Learning Vector Quantization with Adaptive Prototype Addition and Removal

Mihajlo Grbovic and Slobodan Vucetic

***Abstract*— Learning Vector Quantization (LVQ) is a popular class of nearest prototype classifiers for multiclass classification. Learning algorithms from this family are widely used because of their intuitively clear learning process and ease of implementation. They run efficiently and in many cases provide state of the art performance. In this paper we propose a modification of the LVQ algorithm that addresses problems of determining appropriate number of prototypes, sensitivity to initialization, and sensitivity to noise in data. The proposed algorithm allows adaptive addition of prototypes at potentially beneficial locations and removal of harmful or less useful prototypes. The prototype addition and removal steps can be easily implemented on top of the many existing LVQ algorithms. Experimental results on synthetic and benchmark datasets showed that the proposed modifications can significantly improve LVQ classification accuracy while at the same time determining the appropriate number of prototypes and avoiding the problems of initialization.**

I. Introduction

Learning Vector Quantization (LVQ) has been introduced by T. Kohonen [27] as a simple, universal and efficient classification algorithm and has since found many applications and extensions [21]. LVQ is a class of learning algorithms for nearest prototype classification (NPC). Like the nearest neighbor method [22], LVQ is a local classification algorithm, where the classification boundaries are approximated locally. The difference is that instead of using all the points in the training dataset, LVQ uses only a set of appropriately chosen prototype vectors. This way the classification method is much more efficient, because the number of vectors that should be stored and compared with is significantly reduced. In addition, a carefully chosen prototype set can greatly increase classification accuracy on noisy problems.

LVQ is introduced as an algorithm that efficiently learns the appropriate prototype positions used for classification. It is defined by a set of P prototypes $\{(m_j, c_j), j = 1\ldots P\}$, where m_j is a K-dimensional vector in the feature space, and c_j is its class label. Typically, the number of prototypes is larger than the number of classes. This way, each class is represented with more than one prototype. Given an unlabeled data point x_u, its class label y_u is determined as the class c_q of its nearest prototype m_q

$$y_u = c_q,\ q = arg\ min_j\ d(x_u, m_j) \tag{1}$$

where d is the Euclidean distance. Other distances measures can be used as well, depending on the problem at hand. The space required to store an LVQ classifier is $\Theta(PK)$ and the time needed to classify an unlabeled point is $\Theta(PK)$. The number of prototypes P can be selected to meet the computational constraints. Clearly, the number of prototypes used represents a tradeoff between computational cost and classifier complexity.

The training of LVQ starts with placing the prototypes at some initial positions in the input space. In the simplest scenario, assuming there are C classes with balanced class distributions, P/C prototypes of each of the C classes are selected by random sampling from training examples of the corresponding class. LVQ algorithm then sequentially scans the training data points, possibly in multiple rounds and in an arbitrary order. If the LVQ classifier agrees with the actual class of a training data point, the prototypes are not modified. Otherwise, location of one or more prototypes is updated in an attempt to increase the LVQ accuracy.

There are several different LVQ algorithms that deal with the updates of the prototypes in different ways. Description of their three main variants, LVQ1, LVQ2, and LVQ3 can be found in [27], while some viable alternatives (LFM, LFMW, LVQ+/-) are described in [17]. The common for all is the idea to move the nearest prototype away from the incorrectly classified training point and to (optionally) move the nearest prototype of the correct class towards it.

LVQ2.1 has been shown to provide good NPC classifiers and is commonly used as a representative of LVQ algorithms. Given a training data point (x,y), three conditions have to be met for LVQ2.1 to update its prototypes: 1) Class of the prototype closest to x has to be different from y, 2) Class of the next closest prototype has to be equal to y, and 3) x must fall near the hyperplane at the midpoint between the closest (let us denote it as m_j) and the second closest prototype (let us denote it as m_k). These two prototypes are then modified as

$$m_j(t{+}1) = m_j(t) - \alpha(t)\ (x - m_j(t)) \tag{2}$$

$$m_k(t{+}1) = m_k(t) + \alpha(t)\ (x - m_k(t))$$

where t represents how many updates have been made, and $\alpha(t)$ is a monotonically decreasing function of time. Common initial value for $\alpha(0)$ is 0.03, and it is being decreased as $\alpha(t) = \alpha(t{-}1)\cdot(1{-}i/N)$, $i = 1\ldots N$. Let d_j and d_k be the distances between x and m_j and m_k. Then, x is defined as near the midpoint if $min(d_j/d_k, d_k/d_j) > s$, where s is a

This work was supported by the U.S. National Science Foundation under Grant IIS-0546155.

M. Grbovic and S. Vucetic are with the Center for Information Science and Technology, Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. (S. Vucetic phone: 215-204-5535; fax: 215-204-5082; e-mail: vucetic at ist temple edu).

constant commonly chosen between 0.4 and 0.8. This "window rule" is introduced to prevent prototype vectors from diverging.

Even though LVQ algorithms provide good classification results and are widely used, especially in real time applications such as speech recognition [28, 9], they have their disadvantages. First, the classification results depend on the initial choice of prototypes. If the initialization is not done in a proper way good classification results might never be achieved. Second, since LVQ algorithms typically choose the same number of prototypes per class and the decision on how many prototypes to use is left to the user, there is no guarantee that the prototypes are going to adapt to the data in the best possible way. In multiclass datasets it might happen that some classes have more complicated distributions in the feature space than others. It would seem logical that they get more prototypes. This also becomes an issue when the class distribution is highly unbalanced. Third, LVQ algorithms are not robust to noisy data and prototypes are sometimes trapped in the positions where they are doing more harm than good.

