# Compressed DNN based Automatic Speech Recognition Engine

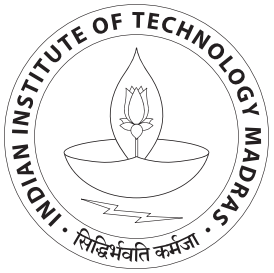
*A Project Report*

*submitted by*

**DEEPALI GARG**

*in partial fulfilment of the requirements*
*for the award of the degree of*

**BACHELOR OF TECHNOLOGY &**
**MASTER OF TECHNOLOGY**




**DEPARTMENT OF ELECTRICAL ENGINEERING**
**INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.**
**18 May, 2019**

# THESIS CERTIFICATE

This is to certify that the thesis entitled **Compressed DNN based Automatic Speech Recognition Engine**, submitted by **Deepali Garg** (**EE14B081**), to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelors of Technology** and **Master of Technology**, is a bona-fide record of the research work carried out by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Dr. Janakiraman Viraraghavan**
Research Guide
Assistant Professor
Department of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 18 May, 2019

# ACKNOWLEDGEMENTS

I would like to thank my guide, **Dr. Janakiraman Viraraghavan**, for his exemplary support and guidance during the past two years of our interaction. He has always guided me in the right direction, motivated me to work harder and pushed me to expand my horizons. He has always provided his intellectual and moral support whenever required and helped me grow both academically and personally. It was a pleasurable and knowledge gaining experience working under his guidance.

I would also like to thank everyone at the Embedded Systems lab. Especially, PhD scholar **Pani Prithvi Raj** and another DD student **Pakala Akhil** for actively discussing the problem and helping me out at places when I was stuck.

Finally, I would like to express my gratitude to my parents, brother and friends for their continuous support.

# ABSTRACT

KEYWORDS: Automatic Speech Recognition (ASR), Hidden Markov Model (HMM), Gaussian Mixture Model (GMM), Weighted Finite-State Transducer (WFST), Deep Neural Networks (DNNs)

The application of speech recognition in devices in which internet connection is slow or unreliable, can not be served by existing software or hardware decoders. The memory, bandwidth and power requirements of speech decoding are excessive. We aim to optimise the existing hardware decoders used for speech recognition, so that we can achieve real-time speech decoders which can be deployed on gadgets with limited battery, and with the need of internet connection. The existing technology uses HMM based decoders, currently a WFST. This requires acoustic modeling and transition modeling. Currently, acoustic modeling is done using GMMs. Each WFST arc (which are usually millions) requires different probability-distribution functions and thus requires different GMM parameters to be stored. This creates a huge load on memory and bandwidth, with ambitious aim to place all the memory on chip and to reduce bandwidth requirement significantly, we try switching the acoustic modeling to DNN-based, with existing literatires promising advantages. This work tried an implementation of DNN-HMM based ASR, and then compression techniques on the DNN in order to reduce the overall memory requirement significantly.

# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** **i**

**ABSTRACT** **ii**

**1 INTRODUCTION** **v**

1.1 Hardware Speech Recognition Engine . . . . . v

1.2 Existing Technology : HMM Framework . . . . . vi

1.3 Existing Architecture . . . . . vii

1.3.1 Front-End . . . . . vii

1.3.2 Viterbi Search . . . . . viii

**2 DNN based Acoustic Modeling** **xi**

2.1 Choosing a Modeling Framework . . . . . xi

2.2 Acoustic Modeling with DNN . . . . . xii

2.2.1 DNN Training . . . . . xii

2.2.2 DNN-HMM execution . . . . . xv

2.2.3 Current Status of Implementation . . . . . xvi

2.3 Comparison : GMM vs DNN . . . . . xvii

**3 DNN Compression** **xviii**

3.1 Quantization Heuristic . . . . . xviii

3.1.1 Motivation . . . . . xviii

3.1.2 Pseudo Code . . . . . xviii

3.1.3 Code . . . . . xix

3.1.4 Compression Results . . . . . xx

3.2 Huffman Encoding . . . . . xxi

3.2.1 Motivation . . . . . xxi

3.2.2 Example . . . . . xxii
3.2.3 Compression results . . . . . xxii

**4 Future Work** **xxiii**

4.1 DNN Model Compression by Re-training . . . . . xxiii
4.2 Hardware Optimization . . . . . xxiii
4.3 Language Modeling . . . . . xxiv

# CHAPTER 1

# INTRODUCTION

## 1.1 Hardware Speech Recognition Engine

The major existing application of speech recognition is in internet-connected devices, in which cloud-based decoders can run in real-time, where decoding adds very little in the latency. The application of speech recognition in devices in which internet connection is slow or unreliable, or in local operations, can not be served by existing software or hardware decoders, since the overhead of using internet-based decoders is high. Such applications include, wearable gadgets, home appliances and industrial equipment.

