

The Polynomial Method Augmented by Supervised Training for Hand-Printed Character Recognition¹

Peter G. Anderson

Computer Science Department
Rochester Institute of Technology,
Rochester, NY 14623-0887
pga@cs.rit.edu

Roger S. Gaborski

Imaging Research Laboratories
Eastman Kodak Company
Rochester, New York 14653-5722 USA
gaborski@kodak.com

¹Presented at The International Conference on Artificial Neural Networks & Genetic Algorithms "ANNGA 93," Innsbruck, Austria, April 1993.

Abstract

We present a pattern recognition algorithm for hand-printed characters, based on a combination of the classical least squares method and a neural-network-type supervised training algorithm. Characters are mapped, nonlinearly, to feature vectors using selected quadratic polynomials of the given pixels. We use a method for extracting an equidistributed subsample of all possible quadratic features.

This method creates pattern classifiers with accuracy competitive to feed-forward systems trained using back propagation; however back propagation training takes longer by a factor of ten to fifty. (This makes our system particularly attractive for experimentation with other forms of feature representation, other character sets, etc.)

The resulting classifier runs much faster in use than the back propagation trained systems, because all arithmetic is done using bit and integer operations.

1 Background

The *augmented polynomial method* is a system for classification of hand printed characters and digits such as ZIP codes, and the characters written on tax forms. This method extends the system presented by in [6] by using iteration to improve classification accuracy. It competes favorably with feed-forward neural network systems trained using back propagation.

1.1 Character Recognition

We only consider digitized characters after a preprocessing step has thinned them to standard width strokes and scaled them into a standard size (we use 30×20 pixels). (We do not address character locating or segmenting issues.)

Broadly speaking, characters of a given classification (i.e., the digit ‘1,’ the character ‘A,’ etc.) form a cluster in some high-dimensional space. An unknown character is classified by determining which cluster it belongs to, for instance, to which cluster centroid is it nearest.

A straightforward but primitive attempt at clustering is to consider the characters’ pixel arrays as points in 30×20 space, and to compute distances between these points using the L^1 or *Hamming* distance. This approach may be satisfactory for matching printed characters of a specified font where error or noise is inverted pixels. Unfortunately, plastic deformations, not inverted pixels, are the deviations from standard for hand-written characters, so this simple-minded distance calculation totally breaks down. Lines that are only slightly displaced from the standard template produce huge differences in the Hamming distance. (We performed this experiment and achieved under 60% correct classification for digits.)

A more effective approach is to determine

a collection of “features” and map the given pixel-coordinate space into a (possibly much higher dimensional) space. This feature mapping is necessarily nonlinear. If the features are well-chosen, the individual character clusters will be much more easily separable; optimistically, the clusters will be linearly separable.

Statistical pattern recognition (see, e.g., [1]) deals with selected features and analytic techniques for determining clusters and cluster membership, whereas neural network pattern recognition seeks to have the features and the discrimination criteria co-evolve through self-organization. The present approach is largely statistical pattern recognition, although it does have an element of iteratively adjusting the cluster boundaries reminiscent of Rosenblatt’s *perceptron training algorithm* [3].

Character recognition techniques, whether statistical, neural, or hybrid, work with a data base of labeled characters partitioned into a *training set* and a *testing set*. The training set is used to construct the recognizer, and the testing set to evaluate its performance. Training tries to get the recognizer to behave as well as it needs to on the training set in order to “generalize” as well as it can on the testing set.

1.2 Srinivasan’s Polynomial Method

Srinivasan’s polynomial classifier works as follows. A data base of labeled, hand written characters are converted to feature vectors, \bar{v} , and are associated with target vectors. The components of the feature vectors are F quadratic polynomials formed from the character’s pixel array to provide evidences of lines throughout the image. The target vector for each character is a standard unit vector $\bar{e}_{k(\bar{v})}$ with the $k(\bar{v})^{th}$ component equal to 1 and all

other components equal to zero, where $k(\bar{v})$ is the classification of the character. Standard numerical techniques [4] are used to determine an $F \times K$ matrix, A , to minimize the squared errors, $\sum_{\bar{v}}(A\bar{v} - \bar{e}_{k(\bar{v})})^2$, where the sum runs over all the feature vectors, \bar{v} . The *weights matrix*, A , is then used to classify unlabeled characters, by determining the largest component in the product $A\bar{w}$, where \bar{w} is the unknown character's feature vector.

