PointMap: A real-time memory-based learning system with on-line and post-training pruning

Norbert Kopčo and Gail A. Carpenter

Department of Cognitive and Neural Systems, Boston University
Boston, Massachusetts USA
kopco@cns.bu.edu, gail@cns.bu.edu

Abstract. A memory-based learning system called PointMap is a simple and computationally efficient extension of Condensed Nearest Neighbor that allows the user to limit the number of exemplars stored during incremental learning. PointMap evaluates the information value of coding nodes during training, and uses this index to prune uninformative nodes either on-line or after training. These pruning methods allow the user to control both *a priori* code size and sensitivity to detail in the training data, as well as to determine the code size necessary for accurate performance on a given data set. Coding and pruning computations are local in space, with only the nearest coded neighbor available for comparison with the input; and in time, with only the current input available during coding. Pruning helps solve common problems of traditional memory-based learning systems: large memory requirements, their accompanying slow on-line computations, and sensitivity to noise. PointMap copes with the curse of dimensionality by considering multiple nearest neighbors during testing without increasing the complexity of the training process or the stored code. The performance of PointMap is compared to that of a group of sixteen nearest-neighbor systems on benchmark problems.

Keywords: memory-based learning, nearest neighbor, on-line pruning, post-training pruning, incremental learning

1 Introduction

The nearest neighbor (NN) algorithm (Cover and Hart, 1967) is the classic *memory-based learning system*. During training, NN forms a *code* by memorizing all inputs and their associated outputs. During testing, NN finds, for a given unknown input, the most similar input(s) stored in the code, and assigns the unknown input to the class of at least one of the chosen exemplars. The simple nearest-neighbor strategy often produces accurate predictions, but may suffer from large memory and computation requirements and sensitivity to noise (Dasarathy, 1991). Methods that have been developed to mitigate these shortcomings include pruning, which eliminates some instances from the code, and prototype generation, which replaces several instances by an abstract representation of important features (Brighton and Mellish, 2002; Wilson and Martinez, 2000). Instance pruning can be incremental, starting with an empty code and adding useful nodes; or decremental, starting with a complete code and removing useless nodes. A given pruning method may favor points at the border of decision regions, at the center, or in between (Lam, Keung, and Liu, 2002).

Many NN rules perform batch learning, requiring access to the whole input set throughout training. Applications often require on-line sequential learning in real time. That is, a system is presented with one training item at a time and does not relearn from scratch each time a new exemplar is arrives. After a number of years in which batch training was a default feature of many learning models, the advantages of on-line training have become the focus of much recent research (e.g., Cauwenberghs and Poggio, 2001; Wilson and Martinez, 2003).

The Condensed Nearest Neighbor (CNN) algorithm (Hart, 1968) is the basic NN algorithm with on-line incremental learning. CNN processes training exemplars sequentially, adding to its code only the exemplars misclassified by their stored nearest neighbor. This can lead to considerable reductions in the size of stored code. When training CNN on noise-free data, the accuracy of the learned code tends to improve as the number of presented input patterns grows, as illustrated in Figure 1a. However, this gain in performance is at the expense of an increase in the number of stored patterns (Figure 1b). An incremental learning rule improves performance without additional memory requirements.

To reduce the stored memory, CNN finds a *consistent subset* (Hart, 1968) of the training data set, i.e., a subset that classifies the remaining data correctly with the nearest neighbor rule. Many (often non-

incremental) methods for finding the consistent subset have been proposed and shown to work well, in particular on noise-free data. See Toussaint (2002) for a review. However, when data are corrupted by noise, many of the noisy data points are by definition a part of the consistent subset. This weakens the ability of consistent-subset algorithms to reduce the size of the stored code, and might lead to worse performance than on the whole training set. Editing rules (e.g., Wilson, 1972) were developed to eliminate only the noisy data from the training set. However, these rules are usually batch-mode, nonincremental, require excessive memory and computation, and produce small reductions in code size.

In this paper we propose a new incremental learning rule for memory-based learning systems. The rule is introduced as an enhancement of the CNN algorithm that satisfies two requirements: 1) on-line learning leads to improved performance without increasing required memory and 2) learning is resistant to noise in the data. In Section 2, a learning algorithm called *PointMap* is introduced to exemplify application of the new learning rule in the context of the CNN algorithm. Like CNN, PointMap adds a new exemplar to the code only in response to a predictive error. The code size is limited in a straightforward way in PointMap: before training starts the user arbitrarily sets a maximum code size. During learning, once the code reaches the maximum, the algorithm first removes one of the stored exemplars before adding a new one. To determine the best candidate for pruning, PointMap stores each exemplar into a node which keeps track of several statistics of the exemplar, such as its predictive accuracy. Based on these statistics the *information value* of each node is computed and the least informative node removed. The information value is updated only for the nearest neighbor, keeping PointMap's computational and memory complexity comparable to the CNN. Final information values may also be used to select nodes for post-training pruning. Both on-line and post-training pruning can help a learning system find an informative subset of coding exemplars among noisy or nonstationary data.

Two sets of simulations are presented that illustrate PointMap behavior. In Section 3, with on-line data sets, PointMap is shown to achieve its computational goals on an example of both noise-free and noisy data. These simulations also illustrate how parameter choice influences PointMap dynamics. In Section 4, with batch data set, PointMap's performance is measured on two benchmark examples and compared to sixteen other NN classifiers. In general, performance of classifiers is evaluated in terms of classification accuracy vs. memory requirements and computational complexity. With PointMap, the user determines the amount of memory used, and classification accuracy depends on the chosen maximum code size. To illustrate this dependence, most simulations were performed repeatedly as the code size parameter varied.

2 PointMap algorithm

PointMap is a memory-based learning system in the family of algorithms that include k-nearest-neighbor (k-NN) (Cover and Hart, 1967), Condensed Nearest Neighbor (CNN) (Hart, 1968), IB2 (Aha, Kibler, and Albert, 1991), and Grow and Learn (Alpaydin, 1997). The memory of each of these algorithms is represented as a set of *coding nodes* which store input / output vector pairs from the training set. A PointMap node also records the number of times it has been chosen as the nearest neighbor of a new input, the number of times it has made a correct prediction, and the number of times it has been deemed critical to the decision. A node that makes a correct prediction is defined as critical with respect to a given training input if the same system without that node would have made an incorrect prediction. These statistics are combined to estimate the information value of each node.

The PointMap training algorithm includes condensed nearest-neighbor selection of the coding node, updating of the information value of the chosen node, and on-line and post-training pruning of nodes with low information value. During testing, PointMap performs a standard k-nearest-neighbor search, assigning each test input to the output class of the largest number of nodes among its k nearest neighbors. Note that some PointMap components described here are not necessary for satisfactory performance, but they illustrate additional system capabilities. In a *default* version, PointMap's information value is based only on predictive accuracy, without post-training pruning, and with single-nearest-neighbor testing. With this setting, the system requires setting of only one free parameter (the maximum code size).

A Matlab implementation of the PointMap code and a sample demo are available at: `http://cns.bu.edu/~pointmap`.

