A Low-Resource ASR Back-End Based on Custom Arithmetic

Xiao Li, Jonathan Malkin, Jeff Bilmes

Abstract—Most contemporary ASR systems running on desktops use continuous-density HMMs (CHMM) with floating-point representations. It is important to reduce their memory and power requirements so that they can be more affordable for portable devices. In this paper, we propose a novel speech recognition back-end based on custom arithmetic, where all floating-point variables are represented by integer indices and all arithmetic operations are replaced by a sequence of table lookups. One critical issue associated with table lookups is what we call an accumulative variable whose dynamic range is either large or unpredictable. Such a variable would introduce much distortion if quantized to low precision, so that the table lookup would incur a great loss of information. We therefore explore different quantization structures dealing with this problem in likelihood evaluation, and present a normalization method for the Viterbi search to make the range of the forward probabilities predictable. Furthermore, we investigate several optimization algorithms on system-wide bit-width allocation. The best algorithm uses 80 Kbytes of tables to achieve all back-end operations with only a slight degradation in system performance. As a side effect, the offline storage for parameters is reduced by 80%, and the memory requirement for online computation is reduced by nearly 70%.

Keywords—speech recognition, low resource, quantization, normalization, optimization

I. INTRODUCTION

The burgeoning development of mobile devices has brought about a great need for a more convenient and friendly user interface. Automatic speech recognition (ASR) will undoubtedly become a dominant method, for it greatly eases the use of hand-held devices and makes the human-machine interaction swift. However, unlike desktop applications with ample memory and incessant power supply, portable devices suffer from limited computational and memory resources and strict power consumption constraints. Most state-of-the-art ASR systems running on desktops use continuous-density HMMs (CHMM) with floating-point arithmetic representations. They are computationally expensive and energy-hungry, making them unaffordable for an embedded system powered by batteries. Therefore, the development of a low-power, low-memory ASR system becomes crucial to the prevalence of speech technologies on mobile devices.

In the literature, there are many techniques to speed up computation at the software level, among which quantization with table lookups has been extensively used. In a discrete HMM (DHMM), for example, state likelihoods of a feature vector can be obtained efficiently via vector quantization (VQ) and its corresponding table lookup. It offers fast computation, though it has significantly worse performance compared to a CHMM. As a compromise, a semi-continuous HMM (SCHMM) welds VQ distortion into the continuous model to achieve both fast computation and high recognition rate [1], [2]. As a further improvement, a discrete mixture HMM (DMHMM) assumes discrete distributions at the scalar or sub-vector level of a mixture model, and applies scalar quantization or sub-vector quantization to the feature vectors [3], [4]. Even in a CHMM, the computational load can be greatly reduced by restricting the precise likelihood computation to the most relevant Gaussians using VQ [5], [6]. In addition, quantization techniques also contribute to a compact representation of model parameters, which not only saves memory but also reduces computational cost [7], [8].

On the other end of the spectrum, architecture-level research has been focusing on substituting the floating point processor with a less costly processor such as a fixed-point DSP. The fixed-point implementation performs arithmetic functions using scaled integers, significantly reducing the bandwidth as well as the complexity of the arithmetic operations. Some MFCC and HMM-based voice command systems built on DSPs have achieved great savings in memory and power consumption [9], [10], [11]. However, although fixed-point arithmetic is faster than its floating-point counterpart, it has potential drawbacks. First, there is substantial loss of information in arithmetic functions like multiplication and logarithm. Secondly, overflow issues have to be dealt with by clipping or shifting numerical values. Finally, fixed-point arithmetic is still not very efficient in utilizing the bandwidth, since it is not optimized for a specific application and a significant portion of fixed-point values might be seldom used.

In this paper, we propose a novel ASR back-end based on custom arithmetic purely via table lookups. More specifically, we represent each floating-point variable by an integer index and replace each arithmetic operation by a table lookup. The goal is appealing, considering the high speed and low power consumption of a table lookup compared to a complicated arithmetic function. On the other hand, the objective also looks daunting, since the back-end might have dozens of variables and operations which could render a prohibitive storage for tables. However, by appropriate quantization techniques and bit-width optimization algorithms, the variables can be quantized to very low precision without deterioration of performance. The tables, therefore, are affordable for most modern embedded systems.

We choose to apply custom arithmetic on the back-end but not on the front-end for the following reasons. The
back-end accounts for 90% of the computational load of the entire system. However, it has fewer free variables than the front-end, and most of its operations are either repetitive, as in the likelihood evaluation, or iterative, as in the Viterbi decoding. Therefore, we expect table-lookup techniques to be especially beneficial to the back-end. As a contrast, the front-end has a relatively low cost in computation but a large variety of variables and operations which would quickly make the lookup-tables prohibitive. In addition, the fixed-point arithmetic for the front-end feature extraction has been well studied and can be implemented by DSPs very efficiently [12].

The paper is organized as follows. We start with the general idea of our proposed custom arithmetic in section II. Section III discusses quantization techniques associated with likelihood evaluation, the bottleneck of the entire recognition engine. Section IV presents a normalized Viterbi search algorithm which makes the table lookup applicable to the whole back-end. In section V, we investigate several heuristics on bit-width optimization. Our system configuration is described in section VI, and experiments and results are given in section VII, followed by concluding remarks in the last section.

II. CUSTOM ARITHMETIC VIA TABLE LOOKUPS

In this section we present a system driven by custom arithmetic, where all the floating calculations are pre-computed and transparent to the online application. We expect that it would offer very fast computation from an architectural perspective. Also we address several potential issues associated with lookup-table design.

