_n[m]));
                 }             if (n > lambda_n[m+1]){
                         filter_bank[m-1][n-1]=0;             }
}    }
```

4. ***Calculating the energy at the output of the filter***

Once the filter bank is constructed, we apply it to the periodogram estimate of the power spectral density of the input and calculate the energy at the output of the filter. In addition, the log of the same values must be computed to make the results more relevant. These must be summed over all frequency bins. The same is done through the following code:

```
for (m = 0; m < 27; m++){
for (i = 0; i < 513; i++){ // for each element in y_mag_firstK

triangleEnergy[m] += y_mag_firstK[i] * filter_bank[m][i]; } }  //
log the triangle energy
for (m = 0; m < 27; m++){
logTriangleEnergy[m] = log10(triangleEnergy[m]);
// logTriangleEnergy is X_m in the lab }
```

5. ***Taking Discrete Cosine Transform to compute ZMFCC values***

In order to determine the ZMFCC values the following summation is used:

$$MFCC(i) = \sum_{m=1}^{M} X_m \cos\left[i \left(m - \frac{1}{2} \right) \frac{\pi}{26} \right], i = 1, \ldots, Z$$

Where M=26 and Z=13.

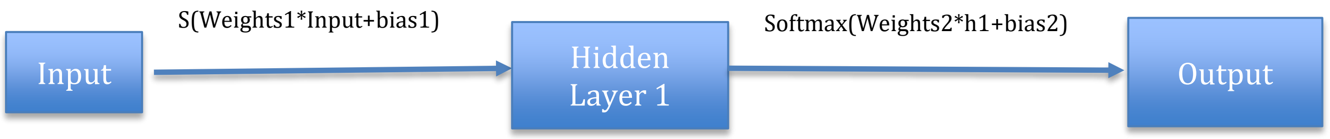
The same is implemented in code as the following:

```
for(i=1; i<=13; i++){ // i's in instructions
       for(k=0; k<26; k++){ // m's in instructions
              Z_MFCC[i-1] += logTriangleEnergy[k] * cos(i*(k-
0.5)*(PI/26.0));      }
 }
```

<u>Machine Learning Process (Neural Network)</u>

1. ***Training the neural network in MATLAB***

Once we have recorded 40 13-long MFCC vectors, we import these into MATLAB which creates a 2-layer neural net for us. The architecture is as follows:



We export the weights and biases and implement the network in CCS.

2. ***Implementing the network in CCS***

   a. ***Preprocessing***

   Preprocessing coefficients from MATLAB are used to preprocess the input MFCCs.

```
// PreProcessing
for(i=0; i<13 ; i++){
// y is a 13-vector
//subtraction
Y[i] = Z_MFCC[i] - x1_step1_xoffset[i];

//multiplication
Y[i] = Y[i] * x1_step1_gain[i];

//Addition
Y[i]= Y[i] + x1_step1_ymin;
}
```

b. ***Input to hidden layer***

The equation showed in the architecture above is used to compute the hidden layer. First, we compute W1*input, then we add the bias1, and then we take an activation function of (W1*input+bias1) to get the output h1 of the first layer.

```
// take the sum for layer 1
// IW_1 is a 5x13 matrix mxn
// sum_Y1 is a 5-vector
for (m=0; m< 5; m++){
 for(i=0; i < 13; i++){
        sum_Y1[m] += IW1_1[m][i] * Y[i];
 }
}
// add the biases for layer 1
for (m=0; m<5; m++){
 S1[m]=sum_Y1[m]+b1[m];
}