2.1 Condensed nearest neighbor coding

The CNN algorithm performs an on-line search for a minimal predictive subset of the training data set. During training, the algorithm sequentially checks whether an input would be classified correctly by the

Table 1 PointMap variables

Description	Parameter
Training set index	$p = 1 \ldots P$
Current input vector	$\mathbf{I}=\mathbf{I}_p \equiv (\mathbf{I}_{p1} \ldots \mathbf{I}_{pi} \ldots \mathbf{I}_{pM})$
Correct output class for the current input	$O = O_p$
Code index (stored coding nodes)	$j = 1 \ldots C$
Code vector j	$\mathbf{w}_j \equiv (w_{j1} \ldots w_{jM})$
Output class associated with coding node j	Ω_j
Index of the coding node closest to the current input	J
Index of the next closest coding node	J^{next}
Index of coding node for pruning	J^{prune}
Number of inputs for which coding node j won	α_j
Number of inputs for which coding node j won and made a correct prediction	β_j
Number of inputs for which coding node j won, made a correct prediction, and was critical	χ_j
Information value of coding node j	δ_j
Index set of the k nearest neighbors during testing	$\lambda \in \Lambda$

Table 2 PointMap parameters

Description	Parameter	Default value
Criticality parameter	$\gamma \in [0,1]$	0
Maximum code size	$C_{max} \in [1,\infty]$	
Fraction of nodes to be retained after post-training pruning	$\theta \in [0,1]$	1
Number of nearest neighbors used for testing	$k \in [1,C]$	1

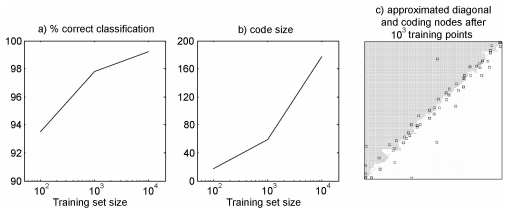

Figure 1. CNN algorithm simulations on the DIAGONAL data set. The training set consists of points uniformly distributed in the unit square, each labeled as lying above or below the diagonal. (a) Test set accuracy as a function of training set size. (b) Number of coding nodes as a function of training set size. (c) Test set response pattern (dark-above / light-below) after training with 10^3 inputs. Squares show locations of the 58 coding points.

candidate nearest neighbor in the current coding set. If yes, the algorithm may proceed to update the statistics of the candidate node, but otherwise does not alter the stored memory. If the prediction is incorrect, the current input is added to the coded memory. Thus, only the input vectors that cause predictive error are stored. Compared to the full training set stored by the *k*-NN algorithm, the reduction in the size of the coding set produced by CNN can be dramatic.

2.2 Information value of coding nodes

In PointMap, the *information value*, or contribution of each node to the overall accuracy of the system, is estimated during learning. A node's information value could, in general, depend on a variety of characteristics such as predictive accuracy, frequency of winning, or distance from a decision boundary. Pilot studies identified two consistently useful performance measures, *criticality* and *predictive accuracy*, to define the information value of PointMap coding nodes. These measures feature simple computations and intuitive functional interpretations. Criticality tends to favor coding nodes that delineate decision boundaries, while predictive accuracy tends to favor nodes that do not encode noise. Updating of these values requires only local computations at the chosen nearest neighbor node, with values updated for only one node per input presentation.

A node is defined as *critical* for a given input if its absence would have caused the system to produce an incorrect prediction. The *criticality fraction* of node j is defined as the ratio of the number of times (χ_j) this node has been critical to correct predictions divided by the number of times (β_j) it has won and made correct

predictions. The *predictive accuracy fraction* of node j is defined as β_j divided by the number of times (α_j) this node has won the nearest-neighbor competition. The information value (δ_j) of a newly added node equals 0. Thereafter, δ_j is defined as the convex combination:

$$\delta_j = (1-\gamma)\frac{\beta_j + 0.5}{\alpha_j + 1} + \gamma\frac{\chi_j}{\beta_j + 1}$$

where the *criticality parameter* $\gamma \in [0,1]$ indicates the contribution that the criticality fraction makes to the information value. The values in the denominators of the equation were increased by 1 to avoid division by 0 for a newly created node. The constant 0.5 is added to the numerator of the predictive accuracy fraction to prevent nodes that happen to make early correct predictions from immediately overwhelming those that produce some initial errors. Pruning decisions are thereby made on the basis of a longer-term average of predictive success than would be the case in the absence of this additional term.

As will be illustrated later (Figure 4), the information value factor admits an intuitive geometric interpretation: the predictive accuracy fraction tends to be highest for nodes that are far from the estimated decision boundaries because nearby inputs tend to be from the same output class. In contrast, the criticality fraction tends to be highest for nodes near a decision boundary, where elimination of a chosen node is most likely to produce a different prediction. Thus the value of the criticality parameter γ often determines the average distance between coding nodes and decision boundaries. Setting $\gamma = 0$ allows predictive accuracy alone to determine the coding set, which therefore tends to lie away from the decision boundary. Setting $\gamma = 1$ produces coding sets which are clustered near decision boundaries for noise-free data.

With noise-free training data, a wide range of γ values can produce reasonable results. However, a noisy node surrounded by ones from the correct category tends to have a high criticality fraction, so $\gamma = 1$ would be a poor parameter choice. Setting $\gamma = 0$, and thus choosing nodes based on predictive accuracy alone, generally works well on both noise-free and noisy data. However, this parameter might produce a code representing one small portion of input space where exemplars of one output class happen to be dense.

The simulation examples below indicate that setting the criticality parameter $\gamma = 0$ (i.e., considering only predictive accuracy) is a reasonable *a priori* choice. The simulations also show that including some criticality factor may improve performance somewhat. The optimum value of γ for a given problem would be estimated by cross-validation.

2.3 On-line pruning

Even without rate-limiting restrictions on memory storage capacity, limiting code size produces benefits for incremental learning systems. First, larger codes require more computation, a factor which may prove critical for real-time performance of large systems. Second, elimination of noisy nodes leads to improved classification accuracy. Third, the code-size maximum provides the user with a free parameter to balance sensitivity to detail against generalization. Requiring the replacement of nodes that become uninformative also allows the system to produce a "concept shift" (Kuh et al.,1991; Salganicoff, 1997) by relearning the entire stored code, if necessary, in a nonstationary environment. Finally, when, as in the PointMap algorithm, node statistics are updated only when the node wins, a smaller code produces more robust estimates by allowing each stored node to win more often.

Many methods could be used to limit the code size. For example, the system might prune only poorly performing nodes without limiting their total number. Or, if 10% of the training data are known to be noisy, a pruning strategy could eliminate one node for every ten added to the code. In order to serve as a general-purpose classifier. PointMap specifies a user-defined hard code size limit, C_{max}. During training, a new input exemplar is added when its nearest neighbor in the code fails to predict the correct output class. Once the size of the stored code reaches C_{max}, the least informative node is pruned before the new exemplar is added.