In order to deal with such weaknesses we introduce a novel modification of the LVQ algorithms called Adaptive LVQ. Our approach is different from basic LVQ algorithms because it lets prototypes adapt to the data during training. Starting with a single prototype per class, we are adding prototypes at appropriate positions and removing prototypes from inappropriate positions in order to optimize the accuracy.

In general, our method can be applied to any algorithm in the LVQ family.

II. Related Work

There have been many attempts to improve the performance of LVQ algorithms. Besides having different versions of LVQ in order to achieve better classification results and better adapt to specific applications, there are whole algorithms that are developed to address the specific weaknesses of LVQ.

The Generalized LVQ (GLVQ) algorithm proposed by Sato [4] is just one of the powerful variants of the Kohonen's LVQ algorithm. GLVQ algorithm has a continuous and differentiable cost function. The updating rule is obtained by minimizing this cost function, ensuring algorithm convergence. Although the GLVQ algorithm has shown promising results in various applications it has been shown that its performance deteriorates when dealing with complex data sets with non linear class boundaries. It also encounters a problem of determining the appropriate number of prototypes and is sensitive to initialization of prototype positions and thus many prototypes may get trapped in local minimal states.

This idea of creating an update rule by minimizing a certain cost function is very popular with today's researchers. As a result, many Soft versions of LVQ algorithm have been developed [24], [20], [2], [3], [15].

The popular Soft LVQ introduced in [24] is similar in design to the VQ classifier using Deterministic Annealing [8]. The problem is treated as optimization, where the probability density that the data point *x* is generated by the model of the correct class is calculated and compared it to the probability density that this point is generated by the models of the incorrect classes. The logarithm of the ratio of the correct vs. the incorrect probability densities serves as the cost function to be maximized. One slight issue with this approach is that it introduces an additional hyperparameter σ, besides the existing α. As stated in [24], one way to use σ is to find the optimal value of this hyperparameter using the validation data set. In this way, the training set has to be examined several times before the start of LVQ, which is time consuming and could also be imprecise in the case when the training data set is very small. The other way to use σ is to calculate its initial value and then anneal it during the training procedure. This way, the above stated problems are reduced.

There is also an issue of the update step α. Some researchers feel that this update step should be different for different prototypes. This idea culminated in development of the optimized learning rate LVQ1 (OLVQ1) algorithm [26] which gathers statistics about each prototype during the training procedure and then modifies their update values accordingly.

In LVQ algorithms and other prototype-based models a data point is usually compared with a prototype according to the Euclidian distance. However, some researchers [19] claim that the specific structure of the data space can and should be accounted for by selecting an appropriate metric. Once a suitable metric is identified, only then can it can be further utilized for the design of good classifiers. In supervised scenarios, auxiliary class information can be used for adapting parameters improving the specificity of data metrics during data processing, as proposed by Kaski for (semi-)supervised extensions of the LVQ [23]. Another metric-adapting classification architecture is the generalized relevance learning vector quantization (GRLVQ) developed by Hammer and Villmann [6].

There has also been a lot of research done in modifying LVQ to perform feature selection during training. Standard LVQ does not discriminate between more and less informative features and their influence on the distance function is equal. Several approaches that deal with this problem can be found in literature [1, 18]. These algorithms modify feature weights and employ a weighted distance function. Influences of features which are frequently contributing to misclassifications of the system are reduced while the influences of very reliable features are increased.

Another idea that can be found in literature is to take more "runners-up" into account [25] because it occurs quite commonly that an input vector has several neighboring codebook vectors that are almost equidistant from it. Therefore, in order to increase the accuracy of learning, the corrections might not only be restricted to the "winner" and the first "runner-up", but could also take the "runners-up" into account. An additional advantage of this approach is that, whereas the "runners-up" are often of the correct class, their simultaneous correction tends to update them more smoothly.

Some researchers extend a local prototype-based learning model by active learning, which gives the learner the capability to select training samples during the model adaptation procedure. It is claimed [10] that by using these active learning strategies the generalization ability of the model could be improved accompanied by a significantly improved learning speed.

Some recent work proposes margin analysis of LVQ to provide theoretical justification for the LVQ algorithm. Roughly speaking, margins measure the level of confidence a classifiers has with respect to its decisions. Margin analysis has become a primary tool in machine learning during the last decade (SVM [29]). Buckingham and Geva [16] were the first to suggest that LVQ is indeed a maximum margin algorithm. They presented a variant named LMVQ and analyzed it. In [14], a geometrically based margin similar to SVM was defined and used for analysis.

Adequate initialization of the codebook vectors is highly important issue with the LVQ algorithm [25]. In general, the optimal numbers of prototypes per class are not easy to derive due to the complexities of class-conditional densities and variability in class distributions. The simplest way to initialize prototypes is to select them by random sampling from the available training points. An alternative way, as suggested by some researchers, is to use the K-means clustering [12], [11] to define the initial locations of prototypes. One option is to perform clustering on each class separately and take cluster centroids as initial prototypes. A slightly different version of K-means initialization can be found in [11] where the placement of prototypes is determined without taking their classification into account. In this case, the prototypes are labeled based on the majority classes assigned to their cluster. This choice, although rather successful in the case of smooth and continuous distributions, does not guarantee any stable or unique distribution in difficult cases where it can even happen that some classes remain without prototype representatives.

