Fast Sequential Implementation of “Neural-Gas” Network for Vector Quantization

Clifford Sze-Tsan Choy and Wan-Chi Siu, Senior Member, IEEE

Abstract—Although the “neural-gas” network proposed by Martinetz et al. in 1993 has been proven for its optimality in vector quantizer design and has been demonstrated to have good performance in time-series prediction, its high computational complexity (O(N log N)) makes it a slow sequential algorithm. In this letter, we suggest two ideas to speed up its sequential realization: 1) using a truncated exponential function as its neighborhood function and 2) applying a new extension of the partial distance elimination method (PDE). This fast realization is compared with the original version of the neural-gas network for codebook design in image vector quantization. The comparison indicates that a speedup of five times is possible, while the quality of the resulting codebook is almost the same as that of the straightforward realization.

Index Terms—Neural-gas network, partial distance elimination, vector quantization.

I. INTRODUCTION

VECTOR QUANTIZATION (VQ) [1] is a process through which a vector space is partitioned, such that an input in each partition is represented by a vector called a codeword or a codevector. Suppose in a K-dimensional Euclidean space \( R^K \), the probability density function (pdf) of vector \( \mathbf{x} \) is given by \( p(\mathbf{x}) \). The distortion introduced by replacing a vector \( \mathbf{x} \) with a codeword \( \mathbf{y} \) is denoted by \( D(\mathbf{x}, \mathbf{y}) \), and the squared distortion measure is a popular one, i.e.,

\[
D(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{K} (x_i - y_i)^2
\]

where \( x_i \) and \( y_i \) are the values of the \( i \)th dimensions of \( \mathbf{x} \) and \( \mathbf{y} \), respectively. Given \( N \) codewords, \( \{w_i\} \) (which are collectively referred to as a codebook), the vector space is partitioned into \( N \) regions \( \Omega_j \) satisfying \( \forall i \neq j \) s.t. \( \Omega_i \cap \Omega_j = \emptyset, \Omega_i \cup \Omega_j = R^K \), and \( \forall i \neq j \) s.t. \( \Omega_i = \{x: D(\mathbf{x}, w_i) \leq D(\mathbf{x}, w_j)\} \). The objective is to design a codebook which minimizes the expected distortion introduced.

The generalized Lloyd algorithm (GLA) [2] is a well-established approach for designing a codebook in vector quantization. The GLA requires a set of training vectors, which approximates the distribution of vectors from the source to be vector-quantized. An initial codebook of \( N \) codewords is randomly selected, which partitions the training set into \( N \) disjoint subsets using the nearest neighbor criterion as described in the previous paragraph. The centroid of each subset is then determined, which forms a new codebook. This reduces the expected distortion introduced on this training set. This process is iterated until either there is no reduction in expected distortion or a prespecified number of iterations has completed. Since the GLA designs the codebook based on the training set, if the distribution of vectors from the source changes with time, a new training set has to be defined and the GLA be reapplied to define a new codebook. Hence, the GLA is considered as an offline or batch approach which is not suitable for designing a codebook which adaptively tracks source characteristics. The \( k \)-means algorithm [3] can be considered as an online or adaptive version of the GLA, in which vectors sampled from the source are used to adaptively modify the codebook and, hence, can adapt to changes in vector distributions. However, the quality of the codebooks from both approaches depends highly on the initial codebooks, which means that they are very likely to produce poor quality codebooks.

Recently, a number of neural networks have been successfully applied to codebook design [4]–[7]. These neural approaches are characterized by their adaptive nature and are highly insensitive to initial conditions. The “neural-gas” network proposed by Martinetz et al. [7] is particularly important since it was proven that in the limit of large \( N \) the density function of codevectors is proportional to \( p(\mathbf{x})^{N/K+2} \) when the squared distortion measure (1) is used. This is equivalent to the density distribution of codevectors of an optimal codebook in the asymptotic limit of large \( N \) with the same measure [8]. In addition, it was demonstrated by Martinetz et al. [7] that the neural-gas network takes a smaller number of learning steps to converge than that of three other algorithms they tested, including the \( k \)-means algorithm [3]. Furthermore, the neural-gas network has also been applied to time-series prediction which outperforms two other neural approaches. Other researchers have also demonstrated various applications of the neural-gas network [9], [10].