A straightforward rule for bounding the memory size of any on-line learner is simply to halt training once the number of stored exemplars reaches a designated maximum. Although this strategy renders the system incapable of adapting to useful new information, at least system performance does not deteriorate over time. Any worthwhile strategy to limit memory size in an on-line learning setting needs to improve upon this baseline procedure. That is, test set performance should continue to improve with further training after the code size has reached its limit. Initial simulations with PointMap showed that meeting this natural constraint is surprisingly nontrivial because forced on-line pruning can easily lead to replacement of a historically informative node by one that will prove to be less informative in the future. One way to mitigate this problem is to bias the system to choose recently added nodes for pruning.

In PointMap, the information value of coding nodes lies between 0 and 1. When a new node is added, its

information value is set equal to 0. This initial value implies that a recently added node is almost always the first eliminated to make room for a new node as soon as the next incorrect prediction is made. This elimination occurs even if the recent node was not involved in making the error. In order to remain in the code for long, a new node must quickly prove useful to other inputs. This strategy helps stabilize an existing code.

2.4 Post-training pruning

At any point during on-line learning, a subset of stored nodes with the smallest information values could be pruned from the coding set. Simulations in Section 3 demonstrate that post-training pruning on the basis of final information values can be useful in some situations. For example, the value of C_{max} (set before training) might turn out to have been larger than the number of nodes needed for accurate classification of a given data set. Also, as discussed in the previous section, the most recently added node often encodes noise. In Section 4.1, a simulation on a small data set shows that post-training pruning of this one node can significantly improve performance. Simulations in Section 3.2 show how post-training pruning can substantially improve test set performance with noisy data when the information value is based on both predictive accuracy and criticality ($\gamma = 0.5$). In this case, many nodes in the noisiest regions may be stored during on-line learning. Since these nodes usually have relatively low predictive accuracy, they tend to be eliminated first in post-training pruning.

2.5 *k*-nearest neighbor testing

Nearest neighbor classifiers often consider more than one nearest neighbor when determining the class of an unknown input. This strategy makes the system less sensitive to noise, because considering more neighbors provides more accurate estimates of the class probabilities at the position of the unknown input in the feature space. In PointMap, the nodes retained by the system are typically the ones from the class with the highest density at a given point in the feature space. Therefore, in most situations it suffices to consider a single nearest neighbor during PointMap training and testing.

An exception to this rule occurs when inputs are high-dimensional. In this situation, memory-based learning systems typically need to store many points to produce sufficiently dense coverage of the space, even though some of the dimensions may be irrelevant to outcome predictions. Although this *curse of dimensionality* (Cybenko, Saarinen, Gray, Wu, and Khrabrov, 1997) also affects PointMap, the problem can be alleviated by using multiple nearest neighbors to predict outcomes during testing. This strategy improves performance because each of the neighbors serves as an independent noisy version of the current input. Voting among the neighbors is like comparing the input to the average of the neighbors from each category, with averaging effectively eliminating noise in irrelevant dimensions. Simulation in Section 4.2 gives an example of a high-dimensional data set, showing that setting the number of neighbors to $k = C_{max}/10$ can be a good *a priori* choice in this situation.

2.6 PointMap training algorithm

PointMap is trained on P input-output pairs $(\mathbf{I}_1,O_1)$, $(\mathbf{I}_2,O_2)$, …, $(\mathbf{I}_p,O_p)$. The input vector $\mathbf{I}_p$ has M components $(\mathbf{I}_{p1}\ldots\mathbf{I}_{pi}\ldots\mathbf{I}_{pM})$, and the integer value O_p represents the output class to which $\mathbf{I}_p$ belongs. The stored code consists of C input-output pairs $(\mathbf{w}_1,\Omega_1)$, …, $(\mathbf{w}_C, \Omega_C)$.

Table 1 lists the variables used in the PointMap algorithm. In addition, four parameters are determined by the user (Table 2). Three of these parameters influence PointMap training: criticality γ biases the system toward choosing coding nodes that are farther from ($\gamma = 0$) or closer to ($\gamma = 1$) decision boundaries; maximum code size C_{max} sets an upper bound on the size of the coding set; and a post-training pruning fraction θ determines how many coding nodes with low information values are discarded before testing. For testing, parameter k specifies the number of neighbors that vote on the output prediction. As illustrated in the following sections, PointMap performs well with parameters fixed to the default values shown in Table 2. Thus, the only user-defined parameter that remains sensitive to its selected value is the maximum code size, C_{max}.

Individual steps of PointMap training are defined as follows. For a Matlab implementation of the PointMap code and a sample demo, see: `http://cns.bu.edu/~pointmap`.

Step 1: Code the first input

Set $p = C = 1;\ \mathbf{w}_1 = \mathbf{I}_1;\ \Omega_1 = O_1;\ \alpha_1 = \beta_1 = \chi_1 = \delta_1 = 0$

Step 2: Present a new input Vector **I** denotes the current training exemplar, and O is its output class.

Increase p by 1

Set $\mathbf{I} = \mathbf{I}_p;\ O = O_p$

Step 3: Choose a candidate coding node J The algorithm searches the stored code for the nearest neighbor of **I** using the L_1 (city-block) metric.

$J = \arg \min_{1 \le j \le C} \left| \mathbf{I} - \mathbf{w}_j \right|$, where $\left| \mathbf{I} - \mathbf{w}_j \right| \equiv \sum_{i=1}^{M} \left| I - w_{ji} \right|$, with ties broken in favor of the smallest index.

Step 4: Update the information value of the candidate node J

Update the number of times α_J that node J has won the nearest-neighbor search:

Increase α_J by 1

If $O_J = O$, update the number of times β_J that node J has produced a correct prediction and the number of times χ_J that node J has been critical to the correct prediction:

Increase β_J by 1

Let $J^{next} = \arg \min_{\substack{1 \le j \le C \\ j \ne J}} \left| \mathbf{I} - \mathbf{w}_j \right|$

If $O_{J^{next}} \ne O$ (or if $C = 1$), increase χ_J by 1

Recompute the information value δ_J of the candidate node J:

$$\delta_J = (1 - \gamma) \frac{\beta_J + 0.5}{\alpha_J + 1} + \gamma \frac{\chi_J}{\beta_J + 1}$$

Step 5: If node J **has made the correct prediction, go to the next training item**

If $O_J = O$, go to Step 7

Step 6: If J **has made an incorrect prediction, add a new coding node**

If $C = C_{max}$, then eliminate the stored node J^{prune} with the smallest information value:

$J^{prune} = \arg \min_{1 \le j \le C_{max}} \delta_j$ (In case of a tie, choose the smallest index.)

For $j = \left(J^{prune} + 1 \right) \ldots C_{max}$

$\mathbf{w}_{j-1} = \mathbf{w}_j;\ \Omega_{j-1} = \Omega_j;\ \alpha_{j-1} = \alpha_j;\ \beta_{j-1} = \beta_j;\ \chi_{j-1} = \chi_j;\ \delta_{j-1} = \delta_j$

Set $C = C_{max} - 1$

Initialize a new node that encodes the current input:

Increase C by 1

$\mathbf{w}_C = \mathbf{I};\ \Omega_C = O;\ \alpha_C = \beta_C = \chi_C = \delta_C = 0$

Step 7: End condition The algorithm performs a single pass through the training set.