Interesting alternatives for the initialization include supervised neural gas (SNG) algorithm [5] and cost-function adaptation [3] that replaces minimum Euclidian distances with harmonic average distances and alleviates the initialization sensitiveness problem.

The number of prototypes to be assigned to each class is another problem to which there is no clear solution. Experimentally it has been found that a good strategy in practice, in absence of any other, is to assign an identical number of prototypes to each class, even when the class distributions are very dissimilar [26]. An alternative would be to assign the number of prototypes proportionally to the class distribution. Another approach is to make sure that the average distances between the adjacent prototypes in each class are smaller than the standard deviations of the respective class samples [26]. This condition has to be satisfied in order to start the training procedure. If this is not the case prototypes are re-initialized by assigning the larger number of prototypes to the classes that didn't satisfy the condition. Finally, one might try all the possible combinations of numbers and perform the comparative classifications [26], but this method is formidably slow.

In this paper, we address the issues of prototype initialization and deciding their number and distribution among classes. In order to find the optimal number of prototypes that provides high classification accuracy we start with the single prototype per class and then let the algorithm decide when and where to add or remove prototypes. This way we avoid the aforementioned initialization issues. The details of the proposed approach are given in the following section.

III. METHODOLOGY

A. Basic Idea of the Algorithm

Adaptive LVQ is a modification that improves LVQ classification performance. It consists of adaptive removal of less useful or harmful prototypes and addition of new prototypes at locations that are potentially beneficial. The idea is to start with an initial, small, and equal number of prototypes per each class. Then, candidate prototypes are added where they can be the most beneficial and prototypes are removed when it results in decrease in classification error.

Instead of fixing the number of prototypes, the adaptive algorithm only sets a threshold B defining an upper bound on the number of LVQ prototypes. To balance the budget B, when the total number of prototypes reaches B new prototypes can be added only after some existing are deleted.

Adaptive LVQ consists of two methods that can be easily appended to an LVQ algorithm of choice: *LVQadd* adds prototypes where current LVQ is making many mistakes, while *LVQremove* removes prototypes if it can result in decrease in classification error. *LVQremove* and *LVQadd* are performed in this order after each pass of the baseline LVQ algorithm through the training data.

B. LVQadd

LVQadd concentrates on misclassified points of each class during LVQ training. Using hierarchical clustering [22, 13] with average linkage, the significant size clusters of misclassified points are determined for each class. Assuming there are M classes, and there are n_j misclassified points of class j, we apply hierarchical clustering on each of these subsets. We can control the number of clusters for each class by setting two bounds: an upper bound, *max_clusters*, on the number of clusters, and a lower bound, *min_cluster_size*, on the cluster size. Then, new prototypes are introduced at positions of cluster centroids.

Since we are working on a budget (B), all new prototypes of all classes are grouped together and sorted by their size and only prototypes from the largest clusters are added to the budget. When the total number of prototypes reaches B, *LVQadd* is suspended until the complementary *LVQremove* does not remove some of them. Default values for *max_clusters* is 5 and for *min_cluster_size* is 3.

C. LVQremove

LVQ-Remove detects prototypes whose removal would result in accuracy increase. Such prototypes are typically outliers or those trapped in the regions with majority of the training data points of different class. During each pass of LVQ through training data statistics are gathered about every prototype and then combined into a unique score. For each prototype m_j its score is measured as

$$Score_j = A_j - B_j + C_j \qquad (3)$$

where A_j counts how many times prototype m_j classified correctly and hasn't been moved, B_j how many times it was moved away as the prototype of the wrong class and C_j how many times it was moved towards as the prototype of the correct class. Prototypes with negative scores are likely to be detrimental to accuracy and as a result they are removed in the *LVQremove* stage of the algorithm.

The *Score* is slightly modified only for the LVQ1 algorithm, where A_j does not exist, so it is set to zero. When Adaptive LVQ is applied to other LVQ algorithms such as LVQ2, LVQ2.1, LVQ3, LFM, LFMW, all three statistics used in *Score* can be positive.

D. The Algorithm

The goal of classification is to build an efficient classifier from N training points, $D = \{(x_i, y_i), i = 1... N\}$, where x_i is a K-dimensional vector in the feature space and y_i is its class label, an integer from a set $\{1…M\}$. We start by defining a LVQ classifier by an initial set of p_0 prototypes by randomly choosing p_0/M prototypes per each class label $\{(m_j, c_j), j = 1… p_0\}$, where m_j is a K-dimensional vector in the feature space, and c_j is its class label. The default value is set to $p_0 = M$.

Next, we define an upper bound B on how many prototypes we can have in total, as well as the initial value of the LVQ hyperparameter α_0. *LVQadd* parameters *max_clusters* and *min_cluster_size* are set to their default values. LVQ training rounds are repeated I times, unless early stopping occurs when training set accuracy stops improving. *LVQremove* and *LVQadd* are performed after each pass of LVQ through the training data. Before the next training round, the training data are shuffled.

