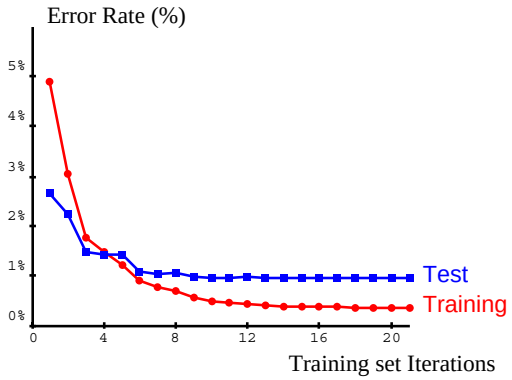

Fig. 5. Training and test error of LeNet-5 as a function of the number of passes through the 60,000 pattern training set (without distortions). The average training error is measured on-the-fly as training proceeds. This explains why the training error appears to be larger than the test error. Convergence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

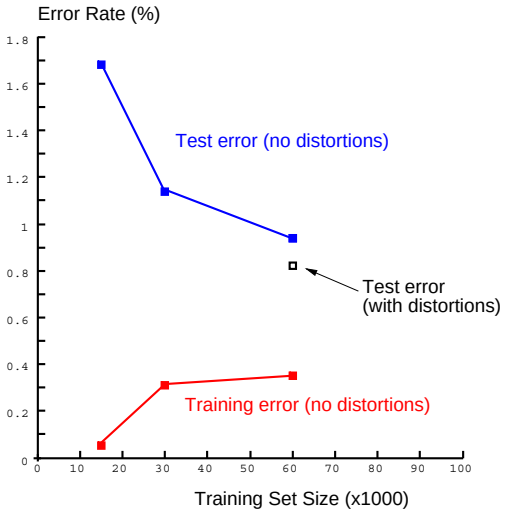

Fig. 6. Training and test errors of LeNet-5 achieved using training sets of various sizes. This graph suggests that a larger training set could improve the performance of LeNet-5. The hollow square show the test error when more training patterns are artificially generated using random distortions. The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

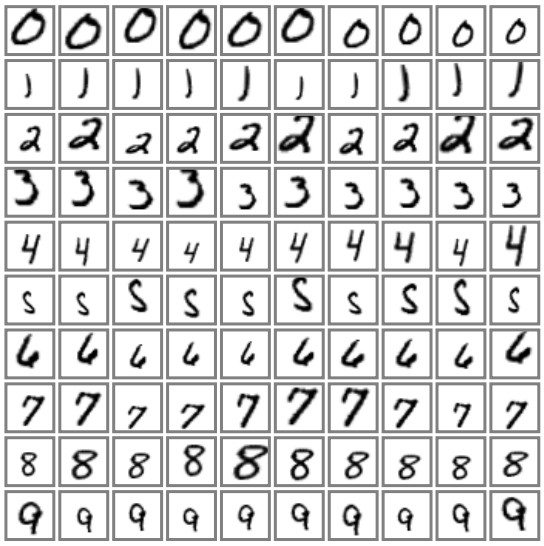

Fig. 7. Examples of distortions of ten training patterns.