Srinivasan's method rapidly creates a reasonably good classification system, achieving approximately 93% on digits that have been normalized in a 16×16 array of pixels.

2 Extending the Polynomial Method

We then iterate a process that starts with that weight matrix, A , which was determined using a small number of features and a small sample of the character data base. Subsequent passes ("epochs") identify those training exemplars that are incorrectly classified or are classified with too small a confidence (i.e., the largest component of $A\bar{v}$ fails to exceed the second largest by a sufficient threshold). These poorly classified characters are replicated in the training to strengthen their correct classification, and negative feedback is used to inhibit strong incorrect classifications.

As a result, our system is able to develop a weights matrix with performance approximately 98.8% on digits, 95.5% on upper case alphabets only, and 90% on mixed characters. We have also experimented with multiple font, machine-printed alphanumeric characters (i.e., 36 classes), and have achieved 98.7%.

3 Detailed Description

3.1 Feature Sets: Line Evidences

3.1.1 Two-Pixel Products

Srinivasan's choice of features was logical products of nearby pixels, which can easily be described in terms of the chess moves, king and big knight. A king feature is the product of two pixels that are a king's move apart; that is, any two pixels chosen from the corners of a 2×2 square of pixels. A big knight feature is the product of two pixels that are diagonally opposite in a 2×3 or 3×2 rectangle of pixels. In this design, almost every pixel can participate in eight features (pixels near the edges are used in fewer features), giving nearly 2,000 features.

We use similar features for our characters that are normalized in a 30×20 array. We describe our features as *n-king* and *n-knight* features, where the parameter n is a small integer that describes the size in number of pixels in the side of a square, and the two pixels multiplied together to form the feature are on the perimeter of that square, related to each other as stretched king or knight moves. So, there are four 5-king features, *ai*, *ck*, *em*, and *og*, and four 5-knight features, *bj*, *dl*, *fn*, and *hp*, centered at the "•," as displayed below:

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>p</i>	.	.	.	<i>f</i>
<i>o</i>	.	•	.	<i>g</i>
<i>n</i>	.	.	.	<i>h</i>
<i>m</i>	<i>l</i>	<i>k</i>	<i>j</i>	<i>i</i>

(the parameter n must be even for the *n-king* *North-South* and *East-West* features to be defined; and n must be of the form $4m + 1$ for the *n-knight* features to be defined.) We have found the following features to be useful: 3-king, 7-king, 5-knight, and 9-knight.

3.1.2 Fuzzy Features

Another type of feature we have experimented with, which seems particularly suitable for this application, we call *fuzzy line features*, and we also label them with chess moves: *fuzzy-n-knight* and *fuzzy-n-king*. As before, these features are extracted from the pixels on the perimeter of an $n \times n$ square, but they are the logical product of the logical sum of pixels. For example, the four fuzzy-3-knight features are built on three-by-three squares of pixels, centered at “•”: $(a+b) \times (h+i)$, $(b+c) \times (g+h)$, $(c+f) \times (d+g)$, and $(a+d) \times (f+i)$, as shown below:

$$\begin{array}{ccccc} a & b & c & & \\ d & \bullet & f & & \\ g & h & i & & \end{array}$$

And the four fuzzy-5-king features are built around the rim of a five-by-five square of pixels; they are: $(b+c+d) \times (j+k+l)$, $(d+e+f) \times (l+m+n)$, $(f+g+h) \times (n+o+p)$, and $(h+i+j) \times (p+a+b)$:

$$\begin{array}{cccccc} a & b & c & d & e & \\ p & \cdot & \cdot & \cdot & f & \\ o & \cdot & \bullet & \cdot & g & \\ n & \cdot & \cdot & \cdot & h & \\ m & l & k & j & i & \end{array}$$

These fuzzy features may be more reliable (lenient) at detecting the presence of lines through the pixel “•” than the first nonfuzzy features.

For hand-written digit recognition we used 1,500 features using 5-knight and 7-king. For machine printed digits and upper case alphabets (a combination of three fonts), we used 7-king, 5-knight, and 9-knight, for a total of 600 features. For upper and lower case alphabetic (a total of 40 different characters—the upper and lower case versions of the letters c, k, o, p, s, t, u, v, w, x, y, and z were treated as the same character), we used 7-king, 5-king,

and fuzzy-9-king, and a total of 1,700 features. These choices were determined experimentally.

