

# Image Segmentation and Recognition

John S. Denker  
Christopher J. C. Burges

## Abstract

We have constructed a system for recognizing multi-character images. This is a nontrivial extension of our previous work on single-character images. It is somewhat surprising that a very good single-character recognizer does not in general form a good basis for a multi-character recognizer. The correct solution depends on three key ideas:

- 1) A method for normalizing probabilities correctly, to preserve information on the quality of the segmentation;
- 2) A method for giving credit for multiple segmentations that assign the same interpretation to the image; and
- 3) A method that combines recognition and segmentation into a single adaptive process, trained to maximize the score of the right answer.

We also discuss improved ways of analyzing recognizer performance.

A major part of this TM is devoted to giving our methods a good theoretical footing. In particular, we do *not* start by asserting that maximum likelihood is obviously the right thing to do. Instead, the problem is formalized in terms of a probability measure; the learning algorithm must then be arranged to make this probability conform to the customer's needs.

This formulation can be applied to other segmentation problems such as speech recognition.

Our recognizer using these principles works noticeably better than the previous state of the art.

## 1 Overview

We begin with a terse overview. Many of the buzzwords used here will be defined more clearly in the following sections.

The task at hand is to build an image recognizer. The input to the recognizer is a multi-character image, represented as an array of pixels. The desired output is the best interpretation of the image, along with a good estimate of the probability that the interpretation is correct. For example, given the ZIP Code image shown in figure 1, the best interpretation is the five-numeral string "35133".



Figure 1: A typical image

As mentioned in the abstract, our recognizer incorporates three key ideas, as discussed in the following three subsections.

## 1.1 Normalization of Probabilities

Our first design for a multi-character recognizer (MCR) naturally used our trusty single-character recognizer (SCR) as a building block. We have since discovered that this is not quite optimal. Specifically, the normalization which is appropriate for an SCR throws away information which is needed for proper segmentation.

The correct strategy has two steps:

- 1) Compute a score for each possible multi-character path (where a “path” specifies the interpretation *and* segmentation) by combining the scores of the individual characters (and whatever other information is available).
- 2) Normalize these scores by dividing by the sum over all paths.

We present below a firm theoretical support for this scheme. In particular, it would be clearly non-optimal to perform the required operations in the reverse order:

- 1') Normalize on a per-character basis.
- 2') Form a multi-character score by multiplying the normalized per-character scores.

## 1.2 Multiple Segmentations

The normalized score for a path gives the joint probability of a given interpretation *and* segmentation. But at the end of the day, the customer cares about the correct interpretation; the segmentation problem is just a step along the way. Therefore we compute the score for a given interpretation by summing over all paths that give that interpretation. This, too, has a firm theoretical foundation. The required sum can be performed very efficiently using the “forward” [10] algorithm.

The difficult step is that we must find the *best* interpretation, i.e. the interpretation that maximizes the aforementioned sum over paths. The Viterbi algorithm [2, 3] can efficiently find the best-scoring path, but it is harder to find the best *sum* directly in closed form. If we could assume that each sum was dominated by its largest term, Viterbi would be exact. In practice, we find that Viterbi, while not always exact, is good enough to identify the one or two interpretations that are *candidates* for having the best sum. We can then run the forward algorithm to check these candidates, i.e. to compute their exact score.

## 1.3 Learning

For a single-character recognizer, we use a neural net to produce a score that reflects the conditional probability of an interpretation, *given* the segmentation. For a multi-character recognizer, we need more detailed information, namely the joint probability of interpretation *and* segmentation. Since the training process determines what the network will actually do, we have designed a training process that takes the needs of the MCR into account.

Specifically, the block diagram of the MCR consists of a large number of SCRs (each of which evaluates the score of a segment) plus a lattice (that finds the best segmentation, i.e. the best way of combining segments). This idea [12, 13, 14, 15, 16] has been around for some time. Obviously “backprop” can be used to train the SCR, but it has not been 100% obvious how to choose the training targets that backprop (supposedly) needs. The new scheme is to treat the whole MCR (lattice and all) as one huge network and perform backprop (or a generalization thereof) on the whole thing.

We use the Baum-Welch algorithm [10] (also known as forward-backward or E-M) to calculate derivatives. This tells us how sensitive the output is to each adjustable parameter in the MCR. Baum-Welch is to forward as backprop is to fprop.

In essence: during training, each parameter is adjusted in a direction that will increase the score of the correct answer. (Since the answers are normalized, this automatically decreases the score of wrong answers.)

This training scheme has two advantages:

- i) As mentioned above, it teaches the neural net to produce information about the quality of the segmentation in addition to information about what the interpretation would be if we were given the correct segmentation.
- ii) It teaches the network to avoid *near mistakes*, not just *mistakes*. That is: under the old training scheme, the highest-scoring path was identified. If it was incorrect, the training process would lower its score. As soon as the wrong path's score dropped below the right path's score, no further training could be applied, unless the wrong path had the same segmentation as the right path. Fortunately for us, in most cases the segmentations *were* the same, so the old training scheme worked quite well in practice. In those cases where the segmentations were different, the training could well produce bad segmentations with scores only infinitesimally lower than the good segmentation — leading to poor robustness. The new training scheme does not have this vulnerability. Since it involves a sum over paths, not a max over paths, it pushes all good paths up and pushes all bad paths down, and keeps pushing regardless of what the best path happens to be.

And finally, the bottom line: we have implemented these ideas and the resulting recognizer just plain works better, as will be discussed in section 5.

## 2 The Dynamic-Programming Approach to Segmentation

For some time now we have had a good single-character recognizer (SCR). If all images were naturally divided into segments containing one character apiece, the multi-character recognizer (MCR) would be straightforward. But in general, real images contain (a) too little connectivity, i.e. fragmented characters consisting of multiple disconnected strokes, and (b) too much connectivity, i.e. characters that nestle, touch, and overlap. We saw both of these problems back in figure 1 – the horizontal stroke that is supposed to form the top of the “5” is disconnected from the body of the “5” and is connected to the following “1” instead. As another example, figure 2 contains several touching characters, and one character nestled under the overhang of another.



