Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Abstract—
Multilayer Neural Networks trained with the backpropagation algorithm constitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal preprocessing. This paper reviews various methods applied to handwritten character recognition and compares them on a standard handwritten digit recognition task. Convolutional Neural Networks, that are specifically designed to deal with the variability of 2D shapes, are shown to outperform all other techniques.

Real-life document recognition systems are composed of multiple modules including field extraction, segmentation, recognition, and language modeling. A new learning paradigm, called Graph Transformer Networks (GTN), allows such multi-module systems to be trained globally using Gradient-Based methods so as to minimize an overall performance measure.

Two systems for on-line handwriting recognition are described. Experiments demonstrate the advantage of global training, and the flexibility of Graph Transformer Networks.

A Graph Transformer Network for reading bank check is also described. It uses Convolutional Neural Network character recognizers combined with global training techniques to provides record accuracy on business and personal checks. It is deployed commercially and reads several million checks per day.

Keywords— **Neural Networks, OCR, Document Recognition, Machine Learning, Gradient-Based Learning, Convolutional Neural Networks, Graph Transformer Networks, Finite State Transducers.**

NOMENCLATURE

- GT Graph transformer.
- GTN Graph transformer network.
- HMM Hidden Markov model.
- HOS Heuristic oversegmentation.
- K-NN K-nearest neighbor.
- NN Neural network.
- OCR Optical character recognition.
- PCA Principal component analysis.
- RBF Radial basis function.
- RS-SVM Reduced-set support vector method.
- SDNN Space displacement neural network.
- SVM Support vector method.
- TDNN Time delay neural network.
- V-SVM Virtual support vector method.

The authors are with the Speech and Image Processing Services Research Laboratory, AT&T Labs-Research, 100 Schulz Drive Red Bank, NJ 07701. E-mail: {yann,leonb,yoshua,haffner}@research.att.com. Yoshua Bengio is also with the Département d'Informatique et de Recherche Opérationelle, Université de Montréal, C.P. 6128 Succ. Centre-Ville, 2920 Chemin de la Tour, Montréal, Québec, Canada H3C 3J7.

I. INTRODUCTION

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques has been a crucial factor in the recent success of pattern recognition applications such as continuous speech recognition and handwriting recognition.

The main message of this paper is that better pattern recognition systems can be built by relying more on automatic learning, and less on hand-designed heuristics. This is made possible by recent progress in machine learning and computer technology. Using character recognition as a case study, we show that hand-crafted feature extraction can be advantageously replaced by carefully designed learning machines that operate directly on pixel images. Using document understanding as a case study, we show that the traditional way of building recognition systems by manually integrating individually designed modules can be replaced by a unified and well-principled design paradigm, called *Graph Transformer Networks*, that allows training all the modules to optimize a global performance criterion.

Since the early days of pattern recognition it has been known that the variability and richness of natural data, be it speech, glyphs, or other types of patterns, make it almost impossible to build an accurate recognition system entirely by hand. Consequently, most pattern recognition systems are built using a combination of automatic learning techniques and hand-crafted algorithms. The usual method of recognizing individual patterns consists in dividing the system into two main modules shown in figure 1. The first module, called the feature extractor, transforms the input patterns so that they can be represented by low-dimensional vectors or short strings of symbols that (a) can be easily matched or compared, and (b) are relatively invariant with respect to transformations and distortions of the input patterns that do not change their nature. The feature extractor contains most of the prior knowledge and is rather specific to the task. It is also the focus of most of the design effort, because it is often entirely hand-crafted. The classifier, on the other hand, is often general-purpose and trainable. One of the main problems with this approach is that the recognition accuracy is largely determined by the ability of the designer to come up with an appropriate set of features. This turns out to be a daunting task which, unfortunately, must be redone for each new problem. A large amount of the pattern recognition literature is devoted to describing and comparing the relative

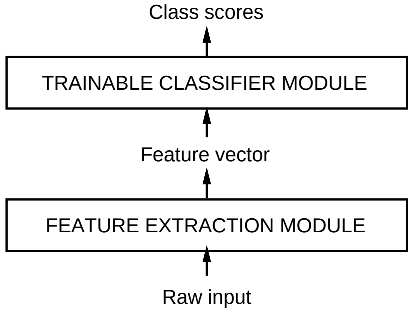

Fig. 1. Traditional pattern recognition is performed with two modules: a fixed feature extractor, and a trainable classifier.

merits of different feature sets for particular tasks.

Historically, the need for appropriate feature extractors was due to the fact that the learning techniques used by the classifiers were limited to low-dimensional spaces with easily separable classes [1]. A combination of three factors have changed this vision over the last decade. First, the availability of low-cost machines with fast arithmetic units allows to rely more on brute-force "numerical" methods than on algorithmic refinements. Second, the availability of large databases for problems with a large market and wide interest, such as handwriting recognition, has enabled designers to rely more on real data and less on hand-crafted feature extraction to build recognition systems. The third and very important factor is the availability of powerful machine learning techniques that can handle high-dimensional inputs and can generate intricate decision functions when fed with these large data sets. It can be argued that the recent progress in the accuracy of speech and handwriting recognition systems can be attributed in large part to an increased reliance on learning techniques and large training data sets. As evidence to this fact, a large proportion of modern commercial OCR systems use some form of multilayer Neural Network trained with back-propagation.

In this study, we consider the tasks of handwritten character recognition (Sections I and II) and compare the performance of several learning techniques on a benchmark data set for handwritten digit recognition (Section III). While more automatic learning is beneficial, no learning technique can succeed without a minimal amount of prior knowledge about the task. In the case of multi-layer neural networks, a good way to incorporate knowledge is to tailor its architecture to the task. Convolutional Neural Networks [2] introduced in Section II are an example of specialized neural network architectures which incorporate knowledge about the invariances of 2D shapes by using local connection patterns, and by imposing constraints on the weights. A comparison of several methods for isolated handwritten digit recognition is presented in section III. To go from the recognition of individual characters to the recognition of words and sentences in documents, the idea of combining multiple modules trained to reduce the overall error is introduced in Section IV. Recognizing variable-length objects such as handwritten words using multi-module systems is best done if the modules manipulate directed graphs. This leads to the concept of trainable *Graph Transformer Network* (GTN) also introduced in Section IV. Section V describes the now classical method of heuristic over-segmentation for recognizing words or other character strings. Discriminative and non-discriminative gradient-based techniques for training a recognizer at the word level without requiring manual segmentation and labeling are presented in Section VI. Section VII presents the promising Space-Displacement Neural Network approach that eliminates the need for segmentation heuristics by scanning a recognizer at all possible locations on the input. In section VIII, it is shown that trainable Graph Transformer Networks can be formulated as multiple generalized transductions, based on a general graph composition algorithm. The connections between GTNs and Hidden Markov Models, commonly used in speech recognition is also treated. Section IX describes a globally trained GTN system for recognizing handwriting entered in a pen computer. This problem is known as "on-line" handwriting recognition, since the machine must produce immediate feedback as the user writes. The core of the system is a Convolutional Neural Network. The results clearly demonstrate the advantages of training a recognizer at the word level, rather than training it on pre-segmented, hand-labeled, isolated characters. Section X describes a complete GTN-based system for reading handwritten and machine-printed bank checks. The core of the system is the Convolutional Neural Network called LeNet-5 described in Section II. This system is in commercial use in the NCR Corporation line of check recognition systems for the banking industry. It is reading millions of checks per month in several banks across the United States.

A. Learning from Data

There are several approaches to automatic machine learning, but one of the most successful approaches, popularized in recent years by the neural network community, can be called "numerical" or *gradient-based learning.* The learning machine computes a function $= ( )$ where is the -th input pattern, and represents the collection of adjustable parameters in the system. In a pattern recognition setting, the output may be interpreted as the recognized class label of pattern , or as scores or probabilities associated with each class. A loss function $= (( ))$, measures the discrepancy between , the "correct" or desired output for pattern , and the output produced by the system. The average loss function $( )$ is the average of the errors over a set of labeled examples called the training set $(^1 \; ^1) \; ( )$. In the simplest setting, the learning problem consists in finding the value of that minimizes $( )$. In practice, the performance of the system on a training set is of little interest. The more relevant measure is the error rate of the system in the field, where it would be used in practice. This performance is estimated by measuring the accuracy on a set of samples disjoint from the training set, called the test set. Much theoretical and experimental work [3], [4], [5] has shown that the gap between the expected error rate on the test set and the error rate on the training set decreases with the number of training samples approximately as

$$= (\quad) \tag{1}$$

where is the number of training samples, is a measure of "effective capacity" or complexity of the machine [6], [7], is a number between 0 5 and 1 0, and is a constant. This gap always decreases when the number of training samples increases. Furthermore, as the capacity increases, decreases. Therefore, when increasing the capacity , there is a trade-off between the decrease of and the increase of the gap, with an optimal value of the capacity that achieves the lowest generalization error . Most learning algorithms attempt to minimize as well as some estimate of the gap. A formal version of this is called structural risk minimization [6], [7], and is based on defining a sequence of learning machines of increasing capacity, corresponding to a sequence of subsets of the parameter space such that each subset is a superset of the previous subset. In practical terms, Structural Risk Minimization is implemented by minimizing + ( ), where the function ( ) is called a regularization function, and is a constant. ( ) is chosen such that it takes large values on parameters that belong to high-capacity subsets of the parameter space. Minimizing ( ) in effect limits the capacity of the accessible subset of the parameter space, thereby controlling the tradeoff between minimizing the training error and minimizing the expected gap between the training error and test error.

B. Gradient-Based Learning

The general problem of minimizing a function with respect to a set of parameters is at the root of many issues in computer science. Gradient-Based Learning draws on the fact that it is generally much easier to minimize a reasonably smooth, continuous function than a discrete (combinatorial) function. The loss function can be minimized by estimating the impact of small variations of the parameter values on the loss function. This is measured by the gradient of the loss function with respect to the parameters. Efficient learning algorithms can be devised when the gradient vector can be computed analytically (as opposed to numerically through perturbations). This is the basis of numerous gradient-based learning algorithms with continuous-valued parameters. In the procedures described in this article, the set of parameters is a real-valued vector, with respect to which ( ) is continuous, as well as differentiable almost everywhere. The simplest minimization procedure in such a setting is the gradient descent algorithm where is iteratively adjusted as follows:

$$= \quad _{-1} \quad (\quad) \tag{2}$$

In the simplest case, is a scalar constant. More sophisticated procedures use variable , or substitute it for a diagonal matrix, or substitute it for an estimate of the inverse Hessian matrix as in Newton or Quasi-Newton methods. The Conjugate Gradient method [8] can also be used. However, Appendix B shows that despite many claims to the contrary in the literature, the usefulness of these second-order methods to large learning machines is very limited.

A popular minimization procedure is the stochastic gradient algorithm, also called the on-line update. It consists in updating the parameter vector using a noisy, or approximated, version of the average gradient. In the most common instance of it, is updated on the basis of a single sample:

$$= \quad _{-1} \quad (\quad) \tag{3}$$

With this procedure the parameter vector fluctuates around an average trajectory, but usually converges considerably faster than regular gradient descent and second order methods on large training sets with redundant samples (such as those encountered in speech or character recognition). The reasons for this are explained in Appendix B. The properties of such algorithms applied to learning have been studied theoretically since the 1960's [9], [10], [11], but practical successes for non-trivial tasks did not occur until the mid eighties.

C. Gradient Back-Propagation

Gradient-Based Learning procedures have been used since the late 1950's, but they were mostly limited to linear systems [1]. The surprising usefulness of such simple gradient descent techniques for complex machine learning tasks was not widely realized until the following three events occurred. The first event was the realization that, despite early warnings to the contrary [12], the presence of local minima in the loss function does not seem to be a major problem in practice. This became apparent when it was noticed that local minima did not seem to be a major impediment to the success of early non-linear gradient-based Learning techniques such as Boltzmann machines [13], [14]. The second event was the popularization by Rumelhart, Hinton and Williams [15] and others of a simple and efficient procedure, the back-propagation algorithm, to compute the gradient in a non-linear system composed of several layers of processing. The third event was the demonstration that the back-propagation procedure applied to multi-layer neural networks with sigmoidal units can solve complicated learning tasks. The basic idea of back-propagation is that gradients can be computed efficiently by propagation from the output to the input. This idea was described in the control theory literature of the early sixties [16], but its application to machine learning was not generally realized then. Interestingly, the early derivations of back-propagation in the context of neural network learning did not use gradients, but "virtual targets" for units in intermediate layers [17], [18], or minimal disturbance arguments [19]. The Lagrange formalism used in the control theory literature provides perhaps the best rigorous method for deriving back-propagation [20], and for deriving generalizations of back-propagation to recurrent networks [21], and networks of heterogeneous modules [22]. A simple derivation for generic multi-layer systems is given in Section I-E.

The fact that local minima do not seem to be a problem for multi-layer neural networks is somewhat of a theoretical mystery. It is conjectured that if the network is oversized for the task (as is usually the case in practice), the presence of "extra dimensions" in parameter space reduces the risk of unattainable regions. Back-propagation is by far the most widely used neural-network learning algorithm, and probably the most widely used learning algorithm of any form.

D. Learning in Real Handwriting Recognition Systems

Isolated handwritten character recognition has been extensively studied in the literature (see [23], [24] for reviews), and was one of the early successful applications of neural networks [25]. Comparative experiments on recognition of individual handwritten digits are reported in Section III. They show that neural networks trained with Gradient-Based Learning perform better than all other methods tested here on the same data. The best neural networks, called Convolutional Networks, are designed to learn to extract relevant features directly from pixel images (see Section II).

One of the most difficult problems in handwriting recognition, however, is not only to recognize individual characters, but also to separate out characters from their neighbors within the word or sentence, a process known as segmentation. The technique for doing this that has become the "standard" is called *Heuristic Over-Segmentation.* It consists in generating a large number of potential cuts between characters using heuristic image processing techniques, and subsequently selecting the best combination of cuts based on scores given for each candidate character by the recognizer. In such a model, the accuracy of the system depends upon the quality of the cuts generated by the heuristics, and on the ability of the recognizer to distinguish correctly segmented characters from pieces of characters, multiple characters, or otherwise incorrectly segmented characters. Training a recognizer to perform this task poses a major challenge because of the difficulty in creating a labeled database of incorrectly segmented characters. The simplest solution consists in running the images of character strings through the segmenter, and then manually labeling all the character hypotheses. Unfortunately, not only is this an extremely tedious and costly task, it is also difficult to do the labeling consistently. For example, should the right half of a cut up 4 be labeled as a 1 or as a non-character? should the right half of a cut up 8 be labeled as a 3?

The first solution, described in Section V consists in training the system at the level of whole strings of characters, rather than at the character level. The notion of Gradient-Based Learning can be used for this purpose. The system is trained to minimize an overall loss function which measures the probability of an erroneous answer. Section V explores various ways to ensure that the loss function is differentiable, and therefore lends itself to the use of Gradient-Based Learning methods. Section V introduces the use of directed acyclic graphs whose arcs carry numerical information as a way to represent the alternative hypotheses, and introduces the idea of GTN.

The second solution described in Section VII is to eliminate segmentation altogether. The idea is to sweep the recognizer over every possible location on the input image, and to rely on the "character spotting" property of the recognizer, i.e. its ability to correctly recognize a well-centered character in its input field, even in the presence of other characters besides it, while rejecting images containing no centered characters [26], [27]. The sequence of recognizer outputs obtained by sweeping the recognizer over the input is then fed to a Graph Transformer Network that takes linguistic constraints into account and finally extracts the most likely interpretation. This GTN is somewhat similar to Hidden Markov Models (HMM), which makes the approach reminiscent of the classical speech recognition [28], [29]. While this technique would be quite expensive in the general case, the use of Convolutional Neural Networks makes it particularly attractive because it allows significant savings in computational cost.

E. Globally Trainable Systems

As stated earlier, most practical pattern recognition systems are composed of multiple modules. For example, a document recognition system is composed of a field locator, which extracts regions of interest, a field segmenter, which cuts the input image into images of candidate characters, a recognizer, which classifies and scores each candidate character, and a contextual post-processor, generally based on a stochastic grammar, which selects the best grammatically correct answer from the hypotheses generated by the recognizer. In most cases, the information carried from module to module is best represented as graphs with numerical information attached to the arcs. For example, the output of the recognizer module can be represented as an acyclic graph where each arc contains the label and the score of a candidate character, and where each path represent a alternative interpretation of the input string. Typically, each module is manually optimized, or sometimes trained, outside of its context. For example, the character recognizer would be trained on labeled images of pre-segmented characters. Then the complete system is assembled, and a subset of the parameters of the modules is manually adjusted to maximize the overall performance. This last step is extremely tedious, time-consuming, and almost certainly suboptimal.

A better alternative would be to somehow train the entire system so as to minimize a global error measure such as the probability of character misclassifications at the document level. Ideally, we would want to find a good minimum of this global loss function with respect to all the parameters in the system. If the loss function measuring the performance can be made differentiable with respect to the system's tunable parameters , we can find a local minimum of using Gradient-Based Learning. However, at first glance, it appears that the sheer size and complexity of the system would make this intractable.