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-

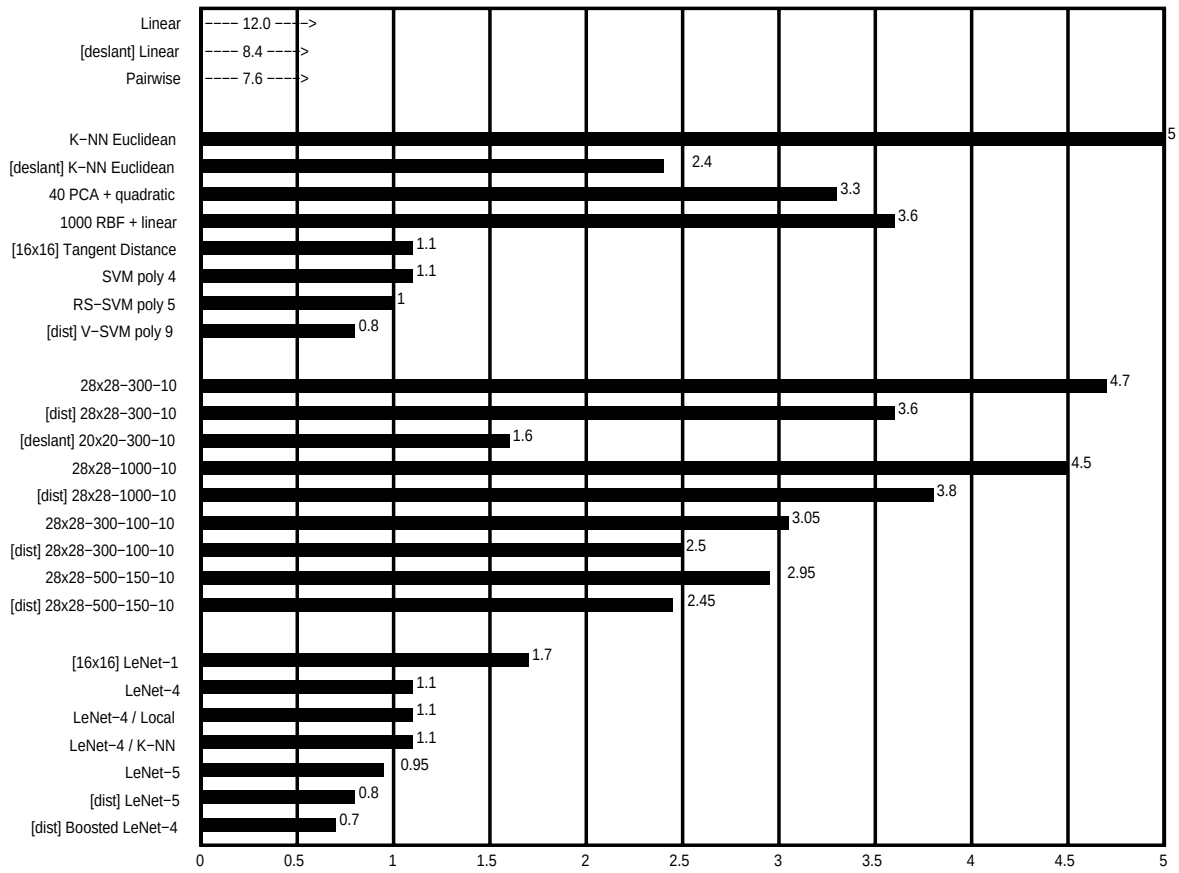

Fig. 9. Error rate on the test set (%) for various classification methods. [deslant] indicates that the classifier was trained and tested on the deslanted version of the database. [dist] indicates that the training set was augmented with artificially distorted examples. [16x16] indicates that the system 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 i/j is trained to produce +1 on patterns of class i, -1 on patterns of class j, and is not trained on other patterns. The final score for class i is the sum of the outputs all the units labeled i/x minus the sum of the output of all the units labeled y/i, for all x and y. 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 $k = 3$. 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 k 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 k-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)

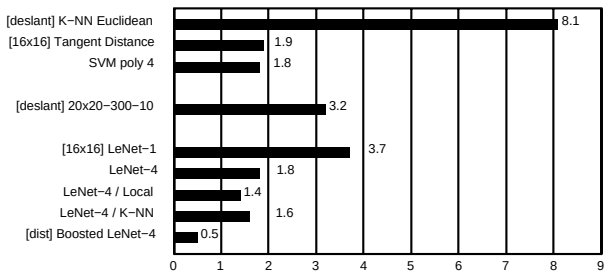

Fig. 10. Rejection Performance: percentage of test patterns that must be rejected to achieve 0.5% error for some of the systems.