Figure 2: Touching and Nestling

The strategy may be summarized as follows

1. Divide the image into cells of manageable size.
2. Combine the cells in various ways to form “segments”
3. Form a sequence of segments (called a “segmentation”) that accounts for the whole image.
4. Assign scores to various possible segmentations.
5. Find the interpretation of the highest-scoring segmentation(s).

One significant advantage of this strategy is that the system does not have to generate the correct segmentation initially: it merely has to create a set of hypothetical segmentations which *contains* the correct segmentation(s)[7, 8]. Previous approaches to this problem envisioned a “pipeline” architecture in which the segmentation process was completed before the single-character recognition process began; in contrast we envision a system in which the segmenter calls a modified single-character recognizer as a subroutine.

The “second generation” segmenter is described more fully elsewhere[7, 8] but will be briefly described here. It starts by cutting the image into cells. The cuts are not necessarily vertical. While the “first generation”

segmenter[6] makes extensive use of “connected component analysis” (CCA) the second-generation system does not; we are working on a segmenter that unifies the cut generation and CCA.

One or more cells are combined to make a segment. The objective is to create segments that contain exactly one character. The second-generation segmenter uses cells in a left-to-right order and only makes segments out of *consecutive* cells; there are one or two examples (out of several thousand) in the training set where this restriction is non-optimal, because one character overhangs another.

Since the number of ways that cells can be combined to make a segment grows combinatorially with the number of cells, we cannot afford to chop the image into ultra-small cells. Therefore the preprocessor embodies heuristics which will identify a set of “good” cuts, including some “obvious” cuts. The cells between good cuts are generally much more than one pixel wide. The set of good cuts will generally contain more cuts than needed.

The image in figure 1 was divided into the seven cells shown in figure 3.



Figure 3: The cells

Segments are sometimes called boxes, in analogy with the “boxes” and “glue” used by the typesetting language  $\text{\TeX}$ .

A segmentation is defined to be a sequence of boxes. Ideally the boxes would abut one another left to right, but in practice we must allow for “glue” between boxes. Positive glue means a sliver of image between the two boxes is skipped; negative glue means two boxes overlap; zero glue means they abut.

Figure 4 shows the segments constructed for the example image; in the corner of each image is a list of numbers designating the cells from which the segment was constructed.

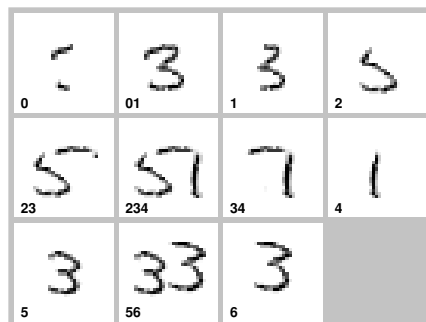


Figure 4: The segments

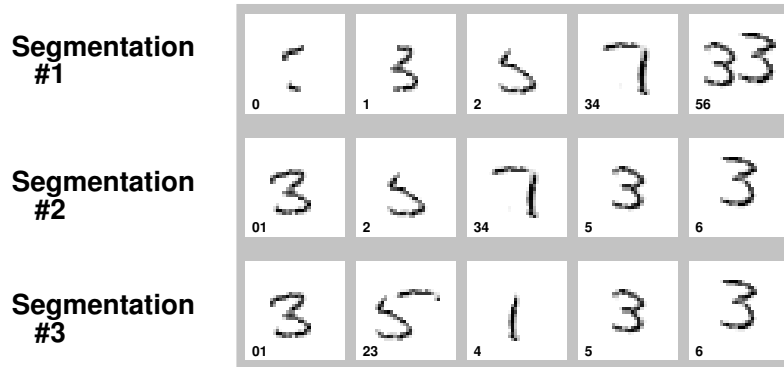


Figure 5: Three segmentations

Figure 5 shows three of the many possible segmentations of our example image. Interestingly, the (2) segment appears in the third slot of the first segmentation and the second slot of the second segmentation. Thus the segmentation algorithm cannot simply keep track of which segments are used; it must keep track of which segment appears in which slot of the answer.

It is advantageous[8, 7] to formulate the segmentation problem as a “best path through graph” problem, as we will now summarize.

We will distinguish between the input space (i.e. image pixels) and the output space (i.e. character interpretations). Borrowing some of the methods used[2] for segmenting speech signals, we will speak of the output space in terms of “time slots.” In the previous examples there were five time slots — because we are expecting a five-digit answer.

At this level of description we have a two-dimensional array of nodes, indexed by (segment-ID, time-slot), where segment-ID is an index into the set of allowed segments for this image.

This can be drawn as shown in figure 6, where each • is a node in the graph. Note there is a special start-node before the first time and to the left of the leftmost segment, and a special end-node after the last time and to the right of the rightmost segment. The arcs constituting all valid segmentations are also shown.

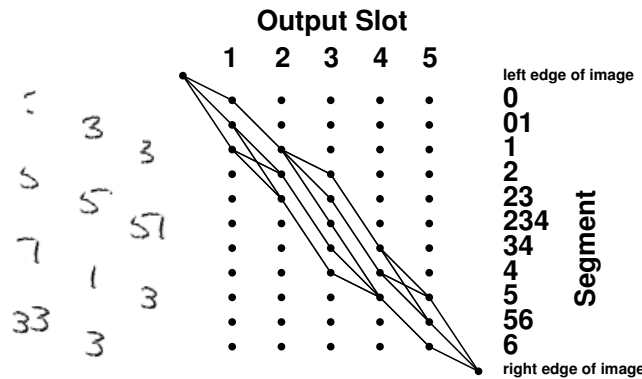


Figure 6: Basic Lattice

For reasons that will become clear in a moment, we subdivide the time axis, as shown in figure 7; the label m is for morning and e is for evening. In this graph, there are two kinds of arcs: glue arcs and recognition arcs. All arcs are directed; we think of them as left to right, although one could just as well solve the problem right to left instead. Recognition arcs (rec-arcs) are daytime arcs, from a morning to an evening. Glue-arcs are nighttime arcs, from an evening to the next morning.