To ensure that the global loss function ( ) is differentiable, the overall system is built as a feed-forward network of differentiable modules. The function implemented by each module must be continuous and differentiable *almost everywhere* with respect to the internal parameters of the module (e.g. the weights of a Neural Net character recognizer in the case of a character recognition module), and with respect to the module's inputs. If this is the case, a simple generalization of the well-known back-propagation procedure can be used to efficiently compute the gradients of the loss function with respect to all the parameters in the system [22]. For example, let us consider a system built as a cascade of modules, each of which implements a function $= (\quad {}_{-1})$, where is a vector representing the output of the module, is the vector of tunable parameters in the module (a subset of), and ${}_{-1}$ is the module's input vector (as well as the previous module's output vector). The input ${}_{0}$ to the first module is the input pattern . If the partial derivative of with respect to is known, then the partial derivatives of with respect to and ${}_{-1}$ can be computed using the backward recurrence

$$
\begin{aligned}
&= \quad (\qquad {}_{-1}) \\
{}_{-1} &= \quad (\qquad {}_{-1}) \qquad (4)
\end{aligned}
$$

where $(\quad {}_{-1})$ is the Jacobian of with respect to evaluated at the point $(\quad {}_{-1})$, and $(\quad {}_{-1})$ is the Jacobian of with respect to . The Jacobian of a vector function is a matrix containing the partial derivatives of all the outputs with respect to all the inputs. The first equation computes some terms of the gradient of ( ), while the second equation generates a backward recurrence, as in the well-known back-propagation procedure for neural networks. We can average the gradients over the training patterns to obtain the full gradient. It is interesting to note that in many instances there is no need to explicitly compute the Jacobian matrix. The above formula uses the product of the Jacobian with a vector of partial derivatives, and it is often easier to compute this product directly without computing the Jacobian beforehand. In By analogy with ordinary multi-layer neural networks, all but the last module are called hidden layers because their outputs are not observable from the outside. more complex situations than the simple cascade of modules described above, the partial derivative notation becomes somewhat ambiguous and awkward. A completely rigorous derivation in more general cases can be done using Lagrange functions [20], [21], [22].

Traditional multi-layer neural networks are a special case of the above where the state information is represented with fixed-sized vectors, and where the modules are alternated layers of matrix multiplications (the weights) and component-wise sigmoid functions (the neurons). However, as stated earlier, the state information in complex recognition system is best represented by graphs with numerical information attached to the arcs. In this case, each module, called a Graph Transformer, takes one or more graphs as input, and produces a graph as output. Networks of such modules are called Graph Transformer Networks (GTN). Sections IV, VI and VIII develop the concept of GTNs, and show that Gradient-Based Learning can be used to train all the parameters in all the modules so as to minimize a global loss function. It may seem paradoxical that gradients can be computed when the state information is represented by essentially discrete objects such as graphs, but that difficulty can be circumvented, as shown later.

II. Convolutional Neural Networks for Isolated Character Recognition

The ability of multi-layer networks trained with gradient descent to learn complex, high-dimensional, non-linear mappings from large collections of examples makes them obvious candidates for image recognition tasks. In the traditional model of pattern recognition, a hand-designed feature extractor gathers relevant information from the input and eliminates irrelevant variabilities. A trainable classifier then categorizes the resulting feature vectors into classes. In this scheme, standard, fully-connected multi-layer networks can be used as classifiers. A potentially more interesting scheme is to rely on as much as possible on learning in the feature extractor itself. In the case of character recognition, a network could be fed with almost raw inputs (e.g. size-normalized images). While this can be done with an ordinary fully connected feed-forward network with some success for tasks such as character recognition, there are problems.

Firstly, typical images are large, often with several hundred variables (pixels). A fully-connected first layer with, say one hundred hidden units in the first layer, would already contain several tens of thousands of weights. Such a large number of parameters increases the capacity of the system and therefore requires a larger training set. In addition, the memory requirement to store so many weights may rule out certain hardware implementations. But, the main deficiency of unstructured nets for image or speech applications is that they have no built-in invariance with respect to translations, or local distortions of the inputs. Before being sent to the fixed-size input layer of a neural net, character images, or other 2D or 1D signals, must be approximately size-normalized and centered in the input field. Unfortunately, no such preprocessing can be perfect: handwriting is often normalized at the word level, which can cause size, slant, and position variations for individual characters. This, combined with variability in writing style, will cause variations in the position of distinctive features in input objects. In principle, a fully-connected network of sufficient size could learn to produce outputs that are invariant with respect to such variations. However, learning such a task would probably result in multiple units with similar weight patterns positioned at various locations in the input so as to detect distinctive features wherever they appear on the input. Learning these weight configurations requires a very large number of training instances to cover the space of possible variations. In convolutional networks, described below, shift invariance is automatically obtained by forcing the replication of weight configurations across space.

Secondly, a deficiency of fully-connected architectures is that the topology of the input is entirely ignored. The input variables can be presented in any (fixed) order without affecting the outcome of the training. On the contrary, images (or time-frequency representations of speech) have a strong 2D local structure: variables (or pixels) that are spatially or temporally nearby are highly correlated. Local correlations are the reasons for the well-known advantages of extracting and combining *local* features before recognizing spatial or temporal objects, because configurations of neighboring variables can be classified into a small number of categories (e.g. edges, corners...). *Convolutional Networks* force the extraction of local features by restricting the receptive fields of hidden units to be local.

A. Convolutional Networks

Convolutional Networks combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: *local receptive fields*, *shared weights* (or weight replication), and spatial or temporal *sub-sampling.* A typical convolutional network for recognizing characters, dubbed LeNet-5, is shown in figure 2. The input plane receives images of characters that are approximately size-normalized and centered. Each unit in a layer receives inputs from a set of units located in a small neighborhood in the previous layer. The idea of connecting units to local receptive fields on the input goes back to the Perceptron in the early 60s, and was almost simultaneous with Hubel and Wiesel's discovery of locally-sensitive, orientation-selective neurons in the cat's visual system [30]. Local connections have been used many times in neural models of visual learning [31], [32], [18], [33], [34], [2]. With local receptive fields, neurons can extract elementary visual features such as oriented edges, end-points, corners (or similar features in other signals such as speech spectrograms). These features are then combined by the subsequent layers in order to detect higher-order features. As stated earlier, distortions or shifts of the input can cause the position of salient features to vary. In addition, elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image. This knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors [32], [15], [34]. Units in a layer are organized in planes within which all the units share the same set of weights. The set of outputs of the units in such a plane is called a *feature map*. Units in a feature map are all constrained to perform the same operation on different parts of the image. A complete convolutional layer is composed of several feature maps (with different weight vectors), so that multiple features can be extracted at each location. A concrete example of this is the first layer of LeNet-5 shown in Figure 2. Units in the first hidden layer of LeNet-5 are organized in 6 planes, each of which is a feature map. A unit in a feature map has 25 inputs connected to a 5 by 5 area in the input, called the *receptive field* of the unit. Each unit has 25 inputs, and therefore 25 trainable coefficients plus a trainable bias. The receptive fields of contiguous units in a feature map are centered on correspondingly contiguous units in the previous layer. Therefore receptive fields of neighboring units overlap. For example, in the first hidden layer of LeNet-5, the receptive fields of horizontally contiguous units overlap by 4 columns and 5 rows. As stated earlier, all the units in a feature map share the same set of 25 weights and the same bias so they detect the same feature at all possible locations on the input. The other feature maps in the layer use different sets of weights and biases, thereby extracting different types of local features. In the case of LeNet-5, at each input location six different types of features are extracted by six units in identical locations in the six feature maps. A sequential implementation of a feature map would scan the input image with a single unit that has a local receptive field, and store the states of this unit at corresponding locations in the feature map. This operation is equivalent to a convolution, followed by an additive bias and squashing function, hence the name *convolutional network*. The kernel of the convolution is the set of connection weights used by the units in the feature map. An interesting property of convolutional layers is that if the input image is shifted, the feature map output will be shifted by the same amount, but will be left unchanged otherwise. This property is at the basis of the robustness of convolutional networks to shifts and distortions of the input.

Once a feature has been detected, its exact location becomes less important. Only its approximate position relative to other features is relevant. For example, once we know that the input image contains the endpoint of a roughly horizontal segment in the upper left area, a corner in the upper right area, and the endpoint of a roughly vertical segment in the lower portion of the image, we can tell the input image is a 7. Not only is the precise position of each of those features irrelevant for identifying the pattern, it is potentially harmful because the positions are likely to vary for different instances of the character. A simple way to reduce the precision with which the position of distinctive features are encoded in a feature map is to reduce the spatial resolution of the feature map. This can be achieved with a so-called *sub-sampling layers* which performs a local averaging and a sub-sampling, reducing the resolution of the feature map, and reducing the sensitivity of the output to shifts and distortions. The second hidden layer of LeNet-5 is a sub-sampling layer. This layer comprises six feature maps, one for each feature map in the previous layer. The receptive field of each unit is a 2 by 2 area in the previous layer's corresponding feature map. Each unit computes the *average* of its four inputs, multiplies it by a trainable coefficient, adds a trainable bias, and passes the result through a sigmoid function. Contiguous units have non-overlapping contiguous receptive fields. Consequently, a sub-sampling layer feature map has half the number of rows and columns

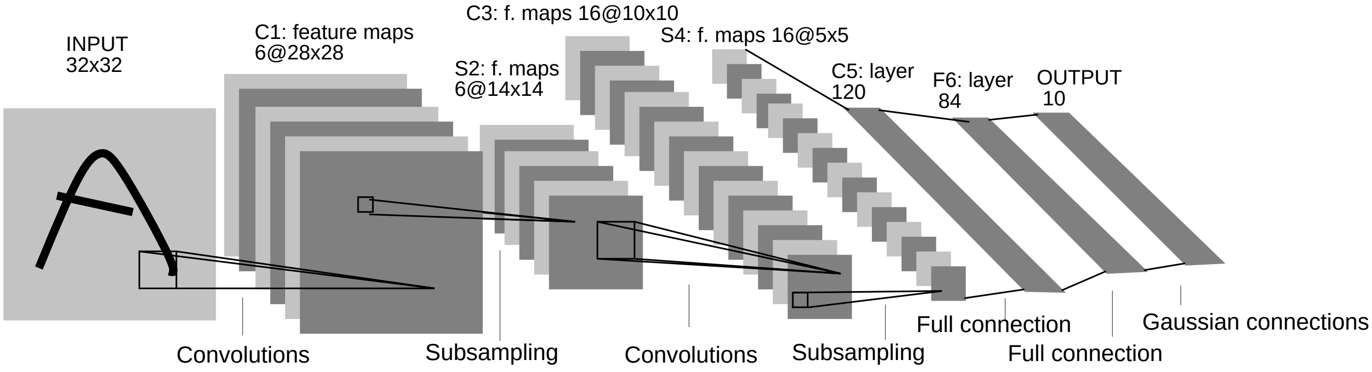

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

as the feature maps in the previous layer. The trainable coefficient and bias control the effect of the sigmoid nonlinearity. If the coefficient is small, then the unit operates in a quasi-linear mode, and the sub-sampling layer merely blurs the input. If the coefficient is large, sub-sampling units can be seen as performing a "noisy OR" or a "noisy AND" function depending on the value of the bias. Successive layers of convolutions and sub-sampling are typically alternated, resulting in a "bi-pyramid": at each layer, the number of feature maps is increased as the spatial resolution is decreased. Each unit in the third hidden layer in figure 2 may have input connections from several feature maps in the previous layer. The convolution/sub-sampling combination, inspired by Hubel and Wiesel's notions of "simple" and "complex" cells, was implemented in Fukushima's Neocognitron [32], though no globally supervised learning procedure such as back-propagation was available then. A large degree of invariance to geometric transformations of the input can be achieved with this progressive reduction of spatial resolution compensated by a progressive increase of the richness of the representation (the number of feature maps).

Since all the weights are learned with back-propagation, convolutional networks can be seen as synthesizing their own feature extractor. The weight sharing technique has the interesting side effect of reducing the number of free parameters, thereby reducing the "capacity" of the machine and reducing the gap between test error and training error [34]. The network in figure 2 contains 340,908 connections, but only 60,000 trainable free parameters because of the weight sharing.

Fixed-size Convolutional Networks have been applied to many applications, among other handwriting recognition [35], [36], machine-printed character recognition [37], on-line handwriting recognition [38], and face recognition [39]. Fixed-size convolutional networks that share weights along a single temporal dimension are known as Time-Delay Neural Networks (TDNNs). TDNNs have been used in phoneme recognition (without sub-sampling) [40], [41], spoken word recognition (with sub-sampling) [42], [43], on-line recognition of isolated handwritten characters [44], and signature verification [45].

B. LeNet-5

This section describes in more detail the architecture of LeNet-5, the Convolutional Neural Network used in the experiments. LeNet-5 comprises 7 layers, not counting the input, all of which contain trainable parameters (weights). The input is a 32x32 pixel image. This is significantly larger than the largest character in the database (at most 20x20 pixels centered in a 28x28 field). The reason is that it is desirable that potential distinctive features such as stroke end-points or corner can appear *in the center* of the receptive field of the highest-level feature detectors. In LeNet-5 the set of centers of the receptive fields of the last convolutional layer (C3, see below) form a 20x20 area in the center of the 32x32 input. The values of the input pixels are normalized so that the background level (white) corresponds to a value of -0.1 and the foreground (black) corresponds to 1.175. This makes the mean input roughly 0, and the variance roughly 1 which accelerates learning [46].

In the following, convolutional layers are labeled Cx, subsampling layers are labeled Sx, and fully-connected layers are labeled Fx, where x is the layer index.

Layer C1 is a convolutional layer with 6 feature maps. Each unit in each feature map is connected to a 5x5 neighborhood in the input. The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. C1 contains 156 trainable parameters, and 122,304 connections.

Layer S2 is a sub-sampling layer with 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C1. The four inputs to a unit in S2 are added, then multiplied by a trainable coefficient, and added to a trainable bias. The result is passed through a sigmoidal function. The 2x2 receptive fields are non-overlapping, therefore feature maps in S2 have half the number of rows and column as feature maps in C1. Layer S2 has 12 trainable parameters and 5,880 connections.

Layer C3 is a convolutional layer with 16 feature maps. Each unit in each feature map is connected to several 5x5 neighborhoods at identical locations in a subset of S2's feature maps. Table I shows the set of S2 feature maps

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

combined by each C3 feature map. Why not connect every S2 feature map to every C3 feature map? The reason is twofold. First, a non-complete connection scheme keeps the number of connections within reasonable bounds. More importantly, it forces a break of symmetry in the network. Different feature maps are forced to extract different (hopefully complementary) features because they get different sets of inputs. The rationale behind the connection scheme in table I is the following. The first six C3 feature maps take inputs from every contiguous subsets of three feature maps in S2. The next six take input from every contiguous subset of four. The next three take input from some discontinuous subsets of four. Finally the last one takes input from all S2 feature maps. Layer C3 has 1,516 trainable parameters and 151,600 connections.

Layer S4 is a sub-sampling layer with 16 feature maps of size 5x5. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C3, in a similar way as C1 and S2. Layer S4 has 32 trainable parameters and 2,000 connections.

Layer C5 is a convolutional layer with 120 feature maps. Each unit is connected to a 5x5 neighborhood on all 16 of S4's feature maps. Here, because the size of S4 is also 5x5, the size of C5's feature maps is 1x1: this amounts to a full connection between S4 and C5. C5 is labeled as a convolutional layer, instead of a fully-connected layer, because if LeNet-5 input were made bigger with everything else kept constant, the feature map dimension would be larger than 1x1. This process of dynamically increasing the size of a convolutional network is described in the section Section VII. Layer C5 has 48,120 trainable connections.

Layer F6, contains 84 units (the reason for this number comes from the design of the output layer, explained below) and is fully connected to C5. It has 10,164 trainable parameters.

As in classical neural networks, units in layers up to F6 compute a dot product between their input vector and their weight vector, to which a bias is added. This weighted sum, denoted for unit , is then passed through a sigmoid squashing function to produce the state of unit , denoted by :

$$= ( ) \tag{5}$$

The squashing function is a scaled hyperbolic tangent:

$$( ) = \tanh( ) \tag{6}$$

where is the amplitude of the function and determines its slope at the origin. The function is odd, with horizontal asymptotes at + and . The constant is chosen to be 1 7159. The rationale for this choice of a squashing function is given in Appendix A.

