Common Quantization Methods

We have obtained optimization methods to achieve minimum distortion levels for scalar and vector quantization. Unfortunately, the optimization methods are usually expensive and long. Many simpler methods to achieve (not as good as the optimal one, but) quite good performance have been developed both for scalar and for vector quantization applications. We will list a few of these many methods here.

Common scalar quantizers:

1. **Adaptive step size quantizers:**
   The quantization step size is adjusted according to the dynamic range of the input.

2. **Companded quantization:**
   Especially in speech signals, the small variations of the signal are important for perception. So those variations must be "fine" quantized. On the other hand, if the signal power is high, the large variations need not be quantized that fine, because the ear is insensitive to small changes there. As a result, we must have a quantizer which is dense for small values around zero, and coarse for large values. This can be achieved by applying a distortion to the waveform first, next applying a uniform quantizer, and finally compensating the waveform distortion by an inverse operation. Consider the following block which amplifies small values more than large values:

```
| A compressor block |
```

The dual of this operation should be the following following block:
An expander block

If we perform a scalar quantizer between these blocks:

- Compressor
- Uniform Quantizer
- Expander

then the overall operation is equivalent to applying the non-uniform quantizer below:

This operation is very commonly used in speech and audio coders, even in CD audio coding and telephone speech. Due to the names: compressor and expander, the quantizer is called a companded quantizer.

---

**Common vector quantizers:**

1. **VQ codebook initialization method**: Pairwise Nearest Neighbor (PNN) algorithm:
   
   Due to Equintz (1989). Start with lot's of clusters (as many as number of the whole input vectors). Combine two closest clusters into a single vector and replace these codebook vectors by their average. Continue until you reach to the number of vectors allowed in the codebook.

   The question is to choose which two vectors to combine. Obviously, we have to combine the two clusters $C_i$ and $C_j$ that would result in minimal increase in distortion. The distortion increase given by combining $C_i$ and $C_j$ is given by
where $\mathbf{v}_i$ is the representative vector, and $n_i$ is the number of elements inside cluster $C_i$. Therefore, we have to choose two clusters which has minimum $D_{inc}$ as given in the above equation.

2. **Tree structured VQ**:

   a) Divide the set of points into two groups with two centroids, and therefore, two regions. Name these groups as group0 and group1. Note: We will obtain other groups by splitting the initial groups. So every input vector is first compared with respect to group0 and group1. If, for example, the data belongs to group0, then it will belong to subgroups of group0, and half of the comparisons are eliminated!

   b) Continue splitting each group until a sufficient number of groups are obtained (this corresponds to the codebook size). As an example, the groups under group0 will be called group00 and group01. The groups under the group01 will be group010 and group011, etc.

   In this way, the comparison or classification of an input vector is quite fast. After every comparison, we discard half of the remaining vector comparisons that we should normally make. For example, we start with a data. Compare it to group0 and group1. Suppose it is nearer to group1. Now we do not have to compare it to any other group under group0, any more. Then we compare it to group10 and group11. Suppose it is nearer to group10. Now we do not have to compare it to any group under group11 and under group0, any more. Then we compare it to group100 and group101. Suppose it is nearer to group101. Now we do not have to compare it to any group under group100, group11, or group0. The algorithm goes on like that, and provides a great computational improvement for finding the class of an input vector. The comparisons trace the tree structure given below:

![Tree structure](image)

3. **Pruned tree-structured VQ**:

   Once we build a tree structured VQ, the rate and distortion can be adjusted by carefully pruning sub-groups. Therefore, we initially obtain a high rate tree structured VQ with many resultant codebook vectors, and then remove some of the leaves which reduces the codebook size, but at the same time increases the distortion. Obviously, we have to remove the leaf which increases the distortion less. This corresponds to the best rate-distortion trade off. Breiman, et al proposed the method first, and Gray improved it for rate-distortion optimality.
Note: The structure given in the previous comparison technique for the tree-structured VQ is unfortunately violated in the pruned VQ, however, a better overall VQ is obtained. In fact, if coding efficiency is important, you should choose the pruned tree-structured VQ. On the other hand, if encoding speed is more important, and if you have limited computer resources, you may want to choose the normal tree-structured VQ.