If $p < P$, go to Step 2

Step 8: Post-training pruning Reduce the stored code to a fraction θ of its final on-line size.

Let $C_\theta = \theta * C$

While $C > C_\theta$, sequentially eliminate the stored nodes J^{prune} with the smallest information values:

$$J^{prune} = \arg \min_{1 \le j \le C} \delta_j$$

For $j = \left(J^{prune} + 1 \right) \ldots C$

$\mathbf{w}_{j-1} = \mathbf{w}_j$; $\Omega_{j-1} = \Omega_j$; $\delta_{j-1} = \delta_j$

Reduce C by 1

2.7 PointMap testing algorithm

The predicted output class of a test input is determined by a majority vote of its k nearest neighbors in the stored code. The algorithm performs no further learning or pruning, and does not use information values during testing.

Step 1: Present the new input

Let **I** be the test input.

Step 2: Identify the k nearest neighbors in the stored code

Let $\Lambda \subseteq \{1 \ldots C\}$ be the set of k coding indices such that $|\mathbf{I}\text{-}\mathbf{w}_\lambda| \le |\mathbf{I}\text{-}\mathbf{w}_\lambda|$ for all $\lambda \in \Lambda$ and $j \notin \Lambda$, with ties broken in favor of the smallest index.

Step 3: Predict the output class

The predicted output class is determined by a vote among the k nearest neighbors of the test input. That is, O is taken to be the class with the largest representation in the set $\{\Omega_\lambda : \lambda \in \Lambda\}$. Tie votes may be broken in favor of the smallest output class number or the output class of the nearest neighbor, or by weighting votes according to distances to the test input.

3 PointMap simulations of on-line data

The PointMap algorithm was tested on four simulation examples. The first two sets of simulations evaluate PointMap behavior when trained on on-line data. The goal of these simulations is to illustrate that PointMap achieves its design goals and to show how parameter settings influence performance. The first simulations (Section 3.1) examine PointMap behavior on the noise-free SPRING data set, which features a multi-scale decision boundary. The second simulations (Section 3.2) evaluate performance on the same task with a high degree of noise added to the training set. The remaining two simulations are discussed in Section 4.

3.1 SPRING simulations

The SPRING benchmark was constructed to test PointMap design goals. This section addresses the following questions on the noise-free version of the SPRING example.

Once the stored code has reached its size limit, does further training improve test performance?

Given a sufficiently large code size limit, can incremental training approach optimal performance?

Can post-training pruning be used to estimate how many coding nodes are necessary to accurately represent the data set? That is, if C_{max} is chosen to be larger than a minimum necessary for accurate performance, will post-training pruning be able to eliminate redundant nodes without loss of accuracy?

What is a good *a priori* value of the parameter γ to balance criticality vs. predictive accuracy?

3.1.1 SPRING data and simulation parameters

SPRING input points are uniformly distributed within the unit square, with points in the two output classes located to the left or right of a multi-scale zig-zag decision boundary (Figure 2). The following simulations examine system performance in on-line setting with up to 4×10^6 training exemplars, three code size limits

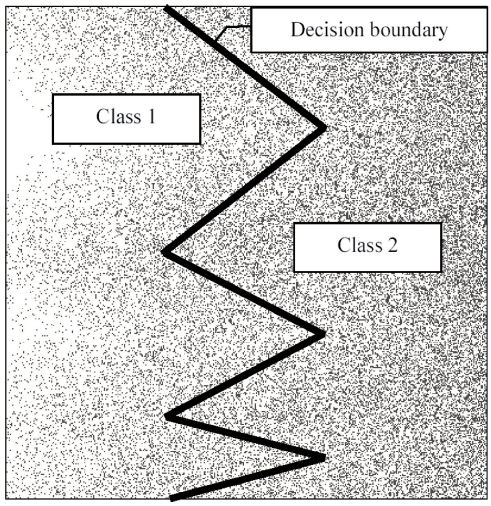

Figure 2. For the SPRING problem, a multi-scale zig-zag marks the ideal boundary between the two classes of points in the unit square. In the noisy version, the probability that a training set point has the wrong label is inversely proportional to its distance to the boundary. Points in the figure represent training exemplars from Class 2 in a noisy SPRING example.

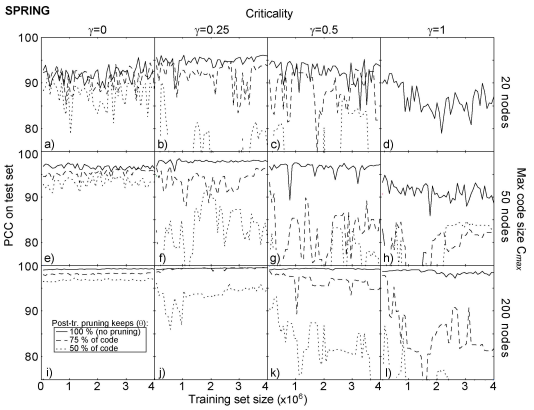

Figure 3. Simulations on the noise-free SPRING example illustrate the roles of the criticality parameter $\gamma = 0.0, 0.25, 0.5, 1.0$, of the maximum code size $C_{max} = 20, 50, 200$, and of the post-training pruning fraction θ=1.0, 0.75, 0.5. When $\gamma = 0$, criticality does not contribute to the information value, which is based on predictive accuracy alone; when $\gamma = 1$, the information value is based only on criticality. In each column, performance is seen to improve as the maximum code size increases. Within each panel, the solid line shows how test set performance varies with the number of training points; the dashed line shows test set performance by the 75% of nodes with highest information values at each stage of training; and the dotted line shows performance by the top 50% of trained nodes. Setting $\gamma = 0$ or 0.25 achieves the best results.

(C_{max} = 20, 50, 200), and four criticality parameters (γ = 0, 0.25, 0.5, 1). After each presentation of 8×10^4 training points, system performance was tested with three levels of post-training pruning (θ= 0.5 (…), 0.75 (---), 1.0 (—)) on a test set grid of 101 × 101 points. The nearest-neighbor parameter k was set equal to 1 during testing.

3.1.2 SPRING results

The solid lines in Figure 3 show how test set performance changes with additional training, after each code has reached its maximum size C_{max}, without post-training pruning (θ= 1). For each criticality value γ (columns), the correct prediction rate increases with C_{max}, reaching close to 100% when C_{max} = 200.

The rows of Figure 3, show that, for all code sizes, performance of the unpruned system is best when information value is based only on predictive accuracy (Figure 3a, 3e, 3i) or when criticality contributes 25% of the information value (Figure 3b, 3f, 3j). Moreover, for these values of γ, test set performance improves slightly with incremental learning for all values of C_{max}. In contrast, with γ= 1, incremental learning actually worsens performance (Figure 3d). In addition, the dashed line in Figure 3j (θ= 0.75) shows that, with C_{max} = 200, this system maintains test performance levels even after pruning the 50 coding nodes with lowest information values.