Finally, the output layer is composed of Euclidean Radial Basis Function units (RBF), one for each class, with 84 inputs each. The outputs of each RBF unit is computed as follows:

$$= ( )^2 \tag{7}$$

In other words, each output RBF unit computes the Euclidean distance between its input vector and its parameter vector. The further away is the input from the parameter vector, the larger is the RBF output. The output of a particular RBF can be interpreted as a penalty term measuring the fit between the input pattern and a model of the class associated with the RBF. In probabilistic terms, the RBF output can be interpreted as the unnormalized negative log-likelihood of a Gaussian distribution in the space of configurations of layer F6. Given an input pattern, the loss function should be designed so as to get the configuration of F6 as close as possible to the parameter vector of the RBF that corresponds to the pattern's desired class. The parameter vectors of these units were chosen by hand and kept fixed (at least initially). The components of those parameters vectors were set to -1 or +1. While they could have been chosen at random with equal probabilities for -1 and +1, or even chosen to form an error correcting code as suggested by [47], they were instead designed to represent a stylized image of the corresponding character class drawn on a 7x12 bitmap (hence the number 84). Such a representation is not particularly useful for recognizing isolated digits, but it is quite useful for recognizing strings of characters taken from the full printable ASCII set. The rationale is that characters that are similar, and therefore confusable, such as uppercase O, lowercase O, and zero, or lowercase l, digit 1, square brackets, and uppercase I, will have similar output codes. This is particularly useful if the system is combined with a linguistic post-processor that can correct such confusions. Because the codes for confusable classes are similar, the output of the corresponding RBFs for an ambiguous character will be similar, and the post-processor will be able to pick the appropriate interpretation. Figure 3 gives the output codes for the full ASCII set.

Another reason for using such distributed codes, rather than the more common "1 of N" code (also called place code, or grand-mother cell code) for the outputs is that non distributed codes tend to behave badly when the number of classes is larger than a few dozens. The reason is that output units in a non-distributed code must be off most of the time. This is quite difficult to achieve with sigmoid units. Yet another reason is that the classifiers are often used to not only recognize characters, but also to reject non-characters. RBFs with distributed codes are more appropriate for that purpose because unlike sigmoids, they are activated within a well circumscribed region of their in-

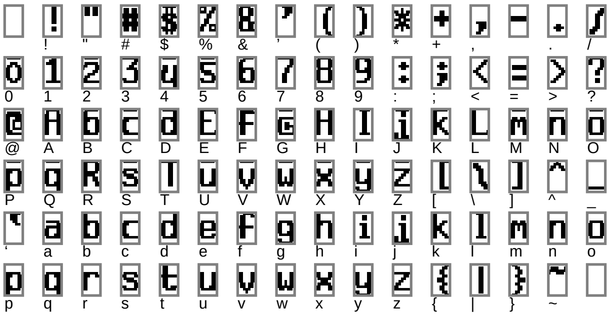

Fig. 3. Initial parameters of the output RBFs for recognizing the full ASCII set.

put space that non-typical patterns are more likely to fall outside of.

The parameter vectors of the RBFs play the role of target vectors for layer F6. It is worth pointing out that the components of those vectors are +1 or -1, which is well within the range of the sigmoid of F6, and therefore prevents those sigmoids from getting saturated. In fact, +1 and -1 are the points of maximum curvature of the sigmoids. This forces the F6 units to operate in their maximally non-linear range. Saturation of the sigmoids must be avoided because it is known to lead to slow convergence and ill-conditioning of the loss function.

C. Loss Function

The simplest output loss function that can be used with the above network is the Maximum Likelihood Estimation criterion (MLE), which in our case is equivalent to the Minimum Mean Squared Error (MSE). The criterion for a set of training samples is simply:

$$(\quad) = \frac{1}{\quad} \quad_{=1} \quad (\qquad) \tag{8}$$

where is the output of the -th RBF unit, i.e. the one that corresponds to the correct class of input pattern . While this cost function is appropriate for most cases, it lacks three important properties. First, if we allow the parameters of the RBF to adapt, ( ) has a trivial, but totally unacceptable, solution. In this solution, all the RBF parameter vectors are equal, and the state of F6 is constant and equal to that parameter vector. In this case the network happily ignores the input, and all the RBF outputs are equal to zero. This collapsing phenomenon does not occur if the RBF weights are not allowed to adapt. The second problem is that there is no competition between the classes. Such a competition can be obtained by using a more discriminative training criterion, dubbed the MAP (maximum a posteriori) criterion, similar to Maximum Mutual Information criterion sometimes used to train HMMs [48], [49], [50]. It corresponds to maximizing the posterior probability of the correct class (or minimizing the logarithm of the probability of the correct class), given that the input image can come from one of the classes or from a background "rubbish" class label. In terms of penalties, it means that in addition to pushing down the penalty of the correct class like the MSE criterion, this criterion also pulls up the penalties of the incorrect classes:

$$(\quad) = \frac{1}{\quad} \quad_{=1} \quad (\quad (\qquad) + \log(\quad^{-} \quad + \qquad ^{-(\qquad)})) \tag{9}$$

The negative of the second term plays a "competitive" role. It is necessarily smaller than (or equal to) the first term, therefore this loss function is positive. The constant is positive, and prevents the penalties of classes that are already very large from being pushed further up. The posterior probability of this rubbish class label would be the ratio of $^{-}$ and $^{-} + \quad ^{-(\quad)}$. This discriminative criterion prevents the previously mentioned "collapsing effect" when the RBF parameters are learned because it keeps the RBF centers apart from each other. In Section VI, we present a generalization of this criterion for systems that learn to classify multiple objects in the input (e.g., characters in words or in documents).

Computing the gradient of the loss function with respect to all the weights in all the layers of the convolutional network is done with back-propagation. The standard algorithm must be slightly modified to take account of the weight sharing. An easy way to implement it is to first compute the partial derivatives of the loss function with respect to each *connection*, as if the network were a conventional multi-layer network without weight sharing. Then the partial derivatives of all the connections that share a same parameter are added to form the derivative with respect to that parameter.

Such a large architecture can be trained very efficiently, but doing so requires the use of a few techniques that are described in the appendix. Section A of the appendix describes details such as the particular sigmoid used, and the weight initialization. Section B and C describe the minimization procedure used, which is a stochastic version of a diagonal approximation to the Levenberg-Marquardt procedure.

III. RESULTS AND COMPARISON WITH OTHER METHODS

While recognizing individual digits is only one of many problems involved in designing a practical recognition system, it is an excellent benchmark for comparing shape recognition methods. Though many existing method combine a hand-crafted feature extractor and a trainable classifier, this study concentrates on adaptive methods that operate directly on size-normalized images.

A. Database: the Modified NIST set

The database used to train and test the systems described in this paper was constructed from the NIST's Special Database 3 and Special Database 1 containing binary images of handwritten digits. NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore it was necessary to build a new database by mixing NIST's datasets.

SD-1 contains 58,527 digit images written by 500 different writers. In contrast to SD-3, where blocks of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identities for SD-1 are available and we used this information to unscramble the writers. We then split SD-1 in two: characters written by the first 250 writers went into our new training set. The remaining 250 writers were placed in our test set. Thus we had two sets with nearly 30,000 examples each. The new training set was completed with enough examples from SD-3, starting at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set was completed with SD-3 examples starting at pattern # 35,000 to make a full set with 60,000 test patterns. In the experiments described here, we only used a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3), but we used the full 60,000 training samples. The resulting database was called the Modified NIST, or MNIST, dataset.

The original black and white (bilevel) images were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as result of the anti-aliasing (image interpolation) technique used by the normalization algorithm. Three versions of the database were used. In the first version, the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. In some instances, this 28x28 field was extended to 32x32 with background pixels. This version of the database will be referred to as the *regular* database. In the second version of the database, the character images were deslanted and cropped down to 20x20 pixels images. The deslanting computes the second moments of inertia of the pixels (counting a foreground pixel as 1 and a background pixel as 0), and shears the image by horizontally shifting the lines so that the principal axis is vertical. This version of the database will be referred to as the *deslanted* database. In the third version of the database, used in some early experiments, the images were reduced to 16x16 pixels. The regular database (60,000 training examples, 10,000 test examples size-normalized to 20x20, and centered by center of mass in 28x28 fields) is available at ˜ . Figure 4 shows examples randomly picked from the test set.

B. Results

Several versions of LeNet-5 were trained on the regular MNIST database. 20 iterations through the entire training data were performed for each session. The values of the global learning rate (see Equation 21 in Appendix C for a definition) was decreased using the following schedule: 0.0005 for the first two passes, 0.0002 for the next three, 0.0001 for the next three, 0.00005 for the next 4, and 0.00001 thereafter. Before each iteration, the diagonal Hessian approximation was reevaluated on 500 samples, as described in Appendix C and kept fixed during the entire iteration. The parameter was set to 0.02. The resulting effective learning rates during the first pass varied between approximately 7 10^{-5} and 0 016 over the set of parameters. The test error rate stabilizes after around 10 passes through the training set at 0.95%. The error rate on the training set reaches 0.35% after 19 passes. Many authors have reported observing the common phenomenon of over-training when training neural networks or other adaptive algorithms on various tasks. When over-training occurs, the training error keeps decreasing over time, but the test error goes through a minimum and starts increasing after a certain number of iterations. While this phenomenon is very common, it was not observed in our case as the learning curves in figure 5 show. A possible reason is that the learning rate was kept relatively large. The effect of this is that the weights never settle down in the local minimum but keep oscillating randomly. Because of those fluctuations, the average cost will be lower in a broader minimum. Therefore, stochastic gradient will have a similar effect as a regularization term that favors broader minima. Broader minima correspond to solutions with large entropy of the parameter distribution, which is beneficial to the generalization error.

Fig. 4. Size-normalized examples from the MNIST database.

The influence of the training set size was measured by training the network with 15,000, 30,000, and 60,000 examples. The resulting training error and test error are shown in figure 6. It is clear that, even with specialized architectures such as LeNet-5, more training data would improve the accuracy.

To verify this hypothesis, we artificially generated more training examples by randomly distorting the original training images. The increased training set was composed of the 60,000 original patterns plus 540,000 instances of

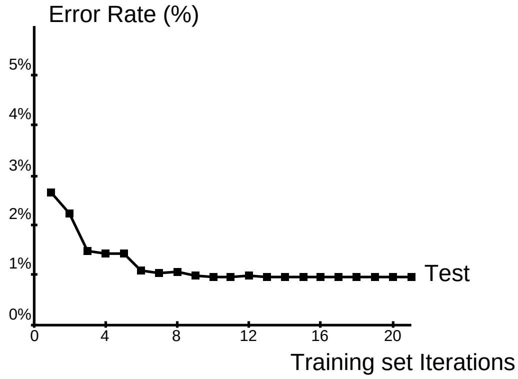

Fig. 5. Training and test error of LeNet-5 as a function of the number of passes through the 60,000 pattern training set (without distortions). The average training error is measured on-the-fly as training proceeds. This explains why the training error appears to be larger than the test error. Convergence is attained after 10 to 12 passes through the training set.

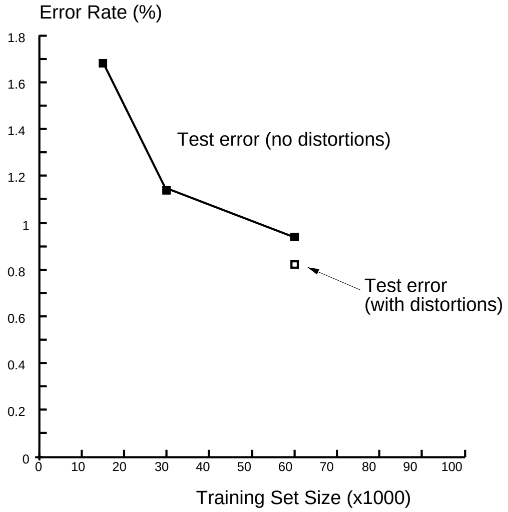

Fig. 6. Training and test errors of LeNet-5 achieved using training sets of various sizes. This graph suggests that a larger training set could improve the performance of LeNet-5. The hollow square show the test error when more training patterns are artificially generated using random distortions. The test patterns are not distorted.

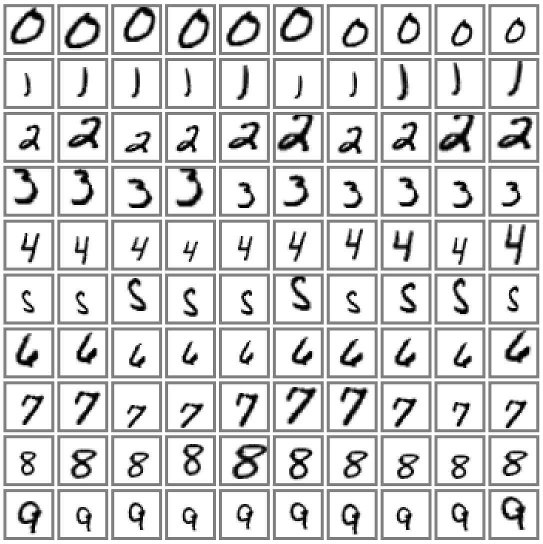

Fig. 7. Examples of distortions of ten training patterns.

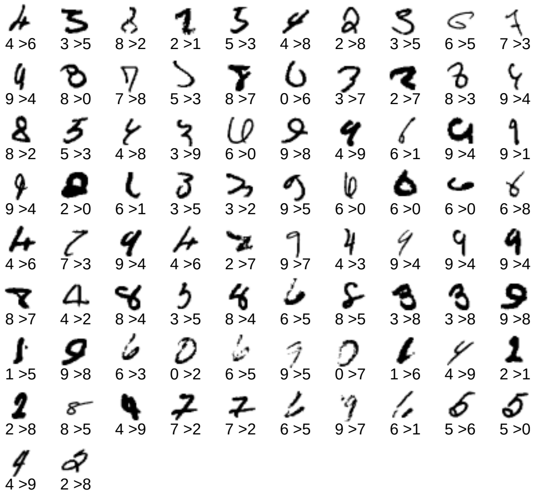

Fig. 8. The 82 test patterns misclassified by LeNet-5. Below each image is displayed the correct answers (left) and the network answer (right). These errors are mostly caused either by genuinely ambiguous patterns, or by digits written in a style that are under-represented in the training set.

distorted patterns with randomly picked distortion parameters. The distortions were combinations of the following planar affine transformations: horizontal and vertical translations, scaling, squeezing (simultaneous horizontal compression and vertical elongation, or the reverse), and horizontal shearing. Figure 7 shows examples of distorted patterns used for training. When distorted data was used for training, the test error rate dropped to 0.8% (from 0.95% without deformation). The same training parameters were used as without deformations. The total length of the training session was left unchanged (20 passes of 60,000 patterns each). It is interesting to note that the network effectively sees each individual sample only twice over the course of these 20 passes.

Figure 8 shows all 82 misclassified test examples. some of those examples are genuinely ambiguous, but several are perfectly identifiable by humans, although they are written in an under-represented style. This shows that further improvements are to be expected with more training data.

C. Comparison with Other Classifiers

For the sake of comparison, a variety of other trainable classifiers was trained and tested on the same database. An early subset of these results was presented in [51]. The error rates on the test set for the various methods are shown in figure 9.

C.1 Linear Classifier, and Pairwise Linear Classifier

Possibly the simplest classifier that one might consider is a linear classifier. Each input pixel value contributes to a weighted sum for each output unit. The output unit with the highest sum (including the contribution of a bias con-

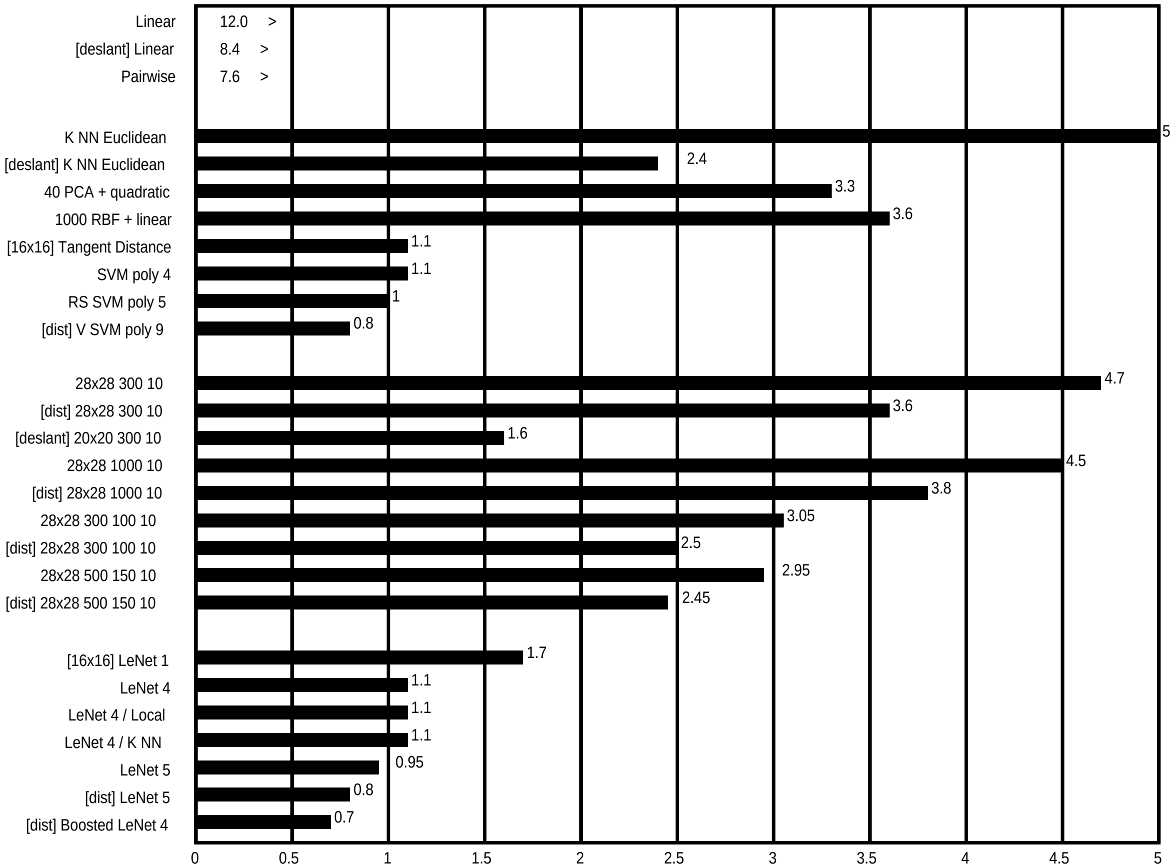

Fig. 9. Error rate on the test set (%) for various classification methods. [deslant] indicates that the classifier was trained and tested on the deslanted version of the database. [dist] indicates that the training set was augmented with artificially distorted examples. [16x16] indicates that the system used the 16x16 pixel images. The uncertainty in the quoted error rates is about 0.1%.

stant) indicates the class of the input character. On the regular data, the error rate is 12%. The network has 7850 free parameters. On the deslanted images, the test error rate is 8.4% The network has 4010 free parameters. The deficiencies of the linear classifier are well documented [1] and it is included here simply to form a basis of comparison for more sophisticated classifiers. Various combinations of sigmoid units, linear units, gradient descent learning, and learning by directly solving linear systems gave similar results.