A. General structure

A high-level programming language allows complex expressions involving multiple operands. We split all such complex expressions into sequences of two-operand operations by introducing intermediate variables. Assume we have a total of $L$ variables $\{V_i\}_{i=1}^{L}$ in the system including parameters computed offline, and assume we have $K$ two-operand functions based on these variables,

$$V_h = F_k(V_i, V_j),$$

where $i,j,h \in \{1..L\}$, $k = 1..K$, and $F_k(\cdot)$ can be an arbitrary arithmetic operation or a sequence of operations. For example, $V_h = (V_i + V_j)^2$ is such a valid two-operand function. In a more general case, the variable can be a vector and the function can have more two operands.

First, we create a codebook for each free variable $V_l$ in the back-end. The codewords are the quantized values of the corresponding variable. The indices of the codewords in a codebook are sequential integers. For an arbitrary floating-point representation $V_l = x$, the closest codeword in the codebook is denoted as $Q_{V_l}(x)$, and its associated index is denoted as $I_{V_l}(x)$. An $n$-bit indexed codebook can accommodate $2^n$ codewords.

Next, we create a table $T_{F_k}$ for each function $F_k(\cdot)$. The address of an entry is determined by the integer indices of the two input variables. The entries of the table are the integer indices of the output variable. If the two inputs and the output have bit-widths of $n_1$, $n_2$ and $n_0$ respectively, the table will take $n_0 \cdot 2^{n_1+n_2}$ bits in storage.

Our goal is to replace all two-operand arithmetic operations as defined in (1) with table lookups. In other words, if $V_i = x$ and $V_j = y$ are provided and we intend to compute $V_h$ according to $F_k(\cdot)$, we can obtain the approximated output via three steps:

1. Retrieve the indices of $V_i$ and $V_j$ in their respective codebooks;
2. Get the output index via the table lookup $T_{F_k}(\cdot)$;
3. Locate the codeword in $V_h$’s codebook by the output index.

The table $T_{F_k}$ should be designed in such a way that, if $z = F_k(x, y)$, we have

$$I_{V_h}(z) = T_{F_k}(I_{V_i}(x), I_{V_j}(y))$$

In the system with a stream of floating-point operations, we achieve the approximated computation by a sequence of table lookups. Ignoring steps 1 and 3, we use the output index directly as the input of the next table lookup, so that all data flow and storage are represented in integer forms, and all arithmetic operations are substituted by table accesses.

B. Benefits from an architectural point of view

Many mobile devices seek software support for floating-point arithmetic operations. Intel XScale processor, for example, was designed for low power mobile devices without a hardware floating-point unit [13]. It takes 35 cycles for a multiplication, 57 cycles for a division and 215 cycles for a logarithm, all in single precision. Although it implements the math library routines with relatively low latencies, it is still costly in computation and power consumption for an application rich in mathematical operations.

In fact, if the variables in the system tend to have low local entropies, it would be efficient to record the results of the calculations for future use. [14] uses cache-like memory tables to store the outputs of particular instruction types. It performs the table lookup in parallel with conventional computation which is halted if the lookup succeeds. It argues that the cycle time of a memo-table lookup is comparable to a cache lookup, which is extremely fast.

Our custom arithmetic calculates the table entries completely offline and stores the tables in non-volatile memories. The physical realization of the lookup-tables is beyond the scope of this work, although it is crucial to the cost and power consumption of the embedded system. No matter how the tables are planted, we expect that an access to the non-volatile memory is generally faster and less power-consuming than a floating-point arithmetic operation. The savings would be more significant with an extended ISA supporting the lookup operations.

C. Issues with table lookup

In spite of the attractions, such a system easily becomes unrealistic if the storage of the tables gets too large. The
tradeoff between the system performance and storage requirement can be controlled in two ways: determine the number of tables and determine their sizes.

The number of the tables depends on how we define variables and functions. Most of the table designs are straightforward, but a few of them are less intuitive. The lookup tables for \( y = \sum_{i=1}^{N} x_i \), for example, can be designed in multiple ways. One way is that we define two variables \( X \) and \( Y \), and only one function \( Y = Y + X \). \( Y \) is initialized as zero and the function is iteratively performed \( N \) times. This approach leads to only one table; but the variable \( Y \) might have a large dynamic range, possibly too large for a single codebook. We call a variable “accumulative variable” if it is involved in such iterative operations. The accumulation process could have a fixed dimension, as in the example above, in which case we call it a “bounded accumulation.” Alternately, the accumulation process could be arbitrarily long, a situation we call an “unbounded accumulation.” Solutions to accumulative variables are crucial to a structure of the operation flow, as in section III, or make modification to the decoding algorithm, as in section IV.

The bit-widths of the variables directly influence the total storage of the tables. After we have defined all the variables \( \{V_i\}_{i=1}^{K} \) and functions \( \{F_k(\cdot)\}_{k=1}^{K} \), we would like to see how much each \( V_i \) can be quantized without degradation in performance. Also, we seek heuristics to find a system-wide optimization method to allocate bits among all the variables in the back-end, which will be discussed in section V.

III. CUSTOM ARITHMETIC FOR LIKELIHOOD EVALUATION

Likelihood evaluation is usually the bottleneck of an ASR system, since it is responsible for 30% to 70% of the total computational load [6]. The observation probabilities are evaluated on every Gaussian component of the system following the same mathematical formula.