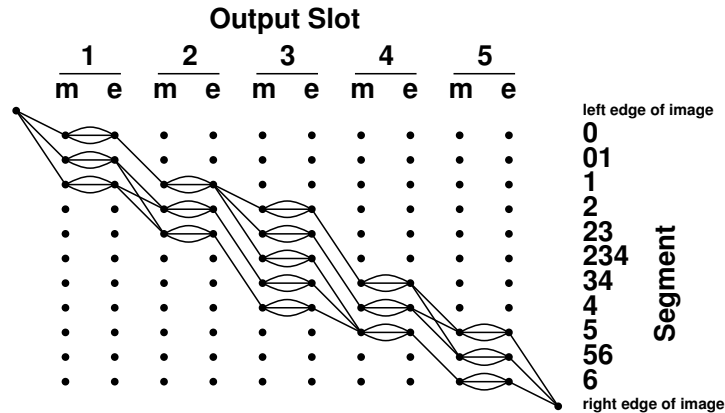


Figure 7: Expanded Lattice

## 2.1 Glue arcs

A given evening node will generally have several glue arcs leaving it. For example, a (1 2) node might originate the following arcs:

- (1 2)  $\rightarrow$  (3) “abut”
- (1 2)  $\rightarrow$  (3 4) “abut”
- (1 2)  $\rightarrow$  (4) “skip”
- (1 2)  $\rightarrow$  (2 3) “overlap”

The parameters of our implementation are currently set to allow only abutment, and possibly skips connected to the start and end nodes.

The number of arcs can be reduced by using the information about obvious cuts; no segment can straddle an obvious cut. This is technically a statement about node-building, but it has a big impact on arc-building, because we do careful pruning of nodes and arcs. We designate “live nodes” as ones that are descendants of the start node and (simultaneously) ancestors of the end node. Arcs to or from a non-live node are deleted from the graph, to prevent unnecessary computation.

## 2.2 Recognition Arcs

We assign each rec-arc four attributes: origin-node, destination-node, interpretation, and score. Note that we can have more than one arc connecting a given pair of nodes; indeed recognition arcs come in groups of  $N$ , where  $N$  is the size of the recognition alphabet ( $N = 10$  for digit recognition). For clarity, only  $N = 3$  recognition arcs per group are drawn in figure 7.

As shown in the figure, recognition arcs connect the morning and evening nodes of a given segment. The score of each rec-arc is determined by calling the single-character recognizer on the corresponding segment, i.e. box of pixels. There is only one call to the SCR per row of the lattice.

## 2.3 Paths, Segmentations, Interpretations

We will consider paths connecting the start-node to the end-node; the path will be a sequence of arcs of the form [G R G R G R G R G] where G stands for glue-arc and R stands for rec-arc.

In this formalism the segmentation corresponding to a given path is independent of the rec-arcs and consists of the list of segments visited by the glue arcs in the path.

Analogously, an “interpretation” is a string of characters, e.g. “07733”. For a five-digit string, there will be

$10^5$  possible interpretations per segmentation. For each path, the segmentation is formed by concatenating the interpretations of the rec-arcs making up the path.

The score of a path is formed by multiplying the scores of the constituent glue-arcs and rec-arcs. Presently all glue-arcs have a score of 1.0; the rec-arc scores are computed by a neural network as discussed in section 4.

To summarize this subsection: if you take a path and ignore the rec-arcs, you get a segmentation consisting of a list of glue-arcs; conversely if you take a path and leave out the glue-arcs, you get an interpretation, which consists of a list of rec-arcs.

### 2.4 Runner-up Information

There are a number of situations in which it is useful to know the scores for more than one interpretation of the image. If nothing else, it is useful during training to know how the bad-guy scores are changing relative to the good-guy scores. (A training scheme that increases the good-guy scores is a failure if it increases bad-guy scores by the same amount.)

The famous Viterbi algorithm[2, 3] operates in terms of paths. It will find the best-scoring path and tell you its score. There exist algorithms that will find the  $K$  best paths through the lattice, but as likely as not, many of these paths will have the same interpretation as the best one (i.e. different segmentations with the same interpretation). It could be quite inefficient to enumerate all these paths and then search for one that differs in interpretation. Therefore we now present an algorithm that directly finds the best path *that differs in interpretation* from the very best path.

We will do this using a lattice that has twice as many nodes as the lattice in figure 7. One part of the new lattice will be called the “gold plane” and will operate as previously described to identify the “gold path,” i.e. the very best interpretation of the image. The other part of the new lattice will be called the silver plane and will be used to compute the “silver path,” i.e. the best path that differs *in interpretation* from the gold path.

The nodes in the new lattice are identified by a 4-tuple of variables (segment-ID, time-slot, m/e, g/s), where m/e specifies morning versus evening, and g/s specifies gold versus silver.

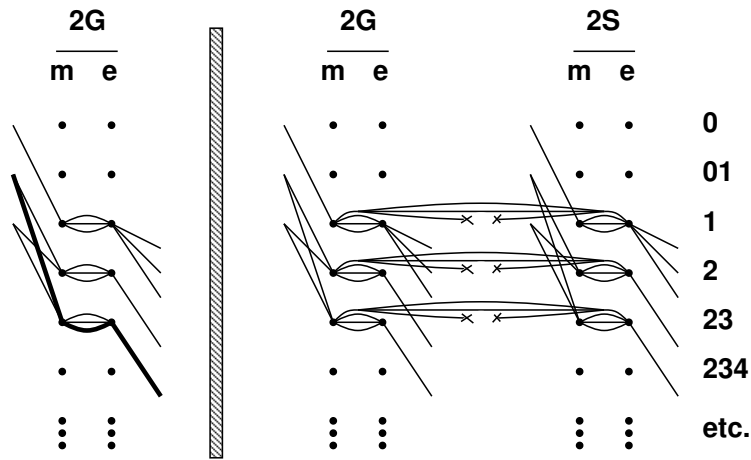


Figure 8: Runner-up Lattice

The left part of figure 8 is a close-up view of part of gold plane. The ordinary Viterbi algorithm has been used to identify the gold path, which is shown with a heavy line. The arcs connecting the gold plane to the silver plane are disabled while the gold path is being computed, and are not shown in this part of the figure.

