18-551 FINAL REPORT
GROUP 9, FALL 2007

JACK AND JILL WENT UP THE HILL TO *BEEP* *BEEP* *BEEP*

GROUP MEMBERS:

JIAMIN BAI, JIAMINB@ANDREW.CMU.EDU

DESMOND HU, LIRENDEH@ANDREW.CMU.EDU

AARON ONG, ABO@ANDREW.CMU.EDU

MINGWEI TAY, MTAY@ANDREW.CMU.EDU

CONTENTS

1. Introduction

Current methods in censorship of live audio and video broadcasts involve a broadcast delay of typically 3 seconds during which a person monitoring the broadcast stream manually censors expletives and other obscenities present.

Invariably, there will be times when the occasional expletive fails to be censored, due either to a lapse in concentration or the delayed reaction time of the human operator. A notable example was of the Superbowl half time show, where viewers got an inappropriate glimpse of Janet Jackson's *beep*'s live on national television.

One way to solve this problem would be to make the process of censorship an automated one, which would reduce the manpower required for such censorship, and therefore reduce the effect of human error. The scope of our project will involve the censorship of target words in live audio transmission, in which phonemes will be detected in the stream of speech, with editing done as close to real time as possible.

2. Prior 551 Projects

Three previous 18551 groups did projects on speech processing, and they all involved speech transformation, in making speech from a source speaker sound like speech from a target speaker. They all had various ways of identifying elements of the speech waveform, as detailed below.

Spring 2001: "Speech Morphing for Space Marines". This group attempted to recognize speech based on a modification of Linear Predictive Coding (LPC) coefficients and the excitation pitch.

Spring 2004, Group 9: "Hey! Stop sounding like me!!". This group used Linear Spectral Frequencies (LSF) instead of LPC coefficients to represent the formant frequencies. They used Dynamic Time Warping (DTW) to time-align the signals during the training phase.

Fall 2006, Group 1: "Voice Transformers: More than meets the ear". This group used DTW and LSF as previously mentioned, and also implemented Pitch-Synchronous Overlap Add (PSOLA) to improve the quality of the output speech, as part of the post processing to make the speech more recognizable. It also mentioned the use of Hidden Markov Models, currently used in state-of-the-art speech recognition.

3. Our Implementation

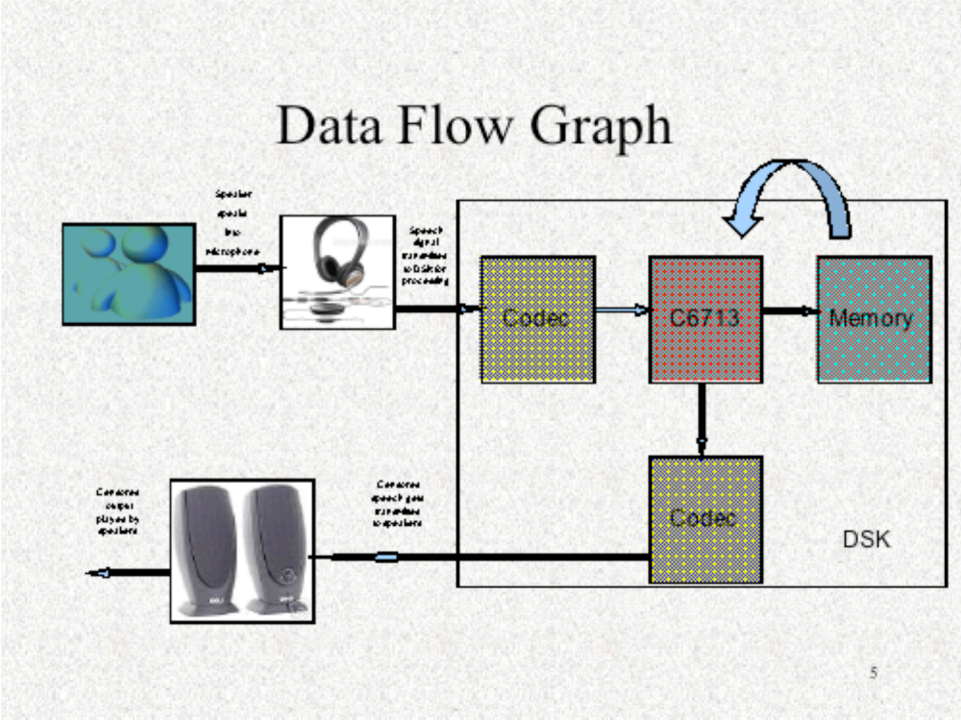
The most unique element of our project is the utilizing of a hybird Artificial Neural Network and Hidden Markov Model as a classifier in the phoneme identification process. None of the previous groups mentioned above have used neural networks before. Also, we will be attempting to get the system to be working in real time for continuous speech.

First, we will be using a set of features to characterize each window (of 8ms), specifically MFCC and their delta values. Instead of merely using the current time window for each feature vector, we will also use previous and future time windows, 24ms and 48ms before and after[1]. As such, the neural network will have 50 input nodes, with 10 MFCC values from each frame. The neural network will also have 30 hidden nodes and 63 output nodes, the latter corresponding

to the 63 different phonemes recognized by TIMIT, the database we are using. The output of the neural network will be a probability distribution according to each of the possible phonemes.

After the obtaining the probability distribution of each window, we will use an approximation of the Viterbi algorithm to find a word, characterized by the most probable phoneme sequence, based on the previous phonemes and the probability of transitions seen. Once a match is found with our target word, the segment of the signal containing the word we are interested in will be beeped out, and the output will be played back, all done in real time.