With γ=1, post-training pruning causes performance to deteriorate drastically. These simulations indicate that, when criticality alone determines the information value, both on-line and post-training pruning methods, which favor points closest to the decision boundary, cause over-fitting. In contrast, note that when predictive accuracy alone determines the information value (γ = 0), even the maximally pruned systems (θ= 0.5) maintain better performance than correspondingly pruned systems with γ > 0. Details of these simulations in the following section illustrate the mechanisms underlying these properties.

3.1.3 SPRING dynamics

Figure 4 illustrates PointMap dynamics for the simulations summarized in Figure 3. Here, each plot shows the decision regions from the beginning (after 8×10^4 inputs) and end (after 4×10^6 inputs) of training, as well as the locations of points in the stored code and the test-set percent correct. These graphs show that, although there is no explicit communication among nodes, on-line pruning helps to create an evenly distributed coding set: compare, for example, beginning and end node distributions in Figure 4f. The rows of Figure 4 also illustrate that the average distance from coding nodes to the decision boundary decreases as the criticality parameter γ increases. In the top row, where the code size is limited to $C_{max} = 20$ nodes, the system performs significantly better with small values of γ, which keeps coding nodes away from the decision boundary, although some contribution of criticality (Figure 4b) is better than none (Figure 4a).

Note that, even with γ fixed, the average distance between coding nodes and the decision boundary also decreases as the maximum code size C_{max} increases. When sufficiently many coding nodes are allowed, the system is able to distribute them near the decision boundary, to fine-tune accurate prediction across multiple scales. A comparison of Figures 4a and 4i shows that nodes cluster near the boundary even with γ=0, a setting which otherwise tends to place nodes far from the boundary. This clustering occurs because coding nodes are added only when a training exemplar makes a predictive error, which, once a code has been established, happens most often at points near the boundary. Figures 4a and 4e show that, as code size increases, larger-scale sections of the decision boundary become accurately delineated, while smaller-scale portions are still approximated. Even with the same number of coding nodes, setting γ=0.25 (Figure 4f) pulls the coding nodes toward the boundary, improving the approximation at smaller spatial scales.

Finally, Figure 4d indicates why setting $\gamma = 1$ produces a poor approximation to the decision boundary. Relying on criticality alone overly favors the selection of pairs of coding nodes that are near one another across the boundary, even when the code is small ($C_{max} = 20$). Each node has a high criticality factor when it makes a correct prediction for a training input, because its removal would produce the incorrect prediction. Tightly clustered pairs produce over-fitting because a small misalignment of the nodes can produce a large error in the approximated decision boundary. With γ=0 or 0.25 (Figure 4a and 4b), several pairs of coding nodes still appear on opposite sides of the boundary, but at some distance, producing a more stable approximation.

3.2 Noisy SPRING

To evaluate PointMap's ability to cope with noise, simulations were performed on an extremely noisy version of the SPRING example. Most memory-based learning systems such as *1*-NN or CNN that look only for the nearest neighbor either perform poorly on this type of example or generate a large code, or both. The performance of these classifiers can be improved by increasing the number of the nearest neighbors (k) making test set predictions. Although larger values of k allow a system to estimate class probabilities in the region around a test input, this computation increases the complexity of the algorithm and does not reduce the code size. Simulations below show that PointMap can find a good solution (95% correct) to the noisy SPRING problem with a relatively small coding set using only k=1 nearest neighbor during both training and testing.

3.2.1 Noisy SPRING data and simulation parameters

For the noisy SPRING example, output class labels were randomly swapped on a subset of the training set. The probability of a class swap was 50% for points at the decision boundary, with the swapping probability decreasing with distance between input points and the boundary. Figure 2 shows 10^5 training exemplars assigned to Class 2. Note that this is extreme level of noise corruption, chosen to examine system behavior under difficult conditions.

The following simulations examine PointMap performance with one code size limit ($C_{max} = 200$), two criticality parameters (γ=0, 0.5), and three levels of post-training pruning (θ= 0.1 (...), 0.5 (---), 1.0 (—)). As for the SPRING simulations, system performance was tested on a grid of 101×101 noise-free points after each presentation of 8×10^4 training points, with $k = 1$ for nearest-neighbor search. The simulations were also performed with $C_{max} = 20$ and 50, and with γ=0.25. Observed performance was much worse than in the simulations reported here.

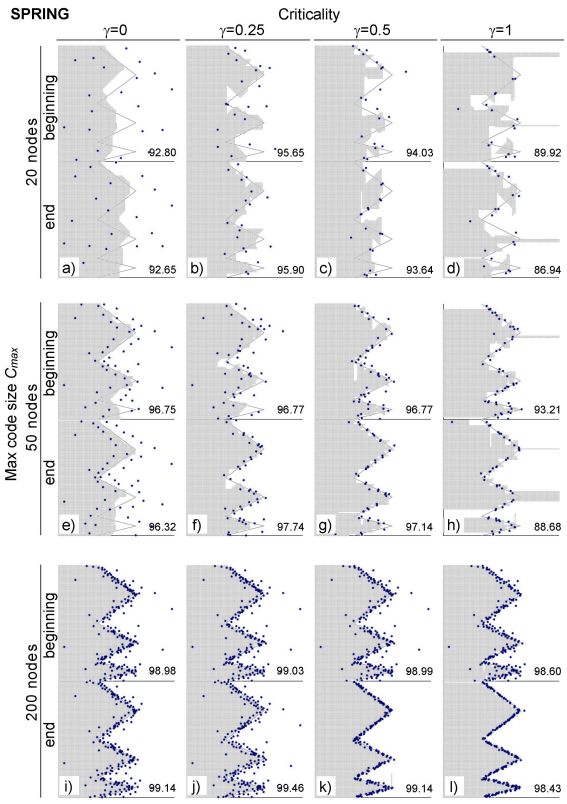

Figure 4 Initial and final SPRING coding node distributions, for simulations of Figure 3, without post-training pruning (θ=1). Each panel shows the predicted decision region and stored coding points after an early training phase (8×10^4 inputs) and at the end of the simulation (4×10^6 inputs). These simulations show that, once a code has achieved its maximum size, additional training does not automatically improve performance. In fact, at each memory size with γ=1, on-line pruning pulls stored points closer to the decision boundary, and these changes in the code training lead to a deterioration of test-set accuracy. In contrast, with γ=0 or 0.25, on-line pruning improves accuracy at each code size.