After the gold path has been identified, the system is ready to compute the silver path. Each evening

node in the silver plane has  $2N - 1$  incoming arcs:  $N$  from the corresponding morning node in the silver plane, and  $N - 1$  from the corresponding morning node in the gold plane. The missing arc is the one whose interpretation matches the gold path’s interpretation for that output time-slot. This is portrayed in the right part of figure 8. The column headed “2G” represents time slot 2 in the gold plane, and the column headed “2S” is the corresponding part of the silver plane.

Glue arcs stay in their plane.

There is only one start node, and it is in the gold plane. The end node of the gold path is in the gold plane, and the end node of the silver path is defined to be in the silver plane. Any path that reaches the silver plane must have differed (in interpretation!) from the gold path in at least one place, because the only arcs that go from one plane to the other are restricted to differ in the required way. It is of course possible that the silver path differs from the gold path in more than one slot.

The computation for the silver path is of the same order of complexity as for the gold path — which means it is negligible compared to the non-recurring cost of computing the rec-arc scores. Also note that during the computation of the runnerup, no nodes in the gold plane need be updated.

### 3 Assigning Probabilities to Competing Interpretations

We assign a number called an “ $R$ -score” to each possible outcome of the segmentation/recognition process. In general,  $R$  will be a function of the interpretation  $C$ , the segmentation  $S$ , and the image  $I$ .

In the first part of this section we will be intentionally vague about the meaning of the  $R$ -score, in order to maximize the generality of some theoretical results. But to fix ideas, it won’t hurt if you visualize  $R(C, S, I)$  as equaling the probability  $P(C, S, I)$ , i.e. the joint probability that the data set contains a data point where character-string  $C$  and segmentation  $S$  are attached to image  $I$ . For starters, we will explicitly require that  $R$  be non-negative.

As discussed in the previous section, since we are building upon our single-character recognizer (SCR), one way of constructing the  $R$ -score of a segmentation would be to multiply the  $r$ -scores of the constituent segments, where the  $r$ -score (with a little  $r$ ) is computed by the SCR. Fancier factorizations of  $R$  will be considered below.

Let us assume for the moment that the SCR produces properly normalized  $r$ -scores. That is, for each single character, the  $r$ -scores add up to unity. This will guarantee that  $R$ -scores of a given segmentation will also be normalized. Therefore we don’t need to worry about the case where one interpretation has an  $R$ -score of .99, while some other interpretation has an  $R$ -score of .95 (which would be a problem since the alternatives would add up to more than unity).

On the other hand, there is no satisfactory way the SCR can be normalized so that scores of *different* segmentations can be compared. In particular all the segments in the second and third segmentations shown in figure 5 would receive good  $r$ -scores, so both segmentations would be assigned good  $R$ -scores.

It absolutely will not suffice to normalize these  $R$ -scores by dividing by the total of all  $R$ -scores. This is infeasible to compute and gives absurd results anyway.

Therefore we must retract the assumption that the  $r$ -scores are normalized (i.e. per-character normalization) and find a new way to normalize the  $R$ -scores (i.e. per-image normalization).

To formalize the task: we want to find the most-probable character string, and estimate its probability, given the image.

We will use the following notation:

- $I$  := the “image”
- $C$  := the “interpretation” (a string of characters)
- $C_t$  := the character in the  $t$ th position of  $C$
- $S$  := the “segmentation” (a string of segments)
- $S_t$  := the segment in the  $t$ th position of  $S$

The single-character recognizer returns an  $r$ -score which depends on  $S_t$ ,  $C_t$ , and  $I$ . Note: it is more common to think of the recognizer as returning a vector of scores, depending on  $S_t$  and  $I$ , but it is formally equivalent and more convenient for present purposes to pick out the score for each  $C_t$  separately. For example, if we are considering the image in figure 1 and its segment (01) as shown in figure 4, then  $r(\text{“3”}, \text{seg}, \text{img})$  will be large while  $r(\text{“1”}, \text{seg}, \text{img})$  will be small.

Using a set of SCRs (plus whatever else you can think of) let’s assume we can assign a score to a given segmentation and interpretation; call it  $R(C, S, I)$ . Arrange it to be positive by construction. As will be discussed later, we want to train the network so that  $R$  will be more or less a probability, but we don’t want to write it as  $P(C, S, I)$  just yet.

Remark: there is an axiomatic definition of probability measure. Specifically, a *measure* is a mapping from sets to numbers; it must be positive, and the measure of the union of disjoint sets must equal the sum of the measures of the ingredients. A *probability measure* has the further property that it is bounded above; typically we arrange this bound to be unity. Anything that satisfies the axioms can justly be called “a” probability; we will be using several such probabilities, and must be careful not to call any of them “the” probability without qualification.

John Bridle[4, 5] suggested computing the quantity

$$Q(C|S, I) := \frac{R(C, S, I)}{\sum_{C'} R(C', S, I)} \quad (1)$$

where the sum runs over all possible interpretations. This is called “softmax” normalization.

It is easy to verify that  $\sum_C Q(C|S, I) = 1$ , and that  $Q(C|S, I)$  is positive. Therefore  $Q$  satisfies the axioms of probability, and the placement of the “given” bar on the LHS of equation 1 is consistent with the usual notation of conditional probabilities. For a single-character recognizer where the segmentation  $S$  can be considered a “given,” this scheme (with an elaboration to be discussed below) is an excellent way of doing the normalization. For suitable functions  $R$ , this gives a good estimate of the actual probability of the interpretation  $C$  being correct.

Many people were surprised to discover that softmax scores cannot serve as the basis of a multi-character recognizer (MCR). To see this, compute instead

$$Q(C, S|I) := \frac{R(C, S, I)}{\sum_{C', S'} R(C', S', I)} \quad (2)$$

and thence

$$Q(C|I) := \sum_S R(C, S|I) \quad (3)$$

The last formula is very important because the probability  $P(C|I)$  (or the expected cost based thereon) is what the customers care about! In particular they often want to know the max and argmax (w.r.t.  $C$ ) of  $P(C|I)$  for each image  $I$ . If we are to have any hope that  $Q$  will be a good estimate of  $P$ , it must be normalized correctly.  $Q(C|I)$  cannot be computed by summing the softmax result  $Q(C|S, I)$  over  $S$ ; the given-bar is on the wrong side of  $S$ .