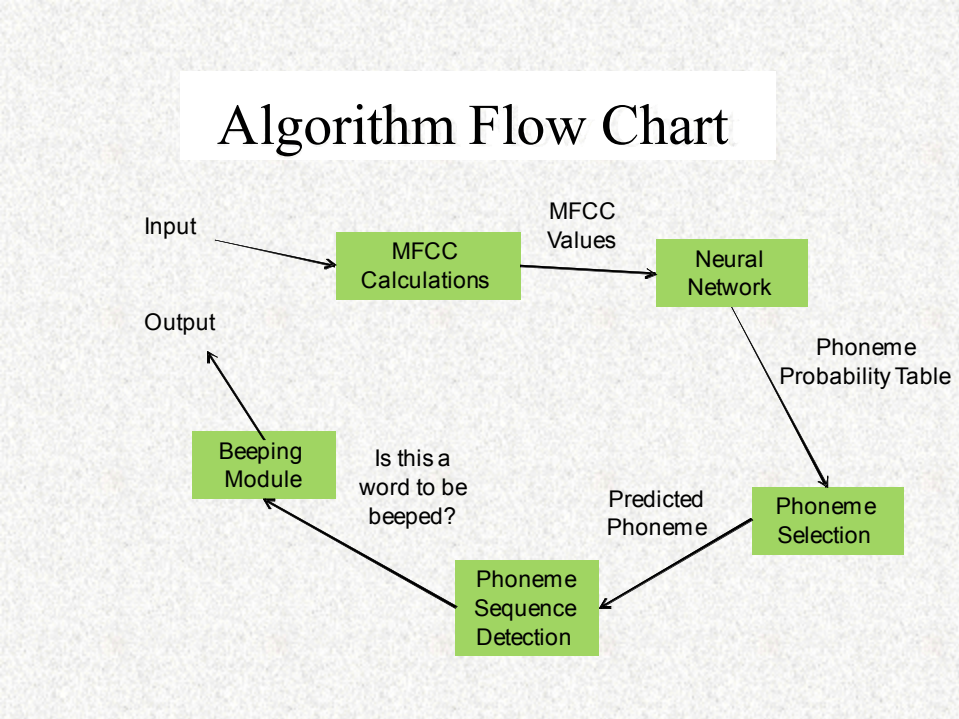
3.1 Data Path Diagram



In our implementation, there is no data transfer between the DSK and the PC. The neural network is contained entirely on the DSK, and the audio inputs and outputs are directly read to the DSK through the codec. All processing and calculations are done on the DSK itself.

4. Algorithms

4.1 Algorithm flowchart



By basing our algorithm on [1], we first obtain 50 MFCC values from the input speech at regular timed windows. These are fed into the input nodes of the neural network. The neural network outputs the probability of a phoneme occurring in the current window. Our segmentation into 63 phonemes was based on the phonemes categorized by TIMIT.

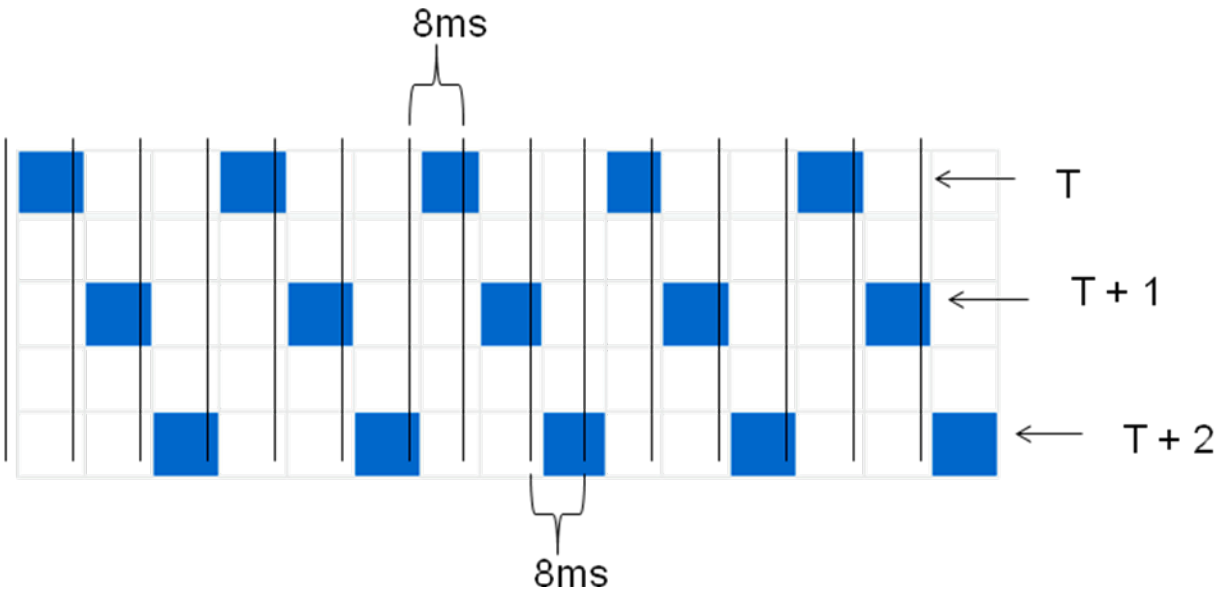
Following that, a modified Viterbi search uses the phoneme probability density function to decide the most likely phoneme present. A detection module will constantly check for certain sequences of phonemes that we wish to censor. If detected, the beeping module will replace the data in the buffer corresponding to the phonemes with a sine wave.