While mapping the modern speech recognition techniques to digital circuits, reduction in power consumption, memory bandwidth to the levels of embedded systems is desired. There are several limitations to these hardware based speech recognition systems. The memory, bandwidth and power requirements of speech decoding are excessive. Digital circuits are fixed functions, they can't be reprogrammed to keep up with the algorithmic and statistical modeling technique developments. existing systems, requires complex external components to realise complete speech-to-text conversion.

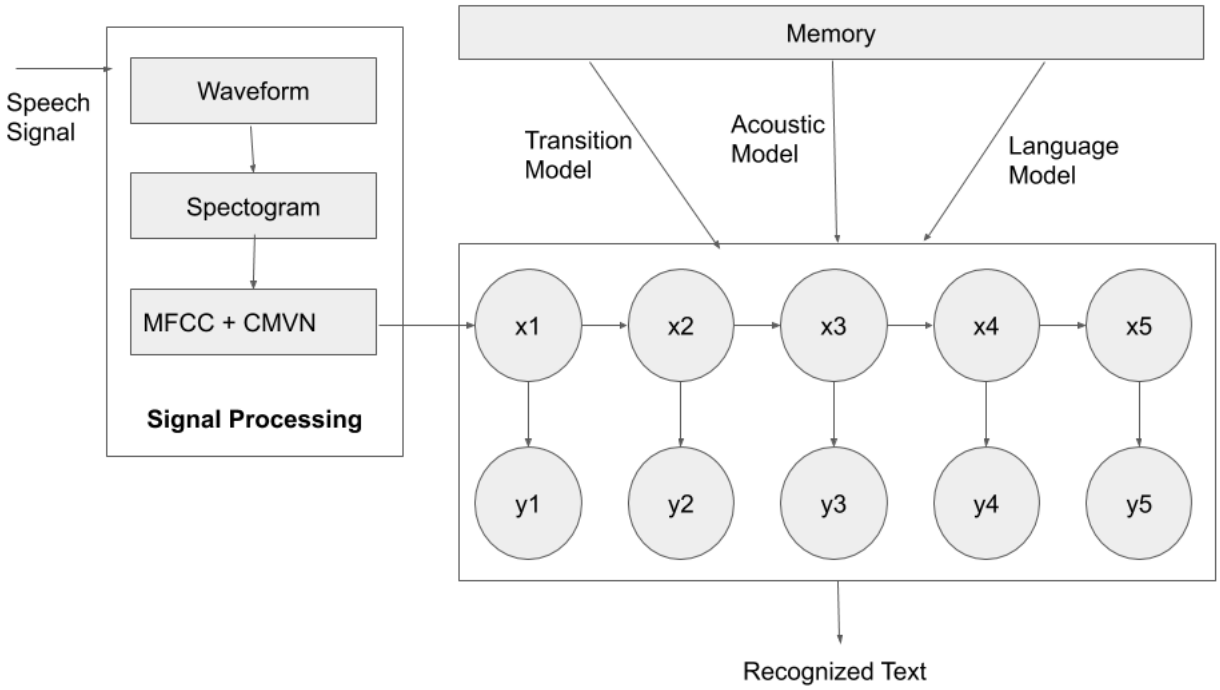
## 1.2 Existing Technology : HMM Framework

Current speech recognition technology uses Hidden Markov Models (HMM). Exploiting the regular structure of connections in Markov Models, approximated linear time inference using Viterbi algorithm makes statistical speech recognition tractable.

The HMM framework requires modeling the dependencies between variables. There is transition model $p(x_{t+1}|x_t)$ and emission model $p(y_t|x_t)$. The $x_t$ are the hidden states, $y_t$ are the observed states. The transition model incorporates information about language - vocabulary and grammatical constraints. The emission model describes how observations change with the hidden states. These models are influences by speaker's vocal tract, microphone and acoustic environment.

The Weighted Finite-State Transducer (WFST) is a state-machine representation that allows each component to be described separately, and then composed together for the entire speech recognition process. All possible transitions between states are expressed as weighted, labeled arcs. A WFST-based decoder can complete tasks of varying complexity in different domains and languages by substituting the appropriate model parameters.

The transition probability $p(x_{t+1}|xt)$ is the weight of the WFST arcs leading from state $x_t$ to $x_{t+1}$. The emission probability $p(y_{t+1}|x_{t+1})$ is specified by acoustic model,typically a Gaussian Mixture model. We in our project intend to replace this acoustic modeling from GMMs to Neural Netwroks to gain hardware advantages.