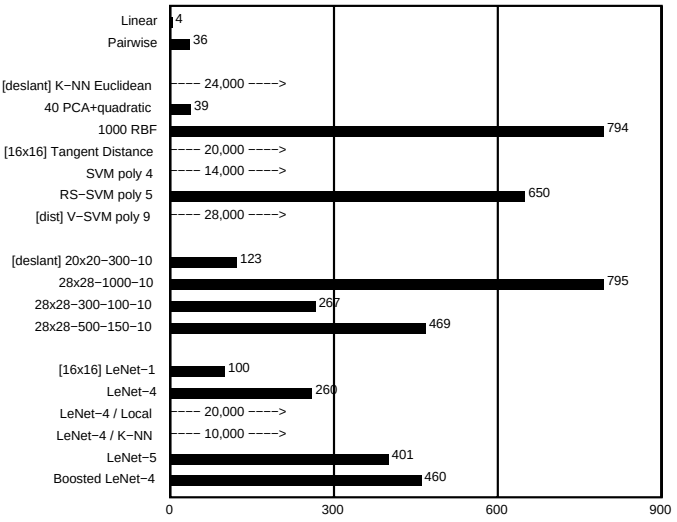

Fig. 11. Number of multiply-accumulate operations for the recognition of a single character starting with a size-normalized image.

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.

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

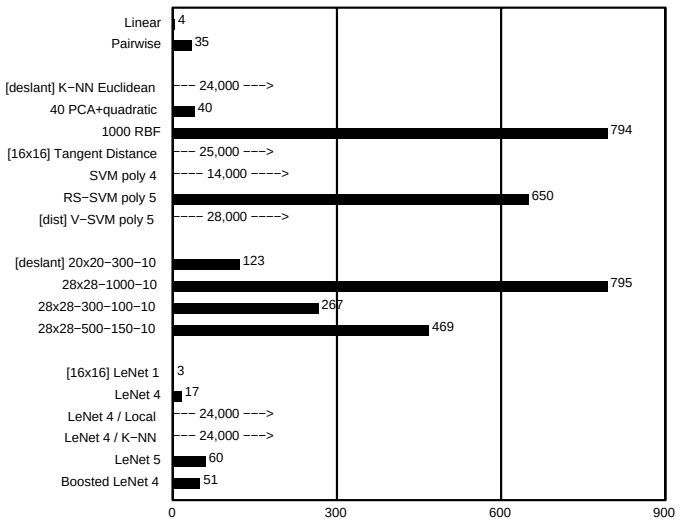

Fig. 12. Memory requirements, measured in number of variables, for each of the methods. Most of the methods only require one byte per variable for adequate performance.

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 `http://www.research.att.com/~yann/ocr`.

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.

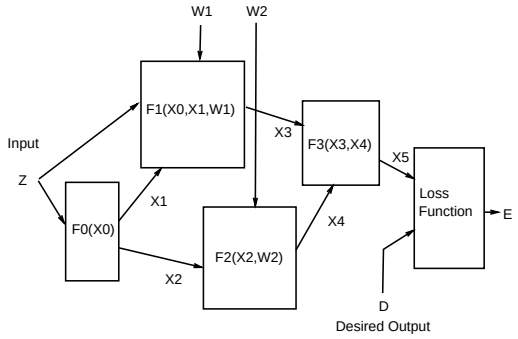

Fig. 14. A trainable system composed of heterogeneous modules.

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 (`W1,W2` in the figure), external inputs and outputs (`Z,D,E`), and intermediate state variables(`X1,X2,X3,X4,X5`).

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 `fprop` 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 `fprop` on module 3 with the arguments `X3,X4,X5`. 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 `fprop` method for the class simply calls the `fprop` 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 `bprop`, for each module class can be defined for that purpose. The `bprop` method of a module takes the same ar-