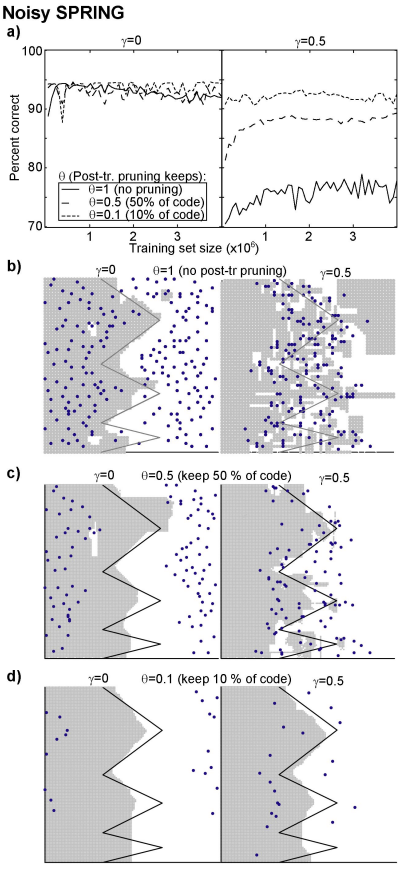

Figure 5. Noisy SPRING simulations with $C_{max}=200$. The information value in the left column is based on predictive accuracy alone (γ=0) and in the right column is equally weighted between predictive accuracy and criticality (γ=0.5). (a) System performance as a function of training set size, averaged across five simulations. (b) Final distribution of coding nodes and decision regions with no post-training pruning. (c) Same as (b), retaining 50% of the trained nodes. (d) Same as (b), retaining 10% of the trained nodes.

3.2.2 Noisy SPRING results

Figure 5 illustrates PointMap performance on the noisy SPRING example, with criticality parameter γ=0 for the left column and γ=0.5 for the right column. Figure 5a shows that, when the information value is based on predictive accuracy alone (γ=0), test set performance reaches about 95%, and remains in that range even when post-training pruning reduces the stored code from 200 to 20 nodes (θ= 0.1).

At first glance, this performance appears manifestly superior to that of the system with γ=0.5, especially without post-training pruning (θ= 1). Closer inspection reveals additional subtleties in this comparison. With γ=0, after an initial stable phase, performance steadily deteriorates with additional training. In contrast, with γ=0.5, performance steadily improves, especially with post-training pruning.

Figure 5b-d, which shows the final stored code and test set decision regions for each criticality setting and each degree of post-training pruning, provides insight into these dynamics. Coding nodes selected on the basis of their predictive accuracy alone (left column) tend to be located at some distance from the actual decision boundary. The bias toward placing the code as far as possible from the decision boundary is

balanced by the fact that a new node is added only in response to a predictive error. However, Figure 5c-d shows that it is the nodes located farthest from the decision boundary (which have information value close to 1) that remain after pruning. Although test set performance remains high for this particular example, the potential for over-generalization with γ=0 is visible where the pruned decision boundary fails to approximate the higher frequency portion of the spring. Slow migration of the code away from the decision boundary also explains the deteriorating performance in the course of on-line learning when criticality is not a factor in the information value.The right column of Figure 5b shows that giving equal weight to criticality and predictive accuracy (γ=0.5) clusters coding nodes in regions with a high concentration of noise, thus producing a poorly defined test set decision boundary. However, because information values are higher away from the actual decision boundary, test set performance of the pruned system steadily improves during training, reaching a level comparable to that of the best performance with γ=0. Moreover, pruning improves the geometry of the decision boundary approximation.

These simulations indicate that a problem characterized by a high degree of noise is best approached by a coding strategy that balances criticality and predictive accuracy during on-line training, followed by post-training pruning. However, validation set selection of the free parameters γ and θ may be time-consuming. The *a priori* parameter selection γ=0 and θ=1, which chooses nodes on the basis of predictive accuracyalone without post-training pruning, provides a quicker if less compact solution.

3.3 Discussion

The simulations presented in this section have shown that PointMap achieves its design goals on examples of noise-free and noisy data. The system is able to improve accuracy online while maintaining a constant memory size, performance is nearly optimum if sufficiently many nodes are available, and post-training pruning can further reduce code size. The simplest form of PointMap, with the information value based only on predictive accuracy, with no post-training pruning, and with single nearest-neighbor testing, realizes the primary design goals. This minimal system allows fast performance with the maximum code size as the single user-defined parameter. The simulations also show that PointMap can incrementally improve operation with parameters fixed at other values. Optimum values vary with the problem, and off-line methods are necessary to determine them.

4 PointMap simulations on benchmark data

The previous sections analyzed the properties of PointMap on synthetic data sets with the goal of exploring PointMap properties in on-line settings. The PointMap algorithm was also tested on two benchmark data sets, with performance compared to that of other memory-based learning systems that do not necessarily learn incrementally. Sections 4.1 and 4.2 examine PointMap performance on benchmark examples (WINE and LED) from the UCI repository of machine learning databases (Blake and Merz, 1998). PointMap results on the WINE and LED examples are compared to those of sixteen variations of the *3*-nearest-neighbors classifier, as analyzed by Wilson and Martinez (2000).

4.1 WINE simulations

The present section compares system performance on the WINE data set, which features a small training set. This example was chosen specifically to challenge PointMap, where large data sets would normally be expected to be needed to estimate coding node information values for pruning.

4.1.1 WINE data and simulation parameters

The WINE data set was obtained from chemical analysis of wines grown in a single region in Italy and derived from three vine varieties. The set consists of 178 13-dimensional input patterns, each belonging to one of three output classes. Although the classes are linearly separable, memory-based learning systems usually do not find the optimum solution because of the small number of available training inputs. PointMap simulations employed the same 10-fold cross validation design used in the Wilson-Martinez analysis. That is, each of ten partitions of the data set served, in turn, as a test set, with training on the remaining set of ~160 items and with results reporting averages across the ten trials.

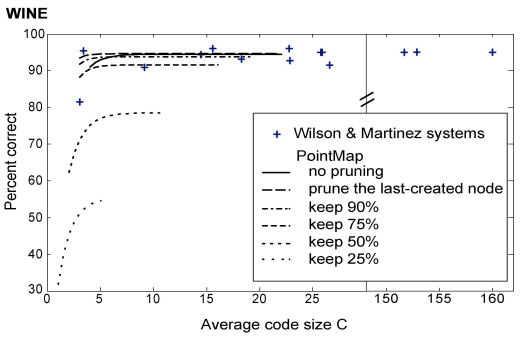

Figure 6. WINE simulations with γ=0.15. For each system, classification accuracy, as a function of memory size, is averaged across 10-fold cross validation trials. Crosses denote the sixteen *3*-NN classifiers reported by Wilson and Martinez (2000). PointMap results are plotted by exponential fit for the unpruned system (solid) curve, with progressively shorter dashs marking increasing levels of post-training pruning.

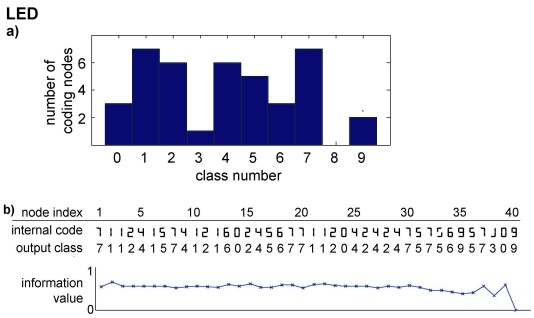

Figure 7. Preliminary PointMap simulation of the LED example with γ=0, C_{max} = 40, 100 training epochs, and no post-training pruning (θ=1). (a) Histogram of the number of coding nodes for each class at the end of training. (b) For e ach of the 40 stored nodes: its index (with recently created nodes having larger indices), its internal code (from input components #1-7), the class to which it is assigned, and the final estimate of its information value.