4.2 Audio Input Processing

The audio input we used was sampled at 16 kHz. Other studies have used inputs sampled at 8 kHz, but those primarily involved telephone signals[2]. Human voice frequencies generally range from 100 Hz to 6 kHz, so this sampling rate allows for sufficient frequency resolution.

In the processing of the speech signal, we segmented the input into 8ms frames. At each point in time, along with the current 8ms frame, frames at 24ms and 48ms before and after the current frame are also considered, to provide contextual data in predicting the phoneme in the current frame. [1]

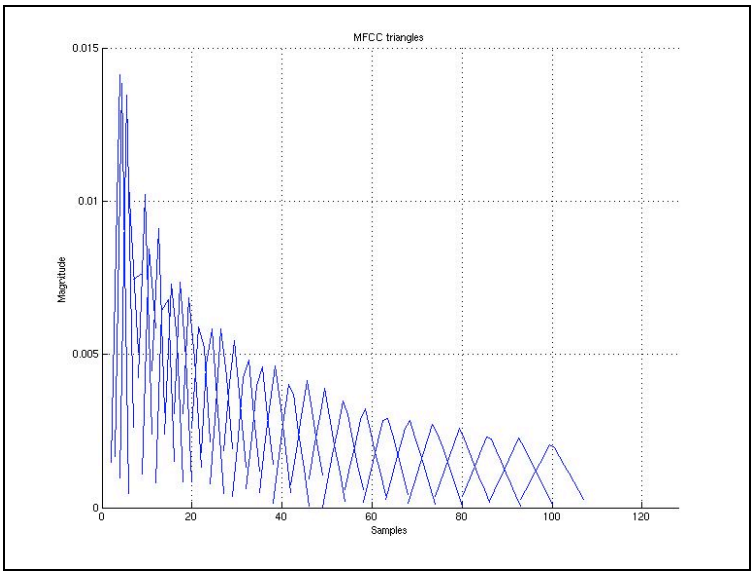
With this method, frames are repeatedly sampled after every third frame, meaning that there is an ‘overlap’ of frames after every three frames are processed.



4.3 MFCC (Mel Frequency Cepstral Coefficients)

Humans tend to recognize speech by grouping frequencies in a logarithmic scale. MFCCs help in this area by utilizing log-energy triangular windows. The FFT is first found for the signal, following which the log amplitudes of the spectrum obtained would be mapped onto the mel scale, using triangular overlapping windows. The Discrete Cosine Transform is taken for the list of mel log-amplitudes, to obtain the MFCCs.

For our project, 10 MFCCs are taken for each 8ms frame, which will provide us with minimal input nodes into our neural network, while maintaining frequency differentiation. With 5 frames being considered at any one time, 50 MFCCs are calculated which serve as the inputs to the neural network.



Triangle windows used to calculate log-energy values in FFTs

4.4 Neural Network (NN)

The neural network used is a feed-forward NN with sigmoid units. There are 50 input nodes, corresponding to the MFCCs calculated for the time frames. We used one hidden layer with 30 hidden nodes, which would allow for sufficient generality after our intended training. Lastly, the NN had 63 output nodes, corresponding to the 63 different phonemes as listed in TIMIT, the database we used to train the NN. The output is in the form of a probability density function, with probabilities given to each of the 63 phonemes, reflecting how probable it was that that particular phoneme was present.

Notably, we included noise/silence as an output node, as if it were one of the phonemes to be recognized. This would serve as a sort of speech detection mechanism, so that the neural network can screen out sound inputs not corresponding to actual phonemes.

It is crucial for us to have sufficient training data as NNs require large amounts of training data for it to be sufficiently accurate in generalizing.

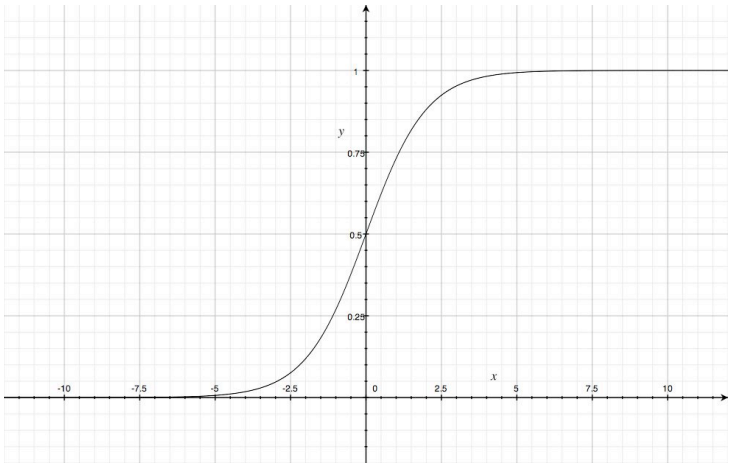
Calculations:

