Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis

Patrice Y. Simard, Dave Steinkraus, John C. Platt
Microsoft Research, One Microsoft Way, Redmond WA 98052
{patrice,v-davste,jplatt}@microsoft.com

Abstract

Neural networks are a powerful technology for classification of visual inputs arising from documents. However, there is a confusing plethora of different neural network methods that are used in the literature and in industry. This paper describes a set of concrete best practices that document analysis researchers can use to get good results with neural networks. The most important practice is getting a training set as large as possible: we expand the training set by adding a new form of distorted data. The next most important practice is that convolutional neural networks are better suited for visual document tasks than fully connected networks. We propose that a simple "do-it-yourself" implementation of convolution with a flexible architecture is suitable for many visual document problems. This simple convolutional neural network does not require complex methods, such as momentum, weight decay, structure-dependent learning rates, averaging layers, tangent prop, or even finely-tuning the architecture. The end result is a very simple yet general architecture which can yield state-of-the-art performance for document analysis. We illustrate our claims on the MNIST set of English digit images.

1. Introduction

After being extremely popular in the early 1990s, neural networks have fallen out of favor in research in the last 5 years. In 2000, it was even pointed out by the organizers of the Neural Information Processing System (NIPS) conference that the term "neural networks" in the submission title was negatively correlated with acceptance. In contrast, positive correlations were made with support vector machines (SVMs), Bayesian networks, and variational methods.

In this paper, we show that neural networks achieve the best performance on a handwriting recognition task (MNIST). MNIST [7] is a benchmark dataset of images of segmented handwritten digits, each with 28x28 pixels. There are 60,000 training examples and 10,000 testing examples.

Our best performance on MNIST with neural networks is in agreement with other researchers, who have found that neural networks continue to yield state-of-the-art performance on visual document analysis tasks [1][2].

The optimal performance on MNIST was achieved using two essential practices. First, we created a new, general set of elastic distortions that vastly expanded the size of the training set. Second, we used convolutional neural networks. The elastic distortions are described in detail in Section 2. Sections 3 and 4 then describe a generic convolutional neural network architecture that is simple to implement.

We believe that these two practices are applicable beyond MNIST, to general visual tasks in document analysis. Applications range from FAX recognition, to analysis of scanned documents and cursive recognition in the upcoming Tablet PC.

2. Expanding Data Sets through Elastic Distortions

Synthesizing plausible transformations of data is simple, but the "inverse" problem – transformation invariance – can be arbitrarily complicated. Fortunately, learning algorithms are very good at learning inverse problems. Given a classification task, one may apply transformations to generate additional data and let the learning algorithm infer the transformation invariance. This invariance is embedded in the parameters, so it is in some sense free, since the computation at recognition time is unchanged. If the data is scarce and if the distribution to be learned has transformation-invariance properties, generating additional data using transformations may even improve performance [6]. In the case of handwriting recognition, we postulate that the distribution has some invariance with respect to not only affine transformations, but also elastic deformations corresponding to uncontrolled oscillations of the hand muscles, dampened by inertia.

Simple distortions such as translations, rotations, and skewing can be generated by applying affine displacement fields to images. This is done by computing for every pixel a new target location with respect to the original location. The new target location, at position (x,y) is given with respect to the previous position. For instance if ?x(x,y)=1, and ?y(x,y)=0, this means that the new location of every pixel is shifted by 1 to the right. If

the displacement field was: ?x(x,y)= αx, and ?y(x,y)= αy, the image would be scaled by α, from the origin location (x,y)=(0,0). Since α could be a non integer value, interpolation is necessary.

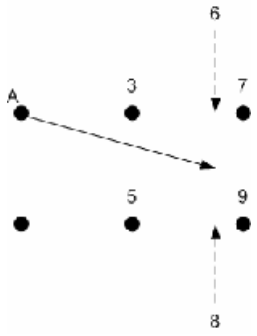

Figure 1. How to compute new grey level for A, at location (0,0) given a displacement ?x(0,0) = 1.75 and ?y(0,0) = -0.5. Bilinear interpolation yields 7.0.