3.1.3 Feature Collection Issues

The cost of a large number, F , of features is borne by “training,” which involves the creation and inversion of an $F \times F$ matrix (complexity $O(F^3)$), the maintenance of a weight matrix of size $K \times F$ (where there are K character classes), and the classification-time creation ($O(F)$) and multiplication ($O(F^2)$) of the feature vector.

We use feature vectors consisting of between 100 and 1,500 features, which are smoothly distributed over the set of all possible (above-described, quadratic) features (see below for a discussion of the smooth subsampling).

3.2 The Pseudo-Inverse

Our character recognition problem is cast in the following general type of framework: given a matrix, X , and a vector, \bar{y} , determine the vector, \bar{a} , that minimizes $\|\bar{r}\| = \|\bar{a}X - \bar{y}\|$. If $\bar{a}X$ is the nearest vector in the Range of X to the given vector, \bar{y} , then the difference, \bar{r} , must be orthogonal to the subspace, $R(X)$, the range of X ; so $\bar{r}X^T = 0$, and $\bar{a}X X^T = \bar{y}X^T$. If XX^T is nonsingular, then $\bar{a} = \bar{y}X^T(XX^T)^{-1}$. $X^T(XX^T)^{-1}$ is called the *Moore-Penrose Pseudo-Inverse of X* [5].

The single-vector problem presented above expands into a multiple-vector problem that we apply to the automatic character classification problem. Suppose that we have a training set consisting of N characters; hence, we have N feature vectors, $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_N$. Form the $F \times N$ matrix, X , whose i^{th} column is \bar{x}_i . Denote the given classification of the i^{th} training vector by k_i , where $1 \leq k_i \leq K$ ($K = 10$ for digit classification problems; $K = 36$ for alphanumeric classification problems). The ideal target for the i^{th} character is \bar{e}_{k_i} , the

standard unit vector in K -space. Form the $K \times N$ matrix, Y , whose columns are $\bar{e}_{k_1}, \bar{e}_{k_2}, \dots, \bar{e}_{k_N}$. We seek a $K \times F$ matrix, A , such that $AX = Y$, or rather, that AX approximates Y in the least squares sense. We determine A using the Moore-Penrose pseudo-inverse: $A = YX^T(XX^T)^{-1}$

A is the weight matrix we seek for character classification. Given an unknown character, we form its feature vector, \bar{x} , and evaluate $\bar{y} = A\bar{x}$, which is a list of “evidences” or “classification strengths” for each of the K classes to which our unknown character can belong. Hopefully, one of these K strengths stands out above the others, and we classify the character with high confidence.

3.3 Iterative Training

The training algorithm described above yields a weight matrix, A , that minimizes the sum of the squared errors, $\sum_{i=1}^N \|A\bar{x}_i - \bar{y}_i\|^2$, where the \bar{x}_i run over the feature vectors of the training data, and $\bar{y}_i = \bar{e}_{k_i}$, the k_i^{th} standard unit vector, denoting that \bar{x}_i is of class k_i . This is approximately what we want. Actually, we want the product vector, $A\bar{x}$, to have its largest component correspond to the correct classification of \bar{x} for as many as possible of the training exemplars, \bar{x} . It is not an average error we want to minimize, but the number of incorrectly classified characters.

To apply our least squares technique to a problem involving a different error criterion, we may increase the representation of the incorrectly classified exemplars and exemplars classified with low confidence (called, together *poorly classified* or *hard to learn* exemplars) in our training set.

As described above, the weight matrix, A , is determined by

$$A = YX^T(XX^T)^{-1} \quad (1)$$

Component matrices in equation (1) are maintained as

$$Z = YX^T = \sum_{i=1}^N \bar{e}_{k_i} \bar{x}_i^T, \quad (2)$$

and

$$W = XX^T = \sum_{i=1}^N \bar{x}_i \bar{x}_i^T \quad (3)$$

where N is the number of training exemplars. If the hard-to-learn \bar{x}_i are repeated with multiplicity m_k ,

$$Z = \sum_{i=1}^N m_i \bar{e}_{k_i} \bar{x}_i^T, \quad (4)$$

and

$$W = \sum_{i=1}^N m_i \bar{x}_i \bar{x}_i^T \quad (5)$$

If an exemplar, \bar{x} , of class k is ill-classified, there is another class, j , such that $(A\bar{x})_j > (A\bar{x})_k - \theta$, where θ is a confidence threshold. We attempt to lower the j^{th} classification strength for \bar{x} by specifying, on at least one epoch, that we want a negative j^{th} classification weight for this exemplar (see the algorithm outlined below).

