Appeared in *Proceedings of the Tenth Int. Conf. on Machine Learning*, pp. 314–321, Morgan Kaufmann, 1993.

Online Learning with Random Representations

Richard S. Sutton and Steven D. Whitehead
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
sutton@gte.com swhitehead@gte.com

Abstract

We consider the requirements of online learning—learning which must be done incrementally and in realtime, with the results of learning available soon after each new example is acquired. Despite the abundance of methods for learning from examples, there are few that can be used effectively for online learning, e.g., as components of reinforcement learning systems. Most of these few, including radial basis functions, CMACs, Kohonen's self-organizing maps, and those developed in this paper, share the same structure. All expand the original input representation into a higher dimensional representation in an unsupervised way, and then map that representation to the final answer using a relatively simple supervised learner, such as a perceptron or LMS rule. Such structures learn very rapidly and reliably, but have been thought either to scale poorly or to require extensive domain knowledge. To the contrary, some researchers (Rosenblatt, 1962; Gallant & Smith, 1987; Kanerva, 1988; Prager & Fallside, 1988) have argued that the expanded representation can be chosen largely *at random* with good results. The main contribution of this paper is to develop and test this hypothesis. We show that simple random-representation methods can perform as well as nearest-neighbor methods (while being more suited to online learning), and significantly better than backpropagation. We find that the size of the random representation does increase with the dimensionality of the problem, but not unreasonably so, and that the required size can be reduced substantially using unsupervised-learning techniques. Our results suggest that randomness has a useful role to play in online supervised learning and constructive induction.

1 Online Learning

Applications of supervised learning can be divided into two types: online and offline. By *offline* supervised learning we mean the classical task in machine learning and statistics: a set of examples is obtained and used to learn a good approximating function before the function is used in the application. In *online* learning, on the other hand, data gathered during the normal operation of the system is used to continually adapt the learned function.

Online learning has several advantages over offline learning. First, online learning is potentially more robust because errors or omissions in the training set can be corrected during operation. Second, training data can often be generated easily and in great quantities when a system is in operation, whereas it is usually scarce and precious before. Being able to use this data sometimes puts online learning at a great advantage; for example, this was probably the most important reason for the success of Tesauro's (1992) champion backgammon-playing program. Finally, online training is necessary in order to learn and track time-varying functions, to continue to adapt to a changing environment. In a broad sense, online learning is essential if we want to obtain *learning* systems as opposed to merely *learned* ones.

Online supervised learning methods are also needed as components of reinforcement learning systems, for example, to learn evaluation functions, policy functions, and action models. In reinforcement learning one learns by trial and error, so it is essential to learn online. In addition, reinforcement learning usually involves some form of temporal-difference learning, e.g., to handle the credit assignment problem involved in learning an evaluation function. Temporal-difference learning inherently involves non-stationarity because the targets for learning are themselves learned quantities that change over time.

What supervised learning methods are suited to online learning? The two most important requirements are

that the methods be *incremental* and able to handle *nonstationary* (time-varying) target functions.

We distinguish two degrees of incrementality. We say a method is *weakly incremental* if it requires relatively little additional memory and computation in order to process one additional example. Most classical machine learning and statistical methods do not meet this standard of incrementality. Methods such as ID3 (Quinlan, 1986), CART (Breiman et al., 1984), INDUCE (Michalski, 1983), and MARS (Friedman, 1988) all take a training set as a whole and perform a computationally intensive process that has to be repeated anew each time examples are added to the training set. Weakly incremental methods include extended decision tree methods such as ID4 (Schlimmer & Fisher, 1986) and ID5R (Utgoff, 1989), and nearest-neighbor and case-based learning methods. However, the memory and (per-example) computational requirements of all of these methods may increase without bound as more examples are seen,[1] which is problematic for long-lived online applications. We say a method is *strictly incremental* if its requirements for memory and (per-example) computation do not increase with the number of examples. Strictly incremental methods include STAGGER (Schlimmer & Granger, 1986) and most connectionist learning methods. Many weakly incremental methods can be converted into strictly incremental variants, e.g., by retaining only a limited number of examples. Such modifications usually involve some loss of the methods' simplicity, ease of use, and theoretical assurances, but are nevertheless an interesting direction of research (e.g., see Moore & Atkeson (1992), Schwartz (1993)).