Figure 1 illustrates how to apply a displacement field to compute new values for each pixel. In this example, the location of A is assumed to be (0,0) and the numbers 3, 7, 5, 9 are the grey levels of the image to be transformed, at the locations (1,0), (2,0), (1,-1) and (2,-1) respectively. The displacements for A are given by ?x(0,0) = 1.75 and ?y(0,0) = -0.5 as illustrated in the figure the arrow. The new grey value for A in the new (warped) image is computed by evaluating the grey level at location (1.75,-0.5) from the original image. A simple algorithm for evaluating the grey level is "bilinear interpolation" of the pixel values of the original image. Although other interpolation schemes can be used (e.g., bicubic and spline interpolation), the bilinear interpolation is one of the simplest and works well for generating additional warped characters image at the chosen resolution (29x29). Interpolating the value horizontally, followed by interpolating the value vertically, accomplishes the evaluation. To compute the horizontal interpolations, we first compute the location where the arrow ends with respect to the square in which it ends. In this case, the coordinates in the square are (0.75, 0.5), assuming the origin of that square is bottom-left (where the value 5 is located). In this example, the new values are: $3 + 0.75 \times (7-3) = 6$; and $5 + 0.75 \times (9-5) = 8$. The vertical interpolation between these values yields $8 + 0.5 \times (6-8) = 7$, which is the new grey level value for pixel A. A similar computation is done for all pixels. If a displacement ends up outside the image, a background value (e.g., 0) is assumed for all pixel locations outside the given image.

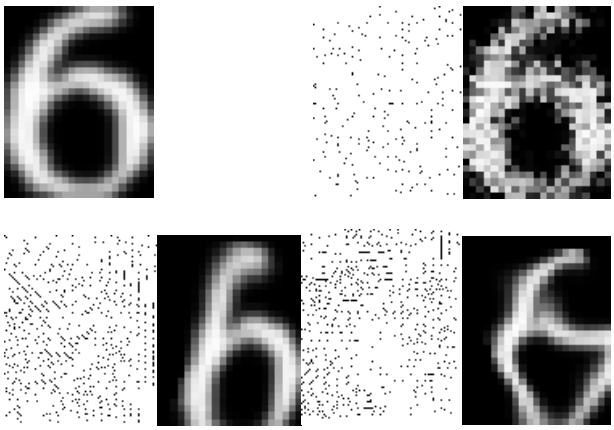

Figure 2. Top left: Original image. Right and bottom: Pairs of displacement fields with various smoothing, and resulting images when displacement fields are applied to the original image.

Affine distortions greatly improved our results on the MNIST database. However, our best results were obtained when we considered elastic deformations. The image deformations were created by first generating random displacement fields, that is ? x(x,y) = rand(-1,+1) and ?x(x,y)=rand(-1,+1), where rand(-1,+1) is a random number between -1 and +1, generated with a uniform distribution. The fields ?x and ?y are then convolved with a Gaussian of standard deviation σ (in pixels). If σ is large, the resulting values are very small because the random values average 0. If we normalize the displacement field (to a norm of 1), the field is then close to constant, with a random direction. If σ is small, the field looks like a completely random field after normalization (as depicted in Figure 2, top right). For intermediate σ values, the displacement fields look like elastic deformation, where σ is the elasticity coefficient. The displacement fields are then multiplied by a scaling factor α that controls the intensity of the deformation.

Figure 2 shows example of a pure random field (α =0.01), a smoothed random field corresponding to the properties of the hand (α =8), and a smoothed random field corresponding to too much variability (α =4). We use α=8 in our experiments. If α is larger than 8, the displacements become close to affine, and if α is very large, the displacements become translations.

3. Neural Networks Architectures for Visual Tasks

We considered two types of architectures neural network architectures for the MNIST data set. The simplest architecture, which is a universal classifier, is a fully connected network with two layers [4]. A more complicated architecture is a convolutional neural

network, which has been found to be well-suited for visual document analysis tasks [3]. The implementation of standard neural networks can be found in textbooks, such as [5]. Section 4 describes a new, simple implementation of convolutional neural networks.

To test our neural networks, we tried to keep the algorithm as simple as possible, for maximum reproducibility. We only tried two different error functions: cross-entropy (CE) and mean squared error (MSE) (see [5, chapter 6] for more details). We avoided using momentum, weight decay, structure-dependent learning rates, extra padding around the inputs, and averaging instead of subsampling. (We were motivated to avoid these complications by trying them on various architecture/distortions combinations and on a train/validation split of the data and finding that they did not help.)