Figure 1 describes the complete Adaptive LVQ2.1 algorithm. Pseudo codes for *LVQadd* and *LVQremove* procedures are described in Figure 2. It should be noted that both *LVQremove* and *LVQadd* are not performed after the last training round of LVQ. This is done because of the nature of *LVQadd* that might be adding prototypes in the noisy regions where they are not needed. Adaptive LVQ can effectively determine which class needs more prototypes and which less. The default choice for the initial number of

Input: Training Set of size N, M classes, Budget B, $p_0=M$, $\alpha_0=0.08$, $\alpha_T=4N$ (update step), $s=0.6$, $I=30$, significant error rate $e_{sig}=10^{-5}$, *max_clusters*=5, *min_cluster_size*=3

```
α=α_0;
t=1;
it=1;
Initialize prototypes
P=p_0;

WHILE (it ≠ I) and (err_it-err_it-1<e_sig)
   FOR i = 1 TO N
      find nearest prototype (m_j,c_j)
      find second nearest prototype (m_k,c_k)
      IF (c_j ≠ y_i) and (c_k = y_i)
         d_j=d(x_i,m_j);
         d_k=d(x_i,m_k);
         IF (min(d_j/d_k, d_k/d_j) > s)
            //update prototypes
            m_j(t+1) = m_j(t) – α(t) (x_i – m_j(t)) ;
            B_j++ ;
            m_k(t+1) = m_k(t) + α(t) (x_i – m_k(t)) ;
            C_k++ ;
         END
      ELSE A_j++ ;
      END
      t++;
      α(t)=α_0*α_T/(α_T+t);
   END
   Calculate classification error err_it
   IF (err_it<err_it-1)
      store current prototypes as Final_Prototypes
   IF NOT Last Iteration
      LVQremove
      LVQadd
   END
   Shuffle data for next iteration
   it++;
END
Calculate test error using Final_Prototypes
```

Figure 1. Pseudo code for Adaptive LVQ2.1 algorithm

prototypes is a single prototype per class, $p_0=M$, but it could be higher if desired.

When computational time and memory for classification is not an issue, budget B can be set to infinity. Then, the *LVQadd* and *LVQremove* of the adaptive algorithm determine the appropriate number of prototypes for each class that maximizes classification accuracy.

When working on a tight budget Adaptive LVQ is a very useful tool as it tries to optimally utilize the available prototypes for classification. Adaptively adding and removing prototypes while working on a budget is much more efficient than using the entire budget right away and spread the prototypes to all classes evenly.

E. Time Complexity

Time complexity of each LVQ round is $\Theta(N \cdot P \cdot K)$. It should be noted that in the Adaptive version P increases from initial p_0 to some final value which can be B at

Procedure **LVQremove**

```
FOR j = 1 TO $P$
    $Score_j=A_j-B_j+C_j$;
END
Find prototypes with negative scores;
Remove prototypes with negative scores;
FOR j = 1 TO $P$
    $A_j=0$;
    $B_j=0$;
    $C_j=0$;
END
```

Procedure **LVQadd**

```
FOR $j$ = 1 TO $M$
  $U_j=\{(x_i,y_i), y_i=j\}$ // Misclassified points in class $j$
  $C_j$= ClusterData($U_j$, $max\_clusters$, $min\_cluster\_size$)
    // $C_j$ – centroids found for class $j$
END
$AllCentorids=\{C_1,C_2,\ldots,C_M\}$;
Sort $AllCentroids$ by cluster size

WHILE P<B
    Remove the first centoid form $AllCentorids$
      and add it to the existing prototypes
    $P$++;
END
```

Figure 2. Pseudo code for *LVQadd* and *LVQremove*

maximum, making the total cost of the LVQ part of the algorithm O($N\cdot P\cdot K$).

The proposed *LVQadd* and *LVQremove* increase the total time only slightly. Since the clustering is performed only on the misclassified data points, the time complexity involved in *LVQadd* follows from the complexity of hierarchical clustering with average linkage and can be calculated as

$$\Theta(\sum_{i=1}^{M} n_i^2 \log n_i) \tag{4}$$

where n_i is the number of misclassified data points with class label i. As the majority of total N data points is already well classified after a single LVQ iteration, *LVQadd* typically uses significantly less training points than N and its time complexity is significantly less than $\Theta(N^2 logN)$. Moreover, after each training round, the fraction of misclassified points is expected to decrease, thus further decreasing the cost of *LVQAdd*.

LVQremove gathers prototype statistics during the LVQ training round and then it simply eliminates the prototypes with the negative score. Therefore, it always scales linearly with the number of prototypes and does not introduce a noticeable increase in the total training time.

IV. Experimental Results

The original LVQ algorithms LVQ1, LVQ2.1, and LVQ3 with k-means prototype initialization were compared to their Adaptive counterparts that start from a single prototype per class and use *LVQAdd* and *LVQRemove* steps to optimize the number of prototypes and their locations. In addition, these algorithms were compared to the popular Soft LVQ (its annealing version) algorithm presented in [24]. We intend to show that our algorithm, that uses simple heuristics, outperforms the basic LVQ algorithms and that it is comparable and, in noisy data cases, even better than the optimization-oriented Soft LVQ algorithm.

Two different sets of experiments are performed. Experimental design and details involved in implementation for both sets of experiments are described in the following sections.