The situation with regard to nonstationarity is very similar. Weakly incremental methods do not naturally handle time-varying target functions, but most can be adapted to do so. Many strictly incremental methods, on the other hand, are well suited to nonstationary problems. The approach we take in this paper is to explore a particular class of connectionist method, described in the next section, that is both strictly incremental and able to handle nonstationarity. As a challenging benchmark, we present results comparing our approach to nearest-neighbor methods on tasks well suited to nearest-neighbor methods.

Multilayer connectionist methods based on gradient descent, such as backpropagation, are typically strictly incremental and able to handle nonstationarity, but have met with mixed success in online applications. The primary problem here is that of "catastrophic interference," in which training on new examples interferes excessively with previously learned examples. This problem has been partially overcome in practice by storing old examples and retraining on them (Lin, 1992), or by learning slowly and training extensively (Tesauro, 1992), but it remains a significant limitation. We also present results comparing our approach to backpropagation in this paper.

[1] In a sense, the memory requirements of decision-tree methods are always bounded by the size of the most complex possible tree. However, as this bound is exponential in the dimensionality of the input space, it is far to high to be useful in practice.

2 Learning with Expanded Representations

The most effective and popular methods for online learning are radial-basis-function networks (e.g., Moody & Darken, 1989), CMAC networks (Albus, 1981), and closely related statistical methods such as Parzen windows (e.g., see Duda & Hart, 1973). Other important online methods are self-organizing maps (Kohonen, 1990), STAGGER (Schlimmer & Granger, 1986), and unsupervised classifiers such as COBWEB (Fisher, 1987), among many others. All of these methods are strictly incremental and able to handle nonstationarity. Moreover, all share a similar two-layer structure: The first layer expands the original input representation into a high dimensional feature space, and the second layer maps that expanded representation to the final answer. The first layer is non-linear and either fixed or learned by an unsupervised learning process, and the last layer is much simpler and is the only part affected by supervised training (see Figure 1). In a radial-basis-function network, for example, the first layer is composed of the radial basis functions, and the last layer is usually a linear mapping or a linear mapping followed by a threshold or sigmoid nonlinearity. In a self-organizing map and in COBWEB the first-layer re-representation is formed by an unsupervised process. This general architecture is the primary focus of this paper. We call it learning with an *expanded representation (ER)*.[2]

The first ER learning system was Rosenblatt's perceptron. Although ignored in most modern treatments, the original perceptron included a preprocessing layer of fixed, random, boolean functions (see also Uhr & Vossler, 1963; Klopf & Gose, 1969). More recently, Gallant and Smith (1987), Kanerva (1988), and Prager and Fallside (1988) have also advocated ER networks in which the first layer consists entirely of fixed, random functions of the original input. They proposed that the random functions be simply linear threshold functions with random weights. This approach may seem naive, but Kanerva's analytic arguments and the results by Gallant and Smith (1987) and Prager and Fallside (1988) suggest that it can perform well. We call ER systems in which the expanded representation

[2] Grouping and clustering methods such as Chapman & Kaelbling (1991), Mahadevan & Connell (1992), and Mahadevan (1992) can also be viewed as ER methods where the final layer is a lookup table rather than a linear mapping.

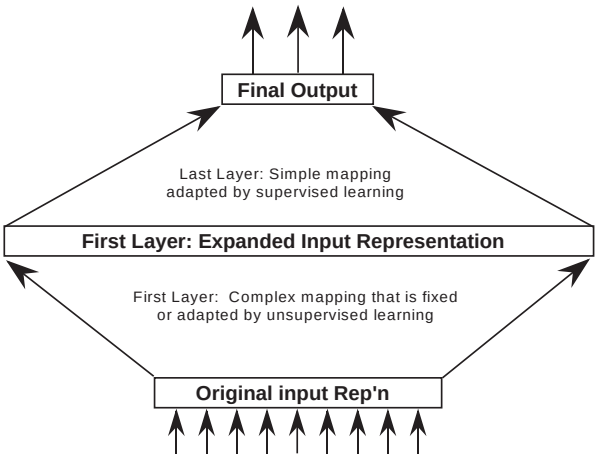

Figure 1: General architecture for learning with an Expanded Representation (ER).

