Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems,
Lausanne, Switzerland, Feb. 12–14, 1996. IEEE Computer Society Press. — Reprint

On-Line Hand-Printing Recognition with Neural Networks

Richard F. Lyon and Larry S. Yaeger

Advanced Technology Group
Apple Computer, Inc.
One Infinite Loop
Cupertino, CA 95014 USA
E-mail: hwr@atg.apple.com

Abstract

The need for fast and accurate text entry on small handheld computers has led to a resurgence of interest in on-line word recognition using artificial neural networks. Classical methods have been combined and improved to produce robust recognition of hand-printed English text. The central concept of a neural net as a character classifier provides a good base for a recognition system; long-standing issues relative to training, generalization, segmentation, probabilistic formalisms, etc., need to resolved, however, to get excellent performance. A number of innovations in how to use a neural net as a classifier in a word recognizer are presented: negative training, stroke warping, balancing, normalized output error, error emphasis, multiple representations, quantized weights, and integrated word segmentation all contribute to efficient and robust performance.

1: Introduction

While on-line handwriting recognition is a problem of long-standing interest and activity, the recent introduction of low-cost portable pen-based computers (e.g., the "Personal Digital Assistant" or PDA category) has focused urgent attention on practical improvements. The classical approach of relying on strong language models or dictionary constraints to limit the search space and the ambiguity inherent in complex pattern recognition problems has not proven to be an acceptable solution in the PDA market. As an alternative, we have been conducting research on more powerful bottom-up classification techniques based on trainable artificial neural networks (ANNs), in combination with comprehensive but weakly-applied language models. To focus our work on a subproblem that is tractable enough to lead to usable products in a reasonable time, we have decided to restrict the domain to hand-printing, so that character-level segmentation is always cued by a pen lift.

There is a rich background on the use of ANNs as classifiers, and in recent years a fair amount of work on the use of ANNs as low-level classifiers in higher-level recognition systems—e.g., as phoneme classifiers in speech recognition systems and as character classifiers in handwriting recognition systems. The integrated segmentation and recognition approach, as used in HMM-based speech recognizers [1] and in "run-on discrete" and cursive handwriting recognition systems [2] provides a reasonable basis for incorporating a low-level classifier as a source of "scores" to be combined in a higher-level search process that seeks overall optimum combinations. But these approaches leave a large number of open-ended questions about how to achieve excellent performance. In this paper, we survey some of our experiences in exploring refinements and improvements. While all the techniques we describe are believed to provide advantages within at least some part of the domain in which we have tested them, we do not provide quantitative results, nor do we comment on exactly what combination of techniques may have been incorporated into products.

The standard multi-layer perceptron (MLP) with contrastive back-propagation training, 0/1 class output targets, and logistic-function sigmoidal squashing functions is the only class of ANN covered in this work, but most of the ideas will be equally applicable in other architectures, other domains, etc.

2: System Overview

It is conventional practice to map high-level recognition problems into minimum-cost-path graph search problems, by defining concatenative sub-units of

the high-level units to be recognized. In our system, words (and non-word strings of characters) are the high-level entities, and characters are the sub-units. A graph is constructed (implicitly) such that paths through the graph correspond to sequences of characters, and additive scores are accumulated along the paths to get scores for words. A search algorithm finds the best path or the several best paths.

2.1: Scores and Segments

Scores in our system are generally thought of as negative log probabilities, or positive costs, so that the minimization of a sum of cost terms is equivalent to maximization of a product of probabilities. We make the usual independence assumptions when we don't know any better, and we convert probability estimates to costs via table lookups. We do not, however, restrict ourselves to strict probabilistic formalisms, and we allow liberal use of fudge factors in combining scores from different modules—maybe someday we will train all the fudge factors by back-propagation, too.

At the lowest level, the inputs are points from a pen tablet; points are grouped into strokes between pen-down and pen-up events. In a cursive or connected recognition system, a segmentation process that maps regions of the lowest-level input into hypothetical character units would need to cut strokes into fragments [2]. But in our printing system, we assume that each stroke belongs to exactly one character. Therefore, we can treat strokes as the bottom-level input, and the segmentation process only needs to hypothesize groupings of strokes into characters, avoiding the harder problem of splitting strokes across multiple characters.

From the bottom up, our system consists of modules for soft grouping or segmentation, soft character classification using an ANN (referred to as "the net"), and N-best graph search. In addition, as shown in Figure 1, there are several modules "on the side" to supply additional context-dependent scores to be added to arcs in the graph during the search process, representing *a priori* sequence probabilities and relative size and position information between characters.

2.2: Searching and Pruning

The finite-state graph implied by the higher-level language model is theoretically amenable to an optimal dynamic programming (Viterbi) search, but the state space is too large to make this approach attractive. Instead, a beam search is used, with unlikely partial paths being continuously pruned in accordance with a beam size parameter (number of active hypotheses ending with each input segment).

Additional mechanisms of pruning prevent the system from exploring very unlikely paths. In particular, the net "proposes" possible characters from its bottom-up information, pruning away character-level hypotheses with essentially zero probability—for many incorrect segments, no character is proposed, saving lots of work that might otherwise be associated with the large number of incorrect segmentation hypotheses generated.

Paths that converge onto a state are not combined as in a Viterbi search, so obtaining the several best word results at the end of the search process is easy.

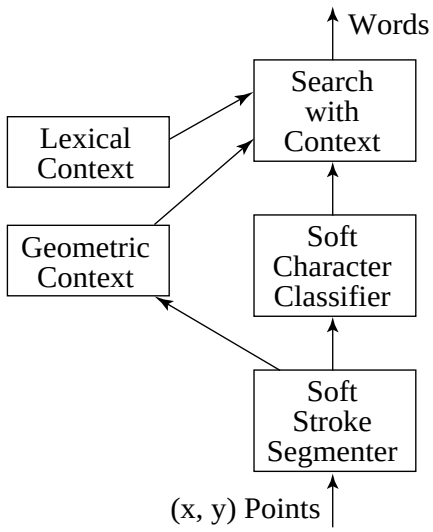

Figure 1. A simplified block diagram of the major components of a system for recognizing hand-printed words.