## 1.3 Existing Architecture

### 1.3.1 Front-End

Speech audio signals are sparse in the sense that time-domain sampling captures redundant information. We use mel-frequency cepstral coefficients (MFCCs) to concisely represent relevant characteristics of the signal. The MFCCs low dimensionality simplifies classification, and channel effects (convolution) are additive in the cepstral domain. Several standalone feature extraction devices have been reported in the literature, including DSPs, FPGAs, fixed-function digital circuits, and low-power analog front-ends. The front-end could be used to provide features to an external (e.g., cloud-based) decoder or to the on-chip decoder.

MFCCs are derived from an audio time series via a chain of conventional signal processing blocks including an FFT, mel-scale bandpass filter bank, and DCT. The

audio is split into a series of overlapping frames 25 ms long with a 10 ms pitch. In addition to computing 12 cepstral coefficients and the log power for each frame, we extract first- and second-order time differences and concatenate them into a 39-dimensional feature vector at 16 bit resolution. Cepstral mean normalization is approximated by subtracting a 10 second moving average from the feature vectors. The FFT exploits the real-valued nature of the input signal to operate on half as many points, and the bandpass filter bank employs two multipliers which are reused across the 26 bands. The circuit has an area of 51.8 k gates (plus 107 kb of SRAM and 66 kb SROM) and requires a clock speed of at least 625 kHz to process a 16 kHz waveform in real-time.

### 1.3.2 Viterbi Search

Viterbi search begins with an empty hypothesis for the utterance text and incorporates a stream of information from the feature vectors to develop a set of active hypotheses, represented by states in the HMM. Each of these hypotheses is modeled using the WFST and GMM; if its likelihood is sufficiently high, it will be saved. The forward pass of Viterbi search propagates a set of hypotheses forward in time, from the active state list of frame to that of frame $t + 1$.

**Hypothesis Fetch :**

Each hypothesis is read from the active state list for the current frame (frame ). The active state list is implemented as a hash table in SRAM with open addressing and collisions resolved by linear probing. We store accepted states for the next frame in a separate SRAM and swap the two SRAMs using multiplexers at the end of each frame. In these hash tables, the key is a unique state ID (the memory address of the

state in the WFST model) and the value is a structure describing the state. Hash table operations become slower as more states are stored (increasing the number of collisions), but this latency is masked by the much longer time required by WFST and GMM operations that access external memory. To reduce the need for memory accesses (which require additional logic and create pipeline stalls), arc labels and other metadata are carried through the pipeline along with state information.

**Arc Fetch :**

Each state ID is an index into the WFST model, from which we can retrieve the parameters of all outgoing arcs. These include all of the information (except a final score) that will be stored in the active state list if the hypothesis is accepted.
While this operation requires far less memory bandwidth than fetching GMM parameters, the arcs retrieved during a typical frame are distributed sparsely across a large memory space: 193 MB for the 5,000 word WSJ model, versus 1 GB or larger for state-of-the-art models. The memory access pattern is sparse and depends on the hypotheses being searched.

**GMM evaluation :**

The acoustic likelihood of the current feature vector (given each hypothesis for the state ) is approximated using a GMM. GMMs used in speech recognition are typically limited to diagonal covariance matrices to reduce the number of parameters, using more mixture components to make up for the shortfall in modeling accuracy.

**Pruning and Storage :**

The state space of a Viterbi search grows exponentially unless a beam or histogram method is used to prune unlikely hypotheses from being stored and considered. A beam pruning stage tests each hypothesis against a threshold (relative to the highest likelihood encountered on the previous frame). Only hypotheses that pass the test are stored into the active state list for frame $t + 1$.Once all hypotheses for a given frame have been processed, a snapshot of the active state list is saved to the external memory.

# CHAPTER 2

# DNN based Acoustic Modeling

## 2.1 Choosing a Modeling Framework

The acoustic likelihood of the current feature vector $y_t$ (given each hypothesis for the state $x_t$) is approximated using a GMM. The speed of GMM evaluation is limited by the rate at which parameters can be fetched from memory. Technically, a separate GMM would be used for each state in WFST, which are usually millions. The states are usually grouped into clusters of 1000 to 10000, by the similarity of their sounds, and share one GMM probability density. Evaluating these probabilities takes up the bulk of power and bandwidth of the chip. This makes it the focus of architectural enhancements, and we aim to replace it with more optimal framework.