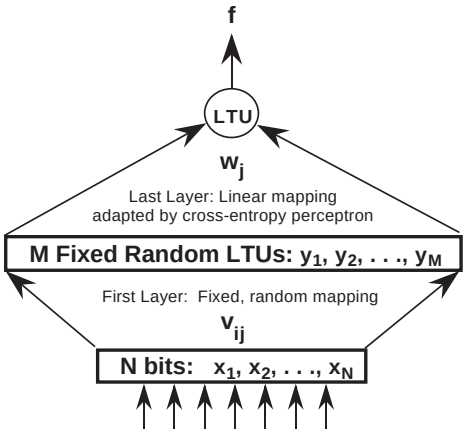

Figure 2: The basic Random Representation (RR) learning system used in this paper.

is constructed with a significant random component *random representation (RR)* systems.

In this paper, the performance of RR methods is measured along three co-varying dimensions: speed of learning, asymptotic error, and number of hidden units required (i.e., size of the random representation). If the number of hidden units is allowed to be arbitrarily large, then it is fairly easy to show that any static target function can be learned to arbitrary accuracy. A more significant result is suggested by Kanerva (1988), who concluded that the speed of learning also becomes very good as the number of hidden units increases, approaching that of nearest-neighbor methods. We present empirical results for a specific RR system that confirm this prediction.

The key question, however, is how many hidden units are needed to obtain desirable speed and accuracy properties? If the number required grows too rapidly with, say, the size of the original input space, or the complexity of the target function, then these methods would be impractical. Gallant and Smith (1987) showed that good results could be obtained with a few hundred hidden units on s set of tasks from the neural network literature such as 7-input parity and right-shift. Is a few hundred a lot? Certainly it is more than is used in typical connectionist networks, but it is no challenge to the memory capacity of today's computers, even conventional serial ones. As proponents of other memory-intensive learning methods have pointed out, memory is cheap; we should not be stingy with our hidden units. Nevertheless, we still seek to obtain the best performance possible with the fewest number of hidden units. We present several innovations that appear to improve this tradeoff for ER systems.

3 A Basic RR Network

In this section we describe a basic RR network based on linear threshold units (LTUs) which we evaluate and extend in subsequent experiments. This system is very close to that described by Gallant and Smith (1987).

The RR system is shown schematically in Figure 2, as an instance of the general architecture of Figure 1. There are n inputs denoted $x_1, x_2, \ldots, x_n$ ($x_i \in \{0, 1\}$) and m hidden units. The hidden units are LTUs, meaning that each unit j is connected to each input by a weight $v_{ij} \in \Re$, and its output y_j is given by

$$y_j = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} v_{ij} x_i \geq \theta_j ; \\ 0, & \text{else.} \end{cases} \qquad (1)$$

For now the weights v_{ij} will be fixed and chosen as either $+1$ or -1, with equal probability. Each unit has one input pattern to which it responds maximally, sometimes called its *prototype*, or address. The prototype for unit j is the input pattern with $x_i = 1$ for those i where $v_{ij} = +1$ and $x_i = 0$ where $v_{ij} = -1$. The threshold θ_j is chosen so that the unit will be active ($y_j = 1$) if slightly more than half of the input bits match the prototype, i.e.,

$$\theta_j = S_j^{min} + \beta n, \qquad (2)$$

where S_j^{min} is the minimum possible value for the jth unit's weighted sum (S_j^{min} is just minus the number of the unit's negative weights) and β is the proportion of the bits that have to match the prototype ($\beta = 0.6$ in all but one of the experiments reported in this paper).

The final unit of our basic RR network is also an LTU. Its output f is given by

$$f = \begin{cases} 1, & \text{if } \sum_{j=0}^{m} w_j y_j \geq 0 \\ 0, & \text{else,} \end{cases} \qquad (3)$$

where $w_j \in \Re$, $j = 1, \ldots, m$, are weights from each hidden unit to the final LTU. The 0th weight, w_0, acts as a negative threshold or bias; the corresponding input signal y_0 is defined always to be 1. All of the weights are initially zero and then adjusted according to a normalized, cross-entropy perceptron rule:

$$\Delta w_j = \frac{\alpha}{N}(f^* - p)y_j\,, \tag{4}$$

where $\alpha > 0$ is a learning-rate parameter ($\alpha = 1$ in all experiments reported in this paper), N is the number of hidden units that are active ($N = \sum_{k=0}^{m} y_k^2$), f^* is the target value for the current example, and p is the estimated probability that $f^* = 1$, given by

$$p = \frac{1}{1 + e^{-\sum_{j=0}^{m} w_j y_j}}. \tag{5}$$

This learning rule is a version of the perceptron suited to binary classification problems (Hinton, 1989). If the overall problem were instead one of minimizing squared error, then an LMS or normalized LMS rule would be more appropriate (see Section 7). This is an online learning procedure because the learning rule (4) is applied once to each example as it is received.