A. Formulation of likelihood evaluation

The continuous density \( b_j(O_t) \) of the observation vector \( O_t \), given a certain state \( q_t = j \), is computed as

\[
b_j(O_t) = \sum_{i \in M_j} w_i \mathcal{N}(O_t; \mu_i, \Sigma_i) \tag{3}
\]

where \( O_t = (x_{t1}, x_{t2}, ..., x_{tD}) \) is the observation vector at time \( t \) with \( D \) the dimensionality of the features; \( M_j \) is a subset of Gaussians belonging to state \( j \); and \( w_i \) is the responsibility of the \( i^{th} \) Gaussian component. All likelihoods \( b_j(t), t = 1..T, j = 1..N \) are computed and cached in memory before Viterbi decoding starts, where \( T \) is the total number of frames and \( N \) is the total number of states.

What we actually compute and store are the log probabilities. In this paper we use the symbol with a bar to represent its logarithm. For example, \( \bar{x} = \log x \). Also, we let \( \oplus \) denote log addition, where the algorithm is given by

\[
x \oplus y = \log(e^x + e^y) = x + \log(1 + e^{-(x-y)}) \tag{4}
\]

if \( x > y \). In this way, we modify (3) as

\[
\bar{b}_j(O_t) = \bigoplus_{i \in M_j} (\bar{w}_i + c_i - \frac{1}{2} \sum_{k=1}^{D} \frac{(x_{tk} - \mu_{ik})^2}{\sigma_{ik}^2}) \tag{5}
\]

where \( \bar{w}_i = \log w_i \) and \( c_i \) is a constant independent of the observation, both of which can be computed offline.

As mentioned earlier, many operations in the likelihood evaluation are repetitive. For example, to evaluate the observation probabilities for an utterance with \( T \) frames in a system with \( M \) Gaussian components, operation \( (x_{tk} - \mu_{ik})^2/\sigma_{ik}^2 \) will be performed \( T \times M \times D \) times, which probably implies millions of floating-point multiplications. Therefore, it would be a great saving if we replace all the Gaussian evaluations with table lookups. Before arriving at variable and function definitions, we will first discuss the solutions to the bounded accumulation in the likelihood evaluation.

B. Solutions to Bounded accumulation

A crucial problem inherent in the likelihood evaluation is that there are two accumulative variables suggested by (5).

One is \( e_i(t) \triangleq \sum_{k=1}^{D} d_{ik}(t) \), where \( d_{ik}(t) \triangleq (x_{tk} - \mu_{ik})^2/\sigma_{ik}^2 \), and the other is \( \bar{b}_j(O_t) \) associated with the log addition. As mentioned previously, a bounded accumulation is one for which there is a fixed number of elements to be added. If those elements have a predictable dynamic range of values, then the total accumulation will also have a larger, but still predictable, dynamic range. This situation leads to two possible algorithms for performing quantized accumulation: a straightforward linear accumulation and a binary tree. The algorithms are presented here and results discussed in section VII.

B.1 Linear accumulation

Linear accumulation is the typical accumulative algorithm, where the next value of a variable equals the current value plus an additional value. In the case of \( e_i(t) \), the variable needs to be initialized to some starting value that must exist in the codebook, typically zero. \( e_i(t) = e_i(t) + d_{ik}(t) \) is then consecutively performed for \( k = 1..D \), and a single table must account for enough values to adequately represent the accumulation’s total dynamic range.

Such an algorithm works especially well for those operations which do not yield a dramatic change in the dynamic range. \( b_j(O_t) = \bigoplus(\cdot) \), for example, can be quantized using this algorithm with fairly good performance, since log addition has a relatively small impact on the range of the value.

B.2 Tree accumulation

An alternative to linear accumulation is to use a binary tree for the operation with a different codebook for each level. This method requires the same number of operations as the linear accumulation, but the operations take place in a different order. The potential advantage is that each
level of the tree can expect to see a smaller dynamic range of values, especially at lower levels. As a result, the total size of the codebooks may be no larger than with the linear accumulation case.

This paper tried two tree accumulation patterns for calculating $e_i(t)$. The patterns differed only at the first level of the tree. The first pattern adds adjacent elements of the vector $\{d_{ik}(t)\}_{k=1}^D$. Based on the empirical observation that the dynamic range of $d_{ik}(t)$ was fairly consistent across all the static features, and that the same held true for the deltas, the second pattern adds the $d_{ik}(t)$ of each feature and its corresponding delta. The result is that the dynamic range of the first level of the tree becomes more consistent, making quantization more successful.

Using a binary tree to add $n$ values requires $\log_2[n]$ tree levels. Since speech recognizers often use features not in a power of 2, this presents a potential problem. Possible solutions are either to treat missing values as zero, or to allow some values to pass over some levels of the tree as shown in figure 1. In the interest of avoiding additional computation, we chose the latter.

The advantage of tree-structure accumulation is more conspicuous for iterative operations with high dimension. For our experiments in section VII where $D = 26$, the linear accumulation works better than the tree-structure one. We therefore choose to use linear accumulation in our custom arithmetic design.

### C. Defining variables and tables

Based on the analysis above, we define 11 variables to be quantized in the likelihood computation, which are categorized in table 1. $x$ is the output variable of the front-end, $m, v, c, w$ are the acoustic model parameters computed offline, $s, d, e, p, q$ are 5 intermediate variables introduced to the system and $b$ is the log state likelihood to be fed into Viterbi decoding.