Our initial weights were set to small random values (standard deviation = 0.05). We used a learning rate that started at 0.005 and is multiplied by 0.3 every 100 epochs.

3.1. Overall architecture for MNIST

As described in Section 5, we found that the convolutional neural network performs the best on MNIST. We believe this to be a general result for visual tasks, because spatial topology is well captured by convolutional neural networks [3], while standard neural networks ignore all topological properties of the input. That is, if a standard neural network is retrained and retested on a data set where all input pixels undergo a fixed permutation, the results would be identical.

The overall architecture of the convolutional neural network we used for MNIST digit recognition is depicted in Figure 3.

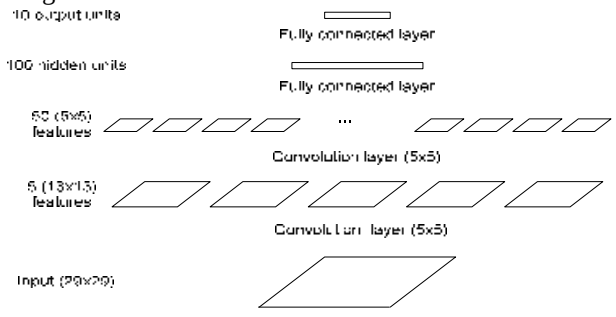

Figure 3. Convolution architecture for handwriting recognition

The general strategy of a convolutional network is to extract simple features at a higher resolution, and then convert them into more complex features at a coarser resolution. The simplest was to generate coarser resolution is to sub-sample a layer by a factor of 2. This, in turn, is a clue to the convolutions kernel's size. The width of the kernel is chosen be centered on a unit (odd size), to have sufficient overlap to not lose information (3 would be too small with only one unit overlap), but yet to not have redundant computation (7 would be too large, with 5 units or over 70% overlap). A convolution kernel of size 5 is shown in Figure 4. The empty circle units correspond to the subsampling and do not need to be computed. Padding the input (making it larger so that there are feature units centered on the border) did not improve performance significantly. With no padding, a subsampling of 2, and a kernel size of 5, each convolution layer reduces the feature size from n to (n-3)/2. Since the initial MNIST input size 28x28, the nearest value which generates an integer size after 2 layers of convolution is 29x29. After 2 layers of convolution, the feature size of 5x5 is too small for a third layer of convolution. The first feature layer extracts very simple features, which after training look like edge, ink, or intersection detectors. We found that using fewer than 5 different features decreased performance, while using more than 5 did not improve it. Similarly, on the second layer, we found that fewer than 50 features (we tried 25) decreased performance while more (we tried 100) did not improve it. These numbers are not critical as long as there are enough features to carry the information to the classification layers (since the kernels are 5x5, we chose to keep the numbers of features multiples of 5).

The first two layers of this neural network can be viewed as a trainable feature extractor. We now add a trainable classifier to the feature extractor, in the form of 2 fully connected layers (a universal classifier). The number of hidden units is variable, and it is by varying this number that we control the capacity, and the generalization, of the overall classifier. For MNIST (10 classes), the optimal capacity was reached with 100 hidden units. For Japanese 1 and 2 stroke characters (about 400 classes), the optimal capacity was reached with about 200 hidden units, with every other parameter being identical.

4. Making Convolutional Neural Networks Simple

Convolutional neural networks have been proposed for visual tasks for many years [3], yet have not been popular in the engineering community. We believe that is due to the complexity of implementing the convolutional neural networks. This paper presents new methods for implementing such networks that are much easier that previous techniques and allow easy debugging.

4.1. Simple Loops for Convolution

Fully connected neural networks often use the following rules to implement the forward and backward propagation:

$$x_j^{L+1} = \sum_i w_{j,i}^{L+1} x_i^L \quad (1.1)$$

$$g_i^L = \sum_j w_{j,i}^{L+1} g_j^{L+1} \quad (1.2)$$

where x_i^L and g_i^L are respectively the activation and the gradient of unit i at layer L, and $w_{j,i}^{L+1}$ is the weight connecting unit i at layer L to unit j at layer $L+1$. This can be viewed as the activation units of the higher layer "pulling" the activations of all the units connected to them. Similarly, the units of the lower layer are pulling the gradients of all the units connected to them. The pulling strategy, however, is complex and painful to implement when computing the gradients of a convolutional network. The reason is that in a convolution layer, the number of connections leaving each unit is not constant because of border effects.