// normalize for layer 1. This gives layer 2's inputs
for (m=0; m<5; m++){
 h1[m] = (2.0/(1+exp(-2.0*S1[m])))-1.0;
}
```

c. ***Hidden layer to output***

The equation shown in the architecture above is used to compute the output. First, compute W2*h1, then we add the bias2, and then we take the softmax of the whole thing. The result is a 4x1 output vector, which tells us the probabilities of each vowel sound (if the first element of the vector has the highest probability then it was probably 'ah', etc).

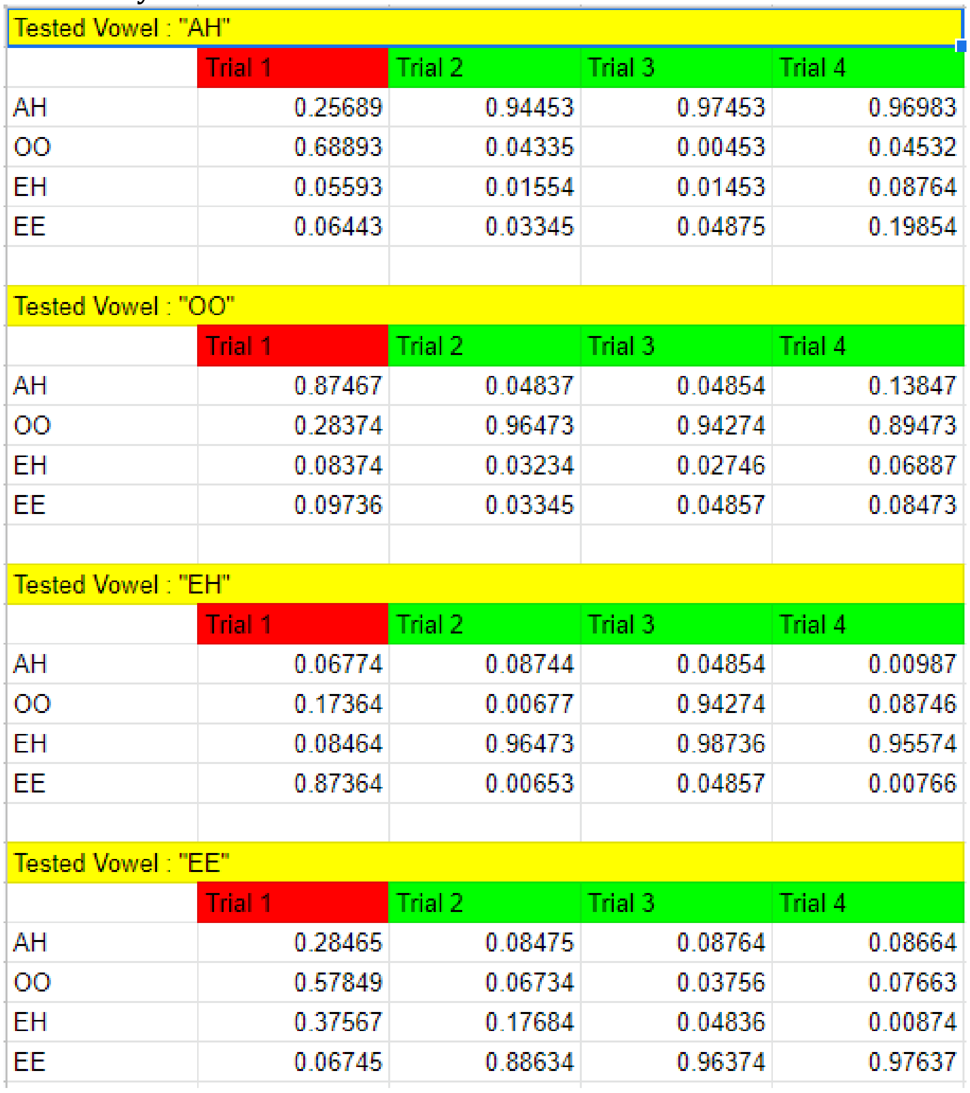
```
// LAYER 2
// take the sum for layer 2
for (m=0; m< 4; m++){
 for(i=0; i < 5; i++){
        sum_Y2[m] += LW2_1[m][i] * h1[i];
 }
}
// add the biases for layer 2
for (m=0; m<4; m++){
 h2[m]=sum_Y2[m]+b2[m];
}