There are 6 functions $F_k(\cdot), k = 1..6$ and hence 6 potential lookup-tables $T_{F_k}$ associated with these variables:

- $F_1(\cdot): s = (x - m)^2$
- $F_2(\cdot): d = s/v$
- $F_3(\cdot): e = e + d$
- $F_4(\cdot): p = e - e/2$
- $F_5(\cdot): q = w + p$
- $F_6(\cdot): b = b \oplus q$

$F_1(\cdot)$ and $F_2(\cdot)$ involve floating-point multiplication and division respectively, and these operations would be performed millions of times for an ordinary isolated word recognition task. $F_6(\cdot)$ would be executed thousands of times with even more expensive computation of logarithm. Therefore, we expect the simple lookup operation will dramatically save cycles as well as power on these functions.

### IV. Custom Arithmetic for Viterbi Search

Viterbi search is another heavy load for a speech recognition engine because the forward probabilities have to be calculated for each state at each frame. We hope the custom arithmetic is applicable to Viterbi search, too, not only to save computation, but also for the consistency of the whole system back-end.

#### A. Formulation of Viterbi search

Viterbi search is carried out by computing the forward probabilities $\alpha_j(t) \triangleq P(O_{1:t}, q_t = j)$ iteratively,

$$\tilde{\alpha}_j(t) = \bigoplus_i (\tilde{\alpha}_i(t-1) + \bar{a}_{ij}) + b_j(O_t) \quad (6)$$

with the final score evaluated as $\log P(O_{1:T}) = \bigoplus_j \tilde{\alpha}_j(T)$. The properties of log addition (4) makes the following approximation valid:

$$\alpha_j(t) = \max_i (\tilde{\alpha}_i(t-1) + \bar{a}_{ij}) + \bar{b}_j(O_t) \quad (7)$$

with the final score $\log P(O_{1:T}) = \max_j \tilde{\alpha}_j(T)$. The error introduced by (7) is very small when the operand with

<table>
<thead>
<tr>
<th>symbol</th>
<th>correspondent in (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x_{ik}$</td>
</tr>
<tr>
<td>$m$</td>
<td>$\mu_{ik}$</td>
</tr>
<tr>
<td>$v$</td>
<td>$\sigma_{ik}^2$</td>
</tr>
<tr>
<td>$c$</td>
<td>$c_i$</td>
</tr>
<tr>
<td>$w$</td>
<td>$\bar{w}_i$</td>
</tr>
<tr>
<td>$s$</td>
<td>$(x_{ik} - \mu_{ik})^2$</td>
</tr>
<tr>
<td>$d$</td>
<td>$d_{ik}(t)$</td>
</tr>
<tr>
<td>$e$</td>
<td>$e_i(t)$</td>
</tr>
<tr>
<td>$p$</td>
<td>$c_i - \frac{1}{2}e_i(t)$</td>
</tr>
<tr>
<td>$q$</td>
<td>$\bar{w}_i - c_i - \frac{1}{2}e_i(t)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$b_j(O_t)$</td>
</tr>
</tbody>
</table>

TABLE I

VARIABLES DEFINED IN LIKELIHOOD EVALUATION
the maximum value is salient compared to other operands. This is often true in the Viterbi decoding, where the score along one path is significantly higher than scores along other paths. We use (7) as an alternate search algorithm for our system, where the computation is greatly saved by eliminating the expensive log additions.

B. Normalization of Viterbi Decoding

A severe problem associated with Viterbi decoding is that the forward probabilities do not have a bounded dynamic range. This is what we defined as an unbounded accumulation.

B.1 Unbounded accumulation

A further derivation of (7) gives that

$$\min_{ij} \bar{a}_{ij} + \max_{j} \bar{b}_j(O_t) \leq \max_{j} \bar{a}_{ij}(t) - \max_{i} \bar{a}_i(t - 1) \leq \max_{ij} \bar{a}_{ij} + \max_{j} \bar{b}_j(O_t)$$

We let

$$r_t \triangleq \min_{ij} \bar{a}_{ij} + \frac{1}{T} \sum_{j} \max_{j} \bar{b}_j(O_t) < 0$$

$$r_h \triangleq \max_{ij} \bar{a}_{ij} + \frac{1}{T} \sum_{j} \max_{j} \bar{b}_j(O_t) < 0$$

Since $\max_{j} \bar{a}_j(1) = 0$, the maximum log forward probability is bounded by

$$r_t t \leq \max_{j} \bar{a}_j(t) \leq r_h t,$$

and the final log likelihood is bounded by

$$r_t T \leq \log p(O_{1:T}) \leq r_h T$$

As suggested by (10) and (11), the log forward probabilities keep drifting away from zero with time, and the possible values of the final score rely heavily on the length of the utterance. Consequently, we can hardly predict the range of the forward probabilities in the stage of codebook design, which causes severe problems for the variable quantization. First, since an utterance in the real application can be arbitrarily long, the forward probabilities might go beyond the dynamic range covered by the codewords, resulting in a huge distortion in quantization. Second, an increasing dynamic range potentially requires more codewords in order to suppress the overall distortion, leading to an exponential growth in table size.

An intuitive solution to the problem is to train the codebook with long utterances. This is by no means an effective solution. The forward probabilities produced by the application are still not guaranteed to be covered by the codebook. And for short utterances which are most likely to occur, many of the codewords would be seldom or never accessed and the resources would be squandered. Therefore, it is necessary to have a normalized version of the forward probability, where the inference is still valid but the dynamic range is restricted regardless of the utterance length.