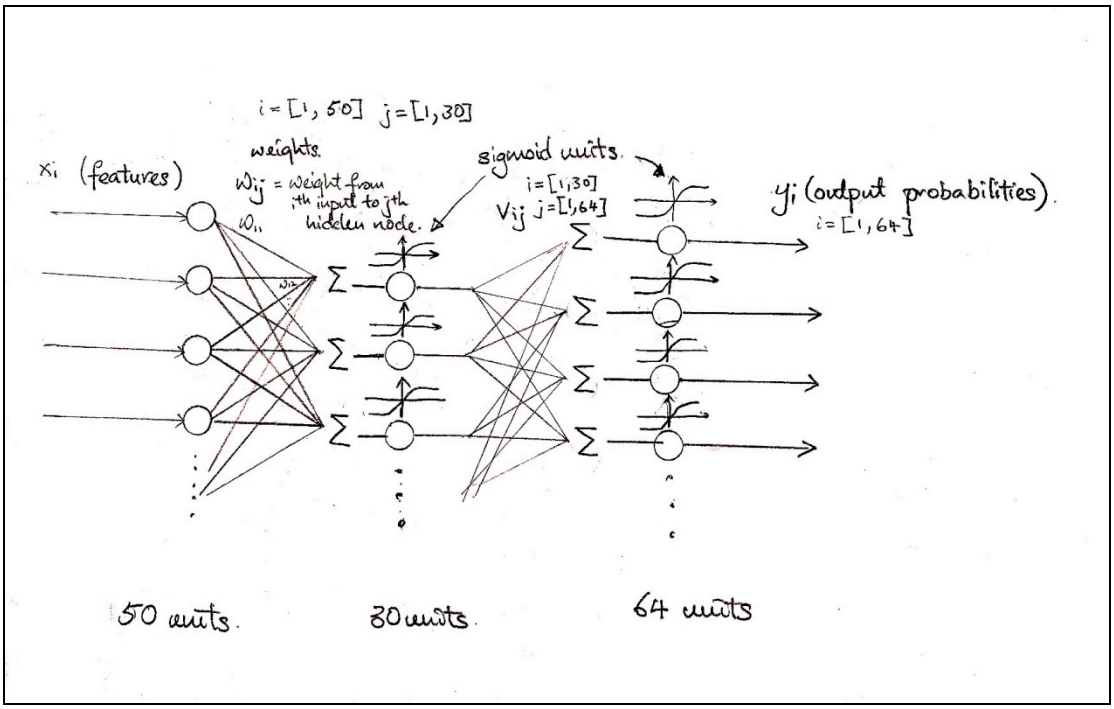
- 50x30+30x63 = 3390 weights
- 3149 x 300 frames = 1.8m frames
- 1.8m / 3390 = 531 frames per weights

With the calculations above, we notice that we have 3390 weights to train. With 3149 sentences and each having an average of 300 frames (or windows), this provides us with 945000training instances. This would mean that we would have about 280 training data for each weight, which is more than sufficient to generate reliable weights for the neural network.

The transfer functions for the hidden layer and the output layer will be the logarithmic sigmoid.

Sigmoid: $Y = 1 / (1 + \exp(-X))$



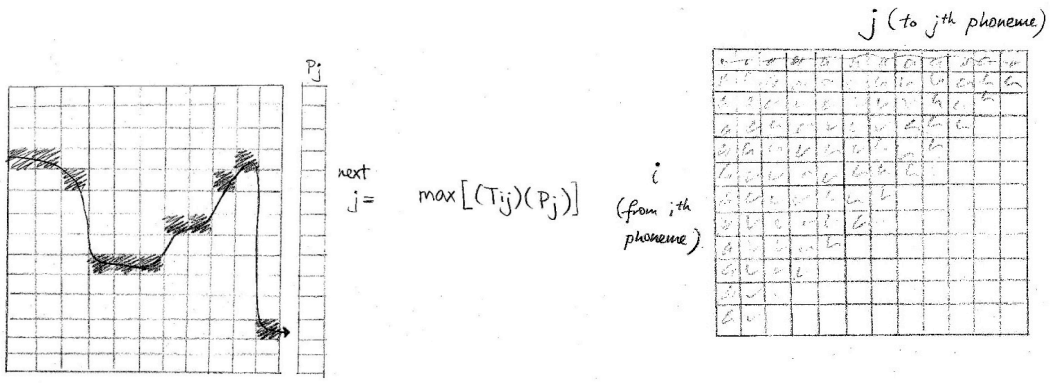


4.5 Phoneme Selection (Viterbi)

The Viterbi algorithm is a dynamic programming algorithm, which we utilize in deciding which phoneme is the most likely one present in the frame being processed. In this case, the most probable phoneme sequence is the most probable previous sequence coupled with the most probable current phoneme, weighted by the most probable transition from the last phoneme to the current phoneme.

- **$Next_j = argmax\ [(T_{ij})(P_j)]$**

The algorithm theoretically allows noisy (incorrect) output from the neural network to be reduced as utterance of phonemes are usually no independent. As such, by weighing the probabilities of the presence of phonemes using past sequences of phonemes and phoneme transition probabilities will help increase the percentage accuracy.



Normally, the Baum-Welch algorithm is used to find the unknown transition probabilities (T_{ij}) as seen in [1]. However, as we have 1.8m frames in our training set, we have sufficient data to estimate the transition table by simply calculating the frequency of the observed transitions in the training set, hence eliminating the need to use the Baum-Welch algorithm.

5. Speed and Memory Issues

Memory Storage

As we intend to have the entire program to be loaded on the DSK, we needed to ensure that there is sufficient memory for the global variables and program to be loaded on the DSK.

We intended to use single precision for the entire project. However, due to slight discrepancies we obtained from the output from the DSK compared to the output from MATLAB, we modified the program to use double precision. This doubled the total memory requirement for all global variables to 8 bytes each. The following shows the calculations for the components that required the most memory.