The confidence threshold, θ , is raised or lowered from one epoch to the next to try to keep the number of ill-classified exemplars near a specified fraction of the training set. We have found that such retraining of 20% of the training exemplars works well. Retraining a larger fraction causes large oscillations. For example, that two character classes ‘S’ and ‘5’ are initially difficult to separate. When too large a fraction of characters are chosen for retraining, we find that, on one epoch all of these characters will be classified as ‘S’, and on the next epoch, all will be classified as ‘5’, and so on. With the 20% retraining fraction, a reasonable boundary between the two classes is determined quickly.

We achieve this by the following type of iteration:

initialize the matrices $Z = 0$ and $W = 0$

/ epoch #1 */*

for every \bar{x}_i

add $\bar{e}_k \bar{x}_i^T$ to Z

add $\bar{x}_i \bar{x}_i^T$ to W

compute $A = ZW^{-1}$

for $epoch = 2 \dots epoch_count$

for every ill-classified \bar{x}_i

/ the correct class k */*

/ must be strengthened */*

/ the incorrect class j */*

/ must be inhibited */*

add $(2\bar{e}_k - \bar{e}_j) \bar{x}_i^T$ to Z

add $\bar{x}_i \bar{x}_i^T$ to W

compute $A = ZW^{-1}$

During the first epoch we consider every \bar{x}_i to be ill-classified. In subsequent epochs, we use the weight matrix, A , as developed in the previous epoch.

3.4 Subsampling the Training Data

The technique of subsampling the data was originally introduced to try to speed up the training process. But it also produced better results: classifiers with better accuracy.

In order to speed up the learning process, we have experimented with subsampling the training data. We use a shuffled data collection, so that in any portion of it we will find approximately equally many exemplars of each character. Starting with a small fraction, say 10%, in epoch one; then we process 20% in epoch two, and so on.

This subsampling gives us the following advantages. Since the classifier functions linearly

(in feature space), it forms boundaries consisting of hyperplanes; the character clusters it forms are represented by convex polytopes. If we can sketch out the convex hulls of these polytopes, we may then avoid training associated with exemplars that are well inside the regions, only focusing attention on those near or on the wrong side of the boundaries.

3.5 Enlarging the Training Set

The training set needs to be big and representative of the characters we want to recognize. However, a lot of the variation in characters can be explained in terms of simple modifications of characters. The technique we use is to translate a given character by eight “king’s moves” (i.e., up one pixel, up and right one pixel, right one pixel, etc.), giving nine training exemplars for the price of one.

The effect of this is that the system learns the training data much more slowly, and its performance on the testing data consequently improves greatly. Because of the way we train and retrain, once the system’s performance on the training data is very good, then very little additional training can take place.

3.6 Feature Subsampling

We have discovered that the sequence of pixels with coordinates $(11k \bmod 28, 11k \bmod 19)$ for $k = 1, \dots, n$ is equidistributed in the 30×20 rectangle of pixels. If we are using an n -king and an m -knight feature, we have approximately eight features centered at every pixel we choose using the above rule. This rule gives us an even covering of the whole image with features, even if the number of features chosen is very small.

4 Implementation Details

4.1 Pseudo-Inverse Calculations

The computation involving the pseudo-inverse, $A = YX^T(XX^T)^{-1}$, seems to require several arrays be built, multiplied, and inverted. Some of these matrices are very large. For example, if we are dealing with N training patterns and F features, and K character classifications, the matrices have the following dimensions:

array	dimensions
A	(K, F)
Y	(K, N)
X	(F, N)

However, because of the simple nature of these matrices and the contexts in which they are used, neither X nor Y needs to be explicitly stored. Recall their definitions: the k^{th} column of Y is a unit vector giving the classification of the k^{th} training pattern; the k^{th} column of X is the binary feature vector of the k^{th} training pattern. Instead of storing X and Y explicitly, we build, on the fly, the matrices $Z = YX^T$ and $W = XX^T$, of dimensions (K, F) and (F, F) , respectively.