A simple improvement of the basic linear classifier was tested [52]. The idea is to train each unit of a single-layer network to separate each class from each other class. In our case this layer comprises 45 units labeled 0/1, 0/2,...0/9, 1/2....8/9. Unit is trained to produce +1 on patterns of class , -1 on patterns of class , and is not trained on other patterns. The final score for class is the sum of the outputs all the units labeled minus the sum of the output of all the units labeled , for all and . The error rate on the regular test set was 7.6%.

C.2 Baseline Nearest Neighbor Classifier

Another simple classifier is a K-nearest neighbor classifier with a Euclidean distance measure between input images. This classifier has the advantage that no training time, and no brain on the part of the designer, are required. However, the memory requirement and recognition time are large: the complete 60,000 twenty by twenty pixel training images (about 24 Megabytes at one byte per pixel) must be available at run time. Much more compact representations could be devised with modest increase in error rate. On the regular test set the error rate was 5.0%. On the deslanted data, the error rate was 2.4%, with = 3. Naturally, a realistic Euclidean distance nearest-neighbor system would operate on feature vectors rather than directly on the pixels, but since all of the other systems presented in this study operate directly on the pixels, this result is useful for a baseline comparison.

C.3 Principal Component Analysis (PCA) and Polynomial Classifier

Following [53], [54], a preprocessing stage was constructed which computes the projection of the input pattern on the 40 principal components of the set of training vectors. To compute the principal components, the mean of each input component was first computed and subtracted from the training vectors. The covariance matrix of the resulting vectors was then computed and diagonalized using Singular Value Decomposition. The 40-dimensional feature vector was used as the input of a second degree polynomial classifier. This classifier can be seen as a linear classifier with 821 inputs, preceded by a module that computes all products of pairs of input variables. The error on the regular test set was 3.3%.

C.4 Radial Basis Function Network

Following [55], an RBF network was constructed. The first layer was composed of 1,000 Gaussian RBF units with 28x28 inputs, and the second layer was a simple 1000 inputs / 10 outputs linear classifier. The RBF units were divided into 10 groups of 100. Each group of units was trained on all the training examples of one of the 10 classes using the adaptive K-means algorithm. The second layer weights were computed using a regularized pseudo-inverse method. The error rate on the regular test set was 3.6%

C.5 One-Hidden Layer Fully Connected Multilayer Neural Network

Another classifier that we tested was a fully connected multi-layer neural network with two layers of weights (one hidden layer) trained with the version of back-propagation described in Appendix C. Error on the regular test set was 4.7% for a network with 300 hidden units, and 4.5% for a network with 1000 hidden units. Using artificial distortions to generate more training data brought only marginal improvement: 3.6% for 300 hidden units, and 3.8% for 1000 hidden units. When deslanted images were used, the test error jumped down to 1.6% for a network with 300 hidden units.

It remains somewhat of a mystery that networks with such a large number of free parameters manage to achieve reasonably low testing errors. We conjecture that the dynamics of gradient descent learning in multilayer nets has a "self-regularization" effect. Because the origin of weight space is a saddle point that is attractive in almost every direction, the weights invariably shrink during the first few epochs (recent theoretical analysis seem to confirm this [56]). Small weights cause the sigmoids to operate in the quasi-linear region, making the network essentially equivalent to a low-capacity, single-layer network. As the learning proceeds, the weights grow, which progressively increases the effective capacity of the network. This seems to be an almost perfect, if fortuitous, implementation of Vapnik's "Structural Risk Minimization" principle [6]. A better theoretical understanding of these phenomena, and more empirical evidence, are definitely needed.

C.6 Two-Hidden Layer Fully Connected Multilayer Neural Network

To see the effect of the architecture, several two-hidden layer multilayer neural networks were trained. Theoretical results have shown that any function can be approximated by a one-hidden layer neural network [57]. However, several authors have observed that two-hidden layer architectures sometimes yield better performance in practical situations. This phenomenon was also observed here. The test error rate of a 28x28-300-100-10 network was 3.05%, a much better result than the one-hidden layer network, obtained using marginally more weights and connections. Increasing the network size to 28x28-1000-150-10 yielded only marginally improved error rates: 2.95%. Training with distorted patterns improved the performance somewhat: 2.50% error for the 28x28-300-100-10 network, and 2.45% for the 28x28-1000-150-10 network.

C.7 A Small Convolutional Network: LeNet-1

Convolutional Networks are an attempt to solve the dilemma between small networks that cannot learn the training set, and large networks that seem overparameterized. LeNet-1 was an early embodiment of the Convolutional Network architecture which is included here for comparison purposes. The images were down-sampled to 16x16 pixels and centered in the 28x28 input layer. Although about 100,000 multiply/add steps are required to evaluate LeNet-1, its convolutional nature keeps the number of free parameters to only about 2600. The LeNet-1 architecture was developed using our own version of the USPS (US Postal Service zip codes) database and its size was tuned to match the available data [35]. LeNet-1 achieved 1.7% test error. The fact that a network with such a small number of parameters can attain such a good error rate is an indication that the architecture is appropriate for the task.

C.8 LeNet-4

Experiments with LeNet-1 made it clear that a larger convolutional network was needed to make optimal use of the large size of the training set. LeNet-4 and later LeNet-5 were designed to address this problem. LeNet-4 is very similar to LeNet-5, except for the details of the architecture. It contains 4 first-level feature maps, followed by 8 subsampling maps connected in pairs to each first-layer feature maps, then 16 feature maps, followed by 16 subsampling map, followed by a fully connected layer with 120 units, followed by the output layer (10 units). LeNet-4 contains about 260,000 connections and has about 17,000 free parameters. Test error was 1.1%. In a series of experiments, we replaced the last layer of LeNet-4 with a Euclidean Nearest Neighbor classifier, and with the "local learning" method of Bottou and Vapnik [58], in which a local linear classifier is retrained each time a new test pattern is shown. Neither of those methods improved the raw error rate, although they did improve the rejection performance.

C.9 Boosted LeNet-4

Following theoretical work by R. Schapire [59], Drucker et al. [60] developed the "boosting" method for combining multiple classifiers. Three LeNet-4s are combined: the first one is trained the usual way. the second one is trained on patterns that are filtered by the first net so that the second machine sees a mix of patterns, 50% of which the first net got right, and 50% of which it got wrong. Finally, the third net is trained on new patterns on which the first and the second nets disagree. During testing, the outputs of the three nets are simply added. Because the error rate of LeNet-4 is very low, it was necessary to use the artificially distorted images (as with LeNet-5) in order to get enough samples to train the second and third nets. The test error rate was 0.7%, the best of any of our classifiers. At first glance, boosting appears to be three times more expensive as a single net. In fact, when the first net produces a high confidence answer, the other nets are not called. The average computational cost is about 1.75 times that of a single net.

C.10 Tangent Distance Classifier (TDC)

The Tangent Distance classifier (TDC) is a nearest-neighbor method where the distance function is made insensitive to small distortions and translations of the input image [61]. If we consider an image as a point in a high dimensional pixel space (where the dimensionality equals the number of pixels), then an evolving distortion of a character traces out a curve in pixel space. Taken together, all these distortions define a low-dimensional manifold in pixel space. For small distortions, in the vicinity of the original image, this manifold can be approximated by a plane, known as the tangent plane. An excellent measure of "closeness" for character images is the distance between their tangent planes, where the set of distortions used to generate the planes includes translations, scaling, skewing, squeezing, rotation, and line thickness variations. A test error rate of 1.1% was achieved using 16x16 pixel images. Prefiltering techniques using simple Euclidean distance at multiple resolutions allowed to reduce the number of necessary Tangent Distance calculations.

C.11 Support Vector Machine (SVM)

Polynomial classifiers are well-studied methods for generating complex decision surfaces. Unfortunately, they are impractical for high-dimensional problems, because the number of product terms is prohibitive. The Support Vector technique is an extremely economical way of representing complex surfaces in high-dimensional spaces, including polynomials and many other types of surfaces [6].

A particularly interesting subset of decision surfaces is the ones that correspond to hyperplanes that are at a maximum distance from the convex hulls of the two classes in the high-dimensional space of the product terms. Boser, Guyon, and Vapnik [62] realized that any polynomial of degree in this "maximum margin" set can be computed by first computing the dot product of the input image with a subset of the training samples (called the "support vectors"), elevating the result to the -th power, and linearly combining the numbers thereby obtained. Finding the support vectors and the coefficients amounts to solving a high-dimensional quadratic minimization problem with linear inequality constraints. For the sake of comparison, we include here the results obtained by Burges and Schölkopf reported in [63]. With a regular SVM, their error rate on the regular test set was 1.4%. Cortes and Vapnik had reported an error rate of 1.1% with SVM on the same data using a slightly different technique. The computational cost of this technique is very high: about 14 million multiply-adds per recognition. Using Schölkopf's Virtual Support Vectors technique (V-SVM), 1.0% error was attained. More recently, Schölkopf (personal communication) has reached 0.8% using a modified version of the V-SVM. Unfortunately, V-SVM is extremely expensive: about twice as much as regular SVM. To alleviate this problem, Burges has proposed the Reduced Set Support Vector technique (RS-SVM), which attained 1.1% on the regular test set [63], with a computational cost of only 650,000 multiply-adds per recognition, i.e. only about 60% more expensive than LeNet-5.

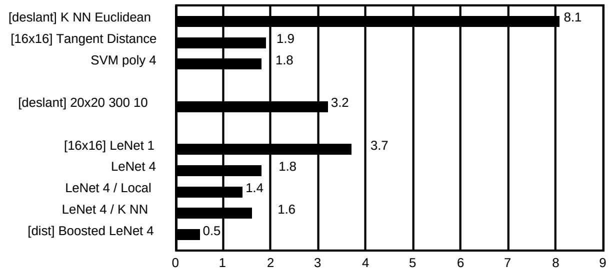

Fig. 10. Rejection Performance: percentage of test patterns that must be rejected to achieve 0.5% error for some of the systems.

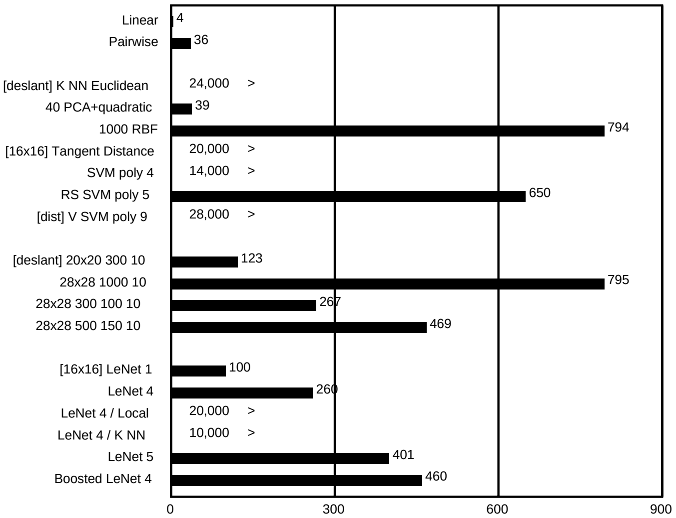

Fig. 11. Number of multiply-accumulate operations for the recognition of a single character starting with a size-normalized image.

D. Discussion

A summary of the performance of the classifiers is shown in Figures 9 to 12. Figure 9 shows the raw error rate of the classifiers on the 10,000 example test set. Boosted LeNet-4 performed best, achieving a score of 0.7%, closely followed by LeNet-5 at 0.8%.

Figure 10 shows the number of patterns in the test set that must be rejected to attain a 0.5% error for some of the methods. Patterns are rejected when the value of corresponding output is smaller than a predefined threshold. In many applications, rejection performance is more significant than raw error rate. The score used to decide upon the rejection of a pattern was the difference between the scores of the top two classes. Again, Boosted LeNet-4 has the best performance. The enhanced versions of LeNet-4 did better than the original LeNet-4, even though the raw

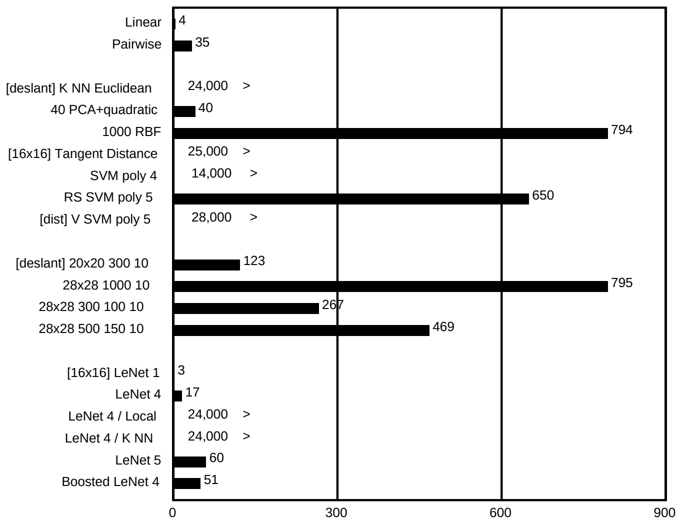

Fig. 12. Memory requirements, measured in number of variables, for each of the methods. Most of the methods only require one byte per variable for adequate performance.

accuracies were identical.

Figure 11 shows the number of multiply-accumulate operations necessary for the recognition of a single size-normalized image for each method. Expectedly, neural networks are much less demanding than memory-based methods. Convolutional Neural Networks are particularly well suited to hardware implementations because of their regular structure and their low memory requirements for the weights. Single chip mixed analog-digital implementations of LeNet-5's predecessors have been shown to operate at speeds in excess of 1000 characters per second [64]. However, the rapid progress of mainstream computer technology renders those exotic technologies quickly obsolete. Cost-effective implementations of memory-based techniques are more elusive, due to their enormous memory requirements, and computational requirements.

Training time was also measured. K-nearest neighbors and TDC have essentially zero training time. While the single-layer net, the pairwise net, and PCA+quadratic net could be trained in less than an hour, the multilayer net training times were expectedly much longer, but only required 10 to 20 passes through the training set. This amounts to 2 to 3 days of CPU to train LeNet-5 on a Silicon Graphics Origin 2000 server, using a single 200MHz R10000 processor. It is important to note that while the training time is somewhat relevant to the designer, it is of little interest to the final user of the system. Given the choice between an existing technique, and a new technique that brings marginal accuracy improvements at the price of considerable training time, any final user would chose the latter.

Figure 12 shows the memory requirements, and therefore the number of free parameters, of the various classifiers measured in terms of the number of variables that need to be stored. Most methods require only about one byte per variable for adequate performance. However, Nearest-Neighbor methods may get by with 4 bits per pixel for storing the template images. Not surprisingly, neural networks require much less memory than memory-based methods.

The Overall performance depends on many factors including accuracy, running time, and memory requirements. As computer technology improves, larger-capacity recognizers become feasible. Larger recognizers in turn require larger training sets. LeNet-1 was appropriate to the available technology in 1989, just as LeNet-5 is appropriate now. In 1989 a recognizer as complex as LeNet-5 would have required several weeks' training, and more data than was available, and was therefore not even considered. For quite a long time, LeNet-1 was considered the state of the art. The local learning classifier, the optimal margin classifier, and the tangent distance classifier were developed to improve upon LeNet-1 – and they succeeded at that. However, they in turn motivated a search for improved neural network architectures. This search was guided in part by estimates of the capacity of various learning machines, derived from measurements of the training and test error as a function of the number of training examples. We discovered that more capacity was needed. Through a series of experiments in architecture, combined with an analysis of the characteristics of recognition errors, LeNet-4 and LeNet-5 were crafted.