Features

Hamming window: 128*8 = 1024 bytes

Double precision FFT requires 128 complex twiddle factors: 128 * 2 * 8 = 2048 bytes

MFCC Triangle windows: 32*16*8 = 4096 bytes

Neural Network

Weights: (30*50 + 30*63 + 30 + 63)*8 = 27864 Bytes

Viterbi

T[ij]: 64*64 *8 = 32768 bytes

Total memory required = 67800 bytes

Since the DSK has 256k bytes of L2 memory, we have sufficient memory for our project.

Speed of Algorithms

Cycles required for major calculations per frame

Estimated:

Features (5 x FFT): 3730 *5 = 18650 cycles

Neural Network: 3420 multiplications/additions = 3420 cycles

Viterbi: (64 multiplications + 64)* 64 + 100 (add that node to list) = 8292 cycles

Total: 30362 cycles per frame

With the speed of the DSK at 225 MHz , which translates to 236 million cycles per second, we would have approximately 1.89 million cycles for each frame of 8ms available for calculations. We aim to complete everything in half of the total cycles as our calculations do not include interrupt handling which occurs every 1/16000 of a second. Therefore, we should aim for a total cycle count of 900,000 cycles. It is clear from the calculations above that we can easily run the algorithm in real time.

Actual Profiled cycle times:

Features (5 x FFT) = 292238 cycles

Neural Network = 17274 cycles

Viterbi = 392895 cycles

Total = 702407 cycles per frame

The calculated results differ drastically from the expected amount because the initial estimates do not consider the cycles needed to handle the interrupts.

Also, our Viterbi algorithm can probably be written be in a more optimized way.

Phoneme Detection Results:

start sample	end sample	ideal output	most probable	Viterbi
0	2120	55	55	55
			11	11
			55	55
2120	4010	12	12	12
4010	4840	33	33	33
4840	6057	31	31	31
6057	8120	37	37	37
8120	8440	59	59	59
8440	8752	2	2	2
8752	9317	29	29	29
9317	10652	51	51	51
10652	11240	59	59	59
11240	11720	2		
11720	13426	38	38	38
			29	29

			51	51
			38	38
13426	14280	27	27	27
14280	15240	63	63	63
15240	15600	6	6	6
15600	18120	11	11	11
18120	21080	47	47	47
21080	21389	5	5	5
			31	31
			5	5
21389	22200	8	8	8
22200	22767	50	50	50
			55	55
			37	37
22767	24061	20	20	20
24061	24480	60	60	60
24480	25107	3	3	3
25107	26151	27	27	27
			55	11
				55
26151	27240	33	33	33
27240	29326	11	11	11
			47	
29326	30040	33	33	33
30040	31707	28	28	28
31707	34520	42	42	42
34520	35885	12	12	12
35885	36386	54	54	54
36386	37263	28	28	28
37263	38600	42	42	42
38600	39227	7	7	7
39227	40637	51	51	51
40637	41302	8	55	55
			59	59
			55	55
41302	43466	42	42	42
43466	44427	26	26	26
44427	46390	29	29	29
46390	48228	34	34	34
48228	50280	51	51	51
50280	55040	55	12	12
			55	55
			12	12
			55	55
			51	51
			55	55
			11	

The first 2 columns show the time frame for each row. The third column is the actual classification from TIMIT. The fourth column is the output from our code if we were to choose

the phoneme present solely from the highest probability as we obtained from the neural network. The fifth column is the output from the Viterbi search with the previous neural network output.

The Viterbi search removed much of the instability present due to noise. Also, both instances in which the phoneme lengths were close to the length of the window of 128 samples (e.g. at sample 11240) were not detected.

Optimization

Initially, we decided to use the FFT in C provided in the TI library (DSPF_SP_CFFTR2_DIT). However, with that FFT, the algorithm was too slow. The calls for 5 FFT (and additional 5 DCT) took an approximate 1.3 million cycles with interrupt handling. Hence, we decided to use the hand optimized assembly code written for the equivalent FFT (with restrictions) and this brought down the time required for the 5 FFT (and additional 5 DCT) to an amazing 250,000 cycles with interrupt handling.

It must also be noted that –o1 optimization was used and not –o3. This is because at –o3, the FFT does not produce the right result. We think that at –o3, the assembly is so heavily optimized such that the processor is unable to tell which part of the assembly code corresponds to which line of C code. Thus during an interrupt, it is possible that the processor handles an interrupt during a crucial step, causing inaccuracies in the FFT calculations.

Also, we initially thought that the sigmoid transfer function would required too much cycles as exponents are likely to require many cycles to compute. Therefore, we wrote a 512-entry lookup table where we could linearly interpolate the values of the exponents required. However, we compared it to using an actual exponent implementation of the transfer function and found of that our implementation of the lookup table was much slower (70,000 cycles) than performing the actual calculation.

On hindsight, we attribute this to our poor implementation of the lookup table as we used a linear search to find the closest values. Also, it could be that the lookup table was not stored in cache as the lookup was done after each calculation of the input to the nodes. This results in having the values of the table removed from cache whenever we need the values of the weights to calculate the input to the hidden and output nodes. This could have been fixed but since we require a very accurate representation of our MATLAB code, we have decided to use the actual computation of the exponent in favor of implementing a better lookup table.

