ADAPTIVE-SEARCH TREE-STRUCTURED RESIDUAL VECTOR QUANTIZATION

Christian B. Peel, Xuegong Liu, Scott E. Budge

Electrical and Computer Engineering Department
Utah State University
Logan, UT 84322-4120
chris.peel@ieee.org

ABSTRACT

Full-search vector quantization (VQ) provides optimal results only with high memory and computational cost. We describe the computational and memory requirements of tree-structured VQ, residual VQ (RVQ), and tree-structured RVQ. We present multiple-rate, adaptive-search implementations of these VQ structures, and simulation results with video sequences. Tree-structured RVQ provides up to 1.5 dB PSNR quality improvement over RVQ, as well as significant perceptual improvement. These algorithms maintain many of the benefits of full-search VQ, while providing trade-offs between computational, storage, and performance requirements.

1. INTRODUCTION

"Today we have no time, but we have technology". Data compression is one way to make efficient use of time. Vector quantization (VQ) is the theoretically optimal way of quantizing, and thus compressing images [1]. To achieve optimality, large dimensional codebooks are needed, but the complexity and memory required by the encoder increase exponentially with codebook size. To overcome these problems, product code techniques such as tree-structured VQ and residual VQ are often used [2].

Tree-structured VQ (TSVQ) overcomes the problem of complexity by searching a group of small codebooks instead of one large codebook. The codebook that is searched at a node of the codebook tree depends upon the vectors chosen in previous nodes. Drawbacks of this method are that it uses more memory than full-search VQ, and also may choose a sub-optimal codevector [1].

Residual VQ (RVQ) overcomes the problem of memory by cascading several stages of VQ. With this method, complexity and memory are linear in the size of the basic codebook. Recently, RVQ has seen commercial application as the basis of Sorenson Vision’s Quicktime codec [3]. While RVQ has low computational and memory complexity, it is significantly lower in performance than full-search VQ. In this paper, we describe tree-structured RVQ, which allows a trade-off between between computation, complexity, and performance.

Section 2 introduces instrumentable vector quantizers, where we describe encoding, codebook design, and the computational complexity and memory requirements for several algorithms, including tree-structured RVQ. Multiple-rate, adaptive-search VQ is described in section 3, and finally simulation results for video sequences are presented in section 4.

2. IMPLEMENTABLE VECTOR QUANTIZERS

In this section we describe product-code vector quantizer structures which have linear growth in complexity. In a product-code vector quantizer, several VQ indices specify a codevector in an equivalent codebook which is the Cartesian product of the codebooks for each index [1, 4]. These structures are implementable in hardware [5] and software [3]. We describe tree-structured VQ, RVQ, and tree-structured RVQ (TRVQ²).

TSVQ greatly reduces the encoder complexity as compared to full-search VQ (FSVQ). The complexity of encoding with TSVQ is linear in the basic codebook size, while the complexity of FSVQ is exponential in the codebook size. Figure 1 shows the basic structure of a TSVQ codebook, where each node represents a vector quantizer. A vector is encoded by first quantizing it with the Level 1 FSVQ codebook, containing codevectors. The index of the vector chosen is used to determine which of N codebooks at Level 2 are used. This process is continued for M levels. Our method of encoding is to determine a level K at which to stop encoding, and to then send K ≤ M indices to the decoder, using K \cdot \log_2(N) bits. These indices would determine the codevector from the Kth codebook to use

²This is also known as multi-stage tree VQ (MSTVQ) in [1].
as the reconstruction vector. The memory required for the TSVQ codebook in numbers of vectors is \( \frac{N^{M+1} - N}{N-1} \), while the memory for the FSVQ codebook which would produce the same number of index bits would be \( N^M \) vectors.