3: Posterior Probabilities

The core of a bottom-up high-level recognition system is a low-level classifier that provides probabilistic scores, rather than decisions, about the classification of low-level segments. A number of authors [3, 4, 5, 6, 7, 8] have shown that contrastive training of MLPs causes them to provide good estimates, in a mean squared error sense, of the *a posteriori* probabilities of each class given the input. Therefore, it makes sense to try to incorporate MLP output scores as probabilities in a probabilistic recognition formalism. Lippmann [3] provides a particularly compelling case for why ANNs used this way should be expected to do a better job than other

recognition approaches. Morgan and Bourlard [4] provide an in-depth look at application of ANNs to speech recognition.

3.1: Frequency Balancing

Both Lippmann [3] and Morgan [4] talk about the problem of converting a net's output, an estimate of posterior probability $P(class|input)$, into the number needed in an HMM or Viterbi search, $P(input|class)$, using Bayes' rule. They both recommend dividing the net output by the prior probability of the class; Morgan: "That is, we divide the posterior estimates from the ANN outputs by estimates of class priors, namely the relative frequencies of each class..."

Using this approach, classes with low frequency in the training set end up with much noisier estimates of $P(input|class)$, due to the division by the low frequency, so the resulting estimates are not really optimized in a mean-squared-error sense as the net outputs were. In addition, output activations that are naturally bounded between 0 and 1, due to the sigmoid, convert to potentially very large probability estimates, requiring a re-normalization step.

Our proposed alternative is to balance the frequencies of the classes in training. We never actually tried the division approach, so we can't say our alternative is necessarily better for word-level recognition—but at least we can say it works a lot better than doing no correction at all.

Since throwing out training samples is a counter-productive way to achieve balance, samples from infrequent classes might be replicated instead. Doing so statically would require handling a greatly expanded set of training patterns, though if replication were done via pointers, this might not be a problem. Alternatively, replication can be done dynamically, integrated into the process of choosing training samples in a random sequence.

In our training process, each epoch consists of a pass over all the training samples, in random sequence. We have included balancing by allowing each randomly-chosen pattern to be presented a variable number of times, where the number is computed probabilistically, and may be zero, one, or more. Many samples from frequent classes are skipped on each epoch, but are used in other epochs. No data gets discarded, and there is no static expansion of the training set.

Based on the number of samples of each class available in the training set, we precompute a replication-factor for each class, which is treated as a real-valued average number of presentations for each sample selected from that class.

A rep-factor less than 1 means there's a probability of skipping the pattern (not training on it, even though it was chosen at random to be the next training pattern); a rep-factor greater than 1 means there will be a chance of repeating the pattern immediately before choosing a next pattern at random.

We compute the rep-factors using a partial-normalization approach. First, for each class i we compute the frequency of that class (number of samples), normalized relative to the average frequency (total number of samples over total number of classes that have any samples):

$$F_i = \frac{n_i}{n_{total}/N}$$

Then we compute a rep-factor for each class as

$$R_i = \left(a/F_i\right)^b$$

with scale factor a in the vicinity of .2 to .8, and exponent b typically 0.5 to 0.9 ($b = 0$ would do nothing, giving $R_i = 1$ for all classes).

The factor $a < 1$ lets us do more skipping than repeating; e.g. for $a = 0.5$, classes with relative frequency equal to half the average will neither skip nor repeat; more frequent classes will skip, and less frequent classes will repeat.

The exponent b lets us compromise with how much normalization of prior probabilities we do. Using $b < 1$ lets the net keep some bias in favor of classes with higher prior probabilities. This bias might help in some cases, depending on the form of the higher-level models used in word recognition. It seems to help to keep b somewhat below 1.

3.2: Normalized Output Error

It occurred to us that having a net trained with a single target of 1 for the correct class and $N-1$ targets of 0 for all the other classes was pushing the outputs too hard toward 0, making it difficult to get meaningful representations of the small probabilities associated with second- and third-choice classes. We found many word errors in which the correct answer had one character which, although apparently "readable," showed essentially zero probability, due to being a poor alternative choice. By normalizing the forces pushing outputs toward 0 targets, relative to the force on one output toward the 1 target, we reasoned that low-probability alternatives would survive with useful posterior probability estimates.

We call this approach "NormOutErr" for Normalized Output Error. In short, we reduce the back-propagated error for classifier outputs corresponding to the many "incorrect" classes, relative to the "correct" class, by

"normalizing" those errors relative to the number of incorrect classes.

The effect is to generally raise the neural net outputs, since the modified training pushes much less hard toward zero than standard back-prop does. The outputs are raised such that they no longer converge to an MMSE estimate of $P(class|input)$. Rather, they converge to an MMSE estimate of a nonlinear function $f(P(class|input), A)$ depending on the factor A by which we reduced the error pressure toward zero. This causes the net to do a better job of estimating the probability when it is low, rather than just doing a good job for probabilities near 0.5 (the steepest part of the sigmoid curve) as in the standard approach.

We worked out, using a simple version of the technique of Bourlard and Wellekens [5], what the resulting nonlinear function is. The net will attempt to converge to minimize the modified quadratic error function

$$\left\langle \hat{E}^2 \right\rangle = p \cdot (1-y)^2 + A \cdot (1-p) \cdot y^2$$

by setting its output y for a particular class to

$$y = p/(A - A \cdot p + p)$$

for $p = P(class|input)$, and modification factor A. For small values of p, the activation y is increased by a factor of nearly $1/A$ relative to the conventional case of $y = p$, and for high values of p the activation is closer to 1 by nearly a factor of A.