4 Performance vs Representation Size

The first experiment evaluated the learning performance of the basic RR network as a function of m, the size of the expanded representation in number of random LTUs. We designed the tasks to be suitable for solution by nearest-neighbor methods, so that these methods could serve as challenging benchmarks to the performance of the RR methods.

The tasks were generated randomly as follows. Each input consisted of 16 randomly generated bits, of which 8 were designated as relevant and 8 as irrelevant. For each task, 8 prototypical inputs were also generated and half assigned a target bit of 1 and half a target bit of 0. To determine the target for a randomly generated 16-bit input, it was compared to each of the prototypical inputs. The prototypical input that was closest in hamming distance (based only on the relevant bits) was selected and its target bit used as the target for the input. Ties were broken according to an arbitrary ordering of the prototypical inputs. Finally, noise was introduced by flipping the target bit on a random 5% of the examples.

The basic RR network was run on this problem with random representations ranging in size from $m = 25$ to $m = 5000$. We also ran a simple nearest-neighbor method called *1NN* on this problem. In this method, for each new input we first found its *nearest neighbor*, the previously experienced input closest to it in hamming distance (using all 16 bits). The guess for the new input was then the same as the target bit stored with the nearest neighbor. Ties were broken in favor of the most recent nearest neighbor. Finally, we also ran on this problem a single cross-entropy perceptron using the original input representation. The performance of this perceptron tells us the extent to which the problem is solvable by a linear method.

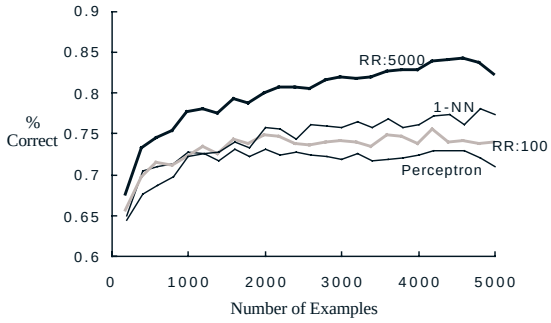

Figure 3: Learning curves on the first task.

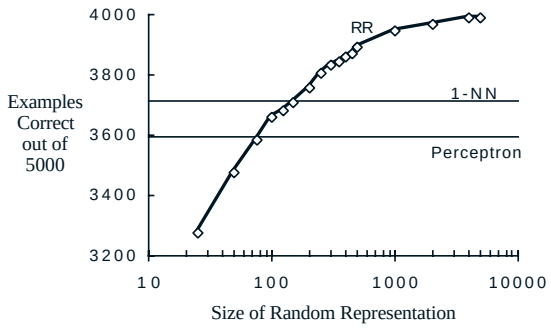

Figure 4: Performance vs. rep'n size on the first task.

Each algorithm was run for 5000 examples. The number of correct guesses by each algorithm was averaged over blocks of 200 examples and then over 20 repetitions of the whole procedure to obtain the learning curves shown in Figure 3. This figure shows the behavior of the RR algorithm with a small representation ($m = 100$) and with a large representation ($m = 5000$). In general, the performance of the method improves with larger representations. Figure 4 is a summary graph that plots the total number of examples guessed correctly by the RR methods as a function of representation size. The flat lines show the performance levels of the perceptron and of the 1NN method. The RR method performed better than both the perceptron and the 1NN method as long as the representation was larger than about 200 units.

Note that the performance of the RR method only improves as the number of hidden units increases. This behavior may be surprising to those accustoming to thinking that excessive memory results in "memorization" of the training examples and poor generalization to new ones. In all of our work with online training with RR systems we have never found this to be

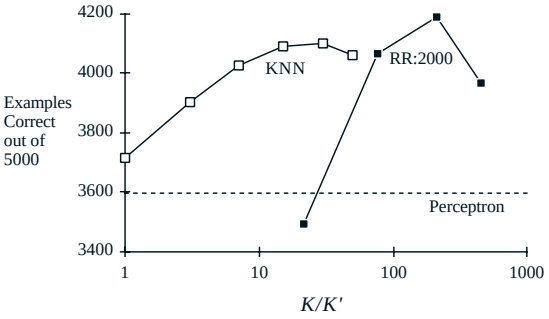

Figure 5: KNN vs. RR on the first task, varying the number of neighbors (KNN) or the average number of active hidden units (RR). For RR, the total number of hidden units was 2000.

a problem.[3] Accordingly, we have not sought ways of simplifying RR networks (e.g., removing little-used hidden units) in the hope of improving generalization.