The Viterbi algorithm if implemented on its entirety would require us to store all previous observed neural network output values. Also, searching for the most probable sequence observed would require a search on all the previously observed output values. However, as the DSK has a limited memory and as we want the program to be real-time, we have decided to implement a simplified Viterbi at the expense of accuracy.

Instead of storing all the previous outputs from the neural network, we only store one previous output from the neural network. We then perform the Viterbi search using only the previous output. This helped to removed some instability in the phoneme detection, as discussed later. We did not use more than one previous output as if we were to implement a close to ideal

one, we should store the outputs from the neural network if the corresponding sound data is still present in the buffer. However, doing this would require storing 375 sets of neural network outputs, which is too large for the DSK to handle.

6. Procedure and Results

Code was available in C for the implementation of the MFCC calculations, the NN and the Viterbi algorithm. However, this proved too difficult to incorporate into our project. As such, we decided to write all of the code for the DSK on our own.

6.1 Using the TIMIT Database

The database that used in our project is the TIMIT database, which consists of 623 speakers from 8 dialects each saying 10 different continuous sentences. Each of the sentences contains approximately 30 phonemes and so there are 6230x30/64 = 2920 instances of each phoneme present in the entire database.

Each sentence spoken by a speaker in TIMIT comes with a sound file, a text document indicating the sentence, and a text document indicating the placement of phonemes in time. It is this last file that indicates the correct output to each time frame within the sound file, so we make use of this information in the training of the neural network.

First of all, the sound files in TIMIT needed to be converted from the NIST/Sphere format to the more recognized Windows WAV format. This was done using a program called Sound eXchange (SoX), which is available off the web.

The TIMIT database is sufficient for the training of our neural network, from the following calculations:

Total Number of Weights in the Neural Network

= 50 input neurons x 30 hidden neurons + 30 hidden neurons + 63 output neurons

= 3390 weights

Total Number of Available Frames

= 3149 Wav Files x 300 Frames

≈ 945000 Frames

Number of Frames available for training per weight

= 945000 / 3390

= 280 frames per weight

Next, this database was further divided up into 3 sets:

1) Test Set – Contains 50% of all the wav files = 3149 files in total. This training set will be used solely for training the neural network to fire the correct neuron when a certain input is fed into it.
2) Validation Set – Contains 35% of the wav files = 2167 files in total. The validation set is used to validate the training results as well as to set certain parameters such as the number of hidden neurons. The number of hidden neurons affects the ability of the neural network to generalize across different scenarios and thus is an important factor for us to check and validate.
3) Test Set – Contains 15% of the wav files = 961 files. This is to test the results of the trained neural network to check if the training was correct.

6.2 Obtaining of Neural Network Weights in MATLAB

6.2.1 Procedure

We used the MATLAB Neural Network toolkit to train the NN to obtain the weights to be used on the NN in the DSK. We also implemented the training and testing modules using the pre-defined functions in the Neural Network toolkit library.

PR: R x 2 matrix of min and max values for MFCC inputs

Si: Size of ith layer, for Nl layers. [30 63] in our case since we have 30 hidden neurons and 63 output neurons.

TFi Transfer function of ith layer (default'tansig'). We used {‘logsig’ ‘logsig’} as we are using sigmoid function as our transfer function for the network. Moreover, the ‘log’ function will make the outputs be in the range of [0,1].

BT FBackpropagation network training function (default = 'trainlm'). We used the ‘trainrp’ (Resilient backpropagation) in our project as it is more stable than the rest of the other training functions such as ‘traingda’ and ‘traingdx’. Moreover, it provides us with a training speed of 3 to 4 times faster than ‘traingda’ and ‘traingdx’.

BLF Backpropagation weight/bias learning function (default='learngdm'). Default value was used.

PF Performance function (default = 'mse'(Mean Square Error)). Default value was used.

The above-mentioned function serves to create our Neural Network of 50 input neurons, 30 hidden neurons and 63 output neurons. The number of columns of the input matrix automatically specifies the number of input neurons.

Next, we calculated the features (MFCC values), which served as inputs for the neural network training.

Wavfile A .wav file that contains the sentence required for computing of features.

Phnfile A pre-processed .txt file that contains all the phonemes tags that are pre-defined by TIMIT and are present in the sentence.

This function takes in a wav file and a text file that contains the phonemes present in the wav file to produce an [R x 113] matrix. This matrix serves as the input to the neural network where R is the number of 8ms frames in the .wav file. The first 50 columns of the feature matrix contain the features of the wav file while the next 63 columns contain information on the correct phoneme present in the frame, in 1-hot encoding. The diagram below serves to illustrate a sample input to the neural network.

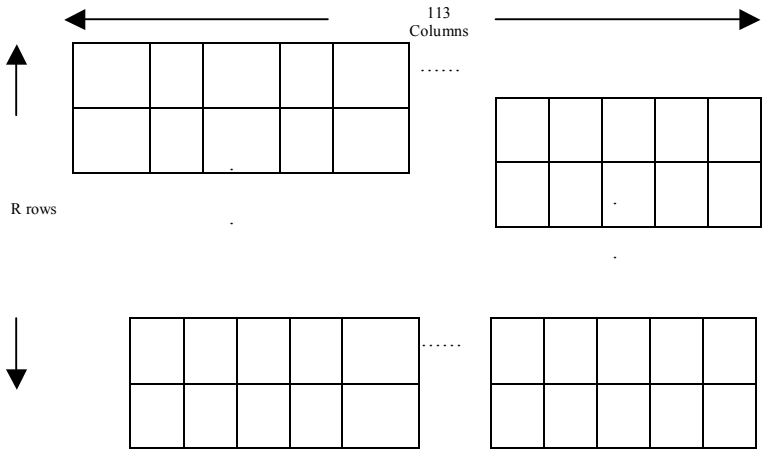


Fig 1.1 Sample input to the Neural Network for training and actual purposes

This matrix is fed into the training function (see Code section for more details). This function pre-processes all the information required for the actual neural network training, and returns the trained network after training with every file.

```
net = training(Features, Net, Goal)
```

Features Features matrix computed from the computefeatures() functioned explained above.