The inverse function, useful for converting back to a probability, is

$$p = y \cdot A/(y \cdot A + 1 - y)$$

We verified the fit of this function by looking at histograms of character-level empirical percentage-correct versus y, as in Figure 2.

The factor A is set according to a partial-normalization formula as

$$A = 1/(d \cdot (N-1))$$

where $d = 1$ yields "complete normalization" for N classes, and $d = 1/(N-1)$ reverts to the standard case. In our printable-ASCII-character-set recognizer, N is 94, so $d = 1$ yields $A = 0.011$, while a more moderate setting of $d = 0.1$ yields $A = 0.11$—still a significant factor change from the conventional case of $A = 1$.

Using the $d = 0.1$, $A = 0.11$ example, the steepest part of the sigmoid where $y = 0.5$ corresponds to $p = 0.10$. Therefore, the net spends its resources disproportionately trying to minimize the error in estimating p values near 0.10.

Since log probabilities are used in the search, a possible strategy would be to try to use the net to estimate log probability directly. One might try to minimize mean squared error of the log probability, or calculate error relative to the probability estimate. It's not clear exactly how to do either of these, but it is clear that it might cause a problem in pushing the net to use its resources very disproportionately near $p = 0$, where the relative error is necessarily large. The NormOutErr technique can be viewed as a stable intermediate technique that does not have the singularity near $p = 0$, but allows the emphasis to be moved by a controlled amount toward the low-probability region.

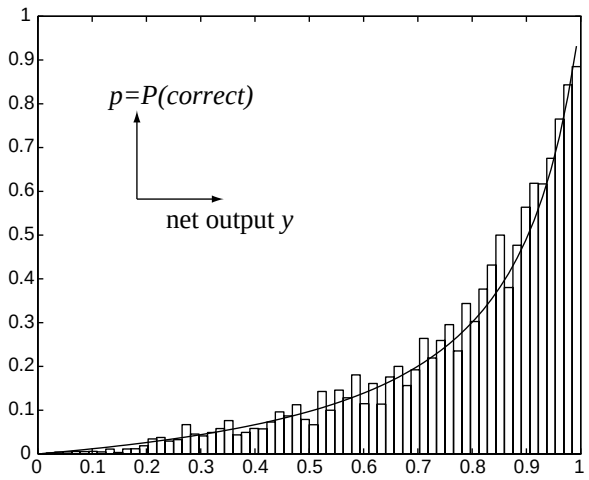

Figure 2. Empirical *p vs. y* histogram for a net trained with *A*=0.11, with the corresponding theoretical curve.

Reducing the back-propagated error by a factor of A on all but one of the net outputs would seem to decrease the effective learning rate by almost a factor of A. But the outputs near zero tend to be almost a factor of A larger, largely compensating for this effect. Of course, there is still some reduction in learning rate, especially for the correct class output in easy cases, which become much closer to the target of 1.

In our tests, the NormOutErr modification always negatively impacted the net's first-choice character-level accuracy, so it would not be a generally useful technique in simple classifiers. However, the improvement in word-level accuracy was impressive, due to the improvement in probability estimates for those low-probability character hypotheses that were important in word-error and near-miss cases.

4: Negative Training

Figure 3 shows an example of a hand-printed word "dog" or "clog." Depending on how it is labeled, the segment consisting of the first stroke might be either a positive example of "c" or a mis-segmentation. There

is no way for a bottom-up segmenter to know which segments are correct, so many wrong segments must be considered at the higher levels.

Therefore, in the normal course of recognition, many segments considered by the net do not correspond to any actual character. The usual practice of training a low-level classifier only on examples of actual characters leaves its response unconstrained for novel patterns that correspond to fragments or combinations of characters. Ideally, we would want the classifier to indicate "none of the above" when it sees an inappropriate segmentation, so that the result of the high-level search process would not likely select a random labeling of a sequence of inappropriate segments.

Figure 3. The "dog/clog/cbg" ambiguous word, exemplifying the segmentation problem in hand-print recognition. Without negative training, it might even be recognized as "%g".

4.1: Negative Patterns

To accomplish this goal, we need to let the net train on examples of the kinds of mis-segmented patterns that it will encounter during recognition. Therefore, the training process needs to be driven from word-level or sentence-level data, and needs to include the segmentation process. Training on a batch of labeled characters is not enough.

We could define an additional class for these "negative patterns," corresponding to "not a character," and treat mis-segmented training samples just like regular character training samples, with a target value of 1 for this new class and 0 for all the others. Training the net would still cause it to provide estimates of $P(class|input)$, but the unwanted implicit conditionalization on correct segmentation would go away. Instead, we treat mis-segmented patterns specially, for two reasons: (1) the "not a character" class output probability is not actually useful to us, so we omit it and use all 0 targets; (2) we maintain independent control of "training factor" and "training probability" of negative patterns in order to trade off the amounts of positive and negative training influence, and their use of training time.

As with NormOutErr, this modification hurts the character-level accuracy of the net as a classifier of correctly segmented characters (by a little bit), but helps overall word recognition (by a lot).

4.2: Strength of Negative Training

For the negative patterns, the back-propagation training force toward 0 targets is not reduced by NormOutErr's factor A, but rather by an independent parameter called the "negative training factor," which is typically between 0.2 and 0.5. In addition, since negative training patterns are quite numerous, we skip most of them on any particular epoch. Rather than use the rep-factor from balancing, as on positive patterns, we use an independent "negative training probability," which is typically between 0.05 and 0.3. The product of these factors yields an effective reduction in the influence of negative patterns.