4. **Shape-structural VQ:**
   Some region shapes repeat itself in 2D. As an example, you can tile the 2D plane using square shapes or rectangles. You can also use hexagonal shapes. In fact, there are many shapes which can completely tile the plane. These shapes are called lattice shapes. If you cluster your region with pre-defined lattice shapes, you obtain a lattice vector quantizer. As a matter of fact, this is equivalent to the uniform quantizer in the scalar case.
   Note: Not all shapes form a lattice. For example, you cannot tile the plane 2D plane using circles, or the 3D hyper-plane using spheres. The tiling must be complete, there must be no overlaps of shapes, and there must be no empty portions left.

5. **Mean removed VQ:**
   Sometimes, it is beneficial to remove the means of the vectors and design the VQ over the zero-mean input vectors. For example, for a vector of size 2x2, an image of size MxN has MxN/4 vectors to be encoded. But if you remove the means of each of these 2x2 vectors, you also have MxN/4 mean values which need to be coded. In some image coding tools, they perform VQ for the 2x2 zero-mean vectors, and perform scalar quantization for the MxN/4 mean values. Clearly, we increase the rate here (decrease compression ratio), but sometimes the improvement in distortion is worth it.

6. **Multi-stage VQ:**
   This is a multi-resolution technique. We first quantize the whole data using a coarse quantizer $Q_1$. The error between the original and the quantized version is the residual detail signal. If we want less error, we, then, quantize the detail signal using another quantizer $Q_2$. Now we have a different detail signal which is the difference between the original, and the sum of quantized original and quantized detail signals. Suppose we have three quantizers $Q_1$, $Q_2$, and $Q_3$. If $X$ is the input signal, we have
   \[ Y_1 = Q_1(X) \]
   \[ Y_2 = Q_2(X - Q_1(X)) \]
   \[ Y_3 = Q_3(X - Q_1(X) - Q_2(X - Q_1(X))) \]
   and the final reconstruction is $\hat{X} = Y_1 + Y_2 + Y_3$.
   Notice that, by using less detail, we could have the reconstruction to be, for example: $\hat{X}_a = Y_1 + Y_2$. However, the quantization error would be more.
   We could use more and more quantizers to quantize the residual detail signal as long as we want less distortion. In this aspect, the resolution of the signal improves by using more and more quantization stages. This can be listed as an algorithm to achieve a given distortion:
   a) Quantize $x$ with the first stage quantizer $Q_1$. 
b) If the residual $\|X - Q_1(X)\|$ is less than some determined threshold, then stop. Here, $Q_1(X)$ is the final quantization.

c) Otherwise, obtain the residual $X_1 = X - Q_1(X)$ and quantize it using $Q_2$. The new overall quantization is now: $Q(X) = Q_1(X) + Q_2(X)$, and the new residual is: $X_2 = X_1 - Q(X)$. Repeat steps (b) and (c) by updating the quantizer indexes and residual errors as long as the residual signal magnitude is above some level.

7. **Index adaptive VQ:**

   We start with a very large codebook that is available to the encoder and to the decoder. The encoder first transmit some information about which vectors inside the codebook it will use. It then encodes the used vectors. For example, suppose we have a large codebook of size 128. The encoder transmits a binary sequence 011010001 indicating that only the second, third, fifth, and ninth vectors will be used. As you see, we are left with 4 vectors only, and they can be represented using 2 bits instead of 7 bits which is necessary for 128 vectors. In some sense, we just select the most suitable vectors from a large selection of codebook vectors. The critical step is to determine which vectors are most appropriate at the encoder side. For this reason, the encoder must first determine a good (or optimum) set of vectors, and then select the nearest vectors to these optimal ones inside a set of, say, 128 vectors.