![A Binary TSVQ Codebook.](image)

A TSVQ codebook is designed with a modification of the generalized Lloyd algorithm [6]. This technique is related to the binary-splitting technique often used to generate initial codebooks. First, an unstructured codebook (of size \( N \)) for Level 1 is designed using the basic generalized Lloyd algorithm. Each of the \( N \) partitions of the training set determined by this codebook is used as the training sets for \( N \) codebooks at Level 2. This process is repeated for every level of the tree, generating \( N^m \) codebooks for the \( m \)th level of the tree.

![Three stages of RVQ.](image)

RVQ has the same low computational complexity as TSVQ, but requires less memory. Figure 2 shows a block diagram of a three stage RVQ encoder. Each stage has its own codebook, which is a full-search quantizer. An encoder determines a stage \( K \) at which to stop encoding, and sends the first \( K \) VQ indices to the decoder. The reconstruction vector at the decoder is simply the direct sum of the \( K \) codevectors chosen. Memory for \( N \cdot M \) codevectors is required for an \( M \) stage RVQ codebook with codebooks at each stage containing \( N \) codevectors.

RVQ codebooks are also designed with a variation on the generalized Lloyd algorithm. RVQ can be considered as a subset of TSVQ, where there is only one codebook at each level of the tree. In our case, the residual vectors after coding the training set at one stage are used as the training set for the next stage. The distortion induced by the current and previous stages is used to determine the partition, while error from subsequent stages is ignored. An alternate method is to jointly optimize all stages. In this method of designing the codebook for one stage, while keeping the rest fixed, the final distortion after encoding to all stages is used [7].

Both TSVQ and RVQ are known to produce suboptimal results. The performance of the RVQ quantizer can be particularly poor, with significant loss of performance after only a few stages [1], while TSVQ requires large codebooks to achieve good performance. TRVQ provides a trade-off between the large memory requirements of TSVQ, and the low performance of RVQ.

![Two stages of TRVQ.](image)

Figure 3 shows the block diagram for a TRVQ encoder. Each quantizer in this example contains three levels of codebooks. With two stages, there are six indices available. The job of the encoder is to determine how many stages of RVQ, and how many levels of TSVQ to use. For example, if each codebook contains sixteen codevectors, the encoder may decide to send four 4-bit indices to the decoder. The first three indices would correspond to one codevector from the third level of the first tree-structured quantizer, and one codevector from the first level of the second quantizer.

Design of codebooks for TRVQ is a straightforward extension of ideas already presented. After designing the first stage quantizer using methods described for TSVQ, the training set is encoded. The residual from this encoding is used as the training set for the second stage quantizer. Succeeding stages are designed the same way.

3. ADAPTIVE-SEARCH VQ FOR VIDEO CODING

The performance of the methods described in section 2 is limited when we are required to use all of the possible \( M \) indices available in the codebook to encode each source vector. If we refer to each quantizer in a structured codebook as a “coding unit,” then a more efficient method would be to select \( K \leq M \) such that \( K \) coding units are used to form the product code to be sent to the decoder. A locally rate-distortion optimal method of selecting \( K \) is to compute the value of \( K \) which minimizes the Lagrangian cost

\[
J_i = D_i + \lambda R_i, \quad i = 1, ..., M,
\]

where \( R_i \) and \( D_i \) are respectively the rate and distortion induced by stopping at the \( i \)th coding unit.

Simulations were performed with an adaptive-search, locally rate-distortion optimal, mean-removed quantizer. The
target channel was constant-bit-rate, so a virtual buffer was used to constrain the rate. The bits produced by the quantization are fed to the buffer, before being passed to the channel at a constant rate. The fullness of the buffer was passed back to the quantization and used as negative feedback [8]. Specifically, a function of the buffer fullness was used as the Lagrangian minimization parameter \( \lambda \) to minimize the rate-distortion cost as illustrated in figure 4. This rate-distortion minimization is related to entropy constrained vector quantization [9]. The mean of each vector was removed before encoding, and the mean-only option was provided as an encoding option. When coding a motion-compensated residual, the option of doing nothing was also considered.

\[\text{Figure 4: Rate control by buffer fullness feedback.}\]

An important feature of this algorithm is its ability to encode at multiple rates. This is achieved merely by extracting data from the buffer at various rates. Huffman codes are used to encode the motion offsets, means, a header indicating how many VQ coding units to use, and the mode. Though a slight benefit was observed for training Huffman tables for each rate, we used tables trained at low rates to encode at all rates.