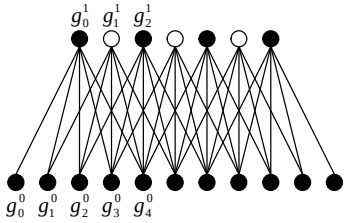

Figure 3. Convolutional neural network (1D)

This is easy to see on Figure 1, where all the units labeled g_i^0 have a variable number of outgoing connections. In contrast, all the units on the upper layer have a fixed number of incoming connections. To simplify computation, instead of pulling the gradient from the lower layer, we can "push" the gradient from the upper layer. The resulting equation is:

$$g_{j+i}^L += w_i^{L+1} g_j^{L+1} \quad (1.3)$$

For each unit *j* in the upper layer, we update a fixed number of (incoming) units *i* from the lower layer (in the figure *i* is between 0 and 4). Because in convolution the weights are shared, *w* does not depend on *j*. Note that pushing is slower than pulling because the gradients are accumulated in memory, as opposed to in pulling, where gradient are accumulated in a register. Depending on the architecture, this can sometimes be as much as 50% slower (which amounts to less than 20% decrease in overall performance). For large convolutions, however, pushing the gradient may be faster, and can be used to take advantage of Intel's SSE instructions, because all the memory accesses are contiguous. From an implementation standpoint, pulling the activation and pushing the gradient is by far the simplest way to implement convolution layers and well worth the slight compromise in speed.

4.2. Modular debugging

Back-propagation has a good property: it allows neural networks to be expressed and debugged in a modular fashion. For instance, we can assume that a module *M* has a forward propagation function which computes its output *M(I,W)* as a function of its input *I* and its parameters *W*. It also has a backward propagation function (with respect to the input) which computes the input gradient as a function of the output gradient, a gradient function (with respect to the weight), which computes the weight gradient with respect to the output gradient, and a weight update function, which adds the weight gradients to the weights using some updating rules (batch, stochastic, momentum, weight decay, etc). By definition, the Jacobian matrix of a function *M* is defined to be $J_{ki} \equiv \frac{\partial M_k}{\partial x_i}$ (see [5] p. 148 for more information on Jacobian matrix of neural network). Using the backward propagation function and the gradient function, it is straightforward to compute the two Jacobian matrices $\frac{\partial I}{\partial M(I,W)}$ and $\frac{\partial W}{\partial M(I,W)}$ by simply feeding the (gradient) unit vectors $\Delta M_k(I,W)$ to both of these functions, where *k* indexes all the output units of *M*, and only unit *k* is set to 1 and all the others are set to 0. Conversely, we can generate arbitrarily accurate estimates of the Jacobians matrices $\frac{\partial M(I,W)}{\partial I}$ and $\frac{\partial M(I,W)}{\partial W}$ by adding small variations ε to *I* and *W* and calling the *M(I,W)* function. Using the equalities:

$$\frac{\partial I}{\partial M} = F\left(\frac{\partial M}{\partial I}\right)^T \quad \text{and} \quad \frac{\partial W}{\partial M} = F\left(\frac{\partial M}{\partial W}\right)^T$$

where *F* is a function which takes a matrix and inverts each of its elements, one can automatically verify that the forward propagation accurately corresponds to the backward and gradient propagations (note: the back-propagation computes $F(\partial I/\partial M(I,W))$ directly so only a transposition is necessary to compare it with the Jacobian computed by the forward propagation. In other words, if the equalities above are verified to the precision of the machine, learning is implemented correctly. This is particularly useful for large networks since incorrect implementations sometimes yield reasonable results. Indeed, learning algorithms tend to be robust even to bugs. In our implementation, each neural network is a C++ module and is a combination of more basic modules. A module test program instantiates the module in double precision, sets $\varepsilon=10^{-12}$ (the machine precision for double is 10^{-16}), generates random values for *I* and *W*, and performs

a correctness test to a precision of 10^{-10}. If the larger module fails the test, we test each of the sub-modules until we find the culprit. This extremely simple and automated procedure has saved a considerable amount of debugging time.