To make this problem clear, consider (again) the competing segmentations in figure 5. Segmentations #2 and #3 are the serious contenders. Most humans prefer segmentation #3, (with interpretation “35133”)

over segmentation #2 (with interpretation “35733”). Segment (34) which appears in segmentation #2 is a perfectly good “7”; so the preference must be based on the judgement that segment (2) is not a good “5”.

Humans can make this judgement, but an MCR based on a softmax SCR will have trouble. The problem is that the conditional probability is nearly 100% that *if* the pixels in segment (2) represent a digit *then* it must be a “5.” Since softmax provides just such a conditional probability, it is doomed.

The MCR needs the *joint* probability that the segmentation is correct *and* the interpretation is correct. Softmax throws away the segmentation-quality information prematurely.

To quantify this, let us factor  $R(C, S, I) = T(C, S, I) U(S)$ , where  $T(C, S, I)$  is constructed to carry very little information about the quality of the proposed segmentation  $S$  — that information being carried instead by  $U(S)$ . To fix ideas, imagine  $T(C, S, I)$  to be the conditional probability  $P(C, I|S)$ , while  $U(S)$  is the “marginal”  $P(S)$ . Plugging in, we get:

$$Q(C|S, I) := \frac{T(C, S, I)U(S)}{\sum_{C'} T(C', S, I)U(S)} \quad (4)$$

Alas, the factor  $U(S)$  drops out of this expression. In contrast, the  $S$ -dependence does not drop out of equation 2, since it has an  $S$  in the numerator and an  $S'$  in the denominator.

As foreshadowed above, let us assume (with modest loss of generality) that the recognition results are context-independent. That is, we assume that the recognition of a given character is independent of the other characters. That allows us to factor  $R$  as

$$R(C, S, I) = \prod_t r(C_t, S_t, I) \quad (5)$$

where the product runs over all output slots  $t$  (e.g. 1 through 5, for ZIP Codes). The output slots are indexed by the variable  $t$  to suggest “time” in analogy to speech segmentation models.

Additional factors expressing the “quality of segmentation” (e.g. the score of glue arcs as discussed in section 2) have been omitted from this expression for clarity; restoring them to this and subsequent expressions is routine.

We will structure the SCR to ensure that  $r$  is positive. As discussed in section 4, we will train the SCR to produce  $r$  values that are an increasing function of the probability that the (individual) character  $c$  goes with (individual) segment  $s$ .

Plugging in, we expect that the customers will want to know the max and argmax (w.r.t.  $C$ , for a given  $I$ ) of the quantity

$$Q(C|I) = \frac{\sum_{S''} \prod_t r(C_t, S''_t, I)}{\sum_{C'} \sum_{S'} \prod_t r(C'_t, S'_t, I)} \quad (6)$$

It is useful to consider the equivalent expression

$$Q(C|I) = \frac{\sum_{C''} \sum_{S''} \prod_t r(C_t, S''_t, I) \delta_{C''}^C}{\sum_{C'} \sum_{S'} \prod_t r(C'_t, S'_t, I)} \quad (7)$$

where  $\delta$  is the Kronecker delta. This form makes it clear that both numerator and denominator are “lattice sums”; i.e. the sums run over all paths in the lattice. The summand is slightly different for the numerator and denominator. Such sums can be computed very efficiently via the “forward” algorithm.

The forward algorithm is very similar to the Viterbi algorithm. They have the same computational complexity; the former computes a sum where the latter computes a max.

The first step in computing  $\max Q(C|I)$  is to compute the denominator using the forward algorithm, which does exactly what we want. The denominator is independent of  $C$ .

As for the numerator, we need to compute the sum *and* maximize over all interpretations  $C$ . For any particular  $C$ , it is fast (using the forward algorithm) to compute the sum. It is also fast (using Viterbi) to compute the max (over all  $C$ ) of the summand. Viterbi will not compute the max of the whole sum, which is what we would like.

To paraphrase Abraham Lincoln: we can easily maximize one term over all paths, and we can easily evaluate all the terms for some of the paths, but we can't so easily maximize all the terms over all the paths.

We could just approximate the sum by its largest term, in which case Viterbi would give us the answer. A better approximation is to use Viterbi to find the interpretation that contributes the largest term to the sum, and then use forward to evaluate *all* the terms with that interpretation. This gives the exact score  $Q_1$  for that interpretation; the only question is whether it is the genuine maximum. Since  $Q$  is manifestly normalized, if  $Q_1$  is greater than one-half, we know there can be no other possibility.

If  $Q_1$  does not “use up” enough of the probability measure, we can use second-interpretation Viterbi (as discussed in section 2) to identify another contender, and then use forward to compute its score. In principle this process could continue indefinitely. Let  $Q_*$  denote the sum of the scores of the already-checked interpretations; if at any point  $Q_*$  differs from unity by less than the best score already found, then we have certainly identified the winner (and precisely computed its score).

This technique is known in the speech recognition community, but is rarely used, apparently because the number of candidates gets out of hand. In our case, though, one or two candidates usually suffice to “use up” essentially 100% of the probability, at which point we know our method has found the exact answer.

There exist other schemes for finding and/or approximating the max of the sum, but we won't discuss them here.

## Cooperating Segmentations

To reiterate, our MCR assigns scores to interpretations. The score is a ratio: the denominator is a sum over all paths, and the numerator is a restricted sum, restricted to paths having the given interpretation.

We now explain why it is advantageous to evaluate the restricted sum in the numerator, as opposed to making the Viterbi approximation that the sum is equal to its largest term. Consider the two segmentations [(01) (23) (4) (5) (6)] and [(1) (23) (4) (5) (6)]. Since we see from figure 4 that segment (01) and segment (1) are both perfectly good “3”s, both segmentations will receive high scores; let's assume their scores are essentially equal. Both paths will contribute to the sum in the denominator. If only one of them contributed to the numerator, the score would be reduced by a factor of two — a very significant amount.