B.2 An early attempt

$p(q_t = j|O_{1:t})$ has served as a normalized version of the forward probability in [15] [16]. We let $\alpha_j'(t) \triangleq p(q_t = j|O_{1:t}) = \alpha_j(t)/p(O_{1:t})$, with the recursion being

$$\alpha_j'(t) = \frac{p(O_{1:t-1})}{p(O_{1:t})} \left[ \sum_{i} \alpha_i'(t-1)a_{ij} \right] b_j(O_t).$$

Implemented in with logarithm, $\tilde{\alpha}_j'(t) = \tilde{\alpha}_j(t) - \log p(O_{1:t})$, and the recursion becomes

$$\tilde{\alpha}_j'(t) = \log \frac{p(O_{1:t-1})}{p(O_{1:t})} + \left[ \sum_{i} \tilde{\alpha}_i'(t-1) + \tilde{a}_{ij} \right] + \tilde{b}_j(O_t).$$

In each step of the recursion, an offset is added to $\tilde{\alpha}_j(t)'$, keeping the normalized forward probabilities from becoming too negative. Since

$$1 = \sum_{j} \alpha_j'(t) = \frac{p(O_{1:t-1})}{p(O_{1:t})} \sum_{j} \left[ \sum_{i} \alpha_i'(t-1)a_{ij} \right] b_j(O_t),$$

the offset can be computed as

$$\log \frac{p(O_{1:t-1})}{p(O_{1:t})} = - \log \left( \sum_{j} \sum_{i} \alpha_i'(t-1)a_{ij} \right) b_j(O_t).$$

And the final likelihood score can be obtained by summing over all the log values of the scaling factors

$$\log p(O_{1:T}) = \sum_{t} \log \frac{p(O_{1:t-1})}{p(O_{1:t})}.$$  

The normalized probability $\tilde{\alpha}_j'(t)$ obviously has a more compact dynamic range. However, it introduces complicated operations in obtaining the offset and the final likelihood score. Worst of all, the problem of an accumulating dynamic range still exists in (16). Actually this problem cannot be eradicated if we keep $\log p(O_{1:T})$ as the scoring criterion, because the likelihood itself has a dynamic range depending on $T$. As $T$ increases, $\log p(O_{1:T})$ for each model will grow progressively more negative.

B.3 Proposed normalization method

Based on the arguments (10) and (11), we propose a normalized log likelihood score $\log p(O_{1:T}) - rT$, where $r$ is a constant which we will explain later. This is not only a valid criterion with a recursive inference algorithm available, but also one which controls the dynamic range.

This is a valid criterion simply because the offset $-rT$ stays the same for all word candidates and hence has no impact on the final decision. That gives us absolute freedom in choosing the scalar $r$.

Furthermore, under this criterion, the dynamic programming of the inference still applies with the same computational complexity. We let $\eta_j(t) \triangleq \alpha_j(t)e^{-rt}$, the recursion becomes

$$\eta_j(t) = e^{-rt} \left[ \sum_{i} \eta_i(t-1)a_{ij} \right] b_j(O_t).$$
Implemented with logarithm, where \( \bar{\eta}_j(t) = \tilde{\alpha}_j(t) - rt \),

\[
\bar{\eta}_j(t) = \bigoplus_i [\bar{\eta}_i(t-1) + \tilde{a}_{ij}] + \bar{b}_j(O_t) - r \tag{18}
\]

with the approximated version

\[
\bar{\eta}_j(t) = \max_i [\bar{\eta}_i(t-1) + \tilde{a}_{ij}] + \bar{b}_j(O_t) - r. \tag{19}
\]

The final score is evaluated as \( \bigoplus_j \bar{\eta}_j(T) \) or \( \max_j \bar{\eta}_j(T) \).

Finally, a derivation from (10) and (11) gives

\[
\begin{align*}
(r_l - r)t & \leq \max_j \bar{\eta}_j(t) \leq (r_h - r)t \tag{20} \\
(r_l - r)T & \leq \log p(O_{1:T}) \leq (r_h - r)T. \tag{21}
\end{align*}
\]

where the dynamic range of \( \bar{\eta}_j(t) \) is controlled by the scalar \( r \). From (10) and (11). To choose \( r \), we compute the scores of all utterances in the training set evaluated on their own generative word models. And we let

\[
r = \frac{1}{T} E \left[ \log p(O_{1:T}|\text{correct model}) \right], \tag{22}
\]

in an attempt to normalize the highest likelihood score for an utterance to zero. It might be true that some normalized scores evaluated on the incorrect word models are still going negative as \( T \) increases. But that would be at a much lower rate and its effect on the quantization is greatly mitigated, as we can see in section VII.

C. Definitions of variables and functions

Based on the normalized Viterbi search, we introduce symbols to represent the variables as shown in table II. \( b \) is the log state likelihood computed in the likelihood evaluation; \( a \) is the log transition probability computed offline; \( \eta_1, \eta_2 \) are two intermediate variables introduced into the Viterbi decoding, where we let them share the same codebook and simply denote them by \( \eta \).

<table>
<thead>
<tr>
<th>( b )</th>
<th>( b_j(O_t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( \tilde{a}_{ij} )</td>
</tr>
<tr>
<td>( \eta_1 )</td>
<td>( \max_i [\tilde{\alpha}<em>i(t-1) + \tilde{a}</em>{ij}] + \bar{b}_j(O_t) - r )</td>
</tr>
<tr>
<td>( \eta_2 )</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE II**

Variables defined in normalized Viterbi decoding

We define the functions based on (19) as follows

\[
F_1(\cdot) : \eta = \eta + a \quad F_2(\cdot) : \eta = \eta + b - r
\]

Noting that there are comparison operations inherent in “max” of (19), They can be achieved via the comparisons of the integer indices, meaning no extra tables are required.