Despite all of these advantages, the neural-gas network suffers from high computational complexity for its sequential realization—a complexity of \( O(N \log N) \) [7]. Although highly parallel implementation can improve its processing speed, sequential implementation is still more economical, practical, and flexible, especially with the recent wide availability of powerful processors including digital signal processors (DSP’s). It is obvious that with a fast sequential realization, the neural-gas is more feasible in solving problems with larger \( N \) (less error in VQ and prediction). In next section the neural-gas algorithm is briefly reviewed. Then in Section III we will...
show that by limiting the number of weight vectors to be updated, a substantial speed improvement is possible. Finally, experimental results on codebook design for image VQ are presented, followed by a conclusion.

II. THE NEURAL-GAS ALGORITHM

Let \( N \) be the number of neurons in the neural-gas network, \( \mathbf{w}_i \) be the weight vector for the \( i \)th neuron, and \( \mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_N) \) denote the codebook. Suppose \( \mathbf{v} \) is a vector from the training set. Let \( (\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_N) \) be the “neighborhood-ranking” relative to the training vector \( \mathbf{v} \) such that \( \forall p < q, D(\mathbf{v}, \mathbf{w}_p) \leq D(\mathbf{v}, \mathbf{w}_q) \). Hence, for example, \( \mathbf{w}_1 \) is the weight vector closest to \( \mathbf{v} \), while \( \mathbf{w}_2 \) is the second closest, and so on. Then \( k_p(\mathbf{v}, \mathbf{W}) \) is defined for the \( p \)th neuron, such that \( k_p(\mathbf{v}, \mathbf{W}) = p \). The updating rule for the weight vector of the \( i \)th neuron is

\[
\Delta \mathbf{w}_i = \epsilon \cdot h(t)(i_p(\mathbf{v}, \mathbf{W})) \cdot (\mathbf{v} - \mathbf{w}_i)
\]

(2)

where \( h(t) \) is the neighborhood function. Martinetz et al. [7] reported that an exponential function (i.e., \( h(t) = e^{-(t/\lambda)} \)) should give the best overall result as compared to other choices like Gaussians. The values of \( \lambda \) and \( \epsilon \) decrease as learning progresses. As suggested in [7], we adopt the exponential decreasing schedules for these two parameters, so that

\[
\lambda(t) = \lambda t^t_{\text{max}} \\
\epsilon(t) = e^{-t/t_{\text{max}}}
\]

(3)

where \( \lambda_i \) and \( \epsilon_i \) are the initial values of \( \lambda \) and \( \epsilon \), while \( \lambda_f \) and \( \epsilon_f \) are the final values of \( \lambda \) and \( \epsilon \) (i.e., at time \( t = t_{\text{max}} \)).

III. OUR FAST IMPLEMENTATION

We divide the learning process into two phases: the ranking determination phase and the weights updating phase. In its straightforward realization, the former phase is most time-consuming since all distances have to be evaluated and are then sorted to obtain the “neighborhood-ranking.” Let us consider the latter phase first. The processing time of this phase can be reduced by only updating those weight vectors whose learning rate \( \epsilon(t)h(t) \) is nonnegligible. Since \( \epsilon(t) \) is nonincreasing according to (3), only weights with sufficiently large value of \( h(t) \) are updated. This is equivalent to using the truncated exponential function

\[
h'_2(k) = \begin{cases} 
  e^{-(k/\lambda)}, & \text{if } k \leq r \cdot \lambda \\
  0, & \text{otherwise}
\end{cases}
\]

(4)

where \( r \) is a constant to be selected and determines the number of neurons to be updated at each learning step, i.e., those weights with nonzero value of \( h'_2(k) \) are to be updated. According to (3), \( \lambda \) decreases exponentially, and this is also the case for the number of neurons to be updated. We will demonstrate experimentally that when \( r \) is appropriately chosen, appreciable speedup is possible while performance degradation in the codebook introduced by not updating some weight vectors is insignificant.