Net Neural Network

Goal Desired MSE for neural network

Firstly, there is a separation of the features matrix into the input and the target. The input to the neural network is the 1 – 50th columns of the feature matrix while the target is from the 51st - 113th column. Each target of the neural network is a [63 x 1] matrix with all zeroes except for the target output neuron, which will be valued at 1. This tells the neural network to fire that particular neuron when a certain input is provided. Each output neuron corresponds to the phoneme tag that is pre-defined by TIMIT. Next, we fed all these arguments into the "train" function of the MATLAB Neural Network library.

```
[net,tr,Y,E,Pf,Af] = train(net,P,T,Pi,Ai,VV,TV)
```

Net	Network
P	Network inputs (features matrix in our case)
T	Network targets (Target matrix explained above)
Pi	Initial input delay conditions (used default = zeros)
Ai	Initial layer delay conditions (used default = zeros)
VV	Structure of validation vectors (used default = [])
TV	Structure of test vectors (used default = [])

This returns:

Net	New network
Tr	Training record (epoch and perf)
Y	Network outputs
E	Network errors
Pf	Final input delay conditions
Af	Final layer delay conditions

For our neural network training, the following parameters and their corresponding values were used:

Epochs = 1000000

Network Training Goal = 0.0001

The rest of the parameters were assigned their default value.

After retrieving the new network, we passed it back to the training function, which reinitialized the newly calculated weights and retrained the neural network again. This process is repeated until all the training files are processed for training.

After training the neural network, we wrote our own function to test the accuracy of the neural network. This function uses the trained neural network and performs a direct testing of the accuracy of the output of the neural network by performing the following tasks:

1) For every file in the testing set, compute the MFCC values of the wav file and feed the input into the neural network by utilizing the sim() function of the MATLAB Neural Network Library. This function simulates a neural network and is as follows:

```
[Y,Pf,Af,E,Perf] = sim(net,P,Pi,Ai,T)
```

Net Trained Network

P Network inputs (MFCC values)

Pi Initial input delay conditions

(zeros since we are not using a time-delayed neural network)

Ai Initial layer delay conditions

(zeros since we are not using a time-delayed neural network)

T Network targets

(zeros since we developed our own method of validation of results)

This returns:

Y Network outputs

Pf Final input delay conditions

Af Final layer delay conditions

E Network errors

Perf Network performance

2) Using the neural network output returned by the sim() function, we compare it with the actual phoneme present in the 8ms frame that was used as the input for the simulation. Starting at value 0, the errorcount variable keeps track of each wrong phoneme determined by the trained neural network.

3) After iterating through all the testing files, we calculate the final percentage error of the neural network by

Percentage Error = errorcount/totallines * 100%

where totallines is a variable that keeps track of the total number of phonemes present in the testing set since each line of the MFCC matrix corresponds to one phoneme present in the 8ms frame.

6.2.2 Problems

Our first implementation of the training module did not produce the desired neural network due to the fact that we did not input the whole features matrix as an input to the neural network. Instead, we inserted a row of the features matrix as an input and repeated this process until the whole matrix is being used for training. This had resulted in a neural network that is capable of recognizing 1 particular phoneme and consequently an error percentage of the 90%-97% despite the large amount of training data. This high error percentage obtained is due to the fact that the initial method of training had resulted in a neural network that does not recognize the spatial relationship between different phonemes and thus tends to forget the rest of the earlier phonemes that it had been trained before.

Our course of training had not been smooth due to the fact that the CPU used for training purposes was repeatedly being turned off. As a result, we wasted a vast amount of time to retrain the neural network. Coupled with this issue is the fact that the training of the neural network takes almost a day to complete. This was one of the major obstacles to the completion of the project.

With regards to the testing and validation of the trained network, we have tried several means to accomplish the process of testing and validation but to no avail. The PC in the lab does not have sufficient memory for us to generate a validation or test matrix based on the number of wav files allocated for each set: 2167 wav files for validation and 961 files for testing. Our validation set generation was continually stopped by people turning off the computers while the simulation was running. Each generation of the validation and test matrix takes up to 2 days. Due to the time constraint, we decided to train the neural network with no validation and test set.

After implementing all the modules for the C code for the DSK, we had trouble matching the output with the output from MATLAB. Upon closer inspection, it turns out that the output from the neural network was only accurate to only 4 decimal places. Hence, the most probable phoneme detected was different on the DSK compared to MATLAB. We thought that it was a precision issue and therefore changed all the code to use double precision, including the FFT. However, this did not solve the problem. The MFCC values calculated were still only consistent to only 4 to 5 decimal places. We checked the output of each step with MATLAB and eventually found the difference. The cosine values that we store for the DCT was inconsistent with the values generated.