// Softmax to output
for (m=0; m<4; m++){
 sum_exp += exp(h2[m]);
}

for (m=0; m<4; m++){
 output[m] = exp(h2[m])/sum_exp;
}
// END LAYER 2
```

Evaluation of Network Performance

The test time performance is shown in the chart below. Out of four trials for each vowel, about one of them usually turned out to be wrong. The following data shows the output probabilities for each vowel, as a result of the actual spoken vowel sounds (shown in yellow). In the green trials, the vowel was guessed correctly by the network, and in the red trials, the vowel was guessed incorrectly.



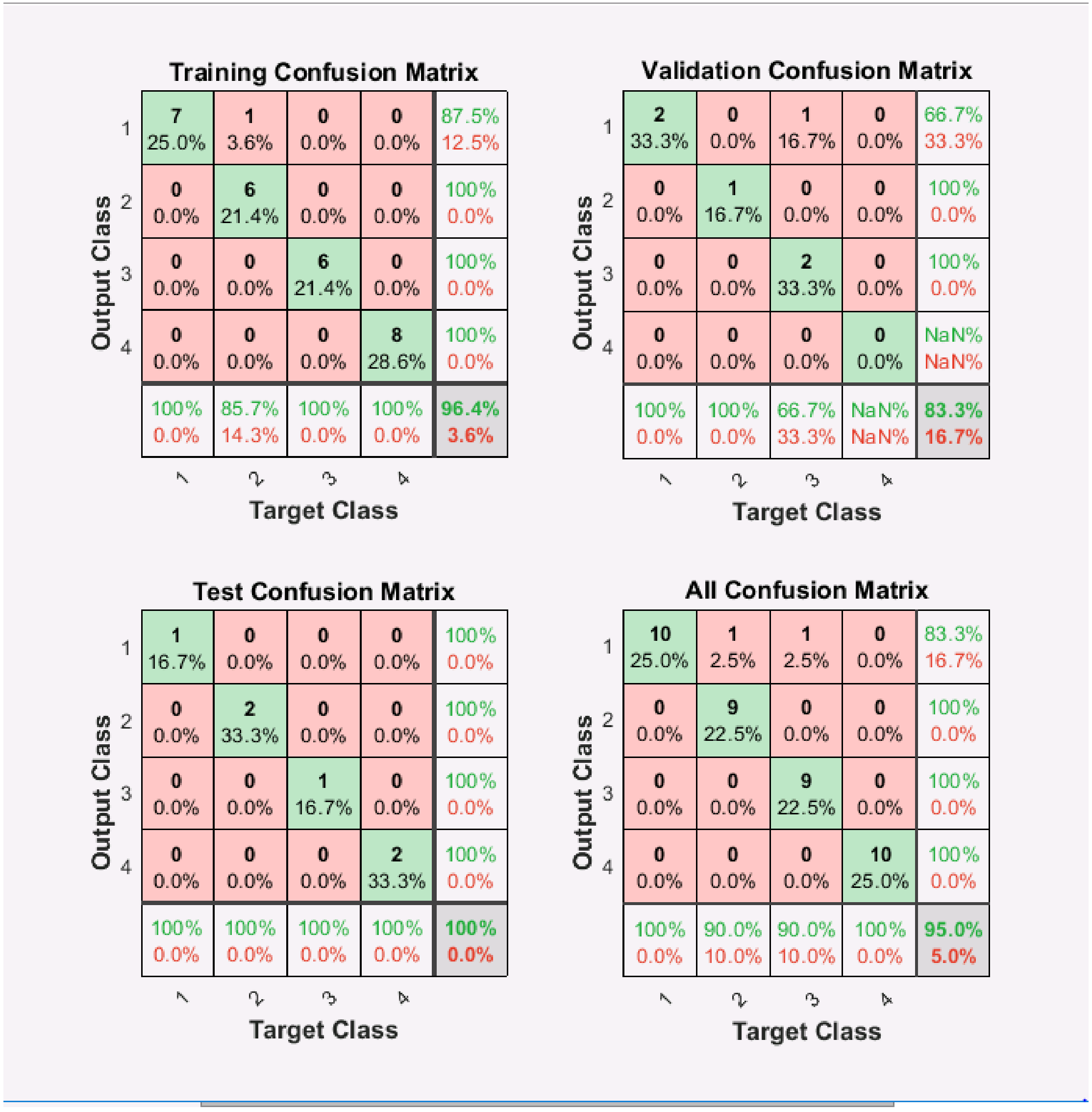
Tested Vowel : "AH"				
	Trial 1	Trial 2	Trial 3	Trial 4
AH	0.25689	0.94453	0.97453	0.96983
OO	0.68893	0.04335	0.00453	0.04532
EH	0.05593	0.01554	0.01453	0.08764
EE	0.06443	0.03345	0.04875	0.19854

Tested Vowel : "OO"				
	Trial 1	Trial 2	Trial 3	Trial 4
AH	0.87467	0.04837	0.04854	0.13847
OO	0.28374	0.96473	0.94274	0.89473
EH	0.08374	0.03234	0.02746	0.06887
EE	0.09736	0.03345	0.04857	0.08473

Tested Vowel : "EH"				
	Trial 1	Trial 2	Trial 3	Trial 4
AH	0.06774	0.08744	0.04854	0.00987
OO	0.17364	0.00677	0.94274	0.08746
EH	0.08464	0.96473	0.98736	0.95574
EE	0.87364	0.00653	0.04857	0.00766

Tested Vowel : "EE"				
	Trial 1	Trial 2	Trial 3	Trial 4
AH	0.28465	0.08475	0.08764	0.08664
OO	0.57849	0.06734	0.03756	0.07663
EH	0.37567	0.17684	0.04836	0.00874
EE	0.06745	0.88634	0.96374	0.97637

Our expected performance from MATLAB is shown below. On the Validation Confusion Matrix, the green diagonal shows the number of times that the vowel was guessed correctly for the samples in the validation set. Any nonzero number in the red squares indicates that the incorrect vowel was guessed. The