Data Sets	Training Size	Test Size	Attributes	*M*
Adult	21048	9114	123	2
Banana	4300	1000	2	2
Gauss	20000	20000	2	2
IJCNN	49990	91701	22	2
Letter	16000	4000	16	26
Pendigits	7494	3498	16	10
Shuttle	42603	14167	9	7
USPS	7291	2007	256	10
Checker	8000	10000	2	2
NChecker	8000	10000	2	2
Synthetic	250	1000	2	2

Table 1. Data Set summaries

A. Data Description

We evaluated Adaptive LVQ algorithm on several benchmark classification datasets from UCI ML Repository as well as on several synthetic data sets. Data set properties are summarized in Table 1. The experiments are performed on both binary and multi-class data sets. The digits data sets *Pendigits* and *USPS* are 10-class datasets of handwritten digits collected from different writers, where each class represent a handwritten digit from 0 to 9. *Letter* data set is a 26-class dataset containing black-and-white rectangular pixel displays of each of the 26 capital letters in the English alphabet in several different fonts. *Shuttle* data set is a 7-class data set, where approximately 80% of the data belongs to class 1. Therefore the default accuracy is about 80%. *IJCNN* is also a highly unbalanced data set. It consists of 2 classes, where 90% of all data belongs to class 1. *Adult* data set contains records of more than 30,000 individuals, each represented with 123 attributes. Prediction task is to determine whether a person makes over 50K a year. *Checkerboard* data set was generated as a uniformly distributed two-dimensional 4x4 checkerboard with alternating class assignments (see Figure 3). *Checkerboard* is a noise-free data set in the sense that each box consists exclusively of examples from a single class. *N-Checkerboard* is a noisy version of *Checkerboard* where class assignment was switched for 15% of the randomly selected examples. For both data sets we used a noise-free *Checkerboard* as a test set. This way a highest reachable accuracy for both *Checkerboard* and *N-Checkerboard* was

100%. *Gauss* data set is a synthetic 2-class data set generated as two Gaussians of different class overlapping in 2D space. *Synthetic* data set, taken from [7] (Figure 4), as well as *Banana* data set are 2-class data sets of two different distributions overlapping in 2D.

B. Classification with an Optimal Number of Prototypes

In the first set of experiments, the proposed Adaptive LVQ algorithms with a large budget *B* were employed to try to maximize accuracy, and, as a by-product, to determine the optimal number of prototypes for each data set. Then, we trained the original LVQ algorithms using the same number of prototypes as the Adaptive LVQ, where k-means was used for prototype initialization (separate clustering was performed for each class).

LVQ algorithms used in these experiments were LVQ1, LVQ2.1 and LVQ3. LVQ1 differs from LVQ2.1 in a way that each time we visit a training point only the nearest prototype is updated by moving it towards (away from) the training point if their class labels match (differ). LVQ3 differs from LVQ2.1 by an additional update rule for the two nearest prototypes if both of them have the same class label as the training point. They are both moved slightly towards the training point, controlled by parameter ε which is scaling the learning rate α.

Both Adaptive LVQ and original LVQ versions used the same hyperparameters α_0=0.08 and update step α_T=4*N*. In the case of LVQ2.1 and LVQ3, the window size was set to s=0.6 and in the case of LVQ3 ε was set to 0.1. They also used the same number of training rounds *I*=30 and the stopping criterion equal to the error rate of $e_{sig}=10^{-5}$. Initial number of prototypes used in Adaptive LVQ algorithms was $p_0=M$, and the *LVQadd* used in all data sets *max_clusters*=5 and *min_cluster_size*=3. The upper bound *B* was set to infinity.

All experiments were repeated 10 times and the average and standard deviations of both the accuracy on the test set and the number of prototypes are reported in Table 2.

As can be seen from the results, Adaptive LVQ achieves very high accuracy on all data sets. More importantly, it successfully determines the appropriate number of prototypes. In most cases, this number is very stable over 10 repetitions. For example, in *Gauss* and *Checkerboard* data sets, where we knew in advance that the best choice of prototypes is 2 and 16, respectively, Adaptive LVQ in most cases found this to be the best choice as well. Among the three alternative original LVQ algorithms, Adaptive LVQ2.1 performed best on average, better than Adaptive LVQ1 and even Adaptive LVQ3 despite the heuristic correction rule supporting the divergence of prototypes.

Finally, when we compare the performance of Adaptive LVQ versions to the performance of original LVQ versions with k-means initialization, dominance of Adaptive versions can be recognized in all cases. Most significant differences in the classification accuracy can be observed on *Ijcnn*, *Gauss*, *Letter* and *N-Checkerboard* data sets. Also, it can be noticed that Adaptive LVQ2.1 achieves the biggest improvement over its original counterpart LVQ2.1.

C. Classification on a Budget

The second set of experiments was performed by restricting the algorithms to a very tight budget. This was done in order to illustrate the behavior of the proposed algorithm in the highly resource constrained applications.

We compared 4 different algorithms, LVQ2.1 with random prototype initialization (Regular LVQ2.1), LVQ2.1 with k-means prototype initialization (K-Means LVQ2.1), our Adaptive LVQ2.1, and annealing version of Soft LVQ. The hyperparameters of LVQ2.1 were set to the same values as in IV.B. Original LVQ2.1 and Soft LVQ algorithms used the same number of prototypes per class, *B*/*M*. The Adaptive LVQ2.1 started with a single prototype per class.

In Soft LVQ, the prototypes θ_l of class *l* were initialized to cluster centroids μ_l of training data and by adding noise $\theta_l=\mu_l$