We find that boosting gives a substantial improvement in accuracy, with a relatively modest penalty in memory and computing expense. Also, distortion models can be used to increase the effective size of a data set without actually requiring to collect more data.

The Support Vector Machine has excellent accuracy, which is most remarkable, because unlike the other high performance classifiers, it does not include *a priori* knowledge about the problem. In fact, this classifier would do just as well if the image pixels were permuted with a fixed mapping and lost their pictorial structure. However, reaching levels of performance comparable to the Convolutional Neural Networks can only be done at considerable expense in memory and computational requirements. The reduced-set SVM requirements are within a factor of two of the Convolutional Networks, and the error rate is very close. Improvements of those results are expected, as the technique is relatively new.

When plenty of data is available, many methods can attain respectable accuracy. The neural-net methods run much faster and require much less space than memory-based techniques. The neural nets' advantage will become more striking as training databases continue to increase in size.

E. Invariance and Noise Resistance

Convolutional networks are particularly well suited for recognizing or rejecting shapes with widely varying size, position, and orientation, such as the ones typically produced by heuristic segmenters in real-world string recognition systems.

In an experiment like the one described above, the importance of noise resistance and distortion invariance is not obvious. The situation in most real applications is quite different. Characters must generally be segmented out of their context prior to recognition. Segmentation algorithms are rarely perfect and often leave extraneous marks in character images (noise, underlines, neighboring characters), or sometimes cut characters too much and produce incomplete characters. Those images cannot be reliably size-normalized and centered. Normalizing incomplete characters can be very dangerous. For example, an enlarged stray mark can look like a genuine 1. Therefore many systems have resorted to normalizing the images at the level of fields or words. In our case, the upper and lower profiles of entire fields (amounts in a check) are detected and used to normalize the image to a fixed height. While this guarantees that stray marks will not be blown up into character-looking images, this also creates wide variations of the size and vertical position of characters after segmentation. Therefore it is preferable to use a recognizer that is robust to such variations. Figure 13 shows several examples of distorted characters that are correctly recognized by LeNet-5. It is estimated that accurate recognition occurs for scale variations up to about a factor of 2, vertical shift variations of plus or minus about half the height of the character, and rotations up to plus or minus 30 degrees. While fully invariant recognition of complex shapes is still an elusive goal, it seems that Convolutional Networks offer a partial answer to the problem of invariance or robustness with respect to geometrical distortions.

Figure 13 includes examples of the robustness of LeNet-5 under extremely noisy conditions. Processing those images would pose unsurmountable problems of segmentation and feature extraction to many methods, but LeNet-5 seems able to robustly extract salient features from these cluttered images. The training set used for the network shown here was the MNIST training set with salt and pepper noise added. Each pixel was randomly inverted with probability 0.1. More examples of LeNet-5 in action are available on the Internet at .

IV. MULTI-MODULE SYSTEMS AND GRAPH TRANSFORMER NETWORKS

The classical back-propagation algorithm, as described and used in the previous sections, is a simple form of Gradient-Based Learning. However, it is clear that the gradient back-propagation algorithm given by Equation 4 describes a more general situation than simple multi-layer feed-forward networks composed of alternated linear transformations and sigmoidal functions. In principle, derivatives can be back-propagated through any arrangement of functional modules, as long as we can compute the product of the Jacobians of those modules by any vector. Why would we want to train systems composed of multiple heterogeneous modules? The answer is that large and complex trainable systems need to be built out of simple, specialized modules. The simplest example is LeNet-5, which mixes convolutional layers, sub-sampling layers, fully-connected layers, and RBF layers. Another less trivial example, described in the next two sections, is a system for recognizing words, that can be trained to simultaneously segment and recognize words, without ever being given the correct segmentation.

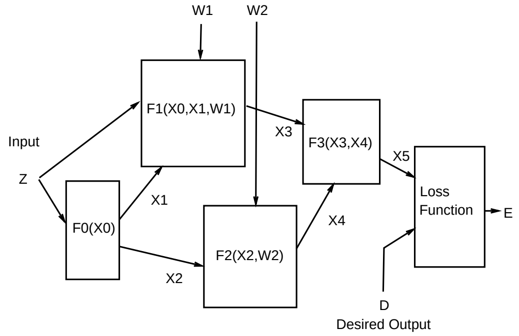

Fig. 14. A trainable system composed of heterogeneous modules.

Figure 14 shows an example of a trainable multi-modular system. A multi-module system is defined by the function implemented by each of the modules, and by the graph of interconnection of the modules to each other. The graph implicitly defines a partial order according to which the modules must be updated in the forward pass. For example in Figure 14, module 0 is first updated, then modules 1 and 2 are updated (possibly in parallel), and finally module 3. Modules may or may not have trainable parameters. Loss functions, which measure the performance of the system, are implemented as module 4. In the simplest case, the loss function module receives an external input that carries the desired output. In this framework, there is no qualitative difference between trainable parameters (in the figure), external inputs and outputs ( ), and intermediate state variables( ).

A. An Object-Oriented Approach

Object-Oriented programming offers a particularly convenient way of implementing multi-module systems. Each module is an instance of a class. Module classes have a "forward propagation" method (or member function) called whose arguments are the inputs and outputs of the module. For example, computing the output of module 3 in Figure 14 can be done by calling the method on module 3 with the arguments . Complex modules can be constructed from simpler modules by simply defining a new class whose slots will contain the member modules and the intermediate state variables between those modules. The method for the class simply calls the methods of the member modules, with the appropriate intermediate state variables or external input and outputs as arguments. Although the algorithms are easily generalizable to any network of such modules, including those whose influence graph has cycles, we will limit the discussion to the case of directed acyclic graphs (feed-forward networks).

Computing derivatives in a multi-module system is just as simple. A "backward propagation" method, called , for each module class can be defined for that purpose. The method of a module takes the same ar-

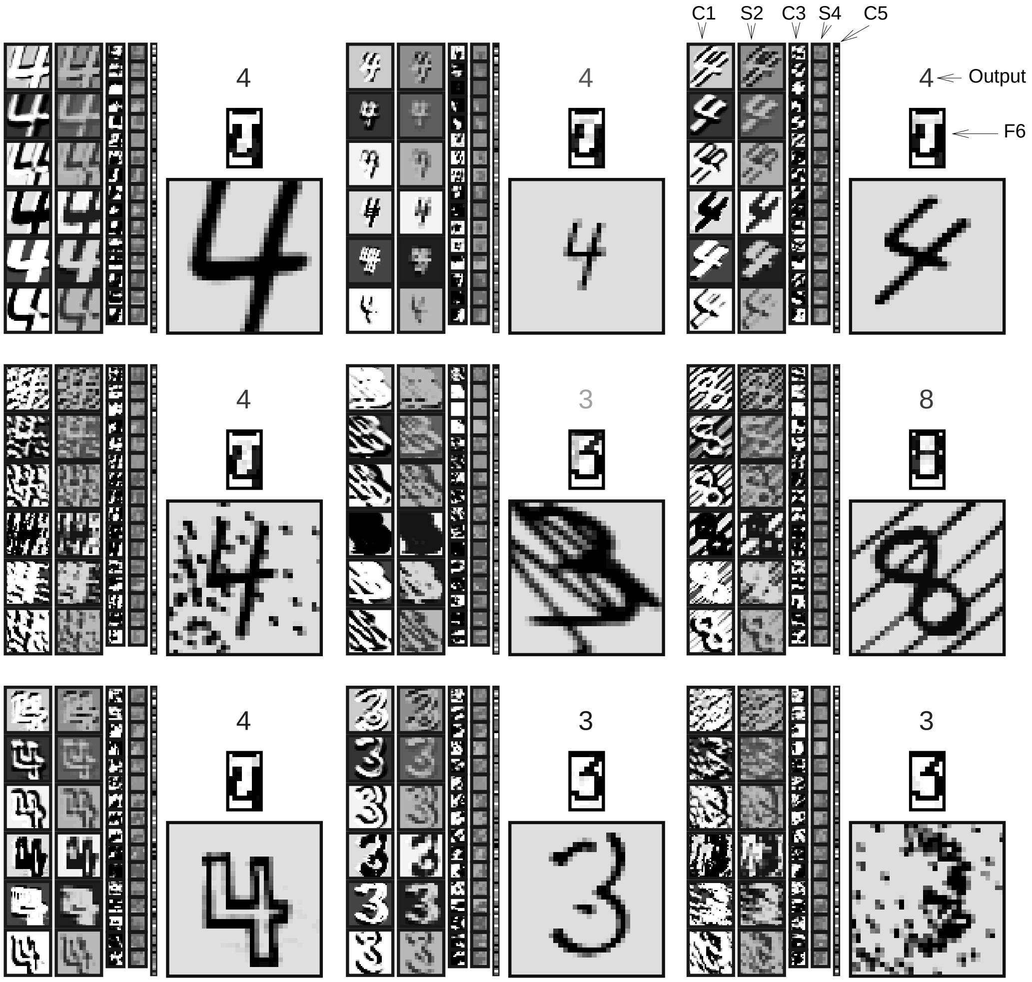

Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).

guments as the method. All the derivatives in the system can be computed by calling the method on all the modules in reverse order compared to the forward propagation phase. The state variables are assumed to contain slots for storing the gradients computed during the backward pass, in addition to storage for the states computed in the forward pass. The backward pass effectively computes the partial derivatives of the loss with respect to all the state variables and all the parameters in the system. There is an interesting duality property between the forward and backward functions of certain modules. For example, a sum of several variables in the forward direction is transformed into a simple fan-out (replication) in the backward direction. Conversely, a fan-out in the forward direction is transformed into a sum in the backward direction. The software environment used to obtain the results described in this paper, called SN3.1, uses the above concepts. It is based on a home-grown object-oriented dialect of Lisp with a compiler to C.

The fact that derivatives can be computed by propagation in the reverse graph is easy to understand intuitively. The best way to justify it theoretically is through the use of Lagrange functions [21], [22]. The same formalism can be used to extend the procedures to networks with recurrent connections.

B. *Special Modules*

Neural networks and many other standard pattern recognition techniques can be formulated in terms of multi-modular systems trained with Gradient-Based Learning. Commonly used modules include matrix multiplications and sigmoidal modules, the combination of which can be used to build conventional neural networks. Other modules include convolutional layers, sub-sampling layers, RBF layers, and "softmax" layers [65]. Loss functions are also represented as modules whose single output produces the value of the loss. Commonly used modules have simple methods. In general, the method of a function is a multiplication by the Jacobian of . Here are a few commonly used examples. The method of a fanout (a "Y" connection) is a sum, and vice versa. The method of a multiplication by a coefficient is a multiplication by the same coefficient. The method of a multiplication by a matrix is a multiplication by the transpose of that matrix. The method of an addition with a constant is the identity.

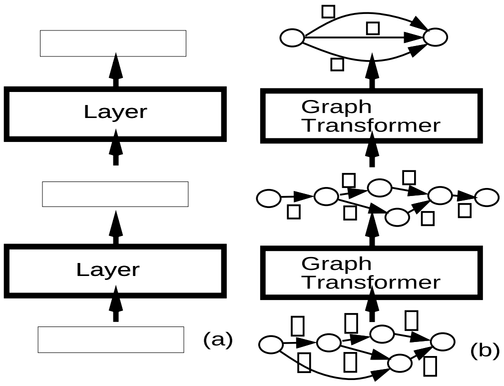

Fig. 15. Traditional neural networks, and multi-module systems communicate fixed-size vectors between layer. Multi-Layer Graph Transformer Networks are composed of trainable modules that operate on and produce graphs whose arcs carry numerical information.

Interestingly, certain non-differentiable modules can be inserted in a multi-module system without adverse effect. An interesting example of that is the multiplexer module. It has two (or more) regular inputs, one switching input, and one output. The module selects one of its inputs, depending upon the (discrete) value of the switching input, and copies it on its output. While this module is not differentiable with respect to the switching input, it is differentiable with respect to the regular inputs. Therefore the overall function of a system that includes such modules will be differentiable with respect to its parameters as long as the switching input does not depend upon the parameters. For example, the switching input can be an external input.

Another interesting case is the module. This module has two (or more) inputs and one output. The output of the module is the minimum of the inputs. The function of this module is differentiable everywhere, except on the switching surface which is a set of measure zero. Interestingly, this function is continuous and reasonably regular, and that is sufficient to ensure the convergence of a Gradient-Based Learning algorithm.

The object-oriented implementation of the multi-module idea can easily be extended to include a method that propagates Gauss-Newton approximations of the second derivatives. This leads to a direct generalization for modular systems of the second-derivative back-propagation Equation 22 given in the Appendix.

The multiplexer module is a special case of a much more general situation, described at length in Section VIII, where the architecture of the system changes dynamically with the input data. Multiplexer modules can be used to dynamically rewire (or reconfigure) the architecture of the system for each new input pattern.

C. Graph Transformer Networks

Multi-module systems are a very flexible tool for building large trainable system. However, the descriptions in the previous sections implicitly assumed that the set of parameters, and the state information communicated between the modules, are all fixed-size vectors. The limited flexibility of fixed-size vectors for data representation is a serious deficiency for many applications, notably for tasks that deal with variable length inputs (e.g continuous speech recognition and handwritten word recognition), or for tasks that require encoding relationships between objects or features whose number and nature can vary (invariant perception, scene analysis, recognition of composite objects). An important special case is the recognition of strings of characters or words.

More generally, fixed-size vectors lack flexibility for tasks in which the state must encode probability distributions over sequences of vectors or symbols as is the case in linguistic processing. Such distributions over sequences are best represented by stochastic grammars, or, in the more general case, *directed graphs in which each arc contains a vector* (stochastic grammars are special cases in which the vector contains probabilities and symbolic information). Each path in the graph represents a different sequence of vectors. Distributions over sequences can be represented by interpreting elements of the data associated with each arc as parameters of a probability distribution or simply as a penalty. Distributions over sequences are particularly handy for modeling linguistic knowledge in speech or handwriting recognition systems: each sequence, i.e., each path in the graph, represents an alternative interpretation of the input. Successive processing modules progressively refine the interpretation. For example, a speech recognition system might start with a single sequence of acoustic vectors, transform it into a lattice of phonemes (distribution over phoneme sequences), then into a lattice of words (distribution over word sequences), and then into a single sequence of words representing the best interpretation.

In our work on building large-scale handwriting recognition systems, we have found that these systems could much more easily and quickly be developed and designed by viewing the system as a networks of modules that take one or several graphs as input and produce graphs as output. Such modules are called *Graph Transformers*, and the complete systems are called *Graph Transformer Networks*, or GTN. Modules in a GTN communicate their states and gradients in the form of directed graphs whose arcs carry numerical information (scalars or vectors) [66].

From the statistical point of view, the fixed-size state vectors of conventional networks can be seen as representing the means of distributions in state space. In variable-size networks such as the Space-Displacement Neural Networks described in section VII, the states are variable-length sequences of fixed size vectors. They can be seen as representing the mean of a probability distribution over variable-length *sequences* of fixed-size vectors. In GTNs, the states are represented as graphs, which can be seen as representing mixtures of probability distributions over structured collections (possibly sequences) of vectors (Figure 15).

One of the main points of the next several sections is to show that Gradient-Based Learning procedures are not limited to networks of simple modules that communicate through fixed-size vectors, but can be generalized to GTNs. Gradient back-propagation through a Graph Transformer takes gradients with respect to the numerical information in the output graph, and computes gradients with respect to the numerical information attached to the input graphs, and with respect to the module's internal parameters. Gradient-Based Learning can be applied as long as differentiable functions are used to produce the *numerical data* in the output graph from the *numerical data* in the input graph and the functions parameters.

The second point of the next several sections is to show that the functions implemented by many of the modules used in typical document processing systems (and other image recognition systems), though commonly thought to be combinatorial in nature, are indeed differentiable with respect to their internal parameters as well as with respect to their inputs, and are therefore usable as part of a globally trainable system.

In most of the following, we will purposely avoid making references to probability theory. All the quantities manipulated are viewed as penalties, or costs, which if necessary can be transformed into probabilities by taking exponentials and normalizing.

V. Multiple Object Recognition: Heuristic Over-Segmentation

One of the most difficult problems of handwriting recognition is to recognize not just isolated characters, but strings of characters, such as zip codes, check amounts, or words. Since most recognizers can only deal with one character at a time, we must first *segment* the string into individual character images. However, it is almost impossible to devise image analysis techniques that will infallibly segment naturally written sequences of characters into well formed characters.