Training at "full strength" on negative patterns would cause too much suppression of the net outputs for certain characters like I, 1, l, o, O and 0, that tend to look the same as fragments of other characters. By varying the training factor and probability, we mitigate this effect while saving lots of training time. It is not clear yet what formal probability model could account for these tradeoffs.

5: Robust Backprop Training

The classical method of stochastic error back-propagation has some classical difficulties that need to be mitigated: (1) making sure the net can learn all the important variations in the training data, as when a class has several very different clusters of examples, including some clusters with very few examples; (2) making sure the net does not learn just minute peculiarities and over-fit to the training data, but rather will generalize to novel testing data. These difficulties are typically seen as being opposed to each other. Techniques such as cross validation stopping and structural risk minimization typically optimize generalization at the expense of thorough learning.

Our approach in this research has been to make sure we can train long and hard, with a sufficiently powerful net architecture, to optimize learning, and train to complete convergence, while avoiding overfitting by using artificial random variations of the training patterns.

5.1: Stroke Warping

To avoid overfitting, we never present the same pattern to the net more than once—in spite of what we said above about repeating patterns, multiple epochs, etc. Instead, we train on patterns that are derived from the training samples, "warped" through a new random transformation on each presentation.

Recall that segments to be classified consist of sets of strokes, and that strokes are sequences of (x,y) coordinates. Sample segments in the training set are similarly stored. The pattern input to the net, however, is a derived representation of the segment; in our system, an object-oriented "patternizer" takes care of constructing various kinds of net inputs from segments. In the training process, but not in the recognition process, we interpose an additional "stroke warping" step in front of the patternizer, to generate random but plausible variations of the sample segment.

Stroke warping is accomplished as (mostly affine) transformations on the (x,y) coordinates of the strokes in a segment, with random transformation parameters; all the points in all the strokes in a segment undergo the same transformation: $(x',y') = M \cdot (x,y)$, where M is a matrix not too far from an identity. Patternizers that derive features such as height, width, local direction, curvature, or image do so from the transformed coordinates.

The "matrix not too far from an identity" concept is a *post hoc* rationalization of what we really do, which is to apply random amounts of rotation, skew, x-size, and y-size factors, plus x and y quadratic nonlinear distortions, within ranges controlled by a set of independent training parameters. These distortions are chosen to represent plausible variations in writing style. The amounts of each distortion to apply are chosen, through cross-validation experiments, as just the amount needed to yield optimum generalization, in combination with our "thorough" training process. Pages of distorted examples are reviewed by eye to verify that they appear to represent a natural range of variation; a brief example is shown in Figure 4.

Figure 4. A few random stroke warpings of the same original "m" data.

Our stroke warping scheme is somewhat related to the ideas of Tangent Prop [9] and Tangent Dist [10], in terms of the use of predetermined families of transformations, but we believe it is much easier to implement. It is also somewhat distinct in applying transformations on the original coordinate data, as opposed to using distortions of images. The voice transformation scheme of Chang and Lippmann [11] is also related, but they use a static replication of the training set through a small number of transformations, rather than dynamic random transformations of an infinite variety.

5.2: Probabilistic Pattern Skipping

We have already mentioned probabilistic skipping of patterns in relation to balancing and negative training probability. In addition, we use a "correct train probability" parameter to sometimes skip training on patterns that are already correctly classified. When correct train probability is very low, like 0.1, most of the training time and the net's learning capability is being directed toward patterns that are more difficult to correctly classify. This is the only way we were able to get the net to learn to correctly classify unusual character variants, such as a 3-stroke "5" as written by only one training writer. This technique is one of several forms of "error emphasis." A related technique is the use of a training subset in which easily-classified patterns are continually replaced by randomly selected patterns from the full set [12].

When using this parameter, which only applies to positive patterns, a pattern selected for training is patternized and presented to the net, and the forward pass is evaluated. If the highest output activation is the one corresponding to the correct class, then a biased coin is flipped to decide whether to skip the back-propagation part of the process.

Patterns that are incorrectly classified are never skipped, but in variants of this scheme they might be repeated an extra time or back-propagated with double the usual learning rate.

Figure 5 illustrates some possible probabilistic skipping parameters and error training factors, in relation to segments from the word shown in Figure 3. Note that the concept of correct as used by correct train probability, and the concept of the correct or "label" class as used by NormOutErr are independent.

Segment	Type	Prob. of Usage		Error Factor	
		Correct	Incorrect	Label Class	Other Classes
cl o g	POS	0.5	1.0	1.0	0.1
c cl o l lo log og	NEG	0.18		NA	0.3

Figure 5. Positive and Negative segments from the example word "dog," with typical corresponding training probabilities and error factors.

5.3: Annealing

Many discussions of back-propagation seem to assume the use of a single fixed learning rate. We view the stochastic back-propagation process as more of a simulated annealing, with a learning rate starting very high and decreasing only slowly to a very low value. We typically start with a rate of 1.0, and reduce the rate by a "decay factor" of 0.9 until it gets down to about 0.001 (we define learning rate as the change in a weight value relative to $-E \cdot \partial E / \partial w$ in a system where the maximum absolute value of the error E is 1 and the maximum squashing function slope is 1/4). The rate decay factor is applied after any epoch on which the total squared error increased on the training set.

We typically run a few hundred thousand positive training patterns, and epochs take about an hour on a high-end IBM RS6000/Power2 machine. For smaller training sets, rate decay factors somewhat closer to 1 are sometimes used. As a result, the net spends a long time, like a few days or a few trillion instruction cycles, training at rates that are very high compared to the usual practice, and about a week annealing to a fully-trained state. Epochs beyond the first one only cut the character-level classification error rate by about a factor of two, but those diminishing returns are still well worth the price.