## 4 Training

In the previous section, we showed that for any  $r$ -score function, the  $Q$  function derived from it would be normalized like a probability, so it could be considered “a” probability in the abstract, formal sense. Our task is not complete, however, until we show how to create an  $r$ -score function that leads to “the” probability, or at least to “a” probability that meets the customer's needs.

We work within the framework of supervised learning. That is, for each image  $I^\alpha$  in the training set, there is a “desired” interpretation  $C^{*\alpha}$ . A straightforward learning principle is to say that we want to increase the expected score of the right answer,  $\langle Q(C^{*\alpha} | I) \rangle$ , where the expectation involves averaging over all elements  $\alpha$  in the training set. Because  $Q$  is normalized, this principle automatically implies that the score of undesired interpretations should decrease.

Consider the lattice for our example image and its 11 segments. The lattice calculates one final output: the  $Q$  score for the desired answer. This number is a function of 110 numbers that are input to the lattice, namely the scores of the recognition arcs (ten per segment). It is useful to calculate the sensitivity of the output to

each of these inputs. This can be done using the Baum-Welch algorithm (also called the forward-backward algorithm). Specifically, Baum-Welch will give 110 partial derivatives, one for each input. The meaning of each one is clear: if the derivative is  $x$ , it means that if the score of that rec-arc went up by one unit, the final  $Q$  score would go up by  $x$  units (to first order).

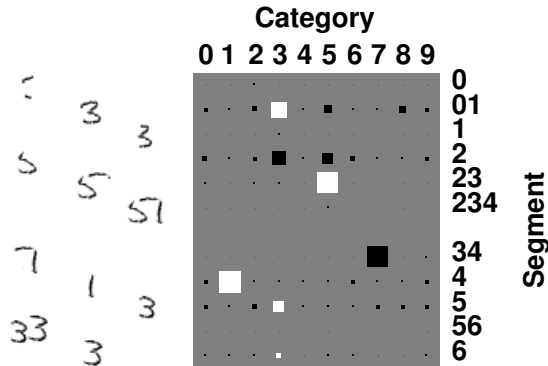


Figure 9: Derivatives w.r.t. Recognition Arcs

The situation for our example is shown in figure 9. A white box indicates a positive derivative, while a black box denotes a negative derivative. We can see that increasing the response of “3” detector neuron to segment (01) will improve the overall score. It also makes sense that it would hurt the overall score if the (2) segment were recognized as a “3” or a “5” — because that would increase the  $R$ -score of paths that give the wrong answer. The strong negative derivative on the “7” response to segment (34) will be discussed later.

In our recognizer, the  $r$ -scores come from a feed-forward neural network called LeNet[1]. The ten output signals of LeNet are exponentiated to give the  $r$ -scores. In the example, the whole MCR consists of the lattice plus 11 copies of LeNet (one for each segment). All 11 copies share the same set of weights.

Now, we can consider the whole MCR as a complicated neural network. We can backpropagate through the whole shebang. The objective function, as we said, is the average  $Q$  score of the desired answers. By the chain rule, the gradient in LeNet’s weight space is

$$\frac{\partial Q}{\partial W} = \frac{\partial Q}{\partial r} \cdot \frac{\partial r}{\partial W} \quad (8)$$

where on the RHS we have a dot product between vectors, which involves summing over all 110  $r$ -scores. The first factor is computed using Baum-Welch, while the second factor is computed using standard backprop. To put it another way, the derivatives calculated using Baum-Welch (and displayed in figure 9) are used as initial deltas at the top layer of LeNet. Backprop does not require targets; all it requires is initial deltas.

Taking a small step in weight space in the direction of the gradient is guaranteed to increase the objective function.

The large black blob in column “7” and row (34) of figure 9 means that LeNet will be trained *not* to recognize such a segment as a “7”. Penalizing this image could be a problem, since the segment really looks like a “7”; but fortunately this is not the only image in the training set. We rely on other images to generate a countervailing training signal. In the end, images like this will be recognized as “7”s, but the score (i.e. the estimated probability) will be less than 100%, which seems reasonable.

## Refinements

The learning principle “increase  $Q$ ” was suggested as being reasonable; it was not claimed to be necessary or sufficient. Here are several refinements to the basic picture:

- In particular,  $\log Q$  is at least as attractive an objective function as  $Q$  ... and in fact  $\log Q$  is what we use. Since the log function is steeper near zero, this causes low-scoring patterns to be emphasized, relatively speaking.

We take the logarithm of the  $r$ -score, too. That is, we set the initial deltas to  $\partial \log Q / \partial \log r$ . (Remember that  $\log r$  is the actual output of LeNet.) Observations indicate that with this choice, the deltas are reasonably uniform in size from pattern to pattern.

- If all  $r$ -scores are multiplied by a constant, the  $Q$  value is unchanged. Therefore an additional principle is required to promote a reasonable scale for  $r$ . For starters, we check for saturated output units on LeNet; a special initial delta is applied if necessary to bring them out of saturation.
- The basic formulation above does not allow the winning  $Q$  score (for any particular image) to be less than  $10^{-5}$  (or more generally,  $N^{-T}$ , where  $N$  is the number of characters in the alphabet, and  $T$  is the length of the interpretation). The minimum occurs when all  $N^T$  interpretations are equally likely. We must get rid of the previously-implicit assumption that there are only  $10^5$  interpretations for a ZIP Code image. In particular, we must allow for the possibility that a given segment does not represent any digit at all. The solution is to add a junk category, making  $N' = N + 1$  categories.

When adding a junk category, some care is required, lest the whole formalism fall down. In particular, there might be a very large number of ways of segmenting the image such that LeNet is absolutely sure, correctly, that the resulting characters are junk. This is not a problem in the numerator, since we just constrain the search to find the highest-scoring non-junk path. The pitfall is that high-scoring junk paths could contribute to the denominator, lowering the score of the right answer even when there is not anything actually wrong. This pitfall can be avoided. It is important to realize that the  $R$ -score of a path involves not just the probability of the interpretation, but also the probability of the segmentation.