The recent history of automatic speech recognition [28], [67] is here to remind us that training a recognizer by optimizing a global criterion (at the word or sentence level) is much preferable to merely training it on hand-segmented phonemes or other units. Several recent works have shown that the same is true for handwriting recognition [38]: optimizing a word-level criterion is preferable to solely training a recognizer on pre-segmented characters because the recognizer can learn not only to recognize individual characters, but also to reject mis-segmented characters thereby minimizing the overall word error.

This section and the next describe in detail a simple example of GTN to address the problem of reading strings of characters, such as words or check amounts. The method avoids the expensive and unreliable task of hand-truthing the result of the segmentation often required in more traditional systems trained on individually labeled character images.

A. Segmentation Graph

A now classical method for word segmentation and recognition is called Heuristic Over-Segmentation [68], [69]. Its main advantages over other approaches to segmentation are that it avoids making hard decisions about the segmentation by taking a large number of different segmentations into consideration. The idea is to use heuristic image processing techniques to find candidate cuts of the word or string, and then to use the recognizer to score the alternative segmentations thereby generated. The process is depicted in Figure 16. First, a number of candidate cuts are generated. Good candidate locations for cuts can be found by locating minima in the vertical projection profile, or minima of the distance between the upper and lower contours of the word. Better segmentation heuristics are described in section X. The cut generation heuristic is designed so as to generate more cuts than necessary, in the hope that the "correct" set of cuts will be included. Once the cuts have been generated, alternative segmentations are best represented by a graph, called the *segmentation graph*. The segmentation graph is a *Directed Acyclic Graph* (DAG) with a start node and an end node. Each internal node is associated with a candidate cut produced by the segmentation algorithm. Each arc between a source node and a destination node is associated with an image that contains all the ink between the cut associated with the source node and the cut associated with the destination node. An arc is created between two nodes if the segmentor decided that the ink between the corresponding cuts could form a candidate character. Typically, each individual piece of ink would be associated with an arc. Pairs of successive pieces of ink would also be included, unless they are separated by a wide gap, which is a clear indication that they belong to different characters. Each complete path through the graph contains each piece of ink once and only once. Each path corresponds to a different way of associating pieces of ink together so as to form characters.

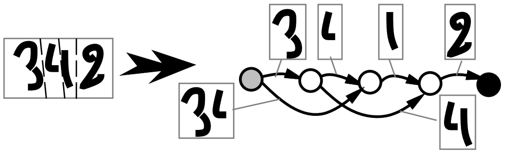

Fig. 16. Building a segmentation graph with Heuristic Over-Segmentation.

B. Recognition Transformer and Viterbi Transformer

A simple GTN to recognize character strings is shown in Figure 17. It is composed of two graph transformers called the *recognition transformer* , and the *Viterbi transformer* . The goal of the recognition transformer is to generate a graph, called the *interpretation graph* or *recognition graph* , that contains all the possible interpretations for all the possible segmentations of the input. Each path in represents one possible interpretation of one particular segmentation of the input. The role of the Viterbi transformer is to extract the best interpretation from the interpretation graph.

The recognition transformer takes the segmentation graph as input, and applies the recognizer for single characters to the images associated with each of the arcs

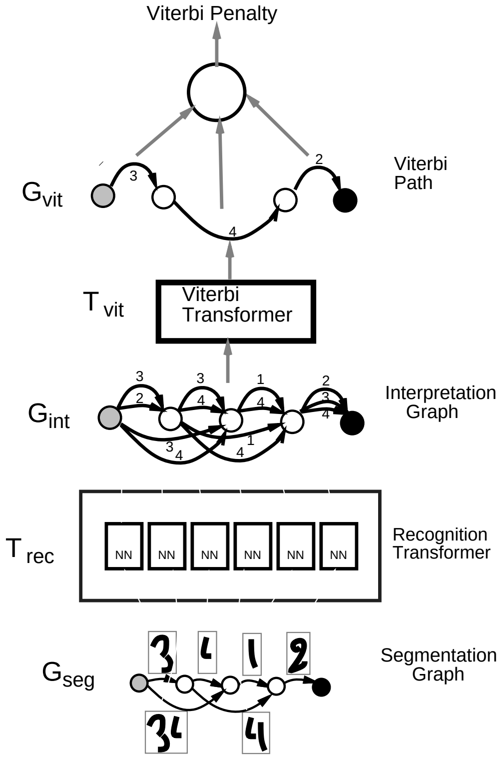

Fig. 17. Recognizing a character string with a GTN. For readability, only the arcs with low penalties are shown.

in the segmentation graph. The interpretation graph has almost the same structure as the segmentation graph, except that each arc is replaced by a set of arcs from and to the same node. In this set of arcs, there is one arc for each possible class for the image associated with the corresponding arc in . As shown in Figure 18, to each arc is attached a class label, and the penalty that the image belongs to this class as produced by the recognizer. If the segmentor has computed penalties for the candidate segments, these penalties are combined with the penalties computed by the character recognizer, to obtain the penalties on the arcs of the interpretation graph. Although combining penalties of different nature seems highly heuristic, the GTN training procedure will tune the penalties and take advantage of this combination anyway. Each path in the interpretation graph corresponds to a possible interpretation of the input word. The penalty of a particular interpretation for a particular segmentation is given by the sum of the arc penalties along the corresponding path in the interpretation graph. Computing the penalty of an interpretation independently of the segmentation requires to combine the penalties of all the paths with that interpretation. An appropriate rule for combining the penalties of parallel paths is given in section VI-C.

The Viterbi transformer produces a graph with a single path. This path is the path of least cumulated penalty in the Interpretation graph. The result of the recognition can be produced by reading off the labels of the arcs along the graph extracted by the Viterbi transformer. The Viterbi transformer owes its name to the famous *Viterbi algorithm* [70], an application of the principle of dynamic programming to find the shortest path in a graph efficiently. Let be the penalty associated to arc , with source node , and destination node (note that there can be multiple arcs between two nodes). In the interpretation graph, arcs also have a label . The Viterbi algorithm proceeds as follows. Each node is associated with a cumulated Viterbi penalty . Those cumulated penalties are computed in any order that satisfies the partial order defined by the interpretation graph (which is directed and acyclic). The start node is initialized with the cumulated penalty $_{\text{start}} = 0$. The other nodes cumulated penalties are computed recursively from the values of their parent nodes, through the upstream arcs = arc with destination = :

$$= \min_{\in} (\; + \;) \qquad (10)$$

Furthermore, the value of for each node which minimizes the right hand side is noted , the minimizing entering arc. When the end node is reached we obtain in $_{\text{end}}$ the total penalty of the path with the smallest total penalty. We call this penalty the *Viterbi penalty*, and this sequence of arcs and nodes the *Viterbi path*. To obtain the Viterbi path with nodes $_1$ and arcs $_1$ $_{-1}$, we trace back these nodes and arcs as follows, starting with = the end node, and recursively using the minimizing entering arc: = , and = until the start node is reached. The label sequence can then be read off the arcs of the Viterbi path.

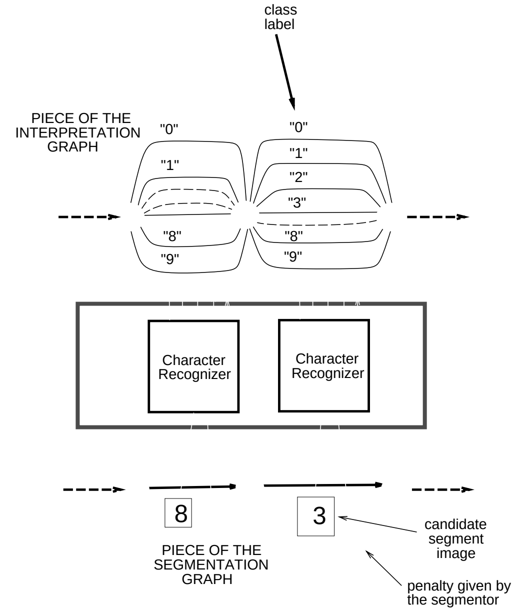

Fig. 18. The recognition transformer refines each arc of the segmentation arc into a set of arcs in the interpretation graph, one per character class, with attached penalties and labels.

VI. Global Training for Graph Transformer Networks

The previous section describes the process of recognizing a string using Heuristic Over-Segmentation, assuming that the recognizer is trained so as to give low penalties for the correct class label of correctly segmented characters, high penalties for erroneous categories of correctly segmented characters, and high penalties for all categories for badly formed characters. This section explains how to train the system at the string level to do the above without requiring manual labeling of character segments. This training will be performed with a GTN whose architecture is slightly different from the recognition architecture described in the previous section.

In many applications, there is enough a priori knowledge about what is expected from each of the modules in order to train them separately. For example, with Heuristic Over-Segmentation one could individually label single-character images and train a character recognizer on them, but it might be difficult to obtain an appropriate set of non-character images to train the model to reject wrongly segmented candidates. Although separate training is simple, it requires additional supervision information that is often lacking or incomplete (the correct segmentation and the labels of incorrect candidate segments). Furthermore it can be shown that separate training is sub-optimal [67].

The following section describes three different gradient-based methods for training GTN-based handwriting recognizers at the string level: Viterbi training, discriminative Viterbi training, forward training, and discriminative forward training. The last one is a generalization to graph-based systems of the MAP criterion introduced in Section II-C. Discriminative forward training is somewhat similar to the so-called Maximum Mutual Information criterion used to train HMM in speech recognition. However, our rationale differs from the classical one. We make no recourse to a probabilistic interpretation, but show that, within the Gradient-Based Learning approach, discriminative training is a simple instance of the pervasive principle of error correcting learning.

Training methods for graph-based sequence recognition systems such as HMMs have been extensively studied in the context of speech recognition [28]. Those methods require that the system be based on probabilistic *generative* models of the data, which provide normalized likelihoods over the space of possible input sequences. Popular HMM learning methods, such as the the Baum-Welsh algorithm, rely on this normalization. The normalization cannot be preserved when non-generative models such as neural networks are integrated into the system. Other techniques, such as discriminative training methods, must be used in this case. Several authors have proposed such methods to train neural network/HMM speech recognizers at the word or sentence level [71], [72], [73], [74], [75], [76], [77], [78], [29], [67].

Other globally trainable sequence recognition systems avoid the difficulties of statistical modeling by not resorting to graph-based techniques. The best example is Recurrent Neural Networks (RNN). Unfortunately, despite early enthusiasm, the training of RNNs with gradient-based techniques has proved very difficult in practice [79].

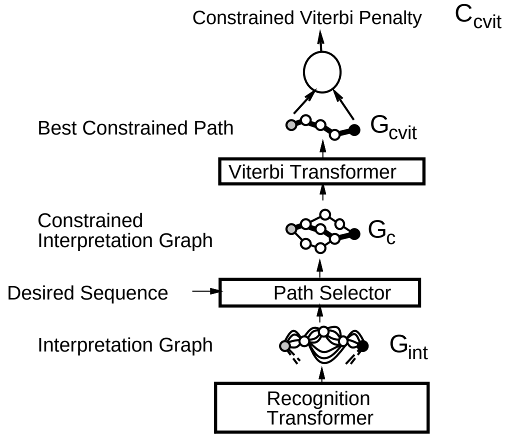

Fig. 19. Viterbi Training GTN Architecture for a character string recognizer based on Heuristic Over-Segmentation.

The GTN techniques presented below simplify and generalize the global training methods developed for speech recognition.

A. Viterbi Training

During recognition, we select the path in the Interpretation Graph that has the lowest penalty with the Viterbi algorithm. Ideally, we would like this path of lowest penalty to be associated with the correct label sequence as often as possible. An obvious loss function to minimize is therefore the average over the training set of the penalty of the path *associated with the correct label sequence* that has the lowest penalty. The goal of training will be to find the set of recognizer parameters (the weights, if the recognizer is a neural network) that minimize the average penalty of this "correct" lowest penalty path. The gradient of this loss function can be computed by back-propagation through the GTN architecture shown in figure 19. This training architecture is almost identical to the recognition architecture described in the previous section, except that an extra graph transformer called a *path selector* is inserted between the Interpretation Graph and the Viterbi Transformer. This transformer takes the interpretation graph and the desired label sequence as input. It extracts from the interpretation graph those paths that contain the correct (desired) label sequence. Its output graph $_{\mathrm{c}}$ is called the *constrained interpretation graph* (also known as *forced alignment* in the HMM literature), and contains all the paths that correspond to the correct label sequence. The constrained interpretation graph is then sent to the Viterbi transformer which produces a graph $_{\mathrm{cvit}}$ with a single path. This path is the "correct" path with the lowest penalty. Finally, a path scorer transformer takes $_{\mathrm{cvit}}$, and simply computes its cumulated penalty $_{\mathrm{cvit}}$ by adding up the penalties along the path. The output of this GTN is the loss function for the current pattern:

$$ {}_{\text{vit}} = {}_{\text{cvit}} \tag{11} $$

The only label information that is required by the above system is the sequence of desired character labels. No knowledge of the correct segmentation is required on the part of the supervisor, since it chooses among the segmentations in the interpretation graph the one that yields the lowest penalty.

The process of back-propagating gradients through the Viterbi training GTN is now described. As explained in section IV, the gradients must be propagated backwards through all modules of the GTN, in order to compute gradients in preceding modules and thereafter tune their parameters. Back-propagating gradients through the path scorer is quite straightforward. The partial derivatives of the loss function with respect to the individual penalties on the constrained Viterbi path ${}_{\text{cvit}}$ are equal to 1, since the loss function is simply the sum of those penalties. Back-propagating through the Viterbi Transformer is equally simple. The partial derivatives of ${}_{\text{vit}}$ with respect to the penalties on the arcs of the constrained graph ${}_{\text{c}}$ are 1 for those arcs that appear in the constrained Viterbi path ${}_{\text{cvit}}$, and 0 for those that do not. Why is it legitimate to back-propagate through an essentially discrete function such as the Viterbi Transformer? The answer is that the Viterbi Transformer is nothing more than a collection of functions and adders put together. It was shown in Section IV that gradients can be back-propagated through functions without adverse effects. Back-propagation through the path selector transformer is similar to back-propagation through the Viterbi transformer. Arcs in ${}_{\text{int}}$ that appear in ${}_{\text{c}}$ have the same gradient as the corresponding arc in ${}_{\text{c}}$, i.e. 1 or 0, depending on whether the arc appear in ${}_{\text{cvit}}$. The other arcs, i.e. those that do not have an *alter ego* in ${}_{\text{c}}$ because they do not contain the right label have a gradient of 0. During the forward propagation through the recognition transformer, one instance of the recognizer for single character was created for each arc in the segmentation graph. The state of recognizer instances was stored. Since each arc penalty in ${}_{\text{int}}$ is produced by an individual output of a recognizer instance, we now have a gradient (1 or 0) for each output of each instance of the recognizer. Recognizer outputs that have a non zero gradient are part of the correct answer, and will therefore have their value pushed down. The gradients present on the recognizer outputs can be back-propagated through each recognizer instance. For each recognizer instance, we obtain a vector of partial derivatives of the loss function with respect to the recognizer instance parameters. All the recognizer instances share the same parameter vector, since they are merely clones of each other, therefore the full gradient of the loss function with respect to the recognizer's parameter vector is simply the sum of the gradient vectors produced by each recognizer instance. Viterbi training, though formulated differently, is often use in HMM-based speech recognition systems [28]. Similar algorithms have been applied to speech recognition systems that integrate neural networks with time alignment [71], [72], [76] or hybrid neural-network/HMM systems [29], [74], [75].

While it seems simple and satisfying, this training architecture has a flaw that can potentially be fatal. The problem was already mentioned in Section II-C. If the recognizer is a simple neural network with sigmoid output units, the minimum of the loss function is attained, not when the recognizer always gives the right answer, but when it ignores the input, and sets its output to a constant vector with small values for all the components. This is known as *the collapse problem.* The collapse only occurs if the recognizer outputs can simultaneously take their minimum value. If on the other hand the recognizer's output layer contains RBF units with fixed parameters, then there is no such trivial solution. This is due to the fact that a set of RBF with fixed distinct parameter vectors cannot simultaneously take their minimum value. In this case, the complete collapse described above does not occur. However, this does not totally prevent the occurrence of a milder collapse because the loss function still has a "flat spot" for a trivial solution with constant recognizer output. This flat spot is a saddle point, but it is attractive in almost all directions and is very difficult to get out of using gradient-based minimization procedures. If the parameters of the RBFs are allowed to adapt, then the collapse problems reappears because the RBF centers can all converge to a single vector, and the underlying neural network can learn to produce that vector, and ignore the input. A different kind of collapse occurs if the width of the RBFs are also allowed to adapt. The collapse only occurs if a trainable module such as a neural network feeds the RBFs. The collapse does not occur in HMM-based speech recognition systems because they are generative systems that produce normalized likelihoods for the input data (more on this later). Another way to avoid the collapse is to train the whole system with respect to a discriminative training criterion, such as maximizing the conditional probability of the correct interpretations (correct sequence of class labels) given the input image.

Another problem with Viterbi training is that the penalty of the answer cannot be used reliably as a measure of confidence because it does not take low-penalty (or high-scoring) competing answers into account.