validation accuracy for the model was 83.3%, and the test accuracy was 100%. This is different from the roughly 75% accuracy we got when we tested our model in CCS. This could be due to discrepancies between the training MFCC data (the 40x13 input) and the test MFCC data that we used when we tested our model.



Training Confusion Matrix

Output Class	1	2	3	4	
1	7 25.0%	1 3.6%	0 0.0%	0 0.0%	87.5% 12.5%
2	0 0.0%	6 21.4%	0 0.0%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	6 21.4%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	8 28.6%	100% 0.0%
	100% 0.0%	85.7% 14.3%	100% 0.0%	100% 0.0%	96.4% 3.6%

Target Class

Validation Confusion Matrix

Output Class	1	2	3	4	
1	2 33.3%	0 0.0%	1 16.7%	0 0.0%	66.7% 33.3%
2	0 0.0%	1 16.7%	0 0.0%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	2 33.3%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
	100% 0.0%	100% 0.0%	66.7% 33.3%	NaN% NaN%	83.3% 16.7%

Target Class

Test Confusion Matrix

Output Class	1	2	3	4	
1	1 16.7%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
2	0 0.0%	2 33.3%	0 0.0%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	1 16.7%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	2 33.3%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%	100% 0.0%

Target Class

All Confusion Matrix

Output Class	1	2	3	4	
1	10 25.0%	1 2.5%	1 2.5%	0 0.0%	83.3% 16.7%
2	0 0.0%	9 22.5%	0 0.0%	0 0.0%	100% 0.0%
3	0 0.0%	0 0.0%	9 22.5%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	10 25.0%	100% 0.0%
	100% 0.0%	90.0% 10.0%	90.0% 10.0%	100% 0.0%	95.0% 5.0%

Target Class

In order to create a more robust and more accurate version of our vowel decoder, we would record much more training data from a variety of voices.