The 1NN method performs relatively poorly on this task in part because the task is noisy and includes irrelevant input dimensions. In these cases, nearest-neighbor methods perform better if they are based not on the single nearest neighbor, but on the majority recommendation of several near neighbors. These are called K-nearest-neighbor (KNN) methods, where K is the number of neighbors considered. Figure 5 shows the total performance of a KNN method as a function of K. If K was chosen within the right range, then the KNN method performed better than the 1NN method.

The β parameter of the RR methods plays a role similar to that of K. β determines the average number of hidden units that respond to an input pattern, which we here call K'. The K' hidden units that respond together determine the network's output, much as the K nearest neighbors together determine the output of the KNN method. In fact, the effects of K and K' on the performance of the two method is very similar, as shown in Figure 5. Within the right range of their respective parameters, the two methods perform roughly equally as well.

5 Unsupervised Learning

The preceding experiments showed that randomly constructed features, if sufficiently numerous, can be useful in learning arbitrary functions. However, the method as presented so far is potentially inefficient. For example, depending on the input distribution, it is possible for many of the randomly-constructed LTUs to be always active or always inactive. Such units do not contribute to learning because they cannot discriminate between inputs. Additionally, units that are active too often can slow learning by causing excessive generalization between inputs with different targets.

Ideally, one would like to construct hidden units whose frequency of activation is normalized to fall within a desirable range, neither too rare nor too common. Because this sort of normalization is based purely on the input distribution, without reference to the targets, it is a form of *unsupervised learning*. It is also desirable to normalize the *density* of activation, that is, the number of unit's that are active at one time. We don't want all the units to be active or inactive at the same times because then they are redundant.

In this section we explore the ability of simple unsupervised learning processes to reduce the size of the representation required by RR methods. It has long been recognized that unsupervised processes can improve the representation used by subsequent linear learning processes. Competitive learning methods (Rumelhart and Zipser, 1986; Kohonen, 1990) have been widely used for this purpose. Sanger (1991), Oja (1983), Linsker (1988), and Földiák (1990) have all argued that the activity of hidden units should be uncorrelated, and have proposed learning methods that detect and eliminate correlations. A significant disadvantage of these latter methods is that they require communication and memory between each pair of units. Generating hidden units at random, as in RR methods, can be seen as an attempt to produce a similar result with no communication at all, relying on statistical properties to ensure that no hidden units are extremely similar.

In this and the following experiments we applied two unsupervised learning techniques. Their overall goal was to keep the frequency and density of activity (as defined above) within a desirable range. Unsupervised learning occurred online, at the same time as supervised learning proceeded in the final layer.

To normalize the frequency of activation of a unit, j, we continually estimated the proportion $\bar{y}_j$ of examples on which it was active ($y_j = 1$):

$$\bar{y}_j \leftarrow \lambda \bar{y}_j + (1 - \lambda) y_j \,, \tag{6}$$

where $\lambda = 0.99$. Our objective was to keep the frequency of activation at $25\% \pm 5\%$. Thus, whenever the estimate $\bar{y}_j$ fell below 0.2 we decremented the threshold θ_j by ϵ, so that the unit would come on more often. Whenever the estimate rose above 0.3, we incremented the threshold by ϵ, so that the unit would come on less often. In the following experiments we used $\epsilon = 0.001$.

To normalize the density—the proportion of units active at one time—we adjusted the weights of each unit so that they would respond maximally to different portions of the input space. As with frequency, we sought to keep the density at $25\% \pm 5\%$. If the density on some time step was less than 0.2 (less than 20% of the units active) then it was considered too low, and

[3] This is may be due more to the use of online training than to the use of RR methods

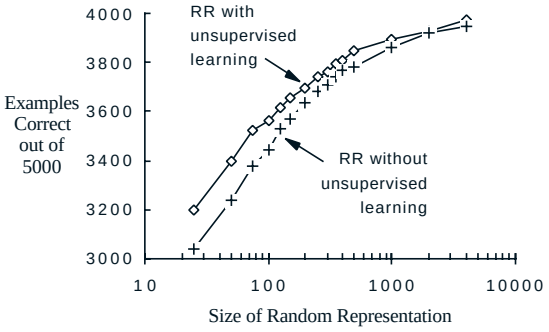

Figure 6: Comparison of RR methods with and without unsupervised learning on a task with many unchanging input bits.