B. Discriminative Viterbi Training

A modification of the training criterion can circumvent the collapse problem described above and at the same time produce more reliable confidence values. The idea is to not only minimize the cumulated penalty of the lowest penalty path with the correct interpretation, but also to somehow increase the penalty of competing and possibly incorrect paths that have a dangerously low penalty. This type of criterion is called *discriminative*, because it plays the good answers against the bad ones. Discriminative training procedures can be seen as attempting to build appropriate separating surfaces between classes rather than to model individual classes independently of each other. For exam-

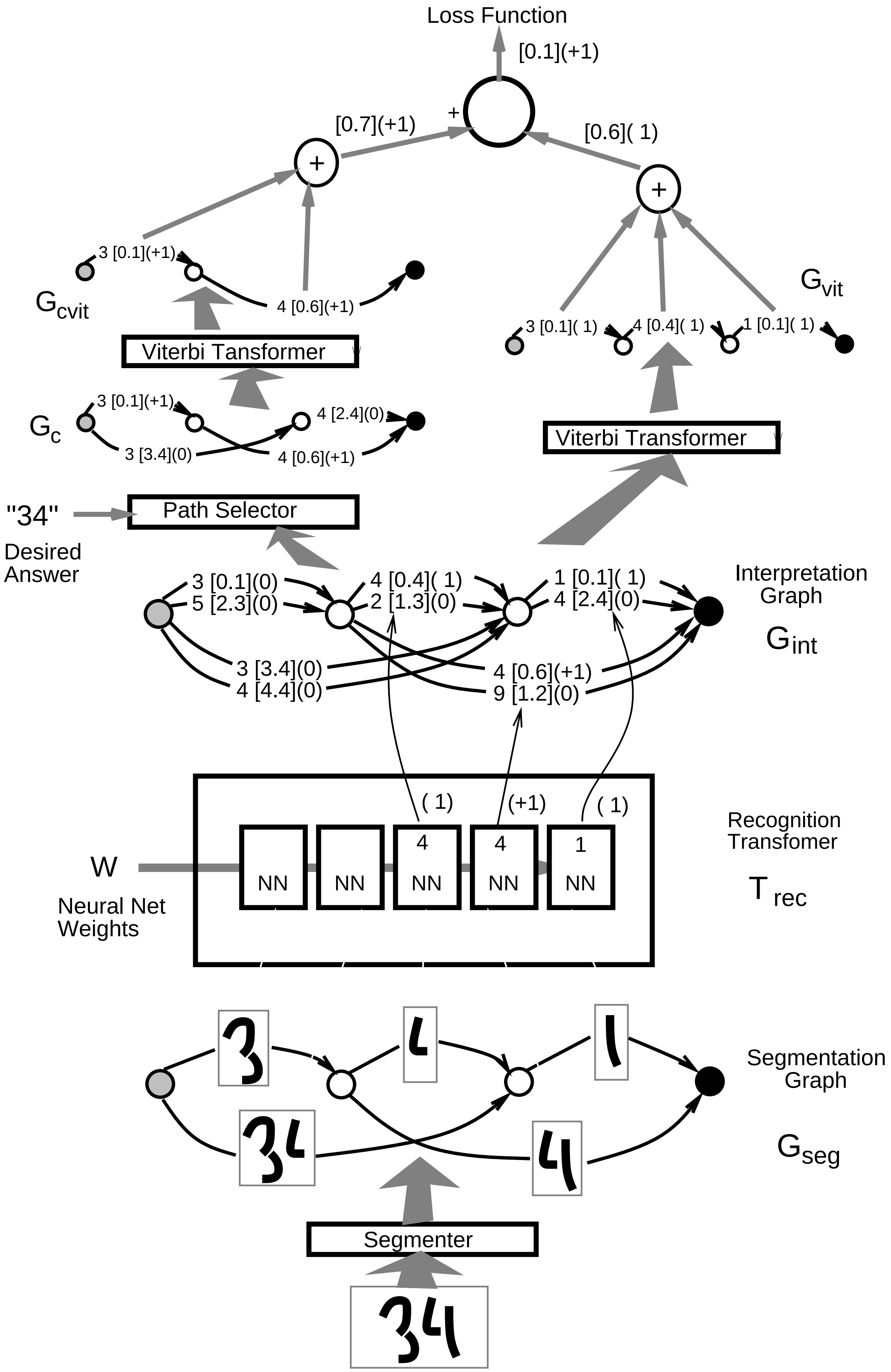

Fig. 20. Discriminative Viterbi Training GTN Architecture for a character string recognizer based on Heuristic Over-Segmentation. Quantities in square brackets are penalties computed during the forward propagation. Quantities in parentheses are partial derivatives computed during the backward propagation.

ple, modeling the conditional distribution of the classes given the input image is more discriminative (focus-sing more on the classification surface) than having a separate generative model of the input data associated to each class (which, with class priors, yields the whole joint distribution of classes and inputs). This is because the conditional approach does not need to assume a particular form for the distribution of the input data.

One example of discriminative criterion is the difference between the penalty of the Viterbi path in the constrained graph, and the penalty of the Viterbi path in the (unconstrained) interpretation graph, i.e. the difference between the penalty of the best correct path, and the penalty of the best path (correct or incorrect). The corresponding GTN training architecture is shown in figure 20. The left side of the diagram is identical to the GTN used for non-discriminative Viterbi training. This loss function reduces the risk of collapse because it forces the recognizer to *increases* the penalty of wrongly recognized objects. Discriminative training can also be seen as another example of *error correction procedure*, which tends to minimize the difference between the desired output computed in the left half of the GTN in figure 20 and the actual output computed in the right half of figure 20.

Let the discriminative Viterbi loss function be denoted $_{\rm dvit}$, and let us call $_{\rm cvit}$ the penalty of the Viterbi path in the constrained graph, and $_{\rm vit}$ the penalty of the Viterbi path in the unconstrained interpretation graph:

$$_{\rm dvit} = \; _{\rm cvit} \qquad _{\rm vit} \tag{12}$$

$_{\rm dvit}$ is always positive since the constrained graph is a subset of the paths in the interpretation graph, and the Viterbi algorithm selects the path with the lowest total penalty. In the ideal case, the two paths $_{\rm cvit}$ and $_{\rm vit}$ coincide, and $_{\rm dvit}$ is zero.

Back-propagating gradients through the discriminative Viterbi GTN adds some "negative" training to the previously described non-discriminative training. Figure 20 shows how the gradients are back-propagated. The left half is identical to the non-discriminative Viterbi training GTN, therefore the back-propagation is identical. The gradients back-propagated through the right half of the GTN are multiplied by -1, since $_{\rm vit}$ contributes to the loss with a negative sign. Otherwise the process is similar to the left half. The gradients on arcs of $_{\rm int}$ get positive contributions from the left half and negative contributions from the right half. The two contributions must be added, since the penalties on $_{\rm int}$ arcs are sent to the two halves through a "Y" connection in the forward pass. Arcs in $_{\rm int}$ that appear neither in $_{\rm vit}$ nor in $_{\rm cvit}$ have a gradient of zero. They do not contribute to the cost. Arcs that appear in both $_{\rm vit}$ and $_{\rm cvit}$ also have zero gradient. The -1 contribution from the right half cancels the the +1 contribution from the left half. In other words, when an arc is rightfully part of the answer, there is no gradient. If an arc appears in $_{\rm cvit}$ but not in $_{\rm vit}$, the gradient is +1. The arc should have had a lower penalty to make it to $_{\rm vit}$. If an arc is in $_{\rm vit}$ but not in $_{\rm cvit}$, the gradient is -1. The arc had a low penalty, but should have had a higher penalty since it is not part of the desired answer.

Variations of this technique have been used for the speech recognition. Driancourt and Bottou [76] used a version of it where the loss function is saturated to a fixed value. This can be seen as a generalization of the Learning Vector Quantization 2 (LVQ-2) loss function [80]. Other variations of this method use not only the Viterbi path, but the K-best paths. The Discriminative Viterbi algorithm does not have the flaws of the non-discriminative version, but there are problems nonetheless. The main problem is that the criterion does not build a margin between the classes. The gradient is zero as soon as the penalty of the constrained Viterbi path is *equal* to that of the Viterbi path. It would be desirable to push up the penalties of the wrong paths when they are dangerously close to the good one. The following section presents a solution to this problem.

C. Forward Scoring, and Forward Training

While the penalty of the Viterbi path is perfectly appropriate for the purpose of recognition, it gives only a partial picture of the situation. Imagine the lowest penalty paths corresponding to several *different* segmentations produced the same answer (the same label sequence). Then it could be argued that the overall penalty for the interpretation should be smaller than the penalty obtained when only one path produced that interpretation, because multiple paths with identical label sequences are more evidence that the label sequence is correct. Several rules can be used compute the penalty associated to a graph that contains several parallel paths. We use a combination rule borrowed from a probabilistic interpretation of the penalties as negative log posteriors. In a probabilistic framework, the posterior probability for the interpretation should be the sum of the posteriors for all the paths that produce that interpretation. Translated in terms of penalties, the penalty of an interpretation should be the negative logarithm of the sum of the negative exponentials of the penalties of the individual paths. The overall penalty will be smaller than all the penalties of the individual paths.

Given an interpretation, there is a well known method, called the *forward algorithm* for computing the above quantity efficiently [28]. The penalty computed with this procedure for a particular interpretation is called the *forward penalty.* Consider again the concept of constrained graph, the subgraph of the interpretation graph which contains only the paths that are consistent with a particular label sequence. There is one constrained graph for each possible label sequence (some may be empty graphs, which have infinite penalties). Given an interpretation, running the forward algorithm on the corresponding constrained graph gives the forward penalty for that interpretation. The forward algorithm proceeds in a way very similar to the Viterbi algorithm, except that the operation used at each node to combine the incoming cumulated penalties, instead of being the function is the so-called operation, which can be seen as a "soft" version of the function:

$$= \mathrm{logadd}_{\in} \left(\quad + \quad \right) \qquad (13)$$

where $_{\mathrm{start}} = 0$, is the set of upstream arcs of node , is the penalty on arc , and

$$\mathrm{logadd}(\;_1 \;\; _2 \quad) = \quad \log(\quad ^{-} \;)_{=1} \qquad (14)$$

Note that because of numerical inaccuracies, it is better to factorize the largest $^{-}$ (corresponding to the smallest penalty) out of the logarithm.

An interesting analogy can be drawn if we consider that a graph on which we apply the forward algorithm is equivalent to a neural network on which we run a forward propagation, except that multiplications are replaced by additions, the additions are replaced by logadds, and there are no sigmoids.

One way to understand the forward algorithm is to think about multiplicative scores (e.g., probabilities) instead of additive penalties on the arcs: score = exp(penalty). In that case the Viterbi algorithm selects the path with the largest cumulative score (with scores multiplied along the path), whereas the forward score is the sum of the cumulative scores associated to each of the possible paths from the start to the end node. The forward penalty is always lower than the cumulated penalty on any of the paths, but if one path "dominates" (with a much lower penalty), its penalty is almost equal to the forward penalty. The forward algorithm gets its name from the forward pass of the well-known Baum-Welsh algorithm for training Hidden Markov Models [28]. Section VIII-E gives more details on the relation between this work and HMMs.

The advantage of the forward penalty with respect to the Viterbi penalty is that it takes into account all the different ways to produce an answer, and not just the one with the lowest penalty. This is important if there is some ambiguity in the segmentation, since the combined forward penalty of two paths $_1$ and $_2$ associated with the same label sequence may be less than the penalty of a path $_3$ associated with another label sequence, even though the penalty of $_3$ might be less than any one of $_1$ or $_2$.

The Forward training GTN is only a slight modification of the previously introduced Viterbi training GTN. It suffices to turn the Viterbi transformers in Figure 19 into *Forward Scorers* that take an interpretation graph as input an produce the forward penalty of that graph on output. Then the penalties of all the paths that contain the correct answer are lowered, instead of just that of the best one.

Back-propagating through the forward penalty computation (the forward transformer) is quite different from back-propagating through a Viterbi transformer. All the penalties of the input graph have an influence on the forward penalty, but penalties that belong to low-penalty paths have a stronger influence. Computing derivatives with respect to the forward penalties computed at each node of a graph is done by back-propagation through the graph

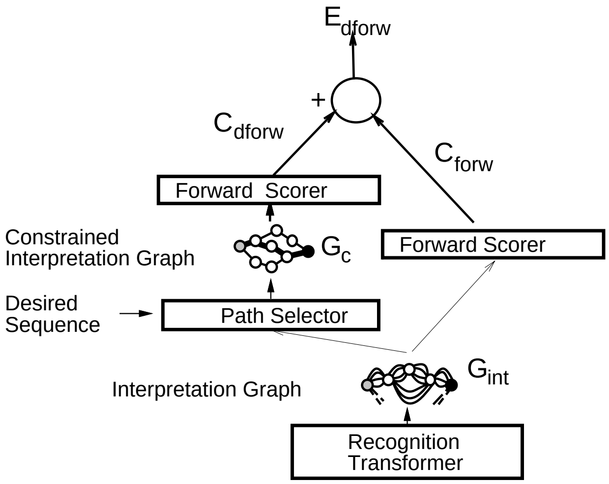

Fig. 21. Discriminative Forward Training GTN Architecture for a character string recognizer based on Heuristic Over-Segmentation.

$$_{\mathrm{c}}$$

$$= {}^{-} \sum_{\in} \qquad {}^{-} \qquad (15)$$

where = arc with source = is the set of downstream arcs from node . From the above derivatives, the derivatives with respect to the arc penalties are obtained:

$$= \qquad {}^{-\;-\;\;+} \qquad (16)$$

This can be seen as a "soft" version of the back-propagation through a Viterbi scorer and transformer. All the arcs in $_{\mathrm{c}}$ have an influence on the loss function. The arcs that belong to low penalty paths have a larger influence. Back-propagation through the path selector is the same as before. The derivative with respect to $_{\mathrm{int}}$ arcs that have an *alter ego* in $_{\mathrm{c}}$ are simply copied from the corresponding arc in $_{\mathrm{c}}$. The derivatives with respect to the other arcs are 0.

Several authors have applied the idea of back-propagating gradients through a forward scorer to train speech recognition systems, including Bridle and his -net model [73] and Haffner and his -TDNN model [81], but these authors recommended discriminative training as described in the next section.

D. Discriminative Forward Training

The information contained in the forward penalty can be used in another discriminative training criterion which we will call the *discriminative forward criterion.* This criterion corresponds to *maximization of the posterior probability of choosing the paths associated with the correct interpretation.* This posterior probability is defined as the exponential of the minus the constrained forward penalty, normalized by the exponential of minus the unconstrained forward penalty. Note that the forward penalty of the constrained graph is always larger or equal to the forward penalty of the unconstrained interpretation graph. Ideally, we would like the forward penalty of the constrained graph to be equal to the forward penalty of the complete interpretation graph. Equality between those two quantities is achieved when the combined penalties of the paths with the correct label sequence is negligibly small compared to the penalties of all the other paths, or that the posterior probability associated to the paths with the correct interpretation is almost 1, which is precisely what we want. The corresponding GTN training architecture is shown in figure 21.

Let the difference be denoted ${}_{\text{dforw}}$, and let us call ${}_{\text{cforw}}$ the forward penalty of the constrained graph, and ${}_{\text{forw}}$ the forward penalty of the complete interpretation graph:

$$ {}_{\text{dforw}} = {}_{\text{cforw}} \qquad {}_{\text{forw}} \tag{17} $$

${}_{\text{dforw}}$ is always positive since the constrained graph is a subset of the paths in the interpretation graph, and the forward penalty of a graph is always larger than the forward penalty of a subgraph of this graph. In the ideal case, the penalties of incorrect paths are infinitely large, therefore the two penalties coincide and ${}_{\text{dforw}}$ is zero. Readers familiar with the Boltzmann machine connectionist model might recognize the constrained and unconstrained graphs as analogous to the "clamped" (constrained by the observed values of the output variable) and "free" (unconstrained) phases of the Boltzmann machine algorithm [13].

Back-propagating derivatives through the discriminative Forward GTN distributes gradients more evenly than in the Viterbi case. Derivatives are back-propagated through the left half of the the GTN in Figure 21 down to the interpretation graph. Derivatives are negated and back-propagated through the right-half, and the result for each arc is added to the contribution from the left half. Each arc in ${}_{\text{int}}$ now has a derivative. Arcs that are part of a correct path have a positive derivative. This derivative is very large if an incorrect path has a lower penalty than all the correct paths. Similarly, the derivatives with respect to arcs that are part of a low-penalty incorrect path have a large negative derivative. On the other hand, if the penalty of a path associated with the correct interpretation is much smaller than all other paths, the loss function is very close to 0 and almost no gradient is back-propagated. The training therefore concentrates on examples of images which yield a classification error, and furthermore, it concentrates on the pieces of the image which cause that error. Discriminative forward training is an elegant and efficient way of solving the infamous *credit assignment problem* for learning machines that manipulate "dynamic" data structures such as graphs. More generally, the same idea can be used in all situations where a learning machine must choose between discrete alternative interpretations.