As strange as this may seem, after the values we need to calculate the DCT are generated, storing them in the memory changed those values. Running short of time, we decided to not store the values and calculate it every time we needed it. This finally solved all our problems

and had our output from the neural network accurate to 9 decimal places. With that, our code on the DSK produces results that closely resembles that in MATLAB.

Another problem that we faced was that we were unable to obtain computers to safely train our neural network. A neural network takes approximately 6 hrs to train after heavy optimization. However, there were times when we were generating the validation and test set that the computer is switched off or hijacked. This greatly impeded our progress.

7. Discussion

Under controlled settings, with the input sound file stored in the DSK, we obtain a 1% error rate on the identification of phonemes. This allows a reasonable detection of words provided the error does not occur within the word itself. However if we were to collect input from the real world, the output from the DSK is very erratic and is not usable.

We think this is because the neural network is not robust enough to handle actual microphone input. It might require more training or training data with more noise.

Also, we are not getting very good phoneme prediction when there is noise as the Viterbi is not implemented in its entirety. Since the dependency is only on the previous probability density, noise would propagate through subsequent frames. Possibilities include either storing a longer history of previous neural network outputs or other metrics such as only recognizing a phoneme after a few continuous frames. The detection could also consider the average length of particular phoneme. These are several improvements that we can make for post processing the output from the neural network. However, in 6 weeks, the complexity of other issues took precedence to algorithm efficiency.

Upon further discussion with Prof Casasent, we re-trained two separate NNs. TIMIT is structured such that each of the 623 speakers say 2 common sentences, and 8 other sentences which may or may not be repeated by other speakers. We removed all the files with the 2 common sentences, and trained the NN using the remaining files in TIMIT. The testing was then done with the files that we removed. For this case, the NN has seen the speakers before, but not the sentences. We obtained a 0.059% error rate.

In the second case, we removed 10 speakers (and all their 10 sentences) and trained the NN using the remaining files. Testing was done using the 100 sentences that we removed. The NN did not see the speakers before, even though it might have seen certain sentences before. This resulted in an error rate of 0.96%.

One last training was done, which was to remove 10 speakers saying 1 different sentence each, and train with the remaining files. Testing was then done with the 10 sentences removed earlier. The NN in this case has seen the speakers before, and also the sentences but spoken by other speakers. The error rate was 13.1%. We suspect this is due to the huge variability in the 10 sentences of the test set.

Professor Casasent also mentioned that we should try using frequency count as a way to determine if any phoneme is present. In an averaged length phoneme, our code will repeatedly

generate the probabilities for it over its duration. Therefore, another way to determine the most probable phoneme would be the phoneme that has the most number of times being the highest probable phoneme within that period.

Because of the limited time, we only managed to write this method of classification for testing on MATLAB. Also, since testing will take about 4 hrs to complete (961 files), we did not test it using the whole test set.

However, we did try against the sound file that we demoed.

Error rate of Viterbi: 17/417 instances = 4.07%

Error rate of new algorithm: 2/41 instances = 4.87%

Although it seems that the new algorithm did not fare as well, but the reason it has a higher percentage error is because the total number of phonemes is much smaller. It does have a better phoneme prediction because of the 41 phoneme utterance, only 2 are wrongly classified. This means that for the most part, the sequences are correctly detected. However, when Viterbi is used to classify the phonemes in frames, we have 17 wrongly classified frames distributed across the entire sentence. The result is that the sequences which the computer can recognize as a match is drastically reduced.

8. Schedule and Assigned Tasks

We initially intended to complete the training of the NN in MATLAB and the various algorithm implementations on the DSK by the update. However, due to the numerous problems we had generating the validation set and training the NN, we were unable to obtain the weights needed to put into the DSK. We had however completed the MFCC implementations in MATLAB. It took several tries to finally get the NN training to work correctly, and we then proceeded to debug the DSK code using an actual file in TIMIT.

Desmond researched using the neural network in MATLAB to train and test the data, and wrote several .m files to train and test the data.

Aaron pre-processed the data files in TIMIT such that they were in a MATLAB-readable format, and converted the phonemes using a one-hot encoding, as well as writing most of this report.

Jiamin wrote the DSK code for the implementations of MFCC calculations, NN and Viterbi, as well as the post-processing of the data after the output from the Viterbi algorithm.

Mingwei implemented the pre-processing of the DSK data, the word detection and beeping module, the MFCC calculations in MATLAB, as well as rewriting the entire training and testing module used in MATLAB upon discovery of several bugs.

9. References

[1] Hosom, J., Cole, R., Fanty, M. (1999) Speech Recognition Using Neural Networks at the Center for Spoken Language Understanding. Retrieved, Oct 1, 2007, from http://cslu.cse.ogi.edu/tutordemos/nnet_recog/recog.html

This reference provided the basis for our project. We used idea of combining Artificial Neural Networks and Hidden Markov Models (Viterbi) from this paper.

[2] Ouzounov, A. (2006) Robust Features and Neural Network for Noisy Speech Detection. *Cybernetics and Information Technologies, 6, 3.*

This paper did a comparison on which features if used on a Neural Network will provide the best results. Although MFCC was not the best feature, it was the easiest for us to implement and yet have a decent
recognition rate.

[3] Davis, S.B., Mermelstein, P. (1980) Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-28, 4.*

In paper 1, 26 features were used (a combination of MFCC and other parameters). However, we were reluctant to use that many input nodes per window. In this paper, it was showed that only using 10 MFCC
coefficients was sufficient to have a decent recognition rate. Also, we used the algorithm provided in the paper to calculate our MFCC values.