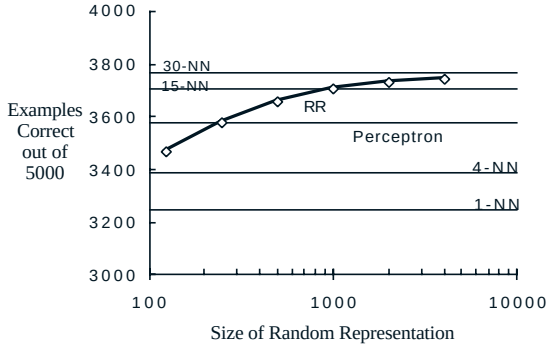

Figure 7: Performance on a task with many irrelevant input bits.

some of the inactive units were selected and changed so as to become more responsive to that time step's input pattern. Each inactive unit was selected for being changed with a probability of 0.0001. Selected units were changed by selecting at random one of their weights that did not agree with the input pattern and flip it so that it did agree. If the selected bit was 1 and the weight was $v = -1$, then the weight was flipped to $v = +1$, and if the input bit was 0 and the weight $+1$, then the weight was flipped to -1. In either case the unit's threshold θ was decreased by 1 so that the primary effect was on density (when the unit comes active) rather than frequency. Conversely, if the density on some time step was greater than 0.3, then it was considered too high, and in a symmetrical way we picked some units at random and flipped one of their bits so that it did *not* agree with the current input. In this case the selection probability was 0.0005, and the threshold was increased by 1 rather than decreased.

The next experiment was designed to directly assess the benefits to the RR algorithm of the unsupervised adaptations. The task was the same as the first experiment (8 relevant inputs, 8 irrelevant) except that 16 additional inputs were added that were always 1. This was done as a simple way to generate tasks with a nonuniform input distribution. Note that in the first experiment all input patterns were equally likely. Such a uniform input distribution is a favorable case for RR approaches and is the case used in most of Kanerva's analyses and Gallant and Smith's experiments. When the distribution is instead *non*uniform, many of the randomly constructed units may respond most strongly to parts of the input space that never occur, or that occur too often. It is here that unsupervised learning may have a significant effect.

Figure 6 summarizes the performance of the RR algorithm with and without unsupervised learning at a variety of representation sizes. The graph shows the numbers of examples correctly guessed by the algorithm out of 5000, plotted against representation size. Within the range where the size of the representation has an effect on performance, unsupervised learning permits the representation to be reduced by roughly 40%.

6 Many Irrelevant Inputs

One of the strengths claimed for RR methods is that they are relatively unaffected by irrelevant inputs. Other online learning methods such as radial basis functions are known to have great difficulty with irrelevant inputs (e.g., Hartman and Keeler, 1991). As we show next, irrelevant inputs also cause difficulties for nearest-neighbor methods.

This experiment was identical to the first experiment (8 relevant input bits, 8 target prototypes, 5000 steps per run, 20 runs) except that it used 22 irrelevant input bits instead of 8. The summary results for the RR method with unsupervised learning and for the KNN methods are shown in Figure 7. In this experiment the KNN method performed worse than the RR method at most values of K tried, and much worse at low values of K.

7 RR vs Backpropagation

The backpropagation algorithm has been widely criticized as being poorly suited to online learning because it learns too slowly, requires repeated passes through the training set, and generalizes too much between examples. This final experiment is meant to briefly illustrate these problems and show that the RR method does not suffer from them.

We sought a simple class of problems well suited to solution by both backpropagation and RR methods. The basic idea was to learn to replicate the behavior of a randomly constructed network. This *target network* consisted of three LTUs, each of which had connections to three randomly selected inputs. The

weights of the connections were set to $+1$ or -1 with equal probability, and each LTU's threshold was set so that the LTU was active and inactive with roughly equal probability. The LTU output values were $+1$ for active and -1 for inactive. The final output of the target network was determined by a linear unit with connections to each LTU (and to a constant bias input) weighted either $+1$ or -1 with equal probability. There were 9 binary inputs signals, which took on the values $+1$ and -1 with equal probability. 50 tasks were created in this way and used to test all algorithms and parameter values. All algorithms were run for 500 examples.