As previously, the derivatives on the interpretation graph penalties can then be back-propagated into the character recognizer instances. Back-propagation through the character recognizer gives derivatives on its parameters. All the gradient contributions for the different candidate segments are added up to obtain the total gradient associated to one pair (input image, correct label sequence), that is, one example in the training set. A step of stochastic gradient descent can then be applied to update the parameters.

E. Remarks on Discriminative Training

In the above discussion, the global training criterion was given a probabilistic interpretation, but the individual penalties on the arcs of the graphs were not. There are good reasons for that. For example, if some penalties are associated to the different class labels, they would (1) have to sum to 1 (class posteriors), or (2) integrate to 1 over the input domain (likelihoods).

Let us first discuss the first case (class posteriors normalization). This local normalization of penalties may eliminate information that is important for locally rejecting all the classes [82], e.g., when a piece of image does not correspond to a valid character class, because some of the segmentation candidates may be wrong. Although an explicit "garbage class" can be introduced in a probabilistic framework to address that question, some problems remain because it is difficult to characterize such a class probabilistically and to train a system in this way (it would require a density model of unseen or unlabeled samples).

The probabilistic interpretation of individual variables plays an important role in the Baum-Welsh algorithm in combination with the Expectation-Maximization procedure. Unfortunately, those methods cannot be applied to discriminative training criteria, and one is reduced to using gradient-based methods. Enforcing the normalization of the probabilistic quantities while performing gradient-based learning is complex, inefficient, time consuming, and creates ill-conditioning of the loss-function.

Following [82], we therefore prefer to postpone normalization as far as possible (in fact, until the final decision stage of the system). Without normalization, the quantities manipulated in the system do not have a direct probabilistic interpretation.

Let us now discuss the second case (using a generative model of the input). Generative models build the boundary indirectly, by first building an independent density model for each class, and then performing classification decisions on the basis of these models. This is not a discriminative approach in that it does not focus on the ultimate goal of learning, which in this case is to learn the classification decision surface. Theoretical arguments [6], [7] suggest that estimating input densities when the real goal is to obtain a discriminant function for classification is a suboptimal strategy. In theory, the problem of estimating densities in high-dimensional spaces is much more ill-posed than finding decision boundaries.

Even though the internal variables of the system do not have a direct probabilistic interpretation, the overall system can still be viewed as producing posterior probabilities for the classes. In fact, assuming that a particular label sequence is given as the "desired sequence" to the GTN in figure 21, the exponential of minus ${}_{\text{dforw}}$ can be interpreted as an estimate of the posterior probability of that label sequence given the input. The sum of those posteriors for all the possible label sequences is 1. Another approach would consists of directly minimizing an approximation of the number of misclassifications [83] [76]. We prefer to use the discriminative forward loss function because it causes

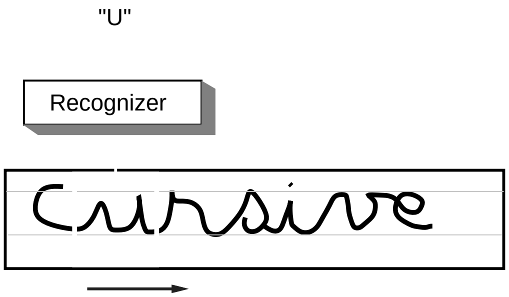

Fig. 22. Explicit segmentation can be avoided by sweeping a recognizer at every possible location in the input field.

less numerical problems during the optimization. We will see in Section X-C that this is a good way to obtain scores on which to base a rejection strategy. The important point being made here is that one is free to choose *any* parameterization deemed appropriate for a classification model. The fact that a particular parameterization uses internal variables with no clear probabilistic interpretation does not make the model any less legitimate than models that manipulate normalized quantities.

An important advantage of global and discriminative training is that learning focuses on the most important errors, and the system learns to integrate the ambiguities from the segmentation algorithm with the ambiguities of the character recognizer. In Section IX we present experimental results with an on-line handwriting recognition system that confirm the advantages of using global training versus separate training. Experiments in speech recognition with hybrids of neural networks and HMMs also showed marked improvements brought by global training [77], [29], [67], [84].

VII. MULTIPLE OBJECT RECOGNITION: SPACE DISPLACEMENT NEURAL NETWORK

.

There is a simple alternative to explicitly segmenting images of character strings using heuristics. The idea is to sweep a recognizer at all possible locations across a normalized image of the entire word or string as shown in Figure 22. With this technique, no segmentation heuristics are required since the system essentially examines *all* the possible segmentations of the input. However, there are problems with this approach. First, the method is in general quite expensive. The recognizer must be applied at every possible location on the input, or at least at a large enough subset of locations so that misalignments of characters in the field of view of the recognizers are small enough to have no effect on the error rate. Second, when the recognizer is centered on a character to be recognized, the neighbors of the center character will be present in the field of view of the recognizer, possibly touching the center character. Therefore the recognizer must be able to correctly recognize the character in the center of its input field, even if neighboring characters are very close to, or touching the central character. Third, a word or character string cannot be perfectly size normalized. Individual characters within a string may have widely varying sizes and baseline positions. Therefore the recognizer must be very robust to shifts and size variations.

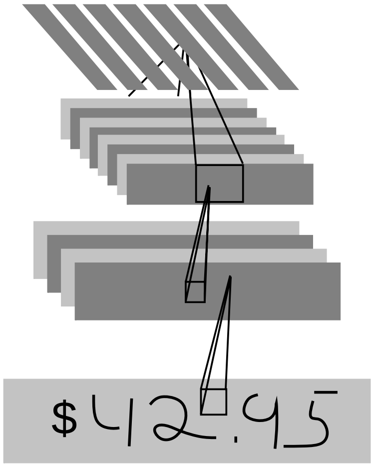

Fig. 23. A Space Displacement Neural Network is a convolutional network that has been replicated over a wide input field.

These three problems are elegantly circumvented if a convolutional network is replicated over the input field. First of all, as shown in section III, convolutional neural networks are very robust to shifts and scale variations of the input image, as well as to noise and extraneous marks in the input. These properties take care of the latter two problems mentioned in the previous paragraph. Second, convolutional networks provide a drastic saving in computational requirement when replicated over large input fields. A replicated convolutional network, also called a *Space Displacement Neural Network* or SDNN [27], is shown in Figure 23. While scanning a recognizer can be prohibitively expensive in general, convolutional networks can be scanned or replicated very efficiently over large, variable-size input fields. Consider one instance of a convolutional net and its *alter ego* at a nearby location. Because of the convolutional nature of the network, units in the two instances that look at identical locations on the input have identical outputs, therefore their states do not need to be computed twice. Only a thin "slice" of new states that are not shared by the two network instances needs to be recomputed. When all the slices are put together, the result is simply a larger convolutional network whose structure is identical to the original network, except that the feature maps are larger in the horizontal dimension. In other words, replicating a convolutional network can be done simply by increasing the size of the fields over which the convolutions are performed, and by replicating the output layer accordingly. The output layer effectively becomes a convolutional layer. An output whose receptive field is centered on an elementary object will produce the class of this object, while an in-between output may indicate no character or contain rubbish. The outputs can be interpreted as evidences for the presence of objects at all possible positions in the input field.

The SDNN architecture seems particularly attractive for recognizing cursive handwriting where no reliable segmentation heuristic exists. Although the idea of SDNN is quite old, and very attractive by its simplicity, it has not generated wide interest until recently because, as stated above, it puts enormous demands on the recognizer [26], [27]. In speech recognition, where the recognizer is at least one order of magnitude smaller, replicated convolutional networks are easier to implement, for instance in Haffner's Multi-State TDNN model [78], [85].

A. Interpreting the Output of an SDNN with a GTN

The output of an SDNN is a sequence of vectors which encode the likelihoods, penalties, or scores of finding character of a particular class label at the corresponding location in the input. A post-processor is required to pull out the best possible label sequence from this vector sequence. An example of SDNN output is shown in Figure 25. Very often, individual characters are spotted by several neighboring instances of the recognizer, a consequence of the robustness of the recognizer to horizontal translations. Also quite often, characters are erroneously detected by recognizer instances that see only a piece of a character. For example a recognizer instance that only sees the right third of a "4" might output the label 1. How can we eliminate those extraneous characters from the output sequence and pull-out the best interpretation? This can be done using a new type of Graph Transformer with two input graphs as shown in Figure 24. The sequence of vectors produced by the SDNN is first coded into a linear graph with multiple arcs between pairs of successive nodes. Each arc between a particular pair of nodes contains the label of one of the possible categories, together with the penalty produced by the SDNN for that class label at that location. This graph is called the *SDNN Output Graph*. The second input graph to the transformer is a *grammar transducer*, more specifically a *finite-state transducer* [86], that encodes the relationship between input strings of class labels and corresponding output strings of recognized characters.The transducer is a weighted finite state machine (a graph) where each arc contains a pair of labels and possibly a penalty. Like a finite-state machine, a transducer is in a state and follows an arc to a new state when an observed input symbol matches the first symbol in the symbol pair attached to the arc. At this point the transducer emits the second symbol in the pair together with a penalty that combines the penalty of the input symbol and the penalty of the arc. A transducer therefore transforms a weighted symbol sequence into another weighted symbol sequence. The graph transformer shown in figure 24 performs a *composition* between the recognition graph and the grammar transducer. This operation takes every possible sequence corresponding to every possible path in the recognition graph and matches them with the paths in the grammar transducer. The composition produces the interpretation graph, which contains a path for each corresponding output label sequence. This composition operation may seem combinatorially intractable, but it turns out there exists an efficient algorithm for it described in more details in Section VIII.

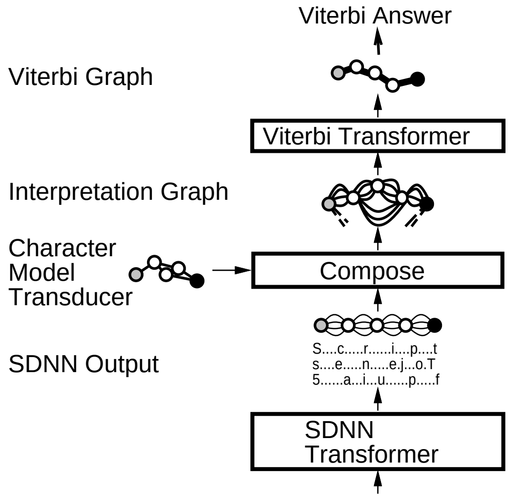

Fig. 24. A Graph Transformer pulls out the best interpretation from the output of the SDNN.

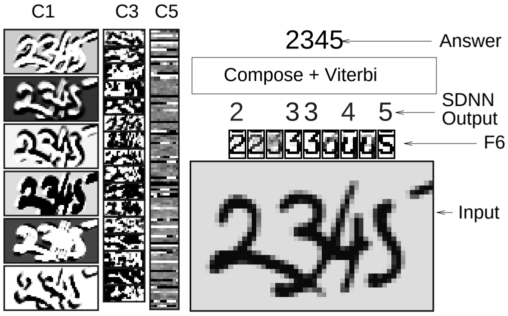

Fig. 25. An example of multiple character recognition with SDNN. With SDNN, no explicit segmentation is performed.

B. Experiments with SDNN

In a series of experiments, LeNet-5 was trained with the goal of being replicated so as to recognize multiple characters without segmentations. The data was generated from the previously described Modified NIST set as follows. Training images were composed of a central character, flanked by two side characters picked at random in the training set. The separation between the bounding boxes of the characters were chosen at random between -1 and 4 pixels. In other instances, no central character was present, in which case the desired output of the network was the blank space class. In addition, training images were degraded with 10% salt and pepper noise (random pixel inversions).

Figures 25 and 26 show a few examples of successful recognitions of multiple characters by the LeNet-5 SDNN. Standard techniques based on Heuristic Over-Segmentation would fail miserably on many of those examples. As can be seen on these examples, the network exhibits striking invariance and noise resistance properties. While some authors have argued that invariance requires more sophisticated models than feed-forward neural networks [87], LeNet-5 exhibits these properties to a large extent.

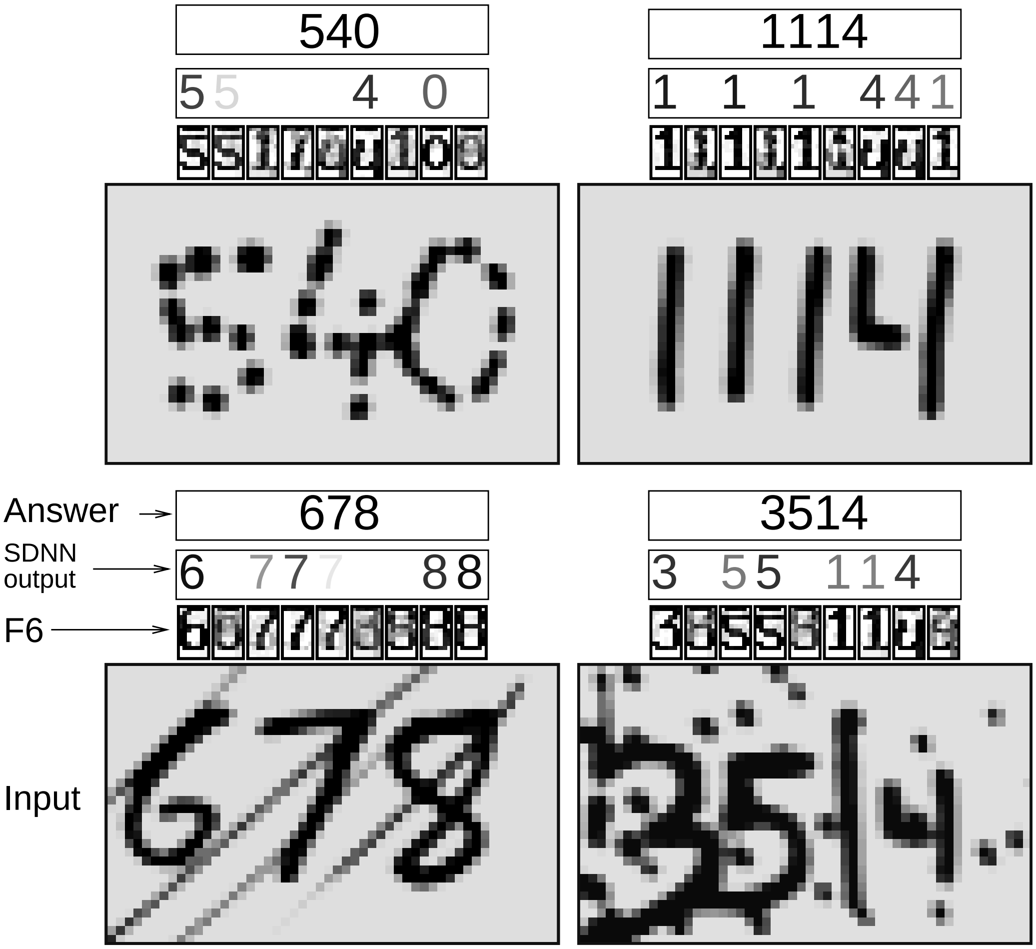

Fig. 26. An SDNN applied to a noisy image of digit string. The digits shown in the SDNN output represent the winning class labels, with a lighter grey level for high-penalty answers.

Similarly, it has been suggested that accurate recognition of multiple overlapping objects require explicit mechanisms that would solve the so-called *feature binding* problem [87]. As can be seen on Figures 25 and 26, the network is able to tell the characters apart, even when they are closely intertwined, a task that would be impossible to achieve with the more classical Heuristic Over-Segmentation technique. The SDNN is also able to correctly group disconnected pieces of ink that form characters. Good examples of that are shown in the upper half of figure 26. In the top left example, the 4 and the 0 are more connected to each other than they are connected with themselves, yet the system correctly identifies the 4 and the 0 as separate objects. The top right example is interesting for several reasons. First the system correctly identifies the three individual ones. Second, the left half and right half of disconnected 4 are correctly grouped, even though no geometrical information could decide to associate the left half to the vertical bar on its left or on its right. The right half of the 4 does cause the appearance of an erroneous 1 on the SDNN output, but this one is removed by the character model transducer which prevents characters from appearing on contiguous outputs.

Another important advantage of SDNN is the ease with which they can be implemented on parallel hardware. Specialized analog/digital chips have been designed and used in character recognition, and in image preprocessing applications [88]. However the rapid progress of conventional processor technology with reduced-precision vector arithmetic instructions (such as Intel's MMX) make the success of specialized hardware hypothetical at best.

Short video clips of the LeNet-5 SDNN can be viewed at .

C. Global Training of SDNN

In the above experiments, the string image were artificially generated from individual character. The advantage is that we know in advance the location and the label of the important character. With real training data, the correct sequence of labels for a string is generally available, but the precise locations of each corresponding character in the input image are unknown.

In the experiments described in the previous section, the best interpretation was extracted from the SDNN output using a very simple graph transformer. Global training of an SDNN can be performed by back-propagating gradients through such graph transformers arranged in architectures similar to the ones described in section VI.