Experiments performed compared three acoustic modeling frameworks in the context of minimizing memory bandwidth : Gaussian Mixture Model, subspace GMM and DNN. The DNNs offered the best tradeoffs between bandwidth and accuracy.

Advantages of DNN :-

- Bandwidth reductions in DNN-based model can be achieved by using smaller networks (for example, 512/256 neurons per layer)
- Scalar quantization of weights and biases can be done, without much loss in accuracy, this can further reduce memory and bandwidth requirements
- Feed-Forward DNNs can be evaluated using a fixed-function SIMD architecture. The architecture can compute likelihood results for multiple frames simultaneously.

## 2.2 Acoustic Modeling with DNN

In a DNN-HMM hybrid system, the DNN is trained to provide posterior probability estimates for the HMM states. Specifically, for an observation $\mathbf{o}_{ut}$ corresponding to time $t$ in utterance $u$, the output $y_{ut}(s)$ of the DNN for the HMM state $s$ is obtained using the softmax activation function:

$$y_{ut}(s) = P(s|\mathbf{o}_{ut}) = \exp\left(a_{ut}(s)\right) / \sum \exp\left(a_{ut}(s)\right)$$

where $a_{ut}(s)$ is the activation at the output layer corresponding to state $s$. The recognizer uses a pseudo log-likelihood of state $s$ given observation $o_{ut}$ ,

$$logp(\mathbf{o}_{ut}|s) = logy_{ut}(s) - logP(s)$$

where $P(s)$ is the prior probability of state $s$ calculated from training data.

### 2.2.1 DNN Training

Since it is a multi-class classification problem, we use cross-entropy as the objective function and optimisation is done through Stochastic Gradient Descent. Cross-entropy between the distribution represented by the reference labels and the predicted distribution $y(s)$, as negative log posterior is defined as :-

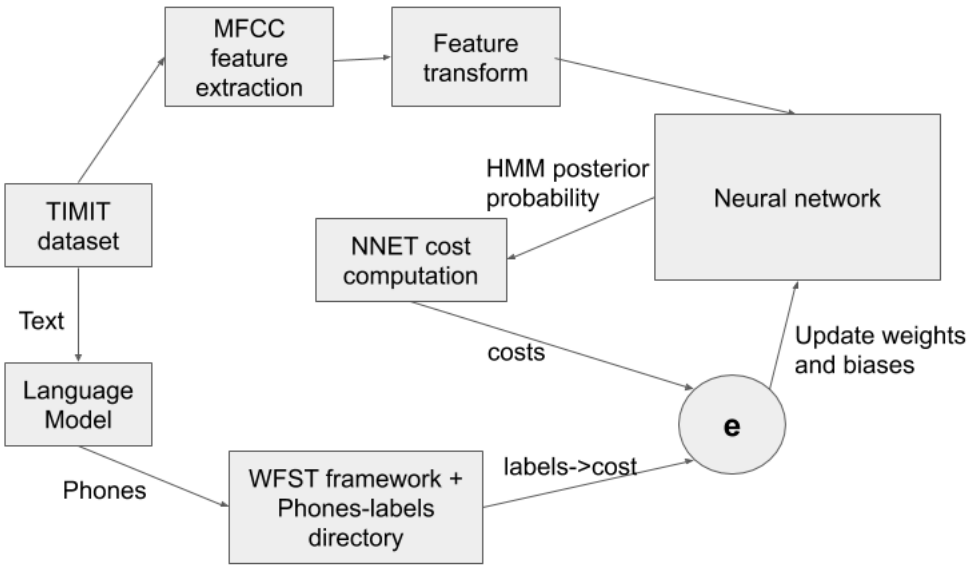
$$F_{CE} = -\sum_{u=1}^{U} \sum_{t=1}^{T_u} logy_{ut}(s_{ut})$$

where $s_{ut}$ is the reference state label at time $t$ for utterance $u$. The required gradient is :-

$$\delta F_{CE} / \delta a_{ut}(s) = -\delta logy_{ut}(s_{ut}) / \delta a_{ut}(s) = y_{ut}(s) - \delta_{s;s_{ut}}$$

where, $\delta_{s;s_{ut}}$ is the Kronecker delta function.

The DNNs are trained on the same features as the GMM-HMM baselines, except that the features are additionally processed. The features are globally normalised to have zero mean and unit variance. Each frame context window size still remains a 40 dimensional feature vector. The input to the neural network is now 11 frames (5 additional frames on each side of the original frame), thus now input to the neural network is a 440 length vector.




The network has 7 layers (i.e., 6 hidden layers), where each hidden layer has 1024 neurons. The input vector to the DNN is of size 440, and the output of the DNN depends on the WFST it is to be trained for, here the size of output of the DNN is 1904.