For the backpropagation algorithm, networks with 4, 40, and 400 hidden units were used. The RR method was run with a range of representation sizes from 10 to 4000. Unsupervised learning was not used in this experiment. For backpropagation, the final layer consisted of a single linear unit. The RR method was adapted from the binary classification form described earlier to a minimum squared error form by replacing equations (3) and (5) with a normalized weighted sum (weighted average):

$$f = p = \frac{\sum_{j=0}^{m} w_j y_j}{\sum_{i=0}^{m} y_j}. \qquad (7)$$

For the backpropagation method, many different values were tried for the learning rates for the first and second layers, and for the variance of the gaussian distribution of initial weights. The results we report are for the best values found. Momentum was not used.

Figure 8 shows average learning curves for the RR method with representation sizes of 100 and 1000, and for backpropagation with 4, 40, and 400 hidden units. The backpropagation results shown are for the best parameter values found after a substantial search. Higher numbers of hidden units were not explored due to the computational expense, but the general trend of the data was toward worse performance at higher numbers of hidden units.

On might wonder why RR methods do so much better than backpropagation on this task. A backpropagation network with a large number of hidden units, and whose initial random weights are chosen with a large variance is in fact almost equivalent to a RR network, However, in this case roughly half the units are on for each input pattern resulting in excessive generalization between input patterns. To get best performance from backpropagation, the number of hidden units must be chosen carefully, neither too small nor too big.

8 Conclusions

In this paper we have begun to experimentally evaluate the performance and utility of random-representation methods for online learning. Our results significantly

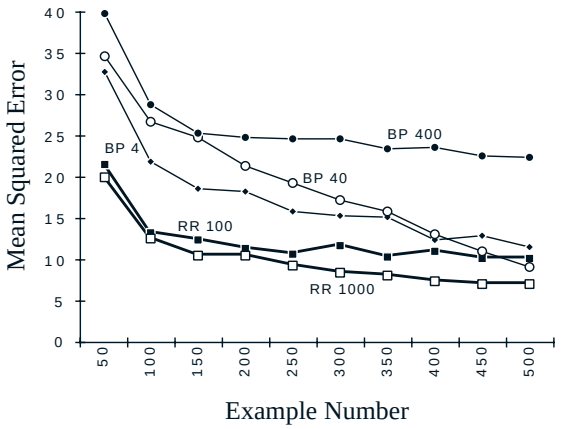

Figure 8: RR vs. Backpropagation.

extend what was previously known about RR methods in that:

- We have shown a rough parity between the performance of RR methods and nearest-neighbor methods, without requiring unreasonably large random representations. This is a promising result because RR methods are more immediately suitable for online learning tasks than nearest-neighbor methods, which are neither strictly incremental nor easily adapted to nonstationary problems. This general result was predicted by Kanerva (1988) for a related system, but had not previously been shown.
- Our results show that RR methods can handle moderately high dimensional input spaces with many irrelevant inputs (e.g., 22 irrelevant bits out of 30) with little increase in representation size. This is in sharp contrast with other ER methods such as radial-basis-function networks, which have great difficulties with these cases.
- We have introduced two simple methods for unsupervised adaptation of randomly constructed representations and shown that they can significantly reduce the required representation size on a problem with a nonuniform input distribution.
- We have presented results showing much faster initial learning by RR methods than by backpropagation networks.

These results show that randomness can play a surprisingly useful role in the search for good representations. Random selection is easy to implement, and, suitably biased, can be an effective search strategy for high-dimensional spaces. Rather than trying to follow gradients, or proposing a small number of carefully chosen features, why not generate a great many at random? What is lost in missteps may be more than made up by the freedom to take simultaneous large steps in a wide variety of directions.

For future work, we are interested in applications to inherently online tasks, such as reinforcement learning tasks, in extensions to continuous inputs and continuous hidden-unit weights, in data-driven methods for biasing the generation of hidden-units (e.g., as in Rogers (1990)), and in multi-layer RR systems. In this paper we have shown that network and nearest-neighbor methods can yield surprisingly similar performance. Does this similarity of performance reflect a deeper similarity between these superficially very different approaches? In our current work we are exploring algorithms that further blur the distinctions between these two classes of methods, in the attempt to get all the advantages of both.

