Deep Big Multilayer Perceptrons For Digit Recognition

Dan Claudiu Cireşan[1,2], Ueli Meier[1,2],
Luca Maria Gambardella[1,2], and Jürgen Schmidhuber[1,2]

[1] IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland,
[2] University of Lugano & SUPSI, Switzerland

Abstract. The competitive MNIST handwritten digit recognition benchmark has a long history of broken records since 1998. The most recent advancement by others dates back 8 years (error rate 0.4%). Good old on-line back-propagation for plain multi-layer perceptrons yields a very low 0.35% error rate on the MNIST handwritten digits benchmark with a single MLP and 0.31% with a committee of seven MLP. All we need to achieve this until 2011 best result are many hidden layers, many neurons per layer, numerous deformed training images to avoid overfitting, and graphics cards to greatly speed up learning.

Keywords: NN (Neural Network), MLP (Multilayer Perceptron), GPU (Graphics Processing Unit), training set deformations, MNIST[3], committee, BP (back-propagation)

Note: This work combines three previously published papers [1–3].

1 Introduction

Automatic handwriting recognition is of academic and commercial interest. Current algorithms are already pretty good at learning to recognize handwritten digits. Post offices use them to sort letters; banks use them to read personal checks. MNIST [4] is the most widely used benchmark for isolated handwritten digit recognition. More than a decade ago, artificial neural networks called Multilayer Perceptrons or MLPs [5–7] were among the first classifiers tested on MNIST. Most had few layers or few artificial neurons (units) per layer [4], but apparently back then they were the biggest feasible MLPs, trained when CPU cores were at least 20 times slower than today. A more recent MLP with a single hidden layer of 800 units achieved 0.70% error [8]. However, more complex methods listed on the MNIST web page always seemed to outperform MLPs, and the general trend went towards more and more complex variants of Support Vector Machines or SVMs [9] and combinations of NNs and SVMs [10] etc. Convolutional neural networks (CNNs) achieved a record-breaking 0.40% error rate [8], using novel

[3] http://yann.lecun.com/exdb/mnist/

elastic training image deformations. Recent methods pre-train each hidden CNN layer one by one in an unsupervised fashion (this seems promising especially for small training sets), then use supervised learning to achieve 0.39% error rate [11, 12]. The biggest MLP so far [13] also was pre-trained without supervision, then piped its output into another classifier to achieve an error of 1% without domain-specific knowledge. Deep MLPs initialized by unsupervised pretraining were also successfully applied to speech recognition [14].

Are all these complexifications of plain MLPs really necessary? Can't one simply train really big plain MLPs on MNIST? One reason is that at first glance deep MLPs do not seem to work better than shallow networks [15]. Training them is hard as back-propagated gradients quickly vanish exponentially in the number of layers [16–18], just like in the first recurrent neural networks [19]. Indeed, previous deep networks successfully trained with back-propagation (BP) either had few free parameters due to weight-sharing [4, 8] or used unsupervised, layer-wise pre-training [20, 15, 11]. But is it really true that deep BP-MLPs do not work at all, or do they just need more training time? How to test this? Unfortunately, on-line BP for hundreds/thousands of epochs on large MLPs may take weeks or months on standard serial computers. But can't one parallelize it? Well, on computer clusters this is hard due to communication latencies between individual computers. Parallelization across training cases and weight updates for mini-batches [21] might alleviate this problem, but still leaves the task of parallelizing fully online-BP. Only GPUs are capable of such fine grained parallelism. Multi-threading on a multi-core processor is not easy either. We may speed up BP using SSE (Streaming Single Instruction, Multiple Data Extensions), either manually, or by setting appropriate compiler flags. The maximum theoretical speedup under single precision floating-point, however, is four, which is not enough. And MNIST is large - its 60,000 images take almost 50MB, too much to fit in the L2/L3 cache of any current processor. This requires to continually access data in considerably slower RAM. To summarize, currently it is next to impossible to train big MLPs on CPUs.

We showed how to overcome all these problems by training large, deep MLPs on graphics cards [1] and obtained an error rate of 0.35% with a deep MLP. Deformations proved essential to prevent MLPs with up to 12 million free parameters from overfitting. To let the deformation process keep up with network training speed we had to port it onto the GPU as well.

At some stage in the classifier design process one usually has collected a set of possible classifiers. Often one of them yields best performance. Intriguingly, however, the sets of patterns misclassified by the different classifiers do not necessarily overlap. This information could be harnessed in a committee. In the context of handwritten recognition it was already shown [22] how a combination of various classifiers can be trained more quickly than a single classifier yielding the same error rate. Here we focus on improving recognition rate using a committee of MLP. Our goal is to produce a group of classifiers whose errors on various parts of the training set differ (are uncorrelated) as much as possible [23].

We show that for handwritten digit recognition this can be achieved by training identical classifiers on data normalized in different ways prior to training.