5.4: Training Schedule

For best overall results, we find it necessary to let some of our many training parameters change during the course of a training run. In particular, the correct train probability needs to start out very low to give the net a chance to learn unusual character styles, but it should finish up at 1.0 in order to not introduce a general posterior probability bias in favor of classes with lots of ambiguous examples.

Since we have a batch-file-oriented training program, we run with fixed parameters for a while, then stop the job, edit the parameters, and start over training from where the net left off. We typically train a net in four "phases" according to parameters such as in Figure 6.

Phase	Epochs	Learning Rate	Correct Train Prob	Negative Train Prob
1	25	1.0 - 0.5	0.1	0.05
2	25	0.5 - 0.1	0.25	0.1
3	50	0.1 - 0.01	0.5	0.18
4	30	0.01 - 0.001	1.0	0.3

Figure 6. A typical multi-phase schedule of learning rates and other parameters for training a character-classifier net. Phase 3 parameters correspond to the numbers in Figure 5.

6: Representation Issues

Probably the single most important success factor in applications of ANNs is the choice of a good representation for input information that the net will operate on. Redundant, distributed, fully decoded kinds of representations generally are more effective than concise, encoded, or implicit representations. For example, an image of a character works better than a set of coordinate values, a dipstick code works better than a set of binary-number bit inputs, etc.

Our research system architecture allows dynamic determination of representations by calling out the names and sizes of patternizers in a net architecture specification file. For example, we might specify a net with 12-by-12 image-splat, a 5-unit aspect ratio dipstick, and a 1-unit height value.

6.1: Image vs. Stroke Representations

We have experimented with a variety of image representations of characters, using anti-aliased gray-scale rendering of strokes into small pixel maps, and with a variety of local stroke features, such as local direction sampled at a fixed number of equal-arc-length intervals circularly encoded as gray scale bumps.

There is not space here to even survey the variety of representations we have experimented with, but suffice it to say that minor variations matter, and not always in the direction that one might expect. As a general result, however, images seem to work best, and 12-by-12 is about the right size if the training set is large enough.

6.2: Multiple representations

The different types of representations mentioned above provide somewhat complementary types of information. One shows what a character looks like, and the other shows how it was written; i.e., stroke order and direction don't matter in an image, but do matter in terms of feature sequences.

Using both input representations together should be expected to provide significant help to the net in distinguishing classes—assuming there is enough data that the extra information does not just lead to worse over-fitting. We find that the effect is indeed quite beneficial.

Exactly what architecture of hidden layers, etc., to use in combining these representations is a significant issue in itself. We favor an architecture in which each of the two major representations has its own first and second hidden layers, which come together only at the output layer. This approach somewhat resembles and is motivated by a "mixture-of-experts" [13] approach or a combination of several rather different types of classifiers, though the layers below the output layer are not really classifiers and are not trained independently.

We experimented with merging the two sides of the net one step lower; i.e., have separate first hidden layers converge onto a common second hidden layer (of about the same size as each of the separate hidden layers in the other approach). Starting from a very large net that we suspected of being too powerful (somewhat over-fitted), we expected that this change, which dropped the net from about 60,000 to about 50,000 weights, would let the net learn important high-order combination features and yet generalize better. This was pure wishful thinking. The net indeed learned combination features and did significantly better on the training set, but did slightly worse on the cross-validation test set.

We rationalize this result into a general lesson about handwriting recognition: if characters can be written in several different ways (stroke orders and directions), and can have a variety of appearances, don't match on the cross-product of those spaces. E.g., don't let the classifier learn that an "S" written from bottom to top always slants a little to the right, just because you had a writer in the training set with that particular combination.

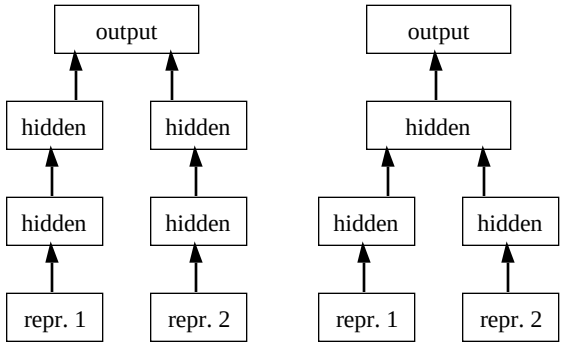

Figure 7. Two architectures for combining different input representations in a classifier net. Each box represents about 100 neuron units, and each heavy arrow corresponds to about 10,000 weights. The more separated architecture (left) seems to work better; the alternative that combines information at a hidden layer (right) is smaller but "too powerful."

7: Geometric Context

It is not possible to reliably estimate a baseline or topline independent of classifying the characters in a word. Our system approach factors the problem by letting the net classify representations that are independent of baseline and size, and then using a separate module to score the relative size and position of adjacent characters, depending on their classification hypotheses. We call this module geometric context or "GeoContext."

GeoContext has proven quite useful for removing case ambiguity and other shape ambiguities, especially for punctuation.

7.1: Relative Size and Placement

It is easy to measure such things as the top and bottom co-ordinates of hypothesized segments, differences between adjacent segments, widths, spaces, etc. For any hypothesized pair of characters, the corresponding expected measurements are constructed, to within an unknown scale factor and offset, from a table of character tops, bottoms, widths, etc., in a standardized space (e.g. in the space defined by baseline=0, topline=1). After scaling and aligning the measurements to the table values, an error vector is

computed to indicate how far the observed relative size and placement are from nominal.

The error vector is modeled by a full multi-variate Gaussian distribution for all characters, so a quadratic error term involving the inverse grand covariance matrix is used directly as a score.

7.2: Pairwise Scores

Notice that the score depends on the pair of character hypotheses for the two segments being considered, since the nominal size and position information for each segment depends on that segment's class. Thus, for example, if a first segment has bottom-up proposed hypotheses of "9", "g", *comma* and *apostrophe* while the second could be "O", "o", "0", and *period*, then 16 different scores will be generated for the possible pairs, to be added to corresponding transitions in the graph search. It is easy to see how pairs like "go", "90", "'.", etc. would give rise to distinguishing sets of scores using this mechanism.