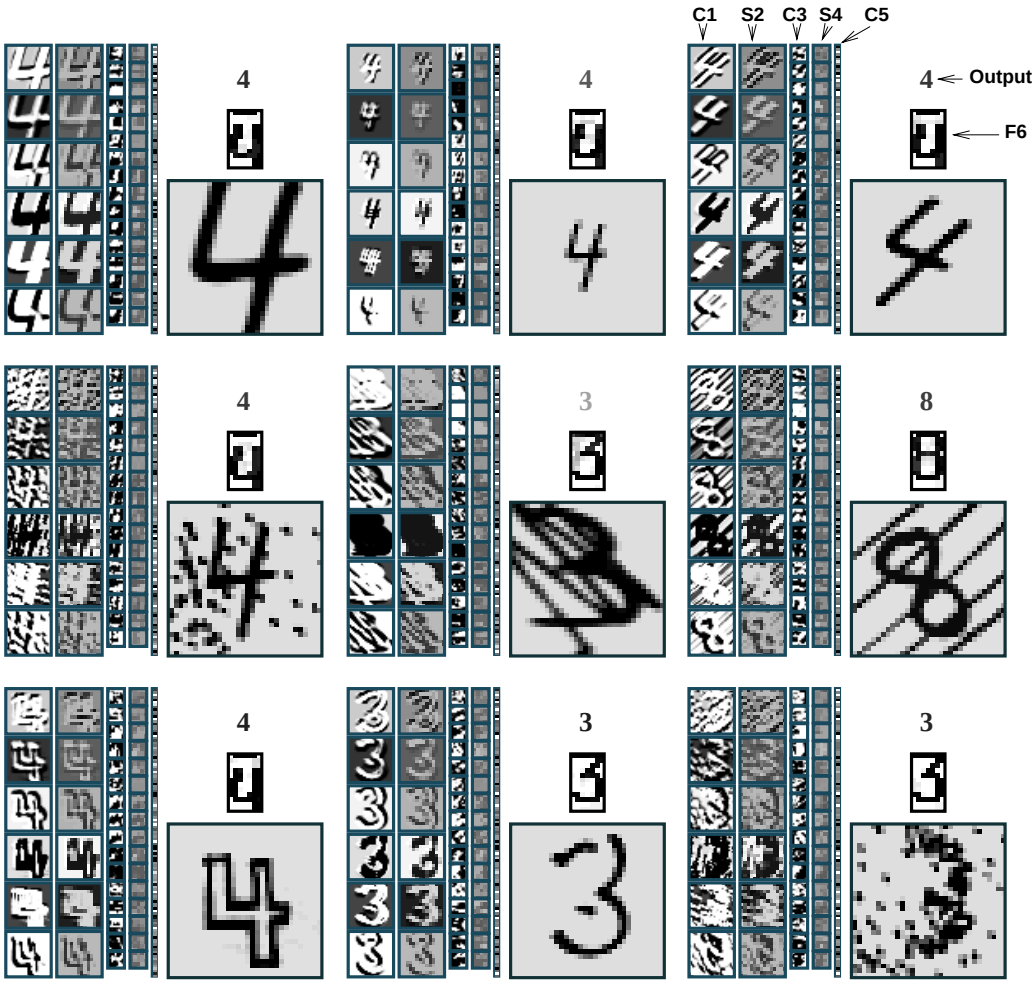

Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).

guments as the `fprop` method. All the derivatives in the system can be computed by calling the `bprop` 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 E 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 multimodular 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 `bprop` methods. In general, the `bprop` method of a function F is a multiplication by the Jacobian of F. Here are a few commonly used examples. The `bprop` method of a fanout (a "Y" connection) is a sum, and vice versa. The `bprop` method of a multiplication by a coefficient is a multiplication by the same coefficient. The `bprop` method of a multiplication by a matrix is a multiplication by the transpose of that matrix. The `bprop` method of an addition with a constant is the identity.

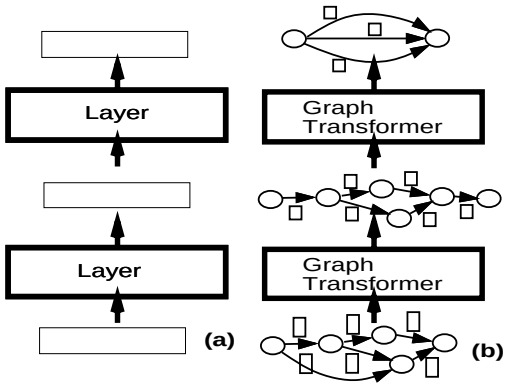

Fig. 15. Traditional neural networks, and multi-module systems communicate fixed-size vectors between layer. Multi-Layer Graph Transformer Networks are composed of trainable modules that operate on and produce graphs whose arcs carry numerical information.

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 `min` 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 `bbprop` 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.

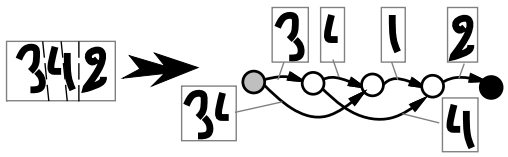

Fig. 16. Building a segmentation graph with Heuristic Over-Segmentation.