In the present implementation, each segment effectively has a junk unit with output level fixed at  $-0.8$ , in units where the activation level of ordinary units' normal range is  $-1.0$  to  $+1.0$ , and extreme range is  $-1.7$  to  $+1.7$ .

- We now return to the issue of the multiplicative scale factor of the  $r$ -scores (which is equivalent to an additive scale on the LeNet output levels). In cases where LeNet is sure of the classification, the junk unit nibbles away a tiny amount of the probability measure. The only way the  $Q$  score can improve is for the magnitude of the LeNet outputs to increase. If unchecked, this would cause a dreadful divergence of the weights. We counter this tendency by applying a tiny amount of “state decay,” which is implemented as an additional small, arbitrary, negative contribution to the initial deltas. It tells the neurons: if you haven't got anything better to do, drift downwards. Eventually they drift down until the junk unit starts to make a significant difference, at which point the delta from Baum-Welch is big enough to balance the state decay.

The presence of the junk unit now imparts an absolute scale to the output neurons. Neurons can be high or low with respect to the junk neuron, which is more meaningful than being high or low with respect to other unmoored neurons.

## 5 Experimental Results

This work used some 11,000 images of ZIP Codes. Approximately 7,000 were chosen at random for the training set, 3,000 for the test set, and 1,000 were reserved for the validation set. About 6% of the images correspond to 9-digit ZIP Codes, and the rest to 5-digit ZIP Codes. They were digitized in black and white at 212 dots per inch. The data was collected by SUNY Buffalo, and is referred to as the “hwb” set by them. All images were lifted from live mail at a mail sorting center, and had been rejected (as unclassifiable) by the “MLOCR” (Multi Line OCR) machines currently used by the U.S. Postal Service.

In most anticipated applications of our recognizer, the cost of punting (i.e. rejecting an image as unclassifiable) is small compared to the cost of a substitution error. To make this clear, consider the following “value model”: the recognizer is offered a sack of mail. For every percent that it classifies correctly, the customer will pay \$1.00, but for every percent that it classifies incorrectly, there is a penalty of \$10.00. There is no cost for rejecting an item since the customers can send the item through the existing system and be no worse off than they are now.

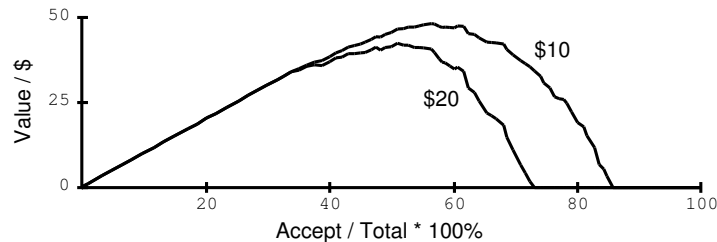


Figure 10: Recognizer Value Curves

The recognizer bases its punting decision on the  $Q$ -score; if the score exceeds a threshold, the item is accepted, otherwise it is rejected. The optimal value of the threshold can be determined by varying the threshold and using the value model, as is done in figure 10. The bottom axis indicates what percentage of the mail is accepted. The upper curve is the value measured on the test set using a \$10.00 penalty, and the lower curve is the same but using a \$20.00 penalty.

Obviously, at 0% acceptance, the value is zero — no payment, no penalty. For small values of acceptance, the value rises with unit slope, because the recognizer can choose to accept only the images that it is sure of. As we move toward the right side of the graph, at some point the recognizer must begin accepting lower-scoring images and taking the attendant risk. The 100% acceptance point is grossly suboptimal.

We find it highly advantageous to think of recognizer performance in these terms, as opposed to (for instance) considering the total number of errors made. It is all too easy to make a design change that is superficially desirable because it reduces the total number of errors, yet unfortunately increases the score of some of the remaining errors (and/or decreases the score of some of the correct items) in such a way that the peak of the value curve is actually lowered.

### Comparison with our previous recognizers

To make a reliable comparison between the old and new ways of building an MCR, we use an idea advocated by Leon Bottou (private communication), which we call the *cross-error matrix*. It emphasizes the distinctions between networks. It is therefore a big win over the low-tech procedure of evaluating each network separately and comparing the scores. All our recognizers perform so well that a validation set of “only” 1000 zips seems frighteningly small; we have to do the statistics carefully lest we get fooled by fluctuations.

Suppose we have two result-files “v2” and “vj” — the scores for two different MCRs on the validation set. Only the network training procedure is different; the network architecture, preprocessor, and everything else are the same. If you know that vj has a punting rate of 37.9% (at 60% correct) while v2 has a punting rate

of 38.6% (also at 60% correct), it is clear that  $v_j$  is better than  $v_2$ , but you ought to wonder whether the difference is statistically significant.

But consider the following table:

		$v_j$			
		Ri	Pu	Wr	Ign
$v_2$	Ri	575	26	0	601
	Pu	25	353	1	379
	Wr	0	7	13	20
	Ign	600	386	14	1000

where the row and column headers are abbreviations for the words Right, Punt, Wrong, and Ignore.

The diagonal is easy to understand: there are 575 cases where *both* networks got the image right (above threshold), 353 cases where both punted it, and 13 cases where both got it badly wrong. There are 1000 cases total, so the counts are easily converted to percentages. The rightmost column tells what happens if we ignore  $v_j$ ; for instance, there are 601 images that  $v_2$  got right (above threshold) and 20 images that it got badly wrong. Similarly, from the bottom row we see that there are 14 images that  $v_j$  got badly wrong. The 14 versus 20 ratio is more informative than the 386 versus 379 ratio, but still not overwhelming.

The most interesting information comes from the 7 and the 1 just off the diagonal. That means that if we exclude the 13 cases that *both* networks got wrong,  $v_j$  makes only 1 mistake that  $v_2$  didn't make, while  $v_2$  makes 7 mistakes that  $v_j$  didn't. Most people consider 1 versus 7 rather more convincing than 14 versus 20.