Acknowledgments

We gratefully acknowledge the helpful comments we have received from Dan Schwartz, Ming Tan, Richard Yee, Chris Atkeson, and the ML93 reviewers.

References

Albus, J.S. (1981) *Brain, Behavior, and Robotics*. Peterborough, NH: Byte Books. Chapter 6, pp. 139–179.

Breiman, L., Friedman, J., Olshen, R., Stone, C.J. (1984) *Classification and Regression Trees*. Belmont, California: Wadsworth.

Duda, R.O., Hart, P.E. (1973) *Pattern Classification and Scene Analysis*. New York: Wiley.

Fisher, D.H. (1987) Knowledge acquisition via incremental conceptual clustering. *Machine Learning 2*: 139–172.

Földiák, P. (1990) Forming sparse representations by local anti-Hebbian learning. *Biological Cybernetics 64*: 165–170.

Friedman, J.H. (1988) Multivariate adaptive regression splines. Technical Report 102, Stanford Univ. Lab. for Computational Statistics.

Gallant, S., Smith, D. (1987) Random cells: an idea whose time has come and gone ... and come again? *Proceeding of the IEEE International Conference on Neural Networks*.

Hartman, E., Keeler, J.D. (1991) Predicting the future: Advantages of semilocal units. *Neural Computation 3*: 566–578.

Hinton, G.E. (1989) Connectionist learning procedures. *Artificial Intelligence 40*: 185–234.

Kanerva, P. (1988) *Sparse Distributed Memory*. Cambridge, MA: MIT Press.

Klopf, A.H., Gose, E. (1963) An evolutionary pattern recognition network. *IEEE Transactions on Systems, Man, and Cybernetics 15*: 247–250.

Kohonen, T. (1990) The self-organizing map. *Proceedings of the IEEE 78*: 1464–1480.

Lin, L.-J. (1992) Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning 8*: 293–322.

Linsker, R. (1988) Self-organization in a perceptual network. *Computer 21*: 105–117.

Mahadevan, S. (1992) Enhancing transfer in reinforcement learning by building stochastic models of robot actions. *Proceedings ML92*: 290–299.

Mahadevan, S., Connell, J. (1992) Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*: 311–365.

Michalski, R.S. (1983) A theory and methodology of inductive learning. In *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.). San Mateo, CA: Morgan Kaufmann.

Moody, J., Darken, C.J. (1989) Fast learning in networks of locally-tuned processing units. *Neural Computation 1*: 281–294.

Moore, A.W., Atkeson, C.G. (1992) An investigation of memory-based function approximators for learning control. MIT AI Laboratory Techical Report.

Oja, E. (1983) *Subspace Methods of Pattern Recognition*. Letchworth, Hertfordshire, UK: Research Studies Press.

Prager, R.W., Fallside, F. (1988) The modified Kanerva model for automatic speech recognition. *Computer Speech and Language 5*: 257–274.

Quinlan, J.R. (1986) Induction of decision trees. *Machine Learning 1*: 81–106.

Rogers, D. (1990) Predicting weather using a genetic memory: a combination of Kanerva's sparse distributed memory and Holland's genetic algorithm. *NIPS-2*, pp. 455–464. San Mateo, CA: Morgan Kaufmann.

Rosenblatt, F. (1962) *Principles of Neurodynamics*. New York: Spartan Books.

Rumelhart, D.E., Zipser, D. (1986) Feature discovery by competitive learning. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, 151–193. Cambridge, MA: MIT Press.

Sanger, T.D. (1991) Optimal hidden units for two-layer nonlinear feedforward neural networks. *Int. J. Pattern Recognition and AI 5*: 545-561.

Schlimmer, J.C., Fisher, D. (1986) A case study of incremental concept induction. *AAAI-86*: 496–501.

Schlimmer, J.C., Granger, R.H., Jr. (1986) Incremental learning from noisy data. *Machine Learning 1*: 317–354.

Schwartz, D. (1993) ATM scheduling with queuing delay predictions. *Proceedings ML-93*.

Tesauro, G. (1992) Practical issues in temporal difference learning. *Machine Learning 8*: 257–278.

Uhr, L., Vossler, C. (1961) A pattern recognition program that generates, evaluates and adjusts its own operators. *Proc. of the Western J. Computer Conference*, 555–569.

Utgoff, P.E. (1989) Incremental induction of decision trees. *Machine Learning 4*: 161–186.