A related technique of evaluating "Adjacency Constraints" that depend on pairwise classification hypotheses has been presented by Parizeau and Plamondon [14]. Their technique requires that ascenders and descenders be identified before the constraints are computed, as the constraints are computed on the "main body" of a character: "...as long as the segmentation/recognition method can isolate the bounding box around the main body of the different hypotheses..." We applaud their insight that the method is "attractive for bypassing the conventional ill-posed problem of zone estimation ... there is no global solution ... combinations of different hypotheses can be accepted or rejected with a local criterion."

7.3: Training GeoContext

In experimenting with relative size and placement measures, it quickly became apparent that our preconceptions about how people would write were not very correct, so we decided to use a data-driven process to set the parameters used in the calculation of GeoContext scores. The trick was to find a good way to train per-character parameters of top, bottom, width, space, etc., in a standardize space, from data without labeled baselines, etc. Since we had a technique for generating an error vector from the table of parameters, we decided to use a back-propagation variant to train the table of parameters to minimize the squared error terms in the error vectors, given all the pairs of adjacent characters and correct class labels from the training set. This approach is not discriminative, but seemed adequate—enough so that we decided to leave the ASCII characters *underscore* and *vertical-bar* in the recognition character set.

8: Word Segmentation

In most of our experimentation, we assume that words are correctly segmented out of the stream of input strokes, e.g. by looking for big gaps, time-outs, etc. For real on-line tests, however, or for processing sentence files not segmented to the word level, word segmentation is an important and difficult additional task. A gap size threshold works to first order, but the number of incorrect word segmentations (splits and joins) can be reduced significantly with more complex strategies.

8.1: Running Averages and Gap Sizes

One simple improvement on a fixed gap threshold is a variable gap threshold that is adjusted to track the size of the writing. After each word is recognized, the resulting character classifications and corresponding segmentations can be used to update a running average estimate of the height of a capital letter, for example, and a proportional gap threshold can be computed. The gap size might also be adjusted according to a running average of the inter-character gaps within words. But no matter how it is adjusted, a threshold is always a threshold, and will result in a discretely errorful behavior when the written gap size crosses to the wrong side of the threshold—a tiny change in writing, across an invisible boundary, can cause an annoying split or join.

8.2: Word-Space as a Character

To have a chance of "optimal" word segmentation, the search process needs to treat the hypothesis of a word break, or space, in much the same way it treats other competing character and string hypotheses. To do so is not necessarily easy, however, if the rest of the system is primarily oriented toward classifying groups of strokes and transitions between them, since the space has no strokes, no bounding box, etc.

We have developed a scheme that starts with two gap thresholds: if a gap is less than a small threshold, then no space is considered; if it exceeds a large threshold, then the word is hard-segmented; between these limits, a space hypothesis will be considered, with a cost term based on a parametric statistical model of gap sizes. Everywhere that a space needs to

be considered, the search process will generate pairs of hypotheses in transitioning from one segment to the next—a hypothesis of no space, continuing a word, and a hypothesis of a word end followed by a space and the beginning of a new word.

Since the high-level language model provides cost terms that reflect the likelihood of a word end for any given string of characters, the bottom-up costs from the gap measurement will trade off against the top-down costs from the language model in arriving at a best overall interpretation that includes possible word breaks. It is thus possible to write whole sentences in which character gaps and word gaps are all about the same, and still get the correct words recognized, in some cases.

Looking for word breaks based on moving in the vertical direction is another whole topic. It is not unusual for lines of writing to have overlapping vertical coordinates, and sometimes even tangled strokes. Even distinguishing return strokes (like i-dot) from new lines of writing can be a challenge, if writing can proceed upward to start a new line.

Integrating all the possibilities of word segmentation and return strokes into the search is still a formidable issue.

9: Hardware Considerations

Running complex word recognition systems of the sort discussed here on low-cost battery-powered hardware is a key problem and opportunity. With sufficient effort, it is possible to squeeze and optimize until a low-power RISC processor can barely keep up with fast printing. With further work, faster processors, specialized hardware support, or other breakthroughs, it should soon be possible to add active adaptation to the user, via continual backprop training.

Some important hardware and performance considerations are discussed in this section.

9.1: Quantized Weights

The work of Asanovic´ and Morgan [15] shows that two-byte (16-bit) weights are about the smallest that can be tolerated in training large ANNs via back-propagation. But memory is expensive in small devices, and RISC processors such as the ARM-610 are much more efficient doing one-byte loads and multiplies than two-byte loads and multiplies, so we were motivated to make one-byte weights work.

Running the net for recognition is a significantly less demanding problem, in terms of precision, than training the net. It turns out that one-byte precision is plenty, if the weights are trained appropriately. In particular, a dynamic range should be fixed, and weights limited to the legal range during training, and then rounded to the requisite precision. For example, we find that a range of values from (almost) –8 to +8 in steps of 1/16 does a good job. Figure 8 shows a typical corresponding distribution of weight values. If the weight limit is enforced during high-precision training, the resources of the net will be adapted to make up for the limit. Since bias weights are few in number, however, and very important, we allow them to use two bytes with essentially unlimited range. We find no noticeable degradation relative to floating-point, four-byte, or two-byte weights using this scheme.