DATA SET	**Adaptive LVQ1**		**K-Means LVQ1**		**Adaptive LVQ2.1**		**K-Means LVQ2.1**		**Adaptive LVQ3**		**K-Means LVQ3**	
	Accuracy	P	Accuracy	P	Accuracy	P	Accuracy	P	Accuracy	P	Accuracy	P
Adult	83.83 (0.21)	50.2 (3.9)	81.81 (0.31)	50	83.51 (0.35)	74.2 (5.3)	77.96 (0.70)	76	81.81 (0.46)	48.3 (11)	81.51 (1.03)	50
Banana	89.51 (0.49)	27.9 (2.7)	87.58 (0.24)	28	88.69 (0.96)	29.5 (2.2)	86.2 (0.56)	30	86.67 (2.27)	21.9 (4.1)	85.94 (1.64)	22
Gauss	91.92 (0.05)	5.5 (2.5)	68.93 (0.11)	6	91.59 (0.46)	4 (2.7)	86.08 (1.53)	4	91.89 (0.11)	6.2 (3.9)	85.80 (1.04)	6
IJCNN	91.05 (0.59)	74.4 (15.7)	74.90 (2.00)	76	98.21 (0.31)	83.5 (8.1)	77.93 (7.63)	84	92.01 (0.97)	12.3 (4.1)	86.36 (2.79)	12
Letter	87.35 (0.49)	350.3 (18.3)	84.84 (0.42)	364	87.33 (0.52)	271.7 (2.7)	81.80 (0.61)	286	81.85 (0.91)	289.2 (2.5)	79.31 (0.78)	312
Pendigits	95.80 (0.30)	109 (8.4)	94.11 (1.13)	110	96.86 (0.26)	34.5 (2.9)	91.08 (1.07)	40	95.62 (0.62)	112.5 (17)	93.36 (1.06)	120
Shuttle	98.07 (0.27)	29.3 (2.9)	96.76 (0.39)	35	99.74 (0.06)	18.8 (3.4)	96.31 (0.06)	21	97.21 (1.02)	13.7 (3.4)	88.49 (0.10)	14
USPS	92.56 (0.36)	207.6 (11.5)	91.99 (0.55)	210	92.31 (0.31)	77.2 (5.2)	90.69 (0.36)	80	91.96 (0.49)	208.8 (19)	91.40 (0.34)	210
Synthetic	88.74 (1.66)	9.7 (2.2)	85.82 (2.42)	10	88.98 (1.29)	9.1 (1.3)	85.57 (2.31)	10	88.61 (1.70)	9.3 (1.8)	86.68 (1.90)	10
CheckB	95.69 (1.58)	17.4 (1.5)	92.18 (0.64)	18	98.63 (0.28)	22.9 (5.4)	92.99 (0.94)	24	97.67 (0.36)	16.8 (1.4)	97.12 (0.59)	18
N-CheckB	93.24 (0.92)	18.8 (4.2)	89.56 (1.83)	20	97.52 (0.46)	21.8 (1.2)	88.26 (2.11)	22	95.53 (0.62)	17.1 (1.6)	73.83 (4.94)	18

Table 2. Performance comparison of Adaptive LVQ and LVQ algorithms with k-means initialization after 30 iterations based on the average classification accuracy and its standard deviation presented in percents

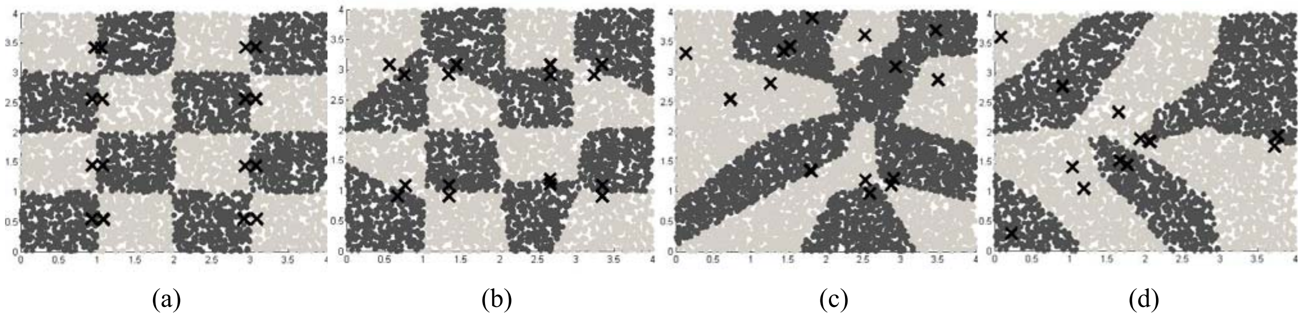

Figure 3. Classification results on the *Checkerboard* data set. The images represent the Test Set results when using the prototypes obtained during the training procedure after 30 iterations of (a) Adaptive LVQ2.1, (b) K-means LVQ2.1, (c) Regular LVQ2.1, (d) Soft LVQ. Dark gray represents class 1, light gray represents class 2 and X's represent the prototype positions

$+0.02\cdot\zeta\cdot\sigma_l\cdot\xi$, where σ_l is the vector of standard deviations of training points with label l along the coordinate axes, ζ is the number of prototypes per class, and $\xi\in[-1\ 1]$ is a number drawn randomly from a uniform distribution. Initial value of α hyperparameter used by Soft LVQ was α_0=0.08 and its update step was α_T=4N. The hyperparamether σ used by Soft LVQ was initialized as the minimal value of within-class variance of all classes and its update step was set to σ_T=6N. Annealing of σ was performed using the schedule $\sigma=\sigma_0\cdot\sigma_T/(\sigma_T+t)$. Learning was terminated after I=30 iterations or when the error improved by less than $e_{sig}=10^{-5}$.

Depending on the data set at hand we decided on the assigned budget. In most datasets we decided on a very tight budget of 20 prototypes, except in the multi-class data sets *Letter* and *Shuttle* where we assigned budgets of 52 and 21 prototypes, respectively. Exceptions were also *Gauss* and *Synthetic* data set where we used the budget of 10 prototypes as well as the *Checkerboard* and *N-Checkerboard* data sets where we assigned a budget of 16 prototypes, as that should be enough to correctly classify all of its 16 cells.