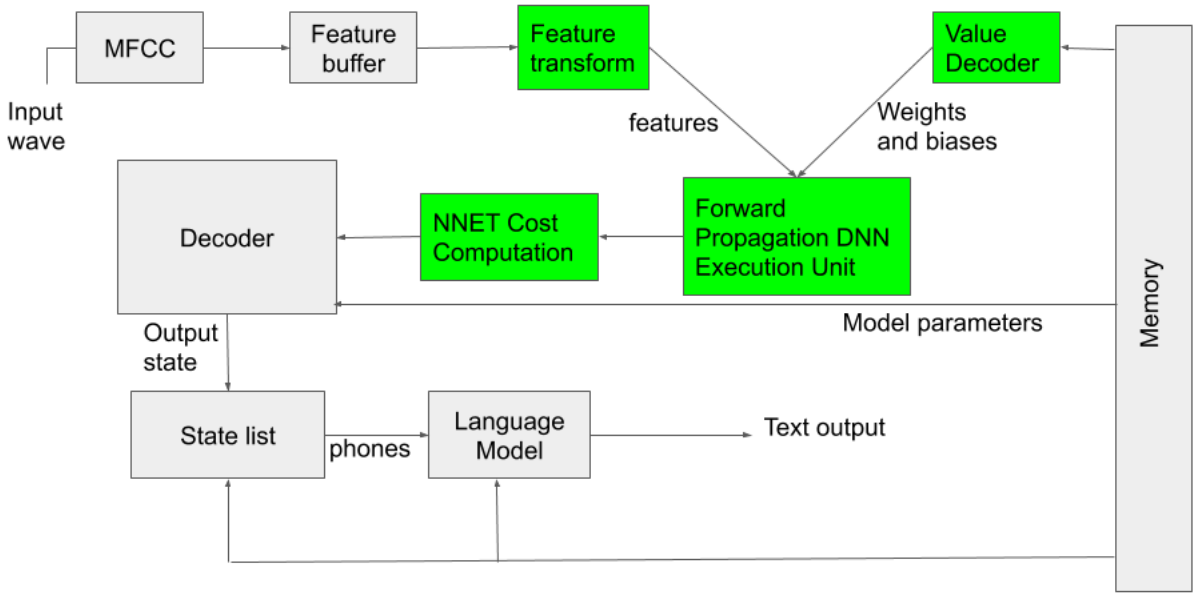
The DNN is initialised with stacked restricted Boltzmann machines, known as RBMs, that are pretrained in a greedy fashion. The Gaussian-Bernoulli RBM is trained with an initial learning rate of 0.01 and the Bernoulli-Bernoulli RBMs with a rate of 0.4. The initial RBM weights are randomly drawn from a Gaussian

N(0, 0.01); the hidden biases of Bernoulli units as well as the visible biases of the Gaussian units are initialized to zero, while the visible biases of the Bernoulli units are initialized as $b_v = log(p/1p)$, where p is the mean output of a Bernoulli unit from previous layer. During pretraining, the momentum m is linearly increased from 0.5 to 0.9 on the initial 50 hours of data, which is accompanied by a rescaling of the learning rate using $1 - m$. Also the L2 regularization is applied to the weights, with a penalty factor of 0.0002.

We use TIMIT Acoustic-Phonetic Continuous Speech Corpus for training. This dataset contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. The TIMIT corpus includes time-aligned orthographic, phonetic and word transcriptions as well as a 16-bit, 16kHz speech waveform file for each utterance.

The utterances and frames are presented to the network in a randomized order while training both of these networks using stochastic gradient descent to minimize the cross-entropy between the labels and network output. The SGD uses mini-batches of 256 frames, and an exponentially decaying schedule that starts with an initial learning rate of 0.008 and halves the rate when the improvement in frame accuracy on a cross-validation set between two successive epochs falls below 0.5%. The optimization terminates when the frame accuracy increases by less than 0.1%. Cross-validation is done on a set of 4000 utterances that are held out from the training data.

### 2.2.2 DNN-HMM execution




The information from the trained neural network (weights and biases) is stored in compressed form in the memory of the ASR engine. Additional decoder units for the comoressed data are to be employed on the chip, these are basic adders, multipliers, shifters or look-up tables for out purpose. There would be SIMD execution units deployed on the chip for computation of the output of this DNN based acoustic modeling.