Before proceeding, let us introduce the following notations. Let \( S_p = \{0, 1, \ldots, P - 1\} \) be a set of \( P \) neuron indexes. With an input vector \( \mathbf{v} \), we define an operator \( R_P = (i_0, i_1, \ldots, i_{P-1}) = \text{Order}_{\mathbf{v}}(\mathbf{v}, S_P) \) which returns an ordered \( P \)-tuple \( R_P \), satisfying the following: \( \forall j \text{ s.t. } i_j \in S_P, \forall j \neq k \text{ s.t. } i_j \neq i_k, \forall j < k \text{ s.t. } D(\mathbf{v}, \mathbf{w}_{i_j}) \leq D(\mathbf{v}, \mathbf{w}_{i_k}) \) and \( \forall j < k \text{ s.t. } D(\mathbf{v}, \mathbf{w}_{i_j}) = D(\mathbf{v}, \mathbf{w}_{i_k}) \text{ s.t. } i_j < i_k \). Furthermore, let us define another operator \( \mathbf{R}_Q = (i_0, i_1, \ldots, i_{Q-1}) = \text{Order}_{\mathbf{v}}(\mathbf{v}, S_Q) \), such that the ordered \( Q \)-tuple \( (Q \leq P) \mathbf{R}_Q \) is the first \( Q \) elements of the ordered \( P \)-tuple \( \text{Order}_{\mathbf{v}}(\mathbf{v}, S_P) \), i.e., \( \forall j < Q \text{ s.t. } i_j = i_j \). For an illustration, let us consider a simple example where \( P = 4 \), such that distances between an input vector \( \mathbf{v} \) and the weight vectors are given by \( D(\mathbf{v}, \mathbf{w}_0) = 30, D(\mathbf{v}, \mathbf{w}_1) = 20, D(\mathbf{v}, \mathbf{w}_2) = 10, \) and \( D(\mathbf{v}, \mathbf{w}_3) = 20 \). Then, \( \mathbf{R}_P \) becomes \((2, 1, 3, 0)\), while \( \mathbf{R}_Q \) is equal to \((2, 1)\).

According to our notation, \( \text{Order}_{N}(\mathbf{v}, \mathbf{S}_N) \) denotes the ordering of indexes relative to input \( \mathbf{v} \) (another form of “neighborhood-ranking”). Let \( F(t) = \min\{t \cdot \lambda(t) + 1, N\} \) where \( \lceil x \rceil \) is the largest integer smaller than or equal to \( x \). Then by using (4), we only have to update weight vectors of those neurons whose indexes correspond to the first \( F(t) \) elements of \( \text{Order}_{N}(\mathbf{v}, \mathbf{S}_N) \), i.e., \( \text{Order}_{N}(\mathbf{v}, \mathbf{S}_N) \). When \( F(t) < N \), not all of the neurons are updated; hence, there is no need to determine the entire “neighborhood-ranking.”

Under this situation, \( \text{Order}_{N}(\mathbf{v}, \mathbf{S}_N) \) is determined with our proposed algorithm (which extends the partial distance elimination (PDE) method [11]) without evaluating and sorting all distances.

In the following, let \( F(<N) \) be the number of neurons to be updated. An operator \( (\mathbf{I}, \mathbf{D}) = \text{Sort}(\mathbf{S}_L)(F \leq L) \) returns two ordered \( F \)-tuples \( \mathbf{I} = (i_0, i_1, \ldots, i_{F-1}) \) and \( \mathbf{D} = (d_0, d_1, \ldots, d_{F-1}) \), such that \( \mathbf{I} = \text{Order}_{\mathbf{v}}(\mathbf{v}, \mathbf{S}_L) \), and \( \forall j \text{ s.t. } d_j = D(\mathbf{v}, \mathbf{w}_{i_j}) \). Then, the algorithm is as follows.