We initialize Z to zero; for each training pattern, if its classification is j , we add its feature vector to the j^{th} column of Z .

We initialize W to zero; for each training exemplar's feature vector, \bar{x} , we add its outer product square, $\bar{x}\bar{x}^T$, to W ; that is, add $x_i x_j$ to W_{ij} for every subscript pair, i, j . An outer product square is symmetrical, so W is symmetrical, and we only have to accumulate W_{ij} for $j < i$, i.e., on the lower triangle. Since the feature vector is a sparse binary vector, we only have to add it to selected rows of W .

An important consequence of this elimination of the explicit storage of X and Y is that there is no storage penalty for a large training set.

4.2 Feature Caching

After the above improvement was made to the program, feature extraction—determining an image's feature vector—turned out to be the most time-consuming step in the training system. Consequently, we create files of feature vectors for the training and testing of images. The feature vectors are sparse binary vectors, so we only store the list of indices with nonzero values. The feature vector files are big, and they take a long time to create, but that creation only happens once, and the files can be reused for other experiments.

5 A Growing Machine

Because of our ordering of the pixel centers for feature extractions, a feature vector with F features is a prefix of a feature vector with $F+F'$ features. Consequently, the corresponding matrices, $Z = YX^T$ and $W = XX^T$, for a small feature vector are submatrices of those for larger feature vectors. The following training algorithm naturally suggests itself.

We determine the largest size feature vector that we believe appropriate (e.g., 1,500 features), and gather feature vectors of that size for every character we process, either in training or testing. However, we will be maintaining classification weight matrices that use only a prefix of that vector, and testing may ignore a feature vector's postfix. For a character used in training, if that character fails to be classified properly by the current weight matrix, its full feature vector will be used to update Z and W . As described above, an epoch consists of a (possibly subsampled or supersampled) pass over the training data. After each epoch, a larger submatrix of Z and W is chosen to compute the new weight matrix.

We obtain several advantages from this system. Since the inversion of W costs time

that is proportional to the cube of the number of features actually used, we will be able to rapidly sketch out a classifier that approximates the classifier we will eventually derive. The inexpensive early classifiers allow us to easily filter the training set, avoiding the cost of determining the outer product squares of feature vectors for correctly classified characters. This eliminates another bottleneck.

The form that our training algorithm takes is shown in Figure 5. In this algorithm:

- F features are used to build Z and W , but only f features are used in classification.
- W_f denotes the upper left $f \times f$ submatrix of W .
- Z_f denotes the first f columns of Z .
- A_f denotes the resulting $K \times f$ weights matrix.

6 Two Training Sessions

6.1 Digits

We trained the polynomial classifier to recognize hand-written digits, using 143,258 training and 15,971 testing exemplars. The training exemplars were shifted to give us four new exemplars for each given one, for a total training set size of 716,290.

The fuzzy-7-knight and the fuzzy-5-king features types were used for 1,500 features. The training started with a feature subset (the f parameter) of 400, and a training exemplar subset of 47,753.

The performance of the training session is shown in Table 1. We found in this example that the artificial enlargement of the training set by shifting kept the training set from ever performing better than the testing set, so that the system never “memorized the noise in the training data” and successfully generalized its learning to the testing data.

The training session took two days on a

SUN SPARCstation 2. A feed-forward network trained using back propagation with similar performance took approximately a month to train.

6.2 Machine Print

Our second experiment here involved machine-printed alphanumeric (36 character classes), and is shown in Table 2. The training set consisted of 22,500 exemplars which was created from a set of 2,500 characters by shifting.

The 5-knight, 7-king, and 9-knight features types were used for 600 features. The training started with a feature subset (the f parameter) of 500, and a training exemplar subset of 2,249. This training session took 68 minutes.

The performance of the training session is shown in Table 2.