In PointMap simulations, each training set was presented in random orders for 500 epochs. Based on estimates obtained from a preliminary study, the PointMap criticality parameter was fixed at γ = 0.15 across all simulations. Testing employed *k*=1 nearest neighbor. Each learning trial was performed on 500 randomly selected training set exemplars. Typically, the stored code stabilized early, but later inputs improved estimation of information values, for post-training pruning. Simulations examined maximum code sizes C_{max} ranging from 3 to 40 and post-training pruning fractions θ = 0.25, 0.5, 0.75, 0.9, 1.0. An additional post-training pruning strategy that eliminated only the last-created node (similar to setting θ = 0.999) was also tested.

4.1.2 WINE results

Figure 6 plots system performance on the WINE example as a function of the average number of input vectors stored in memory. Plotted crosses summarize performance of the sixteen variations of the *3*-NN classifier reported by Wilson and Martinez. These systems produced only small variations in percent correct, averaging 93.5% and with all but one between 90.98% and 96.08%. However, the average sizes of the stored memories ranged from ~3 nodes (81.47% correct) to all the nodes for *3*-NN (94.93% correct).

The solid curve in Figure 6 plots an exponential fit of PointMap performance. Although maximum code sizes up to C_{max} = 40 were tested, actual code sizes never exceeded *C*=23. Simulations with larger C_{max} values thus used no on-line pruning (reducing PointMap to CNN), in which case result variations were caused only by differing orders of input presentation. Although PointMap uses only *k*=1 nearest neighbor during testing, performance is similar to that of the collective Wilson-Martinez systems at each code size. With pruning of the last-created node (widest dashes), PointMap performance improves further at small code sizes, with the system discovering a near-optimal solution with only *C*=3 coding nodes and maintaining performance with larger stored codes. The PointMap on-line pruning strategy is thus seen to succeed on this small data set.

Additional curves in Figure 6 show exponential fits of PointMap performance on WINE simulations following pruning that retains from 90% (dashes) to 25% (dots) of the trained code. These results indicate that post-training pruning causes rapid performance deterioration, even when many more nodes are retained than the minimum needed for optimal performance. This example therefore suggests an *a priori* strategy for small training sets that prunes online after reaching a small upper bound on code size and that then discards only the last node added during training. The parameter γ was fixed at 0.15 in the present simulation, different from the default value of 0. Preliminary simulations (not shown) produce nearly the same results with the default parameter settings.

4.2 LED simulations

The final simulations compare PointMap performance with that of the same sixteen systems as in Section 4.1, again as reported by Wilson and Martinez (2000). Compared to the WINE example, this LED benchmark has more input dimensions, output classes, and training exemplars, and intrinsic noise establishes an upper bound on test set performance. This benchmark is particularly suitable for evaluation of a system's ability to cope with the curse of dimensionality because 17 of its 24 input dimensions contain only noise.

4.2.1 LED data and simulation parameters

The LED task is to identify numbers 0,1,…,9 in a digital display. The first seven components of the binary input vector denote the presence or absence of a segment in an ordinary display. The task is rendered difficult by the presence of seventeen additional input components with randomly assigned binary values. In addition, values in the first seven components are flipped with a 10% probability. Because some flips can change one digit into another (e.g., **6** into **8**), the optimal Bayes classification rate is 74%. Each output class is represented by 1,000 input vectors.

In PointMap simulations, each training set was presented in random orders for 100 epochs. As in the Wilson-Martinez paradigm, performance represents averages from 10-fold cross validation. One simulation (Section 4.2.2) was performed with the default PointMap parameter setting ($\gamma = 0$, $k = 1$, no post-training pruning) and with $C_{max} = 40$ nodes. The remaining simulations (Section 4.2.3) set γ=0.25,. with the maximum code size C_{max} = 30, 100, 300, 1000, and 3000 nodes, with no post-training pruning, and with parameter k chosen by 10-fold cross-validation on the training set.

4.2.2 LED results: default PointMap parameters

The first LED simulation illustrates PointMap dynamics with the information value computation based solely on predictive accuracy (γ=0). Analysis of deficiencies in the resulting code point to potential importance of including criticality in the information value calculation.

In the simulation illustrated in Figure 7a, which shows the number of nodes stored for each class *0* … *9*, PointMap created an unbalanced code, with many nodes assigned to some classes (e.g., *1*, *7*) and few or none to others (e.g., *3*, *8*). This imbalance occurred because randomly flipped line segments have differential effects on different classes. For example, flipping one of the segments in the image **8** could change it into a noise-free representative of class *0*, *6*, or *9*, though still labeled as belonging to class *8*. This property lowers the predictive accuracy, and hence the information value, of nodes that correctly encode *8*. In this example, where the information value is based entirely on predictive accuracy, class *8* was unable to retain any coding nodes. Similarly, one flip could change an image **3** into **9**. Correspondingly class *3* retained only one of the 40 coding nodes, with that node (#38) storing a noisy image (Figure 7b). On the other hand, flipping any one segment in an image **2** would produce a non-digit, which is still likely to be chosen as the nearest neighbor to a test-set exemplar of class *2*. Exemplars for class *2* thus produced high predictive accuracy values and six coding nodes. With the total code restricted in size, such overrepresentation of some classes contributed indirectly to the high test set error rate of others.

Figure 7b shows details of the PointMap code at the end of this simulation. Beneath the index of each coding node is the LED image of its first seven stored components, its assigned output class, and its final information value. Note that the images of only two of the forty stored nodes (#33 and #38) were noisy. This rate is much below chance because the 10% segment flip probability implies that more than half of the input images are noisy. Figure 7b also shows that, except for the most recently added node (#40) retained nodes had similar information values.

With C=40 coding nodes, γ=0 and k=1, PointMap achieved classification accuracy of 49% on the test set. This result is better than the 41% achieved by a standard *1*-NN classifier, storing all 10,000 training inputs, even though PointMap stored only 40 training inputs and it could not classify correctly any testing points from the class *8*. Nonetheless, this accuracy rate is far below the optimal 74%.