8. **The splitting algorithm (grow from scratch):**

   a) First, design a 0 rate code (only one representation):

   ![Diagram](a) 0 bit resolution

   Then add another vector to the codebook to obtain a rate 1 code. This way we have two vectors. Now, run the Lloyd algorithm to optimize the locations of the two vectors. The overall operation is like splitting an initial codebook vector into two:
b) Now add new vectors near the two vectors found above, then run the Lloyd algorithm again. Similar to the above case, we have split the two vectors into four:

(c) Continue splitting until the specified number of vectors are achieved.

9. **Splitting combined with Lloyd algorithm (The Linde Buzo Gray - LBG algorithm):** The algorithm described in 8 is also known as the LBG algorithm. Here, we will present the details. Just like the previous method, the LBG VQ design algorithm is an iterative algorithm which alternatively solves the above two optimality criteria. The algorithm requires an initial codebook $\mathbf{C}^{(0)}$. This initial codebook is obtained by the *splitting* method. You can think of this algorithm, therefore, the same as the Lloyd algorithm, however, the codebook size is increased using the splitting method. In this method, an initial codevector is set as the average of the entire training sequence. This codevector is then split into two. The iterative algorithm is run with these two vectors as the initial codebook. The final two codevectors are split into four and the process is repeated until the desired number of codevectors is obtained.

**Algorithm Steps:**

1. Given $M$ training vectors: $\mathcal{T} = \{x_1, x_2, \ldots, x_M\}$. Fixed stopping threshold: $\epsilon > 0$ to be a ``small'' number.
2. Let $N = 1$ and the average is:

$$c^*_1 = \frac{1}{M} \sum_{m=1}^{M} x_m.$$ 

Calculate distortion:

$$D^*_\text{ave} = \frac{1}{Mk} \sum_{m=1}^{M} ||x_m - c^*_1||^2.$$ 

3. **Splitting:** For $i = 1, 2, \ldots, N$, set

$$c^{(0)}_i = (1 + \epsilon)c^*_i,$$

$$c^{(0)}_{N+i} = (1 - \epsilon)c^*_i.$$ 

Set $N = 2N$ (split all vectors).

4. **Iteration:** Let $D^{(0)}_\text{ave} = D^*_\text{ave}$. Set the iteration index $i = 0$.

   i. For $m = 1, 2, \ldots, M$, find the minimum value of

   $$||x_m - c^{(i)}_n||^2,$$

   over all $n = 1, 2, \ldots, N$. Let $n^*$ be the index which achieves the minimum. Set

   $$Q(x_m) = c^{(i)}_{n^*}.$$ 

   ii. For $n = 1, 2, \ldots, N$, update the codevector

   $$c^{(i+1)}_n = \frac{\sum_{Q(x_m) = c^{(i)}_n} x_m}{\sum_{Q(x_m) = c^{(i)}_n} 1}.$$ 

   iii. Set $i = i + 1$.

   iv. Calculate

   $$D^{(i)}_\text{ave} = \frac{1}{Mk} \sum_{m=1}^{M} ||x_m - Q(x_m)||^2.$$ 

   v. If $(D^{(i-1)}_\text{ave} - D^{(i)}_\text{ave})/D^{(i-1)}_\text{ave} > \epsilon$, go back to Step (i).

   vi. Set $D^*_\text{ave} = D^{(i)}_\text{ave}$. For $n = 1, 2, \ldots, N$, set

   $$c^*_n = c^{(i)}_n$$

   as the final codevectors.

5. Repeat Steps 3 and 4 until the desired number of codevectors is obtained.

Please [click here](#) to see (about 176KB, so may be slow to download) a gif animation which illustrates the running of LBG algorithm on a two-dimensional data.