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* T_{rec}, and the *Viterbi transformer* T_{vit}. The goal of the recognition transformer is to generate a graph, called the *interpretation graph* or *recognition graph* G_{int}, that contains all the possible interpretations for all the possible segmentations of the input. Each path in G_{int} 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 T_{rec} takes the segmentation graph G_{seg} as input, and applies the recognizer for single characters to the images associated with each of the arcs

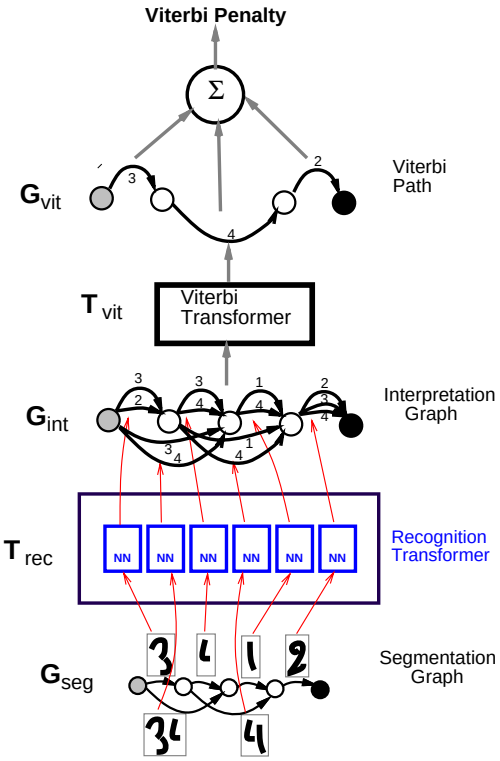

Fig. 17. Recognizing a character string with a GTN. For readability, only the arcs with low penalties are shown.

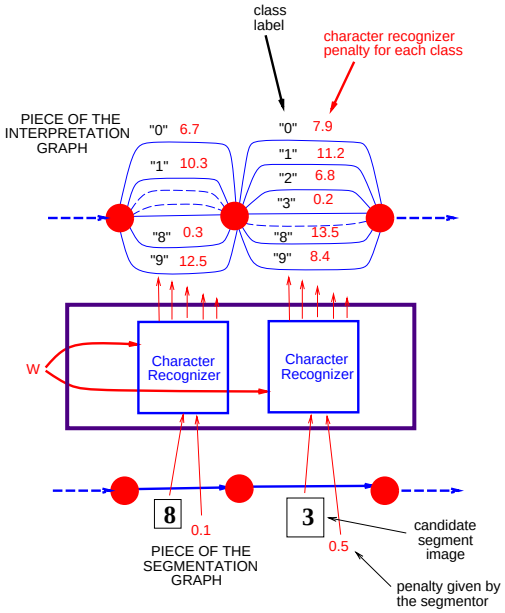

Fig. 18. The recognition transformer refines each arc of the segmentation arc into a set of arcs in the interpretation graph, one per character class, with attached penalties and labels.

in the segmentation graph. The interpretation graph G_{int} 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 G_{seg}. 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 G_{vit} 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 G_{vit} 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 c_i be the penalty associated to arc i, with source node s_i, and destination node d_i (note that there can be multiple arcs between two nodes). In the interpretation graph, arcs also have a label l_i. The Viterbi algorithm proceeds as follows. Each node n is associated with a cumulated Viterbi penalty v_n. 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 $v_{\text{start}} = 0$. The other nodes cumulated penalties v_n are computed recursively from the v values of their parent nodes, through the upstream arcs $U_n = \{\text{arc } i \text{ with destination } d_i = n\}$:

$$v_n = \min_{i \in U_n} (c_i + v_{s_i}). \tag{10}$$

Furthermore, the value of i for each node n which minimizes the right hand side is noted m_n, the minimizing entering arc. When the end node is reached we obtain in v_{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 $n_1 \ldots n_T$ and arcs $i_1 \ldots i_{T-1}$, we trace back these nodes and arcs as follows, starting with $n_T =$ the end node, and recursively using the minimizing entering arc: $i_t = m_{n_{t+1}}$, and $n_t = s_{i_t}$ until the start node is reached. The label sequence can then be read off the arcs of the Viterbi path.