All experiments were repeated 10 times and the average and standard deviations of the accuracy on the test set for all four algorithms are reported in Table 3. The results for *Checkerboard* data set classification are presented both through visualization (Figure 3) and classification accuracy.

Looking at the results in the table it can be seen that Adaptive LVQ2.1 consistently performs better than the Regular LVQ2.1 and K-Means LVQ2.1. Most significant differences can be observed in *Letter*, *Ijcnn*, *Pendigits*, *Checkerboard* and *N-Checkerboard* data sets. When compared to the annealing version of Soft LVQ, Adaptive LVQ2.1 algorithm, in most cases, performs slightly better, except on *Checkerdoard*, *N-Checkerboard* and *Adult* data sets where more noticeable improvements can be observed, and in *Ijcnn* and *Banana* data sets, where it performs slightly worse.

On average Adaptive LVQ2.1 provides classification accuracy of 92.88%. When we compare this to 86.92% (Soft LVQ), 84.87% (K-Means LVQ2.1) and 75.74% (Regular LVQ2.1) we can conclude that our method, on average, achieved considerable improvement in classification accuracy.

V. Conclusion

In this paper we addressed some of the problems that people usually encounter when using LVQ algorithms for classification. After a thorough study of LVQ related work we realized that there has been little progress in resolving the problem of correct initialization, determining optimal number of prototypes, and their distribution among classes. As a result, we proposed a new LVQ algorithm which allows addition and removal of prototypes in an adaptive fashion

ALGORITHM	**Adult M=2**	**Letter M=26**	**Usps M=10**	**Shuttle M=7**	**Ijcnn M=2**	**Banana M=2**	**Pendigits M=10**	**CheckerB M=2**	**N-CheckerB M=2**	**Gauss Data M=2**	**Synthetic Data M=2**
	B_{20}	B_{52}	B_{20}	B_{21}	B_{20}	B_{20}	B_{20}	B_{16}	B_{16}	B_{10}	B_{10}
Regular LVQ 2.1	78.36 (0.90)	61.37 (0.36)	70.05 (3.66)	88.11 (4.95)	56.36 (5.78)	80.99 (4.66)	89.49 (0.10)	69.31 (6.35)	66.30 (5.09)	87.65 (3.40)	85.19 (1.42)
K-Means LVQ 2.1	78.96 (0.29)	61.16 (0.36)	87.49 (0.18)	96.31 (0.06)	75.13 (8.95)	85.88 (1.71)	89.50 (0.11)	92.61 (0.80)	92.89 (0.03)	88.16 (1.85)	85.51 (2.48)
Adaptive LVQ 2.1	83.42 (0.29)	84.59 (0.44)	92.40 (0.42)	99.72 (0.03)	97.89 (0.69)	88.83 (1.38)	96.51 (0.32)	99.43 (0.92)	98.21 (0.27)	91.84 (0.17)	88.94 (1.30)
Soft LVQ	81.15 (0.76)	84.04 (0.25)	91.32 (0.51)	99.41 (0.08)	98.22 (0.06)	89.75 (0.71)	96.12 (0.11)	80.72 (3.20)	58.67 (11.05)	88.11 (0.41)	88.69 (0.19)

Table 3. Performance comparison of Adaptive LVQ2.1, K-means LVQ2.1, Regular LVQ2.1 and Soft LVQ algorithms after 30 iterations based on the average classification accuracy and its standard deviation presented in percents

during the training process. The experimental results showed that our algorithm is very successful. It significantly improves the accuracy of original LVQ algorithms and it is very successful in guiding the allocation of prototypes in learning scenarios with tight budgets.

REFERENCES

[1] A. Cataron and R. Andonie, Energy generalized LVQ with relevance factors. *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 1421-1426, 2004.

[2] A.K. Qin and P.N. Suganthan, A Novel Kernel Prototype-Based Learning Algorithm, *Proceedings of the 17th International Conference on Pattern Recognition*, pp. 621-624, Vol. 4, 2004.

[3] A.K. Qin and P.N. Suganthan, Initialization Insensitive LVQ Algorithm Based on Cost-Function Adaptation, *Pattern Recognition Journal,* pp. 773-776, 2005.

[4] A. Sato, K. Yamada, Generalized learning vector quantization, *Advances in NIPS, MIT Press*, pp. 423-429, Vol. 7, 1995.

[5] B. Hammer, M. Strickert, T. Villmann, Supervised neural gas with general similarity measure, *Neural Processing Letters*, pp. 21-44, 2005.

[6] B. Hammer and T. Villmann. Generalized Relevance Learning Vector Quantization. *Neural Networks*, pp. 1059-1068, 2002.

[7] B.D Ripley, Pattern Recognition and Neural Networks, *Cambridge University Press ISBN* 0 521 46986 7, 1996.

[8] D. Miller, A. Rao, K. Rose, and A. Gersho, A global optimization technique for statistical classifier design, *IEEE Transactions on Signal Processing*, pp. 3108-3121, 1996.

[9] E. McDermott, S. Katagiri, Prototype-based minimum classification error/generalized probabilistic descent training for various speech units, *Computer Speech and Language*, pp. 351-368, Vol. 8, 1994.

[10] F.-M. Schleif, B. Hammer, and Th. Villmann. Margin based Active Learning for LVQ Networks. In *Proc. of ESANN*, pp. 539-545, 2006.