References

- [1] Duda, R. O., and Hart, P. E., *Pattern Classification and Scene Analysis*, John Wiley & Sons, New York, 1973.
- [2] Marvin L. Minsky and Seymour A. Papert, *Perceptrons, Expanded Edition*, The MIT Press, 1988.
- [3] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, *Numerical Recipes in C.*, Cambridge University Press, 1988.
- [4] G. W. Stewart, *Introduction to Matrix Computation*, Academic Press, 1973.
- [5] Uma Srinivasan, “Polynomial discriminant method for handwritten digit recognition,” *SUNY Buffalo Technical Report*, December 14, 1989.

epoch	tss	retrained	ratio	features	testing	training
1	47753	47753	100	400	95.08%	93.67%
2	95507	15714	16	500	97.08%	95.99%
3	143257	16364	11	600	97.31%	96.67%
4	191012	20153	10	700	97.66%	97.16%
5	238762	21273	8	800	97.85%	97.37%
6	286516	25825	9	900	98.11%	97.65%
7	334269	39945	11	1000	98.28%	97.72%
8	382022	31215	8	1100	98.22%	97.93%
9	429775	34003	7	1200	98.26%	97.99%
10	477527	38533	8	1300	98.23%	98.13%
11	525278	40770	7	1400	98.33%	98.17%
12	573032	57852	10	1500	98.38%	98.24%
13	620785	42283	6	1500	98.60%	98.45%
14	668537	43790	6	1500	98.65%	98.53%
15	716290	46752	6	1500	98.65%	98.59%
16	716290	46953	6	1500	98.68%	98.62%
17	716290	47573	6	1500	98.69%	98.64%
18	716290	47900	6	1500	98.68%	98.66%
19	716290	48027	6	1500	98.72%	98.67%
20	716290	48800	6	1500	98.73%	98.68%
21	716290	48062	6	1500	98.71%	98.69%
22	716290	48961	6	1500	98.72%	98.69%
23	716290	48392	6	1500	98.74%	98.70%
24	716290	49373	6	1500	98.74%	98.70%
25	716290	48994	6	1500	98.73%	98.70%
26	716290	49202	6	1500	98.74%	98.71%
27	716290	49245	6	1500	98.75%	98.71%
28	716290	49355	6	1500	98.75%	98.71%
29	716290	49423	6	1500	98.75%	98.71%
30	716290	49754	6	1500	98.74%	

Table 1: Hand-written digit training session. The second column (“tss”) shows a growing subset of the training exemplars. The third column (“retrained”) shows how many “ill-classified” exemplars there were, with the “ratio” column giving the ill-classified percentage. The first ten epochs used a subset of the 1,500 features to sketch out a classifier. The last two columns give the classifier’s performance on testing and training.

```

initialize the matrices  $Z = 0$  and  $W = 0$ 
initialize  $f$ , the number of features to be used

for  $epoch\_count$  iterations
  for every training exemplar's feature vector,  $\bar{x}$ 
    if  $\bar{x}$  is poorly classified by the current  $A_f$ 
      add  $(2\bar{e}_k\bar{x}_i - \bar{e}_j)^T$  to  $Z$ 
      add  $\bar{x}_i\bar{x}_i^T$  to  $W$ 
  increase  $f$ 
  compute  $A_f = Z_f W_f^{-1}$ 

```

Figure 1: The training algorithm for “the growing machine.”

epoch	tss	retrained	ratio	features	testing	training
1	2249	2249	100	510	96.73%	90.53%
2	4499	910	20	520	91.33%	87.48%
3	6750	5760	85	530	95.58%	93.19%
4	9000	1712	19	540	98.20%	97.61%
5	11248	6128	54	550	98.04%	97.58%
6	13500	1383	10	560	98.69%	99.07%
7	15750	10846	68	570	98.85%	98.38%
8	18000	2732	15	580	98.69%	99.08%
9	20250	7826	38	590	98.85%	99.11%
10	22500	4548	20	600	98.53%	99.30%
11	22500	6119	27	600	99.02%	99.35%
12	22500	1779	7	600	98.69%	99.40%
13	22500	4723	20	600	99.02%	99.46%
14	22500	2404	10	600	98.69%	99.45%
15	22500	4529	20	600	99.02%	99.51%
16	22500	4081	18	600	98.69%	99.44%
17	22500	4595	20	600	98.69%	99.44%
18	22500	4318	19	600	98.69%	99.44%
19	22500	5545	24	600	99.02%	99.50%
20	22500	3108	13	600	98.69%	

Table 2: Machine-printed alphanumeric training session. (See the caption for Figure 1.)