4.2.3 LED results: $k > 1$, γ=0.25

All PointMap simulations described so far have based predictions on the output class of single nearest neighbors (k=1). The irrelevant input components #8-24 in the LED example introduce the curse of

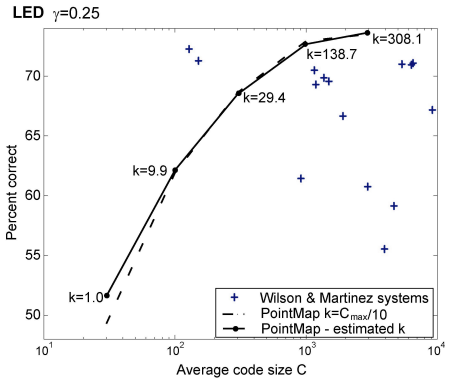

Figure 8. LED simulation with γ=0.25 and no post-training pruning. Crosses mark results from the sixteen 3-*NN* systems reported by Wilson and Martinez (2000). The solid line shows average PointMap performance for C_{max}= 30, 100, 300, 1,000, 3,000; and the number of test-set nearest neighbors k determined by 10-fold cross-validation on the training set. The dotted line shows PointMap results with $k = C_{max}/10$.

dimensionality (Cybenko, Saarinen, Gray, Wu, and Khrabrarov, 1994). As the LED simulations in Section 4.2.2 have illustrated, PointMap with k=1 does not produce satisfactory results under these circumstances. We will now see that higher values of k do produce near-optimal predictions. In these simulations, tie votes (which were rare) were broken in favor of the smallest output class number.

Crosses in Figure 8 represent results obtained by Wilson and Martinez on sixteen types of *3*-NN algorithms. The solid line in Figure 8 plots the average test-set accuracy achieved by PointMap for various values of the maximum code size C_{max}, with the number of nearest neighbors k chosen by 10-fold cross-validation on the training set. The dashed line, which shows that a rule-of-thumb that simply sets $k = C_{max}/10$ produces near-optimal results, indicates the robustness of this parameter choice. Figure 8 shows that PointMap with these settings can achieve nearly optimum performance on the LED data set.

4.2.4 Discussion

LED simulations with default parameters illustrate that the simple PointMap strategy of limiting code size by incrementally selecting nodes with good predictive accuracy improves performance compared to a standard 1-NN or CNN. However, these simulations also illustrate that the resulting code can be unbalanced if the noise in the data has specific characteristics.

The LED simulations with k greater than one illustrate that the strategy of using multiple neighbors can be effective in coping with high dimensional data, if sufficiently many nodes are retained. In the simulation, 10-33% of the training set had to be retained to achieve a near-optimum performance with k proportional to 10% of the code size. LED simulations were performed with γ=0.25. Again, nearly the same results were observed with γ=0.

Note that PointMap selects test-set values of k which are high compared to typical values used with memory-based systems, which are usually less than k=10 (Alpaydin, 1997). The need for large k values might be due to the fact that PointMap uses more than one nearest neighbor only during testing. Not considering k neighbors during training has the advantage of keeping learning fast and simple, while achieving good results. Methods like k-CNN (Gates, 1972) could also be used in this context.

5 Conclusion

This study has introduced a simple and efficient method for limiting code size during on-line learning. Pruning is based on a locally computed measure of the information value of coded items. This value balances an index of criticality, which is high near decision bourndaries, against an index of predictive accuracy, which is high away from these boundaries. The degree of criticality in the information value is determined by a parameter $\gamma \in [0,1]$.

Once a code has reached its designated limit, the stored item with the lowest information value is discarded before a new node is added to memory. Satisfactory performance usually requires γ to be less than 0.5. Setting γ=0, which equates the information value with predictive accuracy alone, produces near-optimal results on a variety of problems. Although larger values of γ appear to produce improved performance with limited training, an over-emphasis on criticality in node selection tends eventually to produce over-fitting of noisy data.

After training, a fraction of nodes with lowest final information values may also be pruned. To create a code of a given size, setting a higher limit during incremental learning, then later applying post-training

pruning, is recommended for large-scale noisy problems. For small-scale problems, setting a lower code size limit is a better stragegy.

On-line pruning methods have here been tested on a memory-based system called PointMap, which is based on the Condensed Nearest Neighbor algorithm. These methods may also be applied to other types of learning systems, to limit code size while improving performance during on-line training of small-scale or large-scale stationary or non-stationary data. Design choices in the definition of PointMap were made to keep the algorithm simple. In addition to providing a new memory-based althorithm, the system thus serves as a template on which to evaluate incremental learning methods. Many alternative choices can be made when applying PointMap.

Acknowledgements

This research was supported by grants from the Air Force Office of Scientific Research (AFOSR F49620-98-1-0108, F49620-01-1-0397, and F49620-01-1-0423) and the Office of Naval Research (ONR N00014-01-1-0624).

References

Aha D.W., Kibler D., and Albert M.K. (1991), "Instance-based learning algorithms," *Machine Learning*, vol. 6, pp. 37-66.

Alpaydin E. (1997), "Voting over multiple condensed nearest neighbors," *Artificial Intelligence Review*, vol. 11, pp. 115-132.

Blake C.L. and Merz C.J. (1998), "UCI repository of machine learning databases," [`http://www.ics.uci.edu/~mlearn/MLRepository.html`], University of California, Department of Information and Computer Science, Irvine.

Brighton H. and Mellish C. (2002), "Advances in instance selection for instance-based learning algorithms," *Data Mining and Knowledge Discovery*, vol. 6, pp. 153-172.

Cauwenberghs, G. and Poggio, T. (2001), "Incremental and decremental support vector machine learning," *Advances in Neural Information Processing Systems*, vol. 13, MIT Press, Cambridge, pp. 409-415.

Cover T.M. and Hart P.E. (1967), "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, pp. 21-27.

Cybenko G.V., Saarinen S., Gray R., Wu Y., and Khrabrov A. (1997), "On the effectiveness of memory-based methods in machine learning," *Dealing with complexity: A neural networks approach,* M. Karny, K. Warwick, V. Kurkova, and J.G. Taylor, Editors, Springer Verlag, New York, pp. 62-75.

Dasarathy B.V. (1991), *Nearest neighbor (NN) norms: NN pattern classification techniques*. IEEE Computer Society Press, Los Alamitos.

Gates, G.W. (1972), "The reduced nearest neighbor rule," *IEEE Trans on Info Theory* vol. 18, pp. 431–433.

Hart P.E. (1968), "The condensed nearest neighbor rule," *IEEE Trans on Info Theory*, vol. 14, pp. 515-516.

Kuh, A., Petsche, T., and Rivest, R.L. (1991), "Learning time-varying concepts," *Advances in Neural Information Processing Systems*, Morgan Kaufmann, pp 183-189.

Lam, W., Keung, C.-K., and Liu, D. (2002), "Discovering useful concept prototypes for classification based on filtering and abstraction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 1075-1090.

Salganicoff, M. (1997), "Tolerating concept and sampling shift in lazy learning using prediction error context switching," *Artificial Intelligence Review*, vol. 11, pp. 133-155.

Toussaint, G. (2002), "Proximity graphs for Nearest Neighbor decision rules: recent progress," *Proceedings of INTERFACE-2002, 34th Symposium on Computing and Statistics, Montreal, Canada.*

Wilson, D.L. (1972), "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 2, pp. 408-420.

Wilson, D.R. and Martinez, T.R. (2000), "Reduction techniques for instance-based learning algorithms," *Machine Learning*, vol. 38, pp. 257-286.

Wilson, D.R. and Martinez, T.R. (2003), "The general inefficiency of batch training for gradient descent learning," *Neural Networks*, vol. 16, pp. 1429-1451.