[11] H. Iwamida, S. Katagiri, E. McDermott, and Y. Tohkura, A Hybrid Speech Recognition System using HMMS with an LVQ-Trained Codebook, *ICASSP*, pp. 489-492, 1990

[12] J. Makhoul, S. Roucos, and H. Gish, Vector quantization in speech coding, *Proc. IEEE*, pp. 1551-1588, vol. 73, 1985.

[13] J. Seo, B. Shneiderman, Interactively exploring hierarchical clustering results, *IEEE Computer*, pp. 80-86, Vol. 35, 2002.

[14] K. Crammer, R. Gilad-Bachrach, A.Navot, and A.Tishby, Margin analysis of the LVQ algorithm, *Proc. 17th Conference on Neural Information Processing Systems*, 2002.

[15] K.L. Wu and M.S Yang, Alternative learning vector quantization, *Pattern Recognition Journal*, pp. 351-362, Vol. 39, 2006.

[16] L. Buckingham and S. Geva, Lvq is a maximum margin algorithm. In *Pacific Knowledge Acquisition Workshop PKAW'2000*, 2000.

[17] M. Biehl, A. Ghosh, B. Hammer, Dynamics and Generalization Ability of LVQ Algorithms, *The Journal of Machine Learning Research*, pp. 323-360, Vol. 8, 2007.

[18] M. Pregenzer, G. Pfurtscheller, D. Flotzinger, Automated feature selection with a distinction sensitive learning vector quantizer, *Neurocomputing*, pp. 19-29, Vol. 11, 1996.

[19] M. Strickert, U. Seiffert, N. Sreenivasulu, W. Weschke, T. Villmann, B. Hammer, Generalized Relevance LVQ (GRLVQ) with Correlation Measures for Gene Expression Analysis, *Neurocomputing* , pp. 651-659, Vol. 69, 2006.

[20] N.B. Karayiannis, M. M. Randolph-Gips, Soft learning vector quantization and clustering algorithms based on mean-type aggregation operators, *Int. J. Fuzzy Syst.*, pp. 90-94, Vol. 4, 2002.

[21] Neural Networks Research Centre Helsinki Univ. of Tech., Bibliography on the Self-Organizing Map (SOM) and Learning Vector Quantization (LVQ), http://liinwww.ira.uka.de/bibliography/Neural /SOM.LVQ.html, 2002.

[22] R. O. Duda, and P.E. Hart, Pattern classification and scene analysis, *John Wiley & Sons*, 1973.

[23] S. Kaski, Bankruptcy analysis with self-organizing maps in learning metrics, *IEEE Transactions on Neural Networks*, pp. 936-947, Vol. 12, 2001.

[24] S. Seo and K. Obermayer. Soft learning vector quantization, *Neural Computation*, pp. 1589-1604, Vol. 15, 2003.

[25] T. Kohonen, Improved versions of learning vector quantiztion, *International Joint Conference on Neural Networks*, pp. 545-550, Vol. 1, 1990.

[26] T. Kohonen, J. Kangas, J. Laaksoonen, and K. Torkolla, LVQ-PAK Learning vector quantization program package, *Lab. Comput. Inform. Sci. Rakentajanaukio, Finland*, *Tech. Rep. 2C*, 1992.

[27] T. Kohonen, The Self-organizing Map, *Proceedings of the IEEE*, pp. 1464-1480, 1990.

[28] T. Komori, and S. Katagiri, Application of a probabilistic descent method to dynamic time warping-based speech recognition. *In IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 497-500, 1994

[29] V.N. Vapnik. The Nature of Statistical Learning Theory, *Springer-Verlag*, 1995.

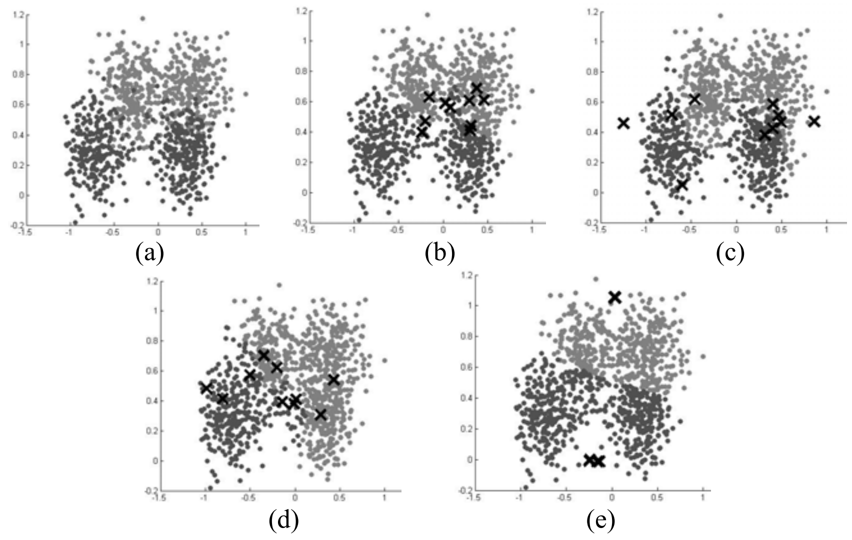

Figure 4. Classification results on the *Synthetic* data set. The images represent the Test Set (a) and results when using the prototypes obtained during the training procedure after 30 iterations of (b) Adaptive LVQ2.1, (c) Regular LVQ2.1, (d) K-means LVQ2.1 and (e) Soft LVQ. Dark gray represents class 1, light gray represents class 2 and X's represent the prototype positions