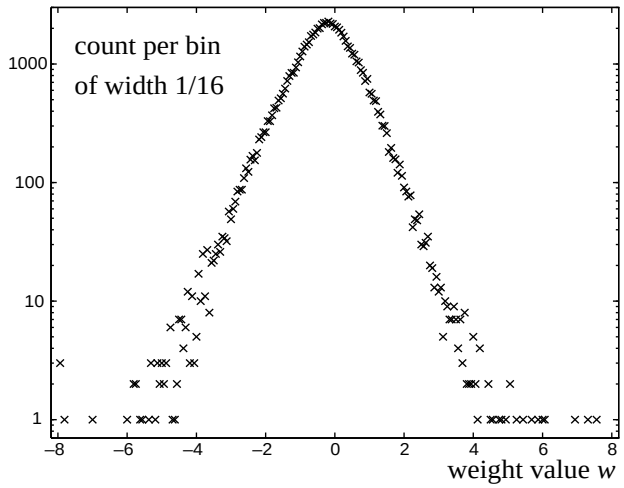

Figure 8. Distribution of weight values in a net with one-byte weights, on a log count scale. Weights with magnitudes greater than 4 are sparse, but important.

We have also developed a scheme for training with one-byte weights, which works as well as, or in some cases a little better than, floating-point training. It uses a temporary augmentation of the weight values with two additional low-order bytes to achieve precision in training, but runs the forward pass of the net using only the one-byte high-order part. Thus any cumulative effect of the one-byte rounded weights in the forward pass can be compensated in the further training. Small weight changes accumulate in the low-order bytes, and only occasionally carry into a change in the one-byte weights used by the net. In a personal product, this scheme could be used for batched adaptation to the user, after which the low-order residuals could be discarded and the temporary memory reclaimed. Perhaps only weights that actually changed from the ROM values would need to be patched in RAM.

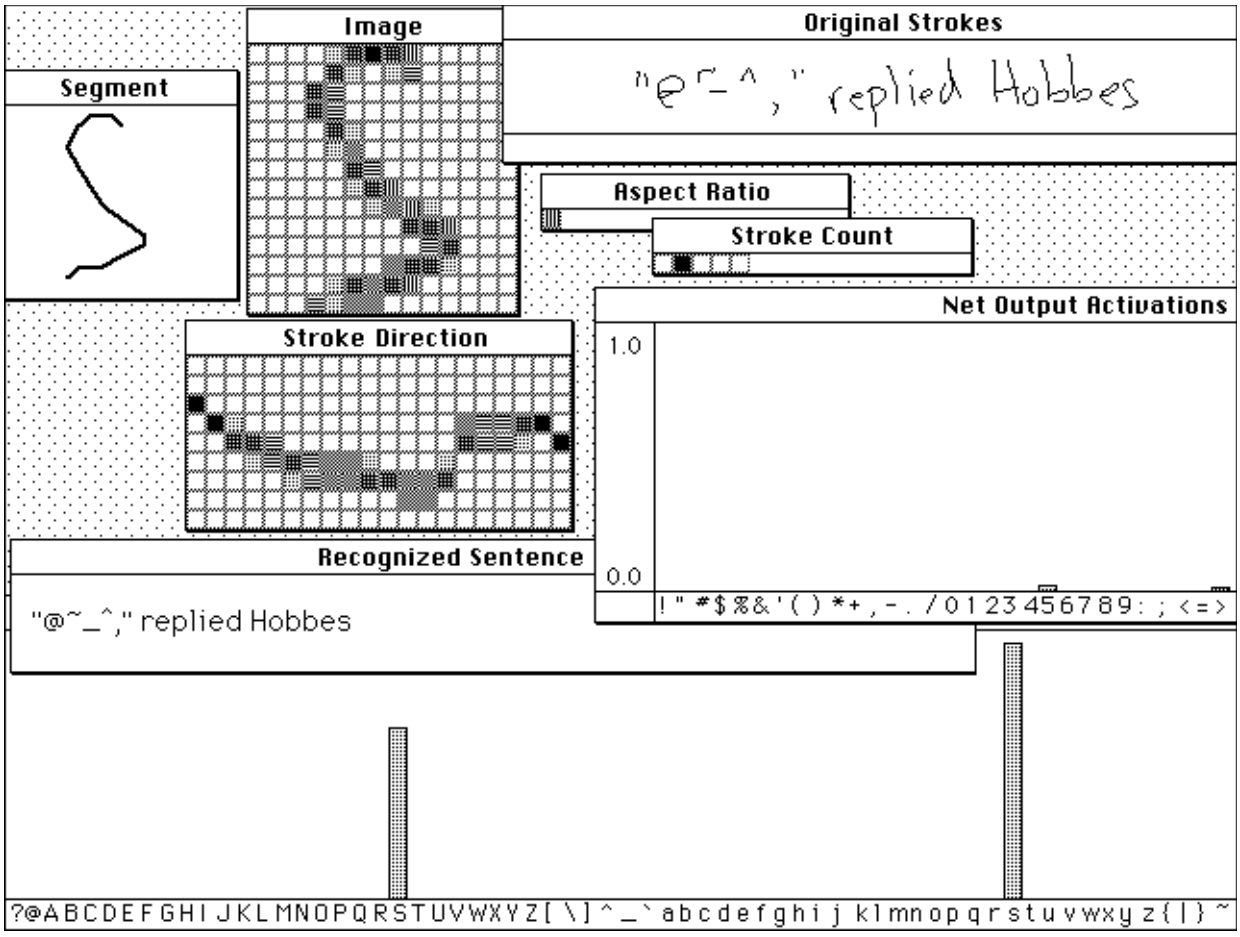

Figure 9. Some of the kinds of windows available in our research recognizer. The net output window is wrapped to fit all 94 printable-ASCII characters in the figure. In the example shown, an intended *hyphen* has been mis-classified as *underscore*.

9.2: Speed, Caching, and Power Efficiency

As long as the processor can keep up with fast writing, why try to speed up the process more? There are several good reasons. Battery-operated devices typically slow down or stop the processor clock when compute cycles are not needed, so using fewer cycles translates directly to using less energy from the limited supply in the battery. In addition, the apparent speed might change substantially, as there tends to be a significant latency between a stroke time-out or line-break and the time that an answer is returned and displayed—the system may keep up fine on multiple-line writing, but feel slow for a single line.