The MFCC feature extraction is done in the front-end, the output of this usually extracted feature extractor is then stored in a feature buffer. There a set of additional transformations applied on this feature extracted, which includes globally normalising the mean and the variance of the features to 0 and 1 respectively, and adding 5 additional frames on each side of the original frame. These operations are exactly identical to the ones performed on the features while DNN was being trained, similar operations are thus required on the input while we are taking the same DNN to use. These additional transforms helps in easier computations in

the DNN, and tries to avoid biasing in the network based on the input data.
With the output of the DNN, for each hypotheses, the cost is calculated. There is a additional cost calculation unit which takes into consideration whether or not softmax layer was applied while training the DNN, and appropriate log-likelihood is calculatedfor cost purposes. The same cost is used along with the arc weight for calculating the total cost of the hypotheses. Again, the hypotheses below a certain threshold are given up during pruning, and approrpiate hypothesis along with the probability is stored in the state list. The Viterbi Search is applied in the same way as it was for GMM-HMM based ASR architecture, just that with DNN-HMM based architecture, the acoustic cost calculation is different.
We get a list of phenotics as output as the DNN was trained over TIMIT corpus of speech signal and their appropriate phonetics. We would additionally need to convert the phones to worded texts for human understanding. This would require us to store the Language Model additionally.
We have tried to maintain the architecture as close to the architecture of GMM-HMM based for easy comparison, better readability and simplicity.

### 2.2.3 Current Status of Implementation

The training was done on TIMIT dataset, by existing Kaldi Software. The trained NNET, WFST model, phones directory, and language model from the Kaldi source has been extracted into the ASR source. The source code executes each of these block described above and is currently returning an output.
Correct output text :

YOUR WARRIORS MUST GROW WEARY OF RESTING ON THEIR SPEARS

INFADOOS

Current output phones :

t p er cl k cl p r ey cl p ey vcl g aa cl aa cl ch cl t r iy vcl b ay cl p ay cl p ey ey cl p

The output is incorrect. We are still debugging, and expect the issue to be resolved once the additional feature extraction (which is done differently in Kaldi source) is set to the correct values in our source scripts as used while training the DNN.

## 2.3 Comparison : GMM vs DNN

Our primary aim to switch from GMM to DNN was to significantly affect the memory required. The memory requirements for storing GNM parameters vs DNN parameters currently is as,

| **GMM file** | **DNN file** |
|---|---|
| 8.8 MB | 10.7 MB |

Though currently larger than GMM size, DNN is promising as we have currently trained a much bigger network of 6 hidden layers and 1024 neurons each. Without much loss in accuracy the size of the DNN can be significantly reduced. Also, since we have also opted for phonetics as output for the current DNN-HMM architecture, the WFST has reduced significantly. Overall memory requirements for both cases are as follows,

| **Earlier GMM-based ASR** | **New DNN-based ASR** |
|---|---|
| 21.1 MB | 12.2 MB |

# CHAPTER 3

# DNN Compression

## 3.1 Quantization Heuristic

### 3.1.1 Motivation

Neural network, because of its bulky nature, consumes lot of memory. Since, we are constrained, DNN compression becomes crucial.

Each weight or bias was stored as a float(32-bit). We intend to store it in 16-bit, such that loss in accuracy of the model is not significant.

Here, We store the float(32-bit) after certain operations as an integer (16-bit). The algorithm described below details the operations required to convert the float into the appropriate integer and then convert the integer back into approximated float.

### 3.1.2 Pseudo Code

- Encoding float into integer :
  - Identifying the range
  - negatively shift each float value by the mean of maximum and minimum value to surround the range evenly across zero
  - This number will be called shifter and will be stored in memory along with the compressed values
  - Divide all the floats with an integer such that all values lie between (-2,2)
  - This number will be called divider and will be stored in memory along with the compressed values
  - Multiply each float by 10,000

- Round up each float to the nearest integer
- The resultant numbers will lie between (-20000,20000), which is $(-2^{15}, 2^{15})$, thus 16-bit integer.

Example :-
a = [0.1215515, 1.021151, -3.518452, 9.211515]
*shifter = -mean(max(a),min(a)) = -2.8465315*
a = [-2.72498, -1.8254105, -6.3649835, 6.3649835]
*divider = floor(abs(max)) = 6*
a = [-0.4541633, -0.3042350, -1.0608305,1.0608305]
*multiplier = 10000*
a = [-4542, - 3042, -10608, 10608]

- Recovering float back from integer :
  - Shifter, divider and multiplier would be already stored
  - Divide each number by the multiplier (here, 10000)
  - Multiply each number by the divider
  - Positively shift each number by the shifter