V. Optimization of Bit-width Allocation

After we tackle the problems with single-variable quantization individually, we will come up with a system-wide optimization algorithm to allocate resources among all variables. Our goal is to find the bit-width allocation scheme which minimizes the cost of resources while maintaining the baseline performance. Suppose we have \( L \) variables \( \{V_i\}_{i=1}^L \) in the system. We define a few notations as follows

- \( fp \) – single-precision floating-point representation;
- \( bw_i \) – the bit-width of \( V_i \). \( bw_i \) can take on any integer value below floating-point precision. We also allow \( bw_i = fp \) meaning \( V_i \) is not quantized.
- \( bw = (bw_1, bw_2, ..., bw_L) \) – a bit-width allocation scheme;
- \( \Delta bw \) – an increment of 1 bit along the direction of \( bw_i \), where \( bw + \Delta bw = (bw_1, ..., bw_i + 1, ..., bw_L) \);
- \( \text{wer}(bw) \) – the word error rate (WER) evaluated at \( bw \);
- \( \text{cost}(bw) \) – total cost of resources evaluated at \( bw \). Note that the cost function here can be defined on memory footprint, power consumption, computation speed and etc, depending on our specific goals. In this paper, we use the total storage of the tables as the cost.

In addition, we define the gradient \( \delta_i \) as the ratio of the decrease in WER to the increase in cost evaluated in the direction of \( bw_i \).

\[
\delta_i(bw) = \frac{\text{wer}(bw) - \text{wer}(bw + \Delta bw_i)}{\text{cost}(bw + \Delta bw_i) - \text{cost}(bw)} \tag{23}
\]

(23) reflects the rate of improvement along the \( bw_i \)'s direction. We can extend this definition to the gradient along multiple directions. For example,

\[
\delta_j(bw) = \frac{\text{wer}(bw) - \text{wer}(bw + \Delta bw_i + \Delta bw_j)}{\text{cost}(bw + \Delta bw_i + \Delta bw_j) - \text{cost}(bw)} \tag{24}
\]

is the gradient along the joint direction of \( bw_i \) and \( bw_j \).

Assuming the baseline system gives a WER of \( \text{BWER} \), our goal can be interpreted as

\[
bw^* = \arg\min_{bw: \text{wer}(bw) \leq \text{BWER}} \text{cost}(bw) \tag{25}
\]

A direct way to find \( bw^* \) is via an exhaustive search, which evaluates the \( \text{wer}(bw) \) and \( \text{cost}(bw) \) at every possible \( bw \) and selects the one that meets our requirements. For example, if all bit-widths from 1 to 32 are allowed for each variable, there would be a total of \( 32^L \) operating points in the \( L \)-dimensional space, which makes the exhaustive search intractable. However, by making certain constraints, the search space can be greatly narrowed. We present several heuristics in the following text, the essential idea being that we start at a point with low cost and reasonable performance, then iteratively increase the bit-width of the variable or group of variables that gives the best improvement until we obtain the satisfactory result. A similar methodology has been used in [17] for floating-point bit-width optimization.
A. Single-variable quantization

Before doing the bit-width optimization, we would like to see how much a single variable can be quantized while maintaining good performance. This will give us an idea of how robust the system is to different types and different levels of quantization noise, from which we can have a good approximation of where to initialize the greedy search and where to end it.

We quantize each variable $V_i$ with an increasing bit-width while keeping all other variables in floating-point precision. Therefore, we gain the plots of “WER vs. bit-width” for all $L$ variables that are shown in section VII.

We assign lower and upper bounds to the bit-widths of the variables according to these plots. We let $m_i$ denote the minimum bit-width to which $V_i$ can be quantized to achieve baseline recognition rate in the single-variable quantization. Also, we set an upper bound $M_i$ for $bw_i$ based on inspection. The optimum point is expected to reside in the hypercube constrained by $m_i \leq bw_i \leq M_i$, $i = 1..L$. The search space, therefore, is greatly narrowed. To start the greedy search, we initialize $bw_{init} = (bw_1 = m_1, bw_2 = m_2, ..., bw_L = m_L)$.

B. Single-dimensional increment based on static gradients

For this algorithm, we allow the increment to take place only in one dimension at a time, based on static gradient information. Here we define the static gradients as $\delta_t^i(bw_i) = \frac{\Delta}{\delta(bw_i)}_{bw_i \rightarrow \text{fp},...,bw_i,\text{fp}}$, which can be obtained offline from the results of the single-variable quantization. The algorithm can be described as follows:

1. Initialize $bw = bw_{init}$. If $wer(bw) \leq \text{BWER}$, output $bw$;
2. Compute the static gradients $\delta_t^i(bw_i)$, $i = 1..L$ at point $bw$, based on the WERs of the single-variable quantization. Choose the direction $k = \text{argmax} \delta_t^i(bw_i)$, and increase $bw_k$ by one if it does not exceed the upper bound $M_k$;
3. Run the test to get the new $wer(bw^*)$. Output $bw^* = bw$ if $wer(bw^*) \leq \text{BWER}$ or there is no improvement in WER observed; otherwise repeat steps 2 and 3.

This algorithm updates $\delta_t^i(bw)$ at each step, which gives the true gradients in the $L$-dimensional space. However, we are not guaranteed to find the global optimum, since we only allow the increment of bit-width in a single direction. It might be the case that no improvement exists along each of the $L$ directions, but it does exist with increments along multiple dimensions. With this algorithm, the search might be stuck in a local optimum.