Algorithm 1: Extended PDE

1) Set \( (\mathbf{I}(F), \mathbf{D}(F)) = \text{Sort}(\mathbf{S}_F) \).
2) For \( p = F \) to \( N - 1 \), do
   a) Set \( \mathbf{D}_{\text{min}} = \mathbf{D}_{F-1}(p) \). Set \( \mathbf{D} = 0 \).
   b) Let \( \mathbf{v} \) be represented by \( (x_1, \ldots, x_K) \) and \( \mathbf{w}_p \) be represented by \( (y_1, \ldots, y_K) \).
      For \( j = 1 \) to \( K \), do
      i) Set \( D = D + (x_j - y_j)^2 \).
      ii) If \( D \geq \mathbf{D}_{\text{min}} \),
         A) Set \( \mathbf{I}(p+1) = \mathbf{I}(p) \).
         B) Set \( \mathbf{D}(p+1) = \mathbf{D}(p) \).
         C) Goto Step 2 to process the next value of \( p \).
      c) Set \( (\mathbf{I}(p+1), \mathbf{D}(p+1)) = \text{Sort}(\{i_{0:p}, i_{1:p}, \ldots, i_{F-2:p}, \ldots, i_{F-1:p}\}) \).
3) Return the two ordered tuples \( \mathbf{I}(N) \) and \( \mathbf{D}(N) \) and terminate.

The aim of the algorithm is to implement the operation \( (\mathbf{I}, \mathbf{D}) = \text{Sort}(\mathbf{S}_N) \) incrementally. Let us suppose that, just before the execution of Step 2a), it is true, for a particular \( p \), that \( (\mathbf{I}(p), \mathbf{D}(p)) = \text{Sort}(\mathbf{S}_p) \),
TABLE I

<table>
<thead>
<tr>
<th>Image</th>
<th>$r = 3$</th>
<th>$r = 6$</th>
<th>$r = 9$</th>
<th>$N = 32$</th>
<th>$N = 64$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena</td>
<td>22.26</td>
<td>22.92</td>
<td>22.92</td>
<td>22.93</td>
<td>23.46</td>
</tr>
<tr>
<td>Tiffany</td>
<td>21.53</td>
<td>21.99</td>
<td>22.00</td>
<td>21.98</td>
<td>22.49</td>
</tr>
<tr>
<td>Baboon</td>
<td>16.40</td>
<td>16.75</td>
<td>16.78</td>
<td>16.78</td>
<td>17.07</td>
</tr>
<tr>
<td>Peppers</td>
<td>21.81</td>
<td>22.41</td>
<td>22.41</td>
<td>22.41</td>
<td>23.15</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Image</th>
<th>$r = 3$</th>
<th>$r = 6$</th>
<th>$r = 9$</th>
<th>$N = 128$</th>
<th>$N = 256$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lena</td>
<td>24.59</td>
<td>25.10</td>
<td>25.10</td>
<td>25.10</td>
<td>25.65</td>
</tr>
<tr>
<td>Tiffany</td>
<td>23.40</td>
<td>23.74</td>
<td>23.75</td>
<td>23.76</td>
<td>24.21</td>
</tr>
<tr>
<td>Baboon</td>
<td>17.76</td>
<td>18.03</td>
<td>18.05</td>
<td>18.05</td>
<td>18.38</td>
</tr>
<tr>
<td>Peppers</td>
<td>24.30</td>
<td>24.65</td>
<td>24.65</td>
<td>24.65</td>
<td>25.27</td>
</tr>
</tbody>
</table>