Example :-
a = [-4542, - 3042, -10608, 10608]
*Multiplier = 10000*
a = [-0.4542, -0.3042, -1.0608, 1.0608]
*Divider = 6*
a = [-2.7252, -1.8252, -6.3648, 6.3648]
*shifter = 2.8465315*
a = [0.1213315, 1.0213315, -3.5182685, 9.2113315]

### 3.1.3 Code

**Encoding float into integer :**

```
# print len(rows), len(columns)
row_num = 0
for row in rows:
    columns = row.split(' ')
    col_num = 0
    for floatnumber in columns:
        float_num = float(floatnumber)
        floatnumber = (float(floatnumber) + addEachElementBy)/divideEachElementBy
        intnumber = round(floatnumber*pow(10,4))
        # print intnumber
        rec_floatnumber = (1.0*intnumber*divideEachElementBy)/pow(10.0,4) - addEachElementBy
        # print float_num, rec_floatnumber
        if(float_num - rec_floatnumber > 10):
            print float_num, rec_floatnumber, float_num - rec_floatnumber, row_num, col_num
        error = error + float_num - rec_floatnumber
        ofile.write(struct.pack('<h', intnumber))
        col_num = col_num +1
    row_num = row_num + 1
```

**Recovering float back from integer :**

```
for i in range(num_rows-1):

    for k in range(num_cols-1):

        datapoint,j = read_2byte_int(data,j)

        # print datapoint
        datapoint = (1.0*datapoint*multiplyEachElementBy)/pow(10.0,4)-subtractEachElementBy + avg_error
        sum_ = sum_ + datapoint
        num_ = num_ + 1
        ofile.write(str(datapoint)+" ")

    datapoint,j = read_2byte_int(data,j)
    # print datapoint
    datapoint = (1.0*datapoint*multiplyEachElementBy)/pow(10.0,4)-subtractEachElementBy + avg_error
    sum_ = sum_ + datapoint
    num_ = num_ + 1
    ofile.write(str(datapoint))

    ofile.write(";")

for k in range(num_cols-1):

    datapoint,j = read_2byte_int(data,j)

    # print datapoint
    datapoint = (1.0*datapoint*multiplyEachElementBy)/pow(10.0,4)-subtractEachElementBy + avg_error
    sum_ = sum_ + datapoint
    num_ = num_ + 1
    ofile.write(str(datapoint)+" ")

datapoint,j = read_2byte_int(data,j)
# print datapoint
datapoint = (1.0*datapoint*multiplyEachElementBy)/pow(10.0,4)-subtractEachElementBy + avg_error
sum_ = sum_ + datapoint
num_ = num_ + 1
ofile.write(str(datapoint))
ofile.write("\n")
```

### 3.1.4 Compression Results

**DNN output value :**

| Output Value | Compressed output value | Error |
|---|---|---|
| 1.6178E-07 | 1.6085E-07 | 9.3415E-10 |
| 1.4132E-11 | 1.4065E-11 | 6.6656E-14 |
| 1.6668E-10 | 1.6605E-10 | 6.2548E-13 |
| 0.002924 | 0.002932 | -7.8319E-06 |
| 7.71347E-09 | 7.7174E-09 | -4.00035E-12 |
| 2.5043E-09 | 2.48596E-09 | 1.83713E-11 |