C. Single-dimensional increment based on dynamic gradients

Like the first algorithm, this algorithm only allows single-dimensional increments, but it uses the true gradient $\delta_t^i(bw)$ as a measure of improvement, which requires online evaluations:

1. Initialize $bw = bw_{init}$. If $wer(bw) \leq \text{BWER}$, output $bw$;
2. Evaluate the gradients $\delta_i(bw)$, $i = 1..L$ on current $bw$ according to (23), where $L$ tests are needed to obtain the WERs;
3. Choose the direction $k = \text{argmax} \delta_k(bw)$, and increase $bw_k$ by one if it does not exceed the upper bound $M_k$;
4. Run the test to get the new $wer(bw^*)$. Output $bw^*$ if $wer(bw^*) \leq \text{BWER}$ or there is no improvement in WER observed; otherwise repeat steps 2, 3 and 4.

This algorithm updates $\delta_t^i(bw)$ at each step, which gives the true gradients in the $L$-dimensional space. However, we are not guaranteed to find the global optimum, since we only allow one- or two-dimensional increments, leading to $L + \binom{L}{2}$ possible candidates. We could extend it to include triplet increments, but it would take an intolerably long time to finish. The algorithm is described as follows:

1. Initialize $bw = bw_{init}$. If $wer(bw) \leq \text{BWER}$, output $bw$;
2. Evaluate the gradients $\delta_i(bw)$ and $\delta_{ij}(bw)$, $i = 1..L$, $j = 1..L$ on current $bw$ according to (23) and (24) respectively, where $L + \binom{L}{2}$ tests are needed to obtain the WERs;
3. Choose the direction $k$ or a pair of directions $\{k, l\}$ where $\delta_k(bw)$ or $\delta_{kl}(bw)$ is the maximum among all the single-dimensional and pair-wise increments. Increase the bit-width of $V_k$ or those of $\{V_k, V_l\}$ by one if no one exceeds its upper bound;
4. Run the test to get the new $wer(bw^*)$. Output $bw^*$ if $wer(bw^*) \leq \text{BWER}$ or there is no improvement observed, otherwise repeat steps 2, 3 and 4.
This algorithm is superior to the previous two in the sense that it explores many more candidate points in the search space. Consequently it requires much more computation and takes longer to complete.

VI. System Organization

Our database for training and testing is NYNEX PhoneBook [18], a phonetically-rich speech database designed for isolated-word recognition tasks. It consists of 93,667 isolated-word utterances recorded via telephone channels with an 8,000 Hz sampling rate. Each sample is encoded into 8 bits according to μ-Law. The training and testing sets are defined as in [19].

The acoustic features are the standard MFCCs plus the log energy and their deltas, leading to a 26-dimensional vector. We do not add their second deltas in this task, because they do not help in improving the baseline while they definitely cost more in terms of computation and memory. We apply mean subtraction and variance normalization to both the static and dynamic features in an attempt to make the system robust to noise.

The phone-based CHMMs are concatenated together into word-based models according to their pronunciations, where the transition probabilities between phones are determined by the pronunciation models. These embedded HMMs enable users to define their own voice commands by composing a word from phonemes. Our system has 42 phoneme models, each with 4 emitting states except for the silence model. The state probability distribution is a mixture of 12 diagonal Gaussians, which is an efficient configuration for the PhoneBook task.

The testing set consists of 8 subsets, each with a vocabulary of 75 words. The final WER is an average over them.

The front-end and the back-end are two main components of the recognizer. The back-end has been discussed in detail in the earlier sections. Here we only describe the front-end.

Our front-end consists of active speech detection and feature extraction. It expands each μ-law encoded sample into linear 16-bit PCM, and then creates a frame every 10ms (80 samples), each with a length of 25ms (200 samples). For each frame, we compute a smoothed log energy by putting the current log energy through a low-pass IIR filter. If the smoothed log energy goes above a threshold, we decide that the active speech has begun. We similarly decide upon the endpoint when that falls below a threshold. The thresholds are adaptive to the environment noise level.

Feature extraction is triggered immediately when the active speech is detected. It follows a standard procedure described in [20], where we apply pre-emphasis to each frame, multiply it with the hamming window, take the FFT, then calculate the log energy of the mel frequency bins, and finally perform a DCT to obtain the 13-dimensional static-feature vectors. We add the first order dynamic feature, followed by mean subtraction and variance normalization. The feature vectors obtained are fed into the back-end, where the pattern matching takes place.

Since we propose to apply the custom arithmetic to the back-end, but not to the front-end, we have to add an interface between them where we can convert the feature values into their corresponding integer indices. This is in fact the only place in the system where a codebook search is needed, and one of the biggest costs introduced by the custom arithmetic. At the interface, the codebook of x is loaded and a binary search is performed to get the index \( I_{x}(x) \) for a value of x. After that, all the data representations are in integer forms and all the arithmetic operations are via table lookups.

VII. Experiments and Results

In this section, we first present the results of single-variable quantization including those for accumulative variables, then we evaluate the system-wide optimization schemes on bit-width allocation. Note that since it was hard to obtain a large amount of online microphone input data, we used the 75-word PhoneBook testing sets recorded offline to run the experiments. We also tried using a 600-word testing set and observed similar results.

With regard to the quantization algorithm, we choose to use LBG [21] for all the experiments.

A. Single-variable quantization

In the single-variable quantization experiments, we quantize each variable individually, leaving all other variables at full precision.