where $I(p) = (i_0(p), i_1(p), \ldots, i_{F-1}(p))$, and $D(p) = (d_0(p), d_1(p), \ldots, d_{F-1}(p))$. It is obviously true when $p = F$ according to Step 1. Then, the purpose of Steps 2a–c) is to implement $(I(p + 1), D(p + 1)) = \text{Sort}(S_{p+1})$ from $I(p)$ and $D(p)$ in an efficient way. Indeed, we have to determine whether $p$ is in $I(p + 1)$ or not. Before considering these two cases, let us recall that $d_{F-1}(p)$ is the largest distance between vector $v$ and the weight vectors of neurons in $I(p)$. If $D(v, w_p) \geq d_{F-1}(p)$, we know that $p$ is not in the set $I(p + 1)$. (Hence, when two neurons have the same distance from the input vector $v$, our algorithm favors the smaller index, which agrees with our definition of $\text{Order}_{p+1} I(v, S_{p+1})$.) The above condition does not require a complete evaluation of $D(v, w_p)$; hence, the short-cut evaluation in the PDE is applied in Step 2b). On the other hand, if $D(v, w_p) < d_{F-1}(p)$, $p$ must be an element in the tuple $I(p + 1)$, while $i_{F-1}(p)$ must not be present in it. In fact, the realization of the operator $\text{Sort}(S)$ is different for Steps 1 and 2c). In the former, complete distance evaluations of the $F$ neurons are performed and then sorted (with a complexity of $O(F \log F)$). In the latter, the distances of the $F$ weight vectors from the input are available—note that $D = D(v, w_p)$ in this case, while other $F - 1$ distances are available from $D(p)$. This means that at most $F - 1$ comparisons and copying operations have to be performed in this case.

Since $F$ decreases with learning, initialization time in Step 1 decreases progressively as well, especially when $F$ is small.

Therefore, this is a substantial speed improvement as compared to the sorting of $N$ distances in its straightforward realization. In the degenerated case when $F = 1$, our extended PDE becomes the PDE in [11].

IV. RESULTS

Let us demonstrate the speedup of our implementation by applying the neural-gas network in VQ for images. Each image has $512 \times 512$ pixels, and each vector is obtained from a subblock of size $4 \times 4$ pixels. The quality of the codebook so generated is determined by the signal-to-noise ratio (SNR), where

\[
\text{SNR} = 10 \log_{10} \frac{1}{N} \sum_{v \in \Omega} |v|^2 - \sum_{v \in \Omega} \frac{|v - w_i(v)|^2}{|w_i(v)|^2} \tag{5}
\]

with $\Omega$ denoting the training set, and $w_i(v)$ denoting the weight vector closest to the input vector $v$. In our experiments, we have considered codebook sizes of $N = 32, 64, 128$, and $256$. We select $\lambda_F = 0.01$, $\epsilon_i = 0.5$, and $\epsilon_f = 0.005$, which are according to values used in [7]. The value of $\lambda_i = N/8$ was chosen, since, according to our analysis, this value gives the codebook with the best quality for parameters to be chosen. For each combination of the image to be used and the codebook size for each implementation, three independent trials are
performed. Table I shows the average SNR of the codebook from a direct implementation and our fast implementation with $r = 3, 6,$ and $9.$ It is a general trend that a larger value of $r$ gives better codebook on average. Table II shows the corresponding average processing times consumed for the implementation on a PC with a Pentium processor at 90 MHz. It is observed that the processing time increases slowly with an increase in the value of $r.$ Since the quality of the codebook with $r = 6$ is very close to that from the original implementation, this value is a good compromise between quality and speed. With this value, a speedup of more than five times is possible.

V. CONCLUSIONS

In this letter, we propose a fast implementation of the neural-gas network by limiting the number of weight vectors to be updated to only those with a nonnegligible learning rate. In this way, we can apply an extended PDE method to reduce appreciably its processing time. Experimental results indicate that a speedup of more than five times is possible, while the quality of the codebook is almost the same as that from the original algorithm. With our fast implementation, it is possible to design a better codebook with the same processing time on a sequential machine as compared to the direct implementation.

REFERENCES