**DNN Network file size :**

| Original file | Compressed file |
|---|---|
| 30.6 MB | 15.3 MB |

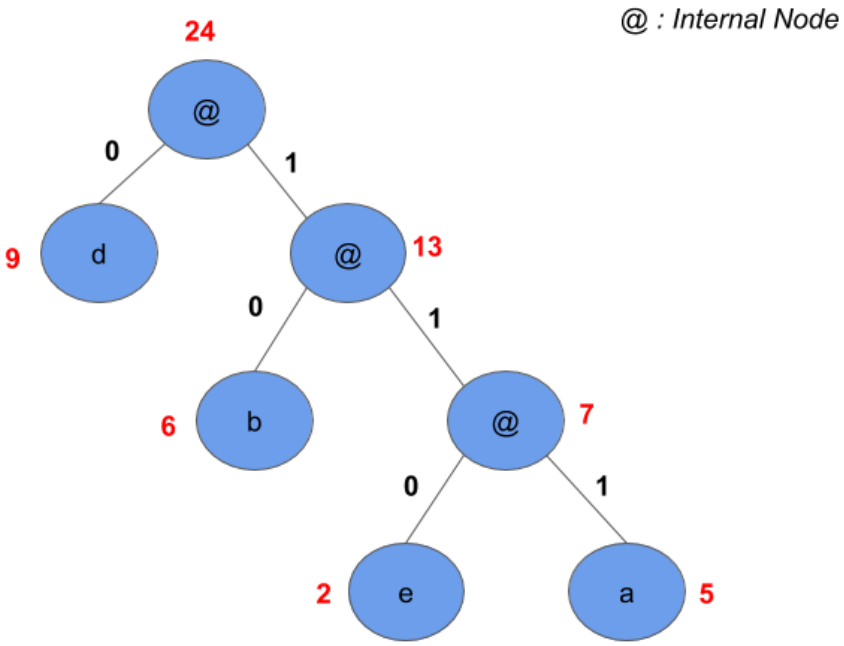
## 3.2 Huffman Encoding

### 3.2.1 Motivation

Common in information theory and computer science, huffman coding is a particular type of optimal prefix-coding technique that is used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol. The algorithm derives this table from the estimated probability or frequency of occurrence for each possible value of the source symbol. Like other entropy-encoding methods, in huffman encoding, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.

### 3.2.2 Example

| char | freq | code |
|---|---|---|
| a | 5 | 111 |
| b | 6 | 10 |
| d | 9 | 0 |
| e | 2 | 110 |




**Decoding :**

0111010
0 111 0 10
d a d b

### 3.2.3 Compression results

| Original file | Huffman Compressed file |
|---|---|
| 15.3 MB | 10.7 MB |

# CHAPTER 4

# Future Work

## 4.1 DNN Model Compression by Re-training

- **Re-training with reduced number of neurons per hidden layer :** We have currently used a big neural network of 6 hidden layers with 1024 neurons each. We can evaluate the effect of performance of the network there would be if we reduce the number of neurons to 512, to 256.
- **Re-training with each value being restricted to 16-bit :** Instead of later pruning each value to a 16-bit, we can also restrict each weight and bias to a 16bit value, so that it trains accordingly and there is a lower loss in accuracy.
- **Parameter Pruning :** Few unimportant weights/biases goes to zero. These values could be tied together and represented with a single character, thus could be represented by much lesser number of bits each. This technique requires re-training, and ensures minimum loss in accuracy.
- **Parameter Sharing :** Few weights/biases will be tied. Thus reducing the number of values required to be stored. This technique requires re-training, and ensures minimum loss in accuracy.

## 4.2 Hardware Optimization

- **SIMD execution :** DNN forward propagation can be executed layer-by-layer in parallel for multiple frames using dedicated SIMD units, which can significantly improve the performance of the system.
- **Specific functions :** DNN forward propagation majorly requires multiplication and addition operations. The sigmoid function at each layer can be approximated to ReLU. We can thus use dedicated functional units like, adder and multiplier can be used to optimise.
- **Compression and Caching techniques to memory and bus requirements**

## 4.3 Language Modeling

- **Optimal conversion model from Phonetics to words :** While mapping phonetics to words there are various approximations required. We might require a proper RNN-LSTM model which can be trained while training the DNN for acoustic modeling. This could be stored in similar compressed way as the acoustic modeling DNN. The memory requirements and accuracy of this and directly using a WFST which maps to word list can be compared.
- **Training phonetics-to-word conversion for the corpus :** The phonetic-word RNN training could first be done within a corpus, to take into account the variations with different speakers and dialects.
- **Build a generalised language model over multiple corpora :** The training could then be expanded to incorporate variations across different corpuses, so that we can possibly develop a generalised langauge model, and can store this information on the ASR system which can be used for speech recognition over various dialects and corpuses.

# References


1. Michael Price, James Glass, Anantha P. Chandrakasan. A Low-Power Speech Recognizer and Voice Activity Detector Using Deep Neural Networks *IEEE Journal of Solid-State Circuits, Volume: 53, Issue: 1, Jan. 2018.*

2. Michael Price, James Glass, Anantha P. Chandrakasan. A 6 mW, 5,000-Word Real-Time Speech Recognizer Using WFST Models *IEEE Journal of Solid-State Circuits, Volume: 50, Issue: 1, Jan. 2015.*

3. Michael Price, James Glass, Anantha P. Chandrakasan. Memory-efficient modeling and search techniques for hardware ASR decoders *Proc. INTERSPEECH, 2016, pp. 18931897.*

4. Song Han, Huizi Mao, William J. Dallay Deep Compression : Compressing Deep Neural Networks with Pruning, Quantization and Huffman Coding *Published as a conference paper at ICLR 2016*

5. Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury Deep Neural Networks for Acoustic Modeling in Speech Recognition

6. Karel Vesely, Arnab Ghoshal, Lukas Burget, Daniel Povey Sequence-discriminative training of deep neural networks *Published as a conference paper at INTERSPEECH 2013*

7. https://kaldi-asr.org/doc/dnn.html