Block-based motion compensation was used to remove temporal redundancy. Simple segmentation into 16x16 or 8x8 blocks was provided as in the H.263 standard [10]. A locally rate-distortion-optimal decision [11] was made between the following four encoding-mode options: 1) No-update (or conditional-update), indicates that the corresponding block from the previously decoded frame is to be used, 2) Intra coding indicates that the block is to be coded without the benefit of motion compensation, 3) A motion residual using 16x16 motion compensation is coded, 4) A motion residual resulting from 8x8 motion compensation is coded.

During decoding, the Huffman code for the mode is first decoded, indicating which of the four encoding options are used. If motion is used, one or four motion offsets are decoded. If VQ is used, a header is first decoded, then a mean and several VQ indices, depending on the value indicated by the header [11]. Finally, a direct sum of a mean vector, VQ codevectors, and a motion compensation vector is made to obtain the decoded vector.

\[\text{Table 1: Memory Requirements}\]

<table>
<thead>
<tr>
<th>Label</th>
<th>Levels</th>
<th>Coding Units</th>
<th>Codebook Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVQ.6</td>
<td>1</td>
<td>6</td>
<td>24576</td>
</tr>
<tr>
<td>RVQ.12</td>
<td>1</td>
<td>12</td>
<td>49152</td>
</tr>
<tr>
<td>TRVQ2.6</td>
<td>2</td>
<td>6</td>
<td>208896</td>
</tr>
<tr>
<td>TRVQ2.12</td>
<td>2</td>
<td>12</td>
<td>417792</td>
</tr>
<tr>
<td>TRVQ3.6</td>
<td>3</td>
<td>6</td>
<td>2236416</td>
</tr>
<tr>
<td>TRVQ3.12</td>
<td>3</td>
<td>12</td>
<td>4472832</td>
</tr>
</tbody>
</table>

\[\text{Figure 5: Rate-PSNR curve for \textit{foreman} sequence.}\]

\[\text{Figure 5 shows the rate-distortion characteristics when using these codebooks to encode the \textit{foreman} sequence.}\]

\[\text{Figure 6 shows the performance obtained when encoding the \textit{mthr_dotr} sequence. These are the H.263 test sequences temporally downsampled to 10 frames per second. These figures illustrate the benefit of TRVQ. As we increase the amount of memory used for the codebooks (increase the number of tree levels), the performance increases. Inspection of these figures reveals that we obtain around one db PSNR better performance from TRVQ2, and one and a half db better from TRVQ3 over RVQ. We note that at low rates, the bit allocation approaches the mean-only case for intra}\]
We have presented multiple-rate, adaptive-search implementations of TSVQ, RVQ, and TRVQ, with a focus on the benefits of using TRVQ to trade-off design requirements for memory and computational resources. As the computational power and memory of computers increase exponentially over time (Moore’s law), we can predict what memory and computational resources will be available to us in the future, and design algorithms accordingly. For example, inspection of figures 5 and 6 reveals around one db PSNR improvement for two-level TRVQ over RVQ. Thus, allowing for more memory in the future, a significant quality improvement can be obtained simply by switching to two-level TRVQ from RVQ, while requiring the same low computational resources as RVQ.

Simulation results for encoding video with RVQ and TRVQ\(^3\) show that TRVQ can provide up to 1.5 db PSNR improvement over RVQ, while maintaining manageable performance. As storage resources increase, TRVQ provides a useful tool when making trade-offs between computational, storage, and performance requirements.

\(^3\)A significant benefit of using VQ for video coding is that decoding is a simple table-lookup. This asymmetry of encoding and decoding is particularly applicable to transmission of video over the Internet.

6. REFERENCES