5. Results

For both fully connected and convolutional neural networks, we used the first 50,000 patterns of the MNIST training set for training, and the remaining 10,000 for validation and parameter adjustments. The result reported on test set where done with the parameter values that were optimal on validation. The two-layer Multi-Layer Perceptron (MLP) in this paper had 800 hidden units, while the two-layer MLP in [3] had 1000 hidden units. The results are reported in the table below:

Algorithm	**Distortion**	**Error**	**Ref.**
2 layer MLP (MSE)	affine	1.6%	[3]
SVM	affine	1.4%	[9]
Tangent dist.	affine+thick	1.1%	[3]
Lenet5 (MSE)	affine	0.8%	[3]
Boost. Lenet4 MSE	affine	0.7%	[3]
Virtual SVM	affine	0.6%	[9]
2 layer MLP (CE)	none	1.6%	this paper
2 layer MLP (CE)	affine	1.1%	this paper
2 layer MLP (MSE)	elastic	0.9%	this paper
2 layer MLP (CE)	elastic	0.7%	this paper
Simple conv (CE)	affine	0.6%	this paper
Simple conv (CE)	elastic	**0.4%**	this paper

Table 1. Comparison between various algorithms.

There are several interesting results in this table. The most important is that elastic deformations have a considerable impact on performance, both for the 2 layer MLP and our convolutional architectures. As far as we know, 0.4% error is best result to date on the MNIST database. This implies that the MNIST database is too small for most algorithms to infer generalization properly, and that elastic deformations provide additional and relevant a-priori knowledge. Second, we observe that convolutional networks do well compared to 2-layer MLPs, even with elastic deformation. The topological information implicit in convolutional networks is not easily inferred by MLP, even with elastic deformation. Finally, we observed that the most recent experiments yielded better performance than similar experiments performed 8 years ago and reported in [3]. Possible explanations are that the hardware is now 1.5 orders of magnitude faster (we can now afford hundreds of epochs) and that in our experiments, CE trained faster than MSE.

6. Conclusions

We have achieved the highest performance known to date on the MNIST data set, using elastic distortion and convolutional neural networks. We believe that these results reflect two important issues.

Training set size: The quality of a learned system is primarily dependent of the size and quality of the training set. This conclusion is supported by evidence from other application areas, such as text[8]. For visual document tasks, this paper proposes a simple technique for vastly expanding the training set: elastic distortions. These distortions improve the results on MNIST substantially.

Convolutional Neural Networks: Standard neural networks are state-of-the-art classifiers that perform about as well as other classification techniques that operate on vectors, without knowledge of the input topology. However, convolutional neural network exploit the knowledge that the inputs are not independent elements, but arise from a spatial structure.

Research in neural networks has slowed, because neural network training is perceived to require arcane black magic to get best results. We have shown that the best results do not require any arcane techniques: some of the specialized techniques may have arisen from computational speed limitations that are not applicable in the 21st Century.

7. References

[1] Y. Tay, P. Lallican, M. Khalid, C. Viard-Gaudin, S. Knerr, "An Offline Cursive Handwriting Word Recognition System", *Proc. IEEE Region 10 Conf.*, (2001).

[2] A. Sinha, *An Improved Recognition Module for the Identification of Handwritten Digits*, M.S. Thesis, MIT, (1999).

[3] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition" Proceedings of the IEEE, v. 86, pp. 2278-2324, 1998.

[4] K. M. Hornik, M. Stinchcombe, H. White, "Universal Approximation of an Unknown Mapping and its Derivatives using Multilayer Feedforward Networks" *Neural Networks*, v. 3, pp. 551-560, (1990).

[5] C. M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, (1995).

[6] L. Yaeger, R. Lyon, B. Webb, "Effective Training of a Neural Network Character Classifier for Word Recognition", *NIPS*, v. 9, pp. 807-813, (1996).

[7] Y. LeCun, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist.

[8] M. Banko, E. Brill, "Mitigating the Paucity-of-Date Problem: Exploring the Effect of Training Corpus Size on Classifier Performance for Natural Language Processing," Proc. Conf. Human Language Technology, (2001).

[9] D. Decoste and B. Scholkopf, “Training Invariant Support Vector Machines”, Machine Learning Journal, vol 46, No 1-3, 2002.