A major consumer of time and energy in running an ANN on a RISC processor is the transfer of weights from main memory, through the cache (if any), to the datapath. The ANN weight set, even for one-byte weights, will not fit in the cache of a small device, so every segment evaluated will stream the whole set of weights through again, possibly flushing everything else out of the cache in the process. Typical co-processor strategies, connecting directly into the processor datapath, would not relieve this problem unless sufficient storage was provided in the co-processor to hold the weights. An alternative strategy would be to attach a very simple co-processor, such as a vector dot-product unit (multiplier-accumulator) directly to the main memory, to avoid pumping the weights through the cache and the processor, and saving cycles and energy. Keeping the net out of the main processor might further improve the caching behavior of the rest of the system, saving more energy. The resulting energy savings could translate directly to longer battery life or lower battery weight, or could be traded for more comprehensive search processes.

An interesting final note on performance tradeoffs comes from an experiment we did with a reduced-size net. We reasoned that a smaller net would use fewer memory and compute cycles, and so speed up the recognition system. What actually happened was that the net was not quite as certain in rejecting lots of incorrect character hypotheses, and thereby made lots of extra work for the search process—the system slowed down overall. So we don't skimp on the net.

10: Conclusions

We believe that we have made a pretty good start at, or have at least scratched the surface of, the wide range of open-ended possibilities for perfecting a system for recognizing on-line hand-printing. We have been extremely encouraged by the success of the ANN approach to classification, but emphasize that the details of the rest of the system will often matter more than the particular low-level classifier used—the ANN is a tool, not a solution.

The innovations we have described are expected to lead to a new level of acceptance of pen-input technology in the PDA market. We are actively pursuing extensions of these techniques to larger alphabets and other writing styles, including cursive.

Acknowledgments: We thank Bill Stafford, Brandyn Webb, Les Vogel, and Michael Kaplan for their many substantial contributions to the experiments and ideas described. We also thank our many colleagues in the neural-net community for their advice, help, and encouragement over the years, and our many colleagues at Apple who have pitched in to help throughout the life of this project.

Some of the techniques described in this paper are the subject of pending US and foreign patent applications.

References

[1] S. Renals and N. Morgan, M. Cohen, and H. Franco "Connectionist Probability Estimation in the Decipher Speech Recognition System," *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing* (San Francisco), pp. I-601–I-604, 1992.

[2] C. C. Tappert, C. Y. Suen, and T. Wakahara, "The State of the Art in On-Line Handwriting Recognition," *IEEE Trans. PAMI,* Vol. 12, pp. 787– 808, 1990.

[3] R. P. Lippmann, "Neural Networks, Bayesian *a posteriori* Probabilities, and Pattern Classification," pp. 83–104 in: *From Statistics to Neural Networks—Theory and Pattern Recognition Applications*, V. Cherkassky, J. H. Friedman, and H. Wechsler (eds.), Springer-Verlag, Berlin, 1994.

[4] N. Morgan and H. Bourlard, "Continuous Speech Recognition—An introduction to the hybrid HMM/connectionist approach," IEEE Signal Processing Mag., Vol. 13, no. 3, pp. 24–42, May 1995.

[5] H. Bourlard and C. J. Wellekens, "Links between Markov Models and Multilayer Perceptrons," *IEEE Trans. PAMI*, Vol. 12, pp. 1167–1178, 1990.

[6] H. Gish, "A Probabilistic Approach to Understanding and Training of Neural Network Classifiers," *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing* (Albuquerque, NM), pp. 1361–1364, 1990.

[7] S. Renals and N. Morgan, "Connectionist Probability Estimation in HMM Speech Recognition," TR-92-081, International Computer Science Institute, 1992.

[8] M. D. Richard and R. P. Lippmann, "Neural Network Classifiers Estimate Bayesian *a Posteriori* Probabilities," *Neural Computation*, Vol. 3, pp. 461–483, 1991.

[9] P. Simard, B. Victorri, Y. Le Cun and J. Denker, "Tangent Prop — A Formalism for Specifying Selected Invariances in an Adaptive Network," in *Advances in Neural Information Processing Systems 4*, Moody et al. (eds.), pp. 895–903, Morgan Kaufmann, 1992.

[10] P. Simard, Y. Le Cun and J. Denker, "Efficient Pattern Recognition Using a New Transformation Distance," in *Advances in Neural Information Processing Systems 5*, Hanson et al. (eds.), pp. 50–58, Morgan Kaufmann, 1993.

[11] E. I. Chang and R. P. Lippmann, "Using Voice Transformations to Create Additional Training Talkers for Word Spotting," in *Advances in Neural Information Processing Systems 7*, Tesauro et al. (eds), pp. 875–882, MIT Press, 1995.

[12] I. Guyon, D. Henderson, P. Albrecht, Y. Le Cun, and P. Denker, "Writer independent and writer adaptive neural network for on-line character recognition," in *From pixels to features III*, S. Impedovo (ed.), pp. 493–506, Elsevier, Amsterdam, 1992.

[13] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive Mixtures of Local Experts," *Neural Computation*, Vol. 3, pp. 79–87, 1991.

[14] M. Parizeau and R. Plamondon, "Allograph Adjacency Constraints for Cursive Script Recognition," *Third International Workshop on Frontiers in Handwriting Recognition* (Pre-Proceedings) , pp. 252–261, 1993.

[15] K. Asanovic´ and N. Morgan, "Experimental Determination of Precision Requirements for Back-Propagation Training of Artificial Neural Networks," TR-91-036, International Computer Science Institute, Berkeley, 1991.