In the example above, we knew all along that  $v_j$  was better than  $v_2$ . The latter uses the network that has been our state-of-the-art standard network for almost a year, but it was not particularly matched to the segmenter (candidate cut generator) and alignment lattice we were using. In contrast,  $v_j$  had been trained (using Baum-Welch and all the latest tricks) with the new segmenter and lattice. A comparison that is more fair (but in some ways harder to interpret) is  $v_1$  versus  $v_j$ ;  $v_1$  lets the standard net use its favorite segmenter and lattice.

		$v_j$			
		Ri	Pu	Wr	Ign
$v_1$	Ri	574	26	0	600
	Pu	26	353	3	382
	Wr	0	7	11	18
	Ign	600	386	14	1000

In this case the advantage is less extreme, but this constitutes reasonably good evidence that the new MCR (built according to the principles presented in this paper) really is better than the old system.

The experiments presented here are preliminary; we expect substantial improvements in the future.

## 6 Remarks

It is worth calling attention to certain ideas that were *not* used in deriving the results of this paper. In particular, we have not used the fashionable but hard-to-justify “principle” of maximum likelihood as our starting point. Rather, our training uses the principle that we want the score of the right answer to increase, on average.

Recall that “likelihood” is a technical term referring to the probability of the training data given the model. In contrast, the derivation presented here revolves around the probability of correct classification, given the training data; this is *not* a likelihood, maximal or otherwise. The mathematics of hidden Markov models[9, 2] (HMMs) — which we have not invoked — tends to focus attention on likelihoods, since the HMM is clearly a data-generating model rather than a classifier per se.

We are quite aware that HMM theory can be used to motivate the construction of a lattice-based classifier that is similar in many respects to our design, but we feel that the derivation presented here is simpler and easier to justify, and makes more clear what probabilities are conditioned on what.

## 7 Conclusions

We have constructed a multi-character recognizer by combining neural networks (to evaluate individual segments) with a lattice (to handle the segmentation problem). We have used measure theory to understand how multiple segmentations contribute to the final score — an estimate of the probability of correct segmentation. We discovered that normal single-character recognizers do not supply the information needed for segmentation, but the neural network can be retrained to provide this information. The multi-character recognizer is trained as a single adaptive system: error-correction information is propagated through the lattice and into the networks.

Preliminary experiments have demonstrated the advantages of this approach, and we have adopted it as the basis of further work.

## Acknowledgements

This report is a snapshot of a long-running collaboration. We are especially indebted to Esther Levin, Yann leCun, Leon Bottou, Craig Nohl, and Yoshua Bengio.

This report was published as a chapter in the book **The Mathematics of Generalization: Proceedings of the SFI/CNLS Workshop on Formal Approaches to Supervised Learning**, Addison Wesley (1994).

## References

- [1] Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel, “Handwritten Digit Recognition with a Back-Propagation Network,” pp. 396–404 in **Advances in Neural Information Processing 2**, David Touretzky, ed., Morgan Kaufman (1990).
- [2] L. R. Rabiner, “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” *Proc. IEEE* **77** (2) 257–286 (February 1989).
- [3] G.D. Forney, Jr., “The Viterbi Algorithm,” *Proc. IEEE*, **61** pp. 268–278 (March 1978).
- [4] J.S. Bridle, “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition,” in **Neuro-computing: Algorithms, Architectures and Applications**, F. Fogelman and J. Héroult, ed., Springer-Verlag (1989).
- [5] J.S. Bridle, “Training Stochastic Model Recognition Algorithms As Networks Can Lead To Maximum Mutual Information Estimation of Parameters,” in **Advances in Neural Information Processing 2**, David Touretzky, ed., Morgan Kaufman (1990).
- [6] O. Matan, J. Bromley, C.J.C. Burges, J.S. Denker, L.D. Jackel, Y. LeCun, E.P.D. Pednault, W.D. Satterfield, C.E. Stenard, and T.J. Thompson, “Reading Handwritten Digits: A ZIP Code Recognition System,” *IEEE Computer* **25**(7) 59–63 (July 1992).
- [7] C.J.C. Burges, O. Matan, Y. LeCun, J.S. Denker, L.D. Jackel, C.E. Stenard, C.R. Nohl, J.I. Ben, “Shortest Path Segmentation: A Method for Training a Neural Network to Recognize Character Strings,” *IJCNN Conference Proceedings* **3** pp. 165–172 (June 1992).

- [8] C.J.C. Burges, O. Matan, J. Bromley, C.E. Stenard, “Rapid Segmentation and Classification of Handwritten Postal Delivery Addresses using Neural Network Technology;” Interim Report, Task Order Number 104230-90-C-2456, USPS Reference Library, Washington D.C., (August 1991).
- [9] Edwin P.D. Pednault, “A Hidden Markov Model for Resolving Segmentation and Interpretation Ambiguities in Unconstrained Handwriting Recognition;” Bell Labs Technical Memorandum 11352-090929-01TM, (1992).
- [10] L. E. Baum, “An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of a Markov Process;” *Inequalities* **3** 1–8 (1972).
- [11] Stephen E. Levinson, “Structural Methods in Automatic Speech Recognition;” *Proc. IEEE*, **73** (11) (November 1985).
- [12] Ofer Matan, Christopher J. C. Burges, Yann LeCun, and John S. Denker, “Multi-Digit Recognition Using a Space Displacement Neural Network;” **Neural Information Processing Systems 4**, J. M. Moody S. J. Hanson and R. P. Lippman, eds., Morgan Kaufmann (1992).
- [13] Y. Bengio, R. deMori, G. Flammia, and R. Kompe, “Global Optimization of a Neural Network – Hidden Markov Model Hybrid;” **Proc. of EuroSpeech 91** (1991)
- [14] M. Franzini, K. Lee, and A. Waibel, “Connectionist Viterbi Training: a new hybrid method for continuous speech recognition;” **Proc. of ICASSP 90** (1990)
- [15] P. Haffner, M. Franzini, and A. Waibel, “Integrating Time-Alignment and Neural Networks for High Performance Continuous Speech Recognition;” **Proc. of ICASSP 91** (1991)
- [16] X. Driancourt, L. Bottou, and P. Gallinari, “Learning Vector Quantization, Multilayer Perceptron and Dynamic Programming: Comparison and Cooperation;” **Proc. of the IJCNN 91** pp. 815-819 (1991).