A.1 Quantization on regular variables

Figure 2 shows the results of the single-variable quantization for all the model parameters in the back-end, which suggests that the floating-point representation of the model parameters is far from a compact form. It is interesting to see that the responsibility \( \mathbf{w} \) and transition probability \( \mathbf{a} \) can each be quantized to 1 bit without much loss of performance. The mean \( \mathbf{m} \) and variance \( \mathbf{v} \) can be represented by 3 and 4 bits respectively, which is reasonable since 3/4 bits in the scalar form implies \( 8^{26}/16^{26} \) possible vectors in the 26-dimensional feature space.

![Figure 2: Single-variable quantization for acoustic model parameters](image)

Figure 3 shows the results for all other free-variables in the back-end except for the accumulated Mahalanobis distance \( \mathbf{e} \) and the forward probability \( \eta \), which we will discuss separately in the following experiments. Note that here we
let \( q \) and \( b \) share the same codebook because their ranges of value have much in overlap. As shown in the figure, all variables here can be individually quantized to below 8 bits. For example, the scalar variable \( x \) of an observation vector and the state likelihood \( b \) can each be represented by 5 bits, which suggests that the online storage for features and evaluated likelihoods are much more than necessary if represented in full precision.

A.2 Linear accumulation vs. tree-structure accumulation

We proposed two quantization approaches in section III for the bounded accumulative variable \( e \). In our case, the operation \( e = e + d \) is repeated 26 times to get the final value of \( e \). It can be realized by one single table, using linear quantization. Alternately, it can have 5 different tables generated by the tree-structure accumulation, each with a relatively small size. For simplicity, we quantized the variables in each level of the tree structure with the same bit-width.

To compute the cost of the linear accumulation, we only consider the two functions in which \( e \) is involved, namely \( F_3(\cdot) \) and \( F_4(\cdot) \). If \( e, d, c \) and \( p \) are quantized to \( bw_e, bw_d, bw_c \) and \( bw_p \) bits respectively, the total size of the tables involved is \( bw_e 2^{bw_e} + bw_c + bw_d 2^{bw_d} + bw_p 2^{bw_p} \) bits. In this single-variable quantization experiment, we do not know the bit-widths of variables other than \( e \). However, we can assume \( bw_d = 5, bw_c = 5 \) and \( bw_p = 6 \) just for the sake of the cost calculation, where the bit-widths are the critical points in figure 3. This gives an approximation on the size of the tables involving \( e \) in the final customized system. We can similarly obtain the plots of “WER vs table size” for the two tree-structure accumulation schemes.

Figure 4 summarizes the results of the experiments on bounded accumulation. Tree-structure 1 denotes the method where adjacent pairs are added at all levels to form the next-level codebook, whereas in tree-structure 2, MFCCs are added to their corresponding deltas at the first level. It can be seen that in order to get baseline recognition rate, we need 6 Kbytes of lookup-tables to realize the operations involving \( e \) using linear accumulation. However, the table size goes up to almost 10 Kbytes for the tree-structure schemes to achieve the same goal. It is worth noting that although the tree-structure accumulation is not superior to the simple linear one in our experiments, it might manifest its advantages when the dimension of the iterative operation is significantly higher.

Figure 4: Single-variable quantization for accumulative variable \( e \)

A.3 Normalization and quantization for forward probability

As stated in section IV, the forward probabilities do not have a bounded dynamic range, which makes codebook design impossible. We therefore applied normalization to restrict the variable to within a certain range. To show the advantage of the normalization on quantization, we extracted samples of the forward probabilities without and with normalization on the same subset of training data, and generated codebooks for each case. Figure 5 shows the “WER vs. bit-width” for both unnormalized and normalized forward probabilities. Obviously the latter outperforms the former by saving 1 bit to achieve the baseline recognition rate. In fact, the longer the utterances, the more significant the advantage of normalization would be. We believe the benefits of normalization would be more conspicuous on a task with longer utterances, such as connected-digit or continuous speech recognition.

Figure 5: Single-variable quantization for forward probability

VIII. Summary and Conclusion

The floating-point and fixed-point units, designed for general-purpose uses, are not optimized for specific applications. For a costly task like speech recognition, they would be more expensive than necessary in terms of memory and power consumption. In this work, we have pro-
posed a speech recognition back-end based on custom arithmetic, where lookup-tables are used in substitute for arithmetic logic units and customized for a CHMM-based system. However, particular attention must to be paid to the table design of iterative operations, because they usually cause a large dynamic range in their outputs. We presented tree-structure accumulation as a solution. It did not perform better than linear accumulation in our experiments, but we expect it to be a promising method for higher-dimensional accumulations. Also, we presented a normalization method for the Viterbi search, which well controls the dynamic range of the forward probabilities, and reduces the corresponding bit-width in single-variable quantization and thereby the table size. Finally, we explored several bit-width allocation algorithms to optimally allocate resources among different tables. We found that the greedy algorithm allowing both single and pair-wise bit-width increment based on dynamic gradients gave fairly good results. Using the best table design schemes, our back-end requires 80 KBytes of tables for an isolate-word recognition task on PhoneBook with only a negligible degradation in recognition rate. It is expected to be significantly faster and less power-consuming than the back-end supported by a floating-point or fixed-point unit. Furthermore, the memory required for parameter storage and online computation can be greatly reduced.

In addition, we are looking forward to hardware support for our custom arithmetic, the amount of savings in cycles and power heavily depends on the physical realization of the lookup-tables and the ISA designed to support lookup operations.

ACKNOWLEDGMENTS

We would like to thank NSF for funding this project under Award-0086032.

REFERENCES