2 Data

MNIST consists of two datasets, one for training (60,000 images) and one for testing (10,000 images). Many studies divide the training set into two sets consisting of 50,000 images for training and 10,000 for validation. Our network is trained on slightly deformed images, continually generated in on-line fashion; hence we may use the whole un-deformed training set for validation, without wasting training images. Pixel intensities of the original gray scale images range from 0 (background) to 255 (max foreground intensity). $28 \times 28 = 784$ pixels per image get mapped to real values $\frac{pixel\ intensity}{127.5} - 1.0$ in $[-1.0, 1.0]$, and are fed into the NN input layer.

3 Architectures

We train 5 MLPs with 2 to 9 hidden layers and varying numbers of hidden units. Mostly but not always the number of hidden units per layer decreases towards the output layer (Table 3). There are 1.34 to 12.11 million free parameters (or weights, or synapses).

We use standard on-line BP [24] , without momentum, but with a variable learning rate that shrinks by a multiplicative constant after each epoch, from 10^{-3} down to 10^{-6}. Weights are initialized with a uniform random distribution in $[-0.05, 0.05]$. Each neuron's activation function is a scaled hyperbolic tangent: $y(a) = A \tanh Ba$, where $A = 1.7159$ and $B = 0.6666$ [4], and a softmax output layer is used. Weight initializaiton and annealing rate are not overly important as long as sensible choices are made.

4 Deforming images to get more training instances

So far, the best results on MNIST were obtained by deforming training images [8], thus greatly increasing their number. This allows for training networks with many weights without overfitting. We combine affine (rotation, scaling and horizontal shearing) and elastic deformations (Figure 1), characterized by the following real-valued parameters:

- σ and α: for elastic distortions emulating uncontrolled oscillations of hand muscles [8];
- β: a random angle from $[-\beta, +\beta]$ describes either rotation or horizontal shearing. In case of shearing, $\tan \beta$ defines the ratio between horizontal displacement and image height;
- γ_x, γ_y: for horizontal and vertical scaling, randomly selected from $[1 - \gamma/100, 1 + \gamma/100]$.

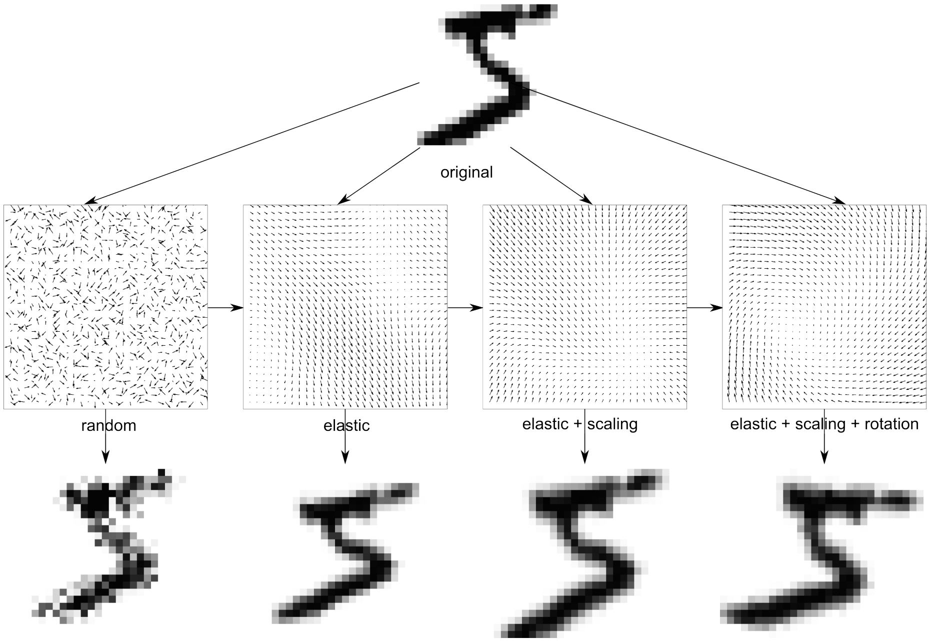


Fig. 1. Original digit (top) and distorted digits (bottom). The digit was distorted with four different displacement fields shown in the middle.

Each affine deformation is fully defined by the corresponding real-valued parameter that is randomly drawn from a uniform interval. Building the elastic deformation field on the other hand consists of three parts: 1) create an initial random distortion vector field, 2) smooth the random distortion field by convolving it with a Gaussian kernel defined by a standard deviation σ, and 3) scale the smoothed deformation field with α, the elastic scaling parameter.

At the beginning of every epoch the entire original MNIST training set gets deformed. Initial experiments with small networks suggested the following deformation parameters: $\sigma = 5.0 - 6.0$, $\alpha = 36.0 - 38.0$, $\gamma = 15 - 20$. Since digits 1 and 7 are similar they get rotated/sheared less ($\beta = 7.5°$) than other digits ($\beta = 15.0°$).

It takes 83 CPU seconds to deform the 60,000 MNIST training images, most of them (75 seconds) for elastic distortions. Only the most time-consuming part of the latter—convolution with a Gaussian kernel—is ported to the GPU. The MNIST training set is split into 600 sequentially processed minibatches of 100 samples each. MNIST digits are scaled from the original 28×28 pixels to 29×29 pixels, to get a proper center, which simplifies convolution. Each batch grid (10×10 images) has 290×290 cells, zero-padded to 310×310, thus avoiding

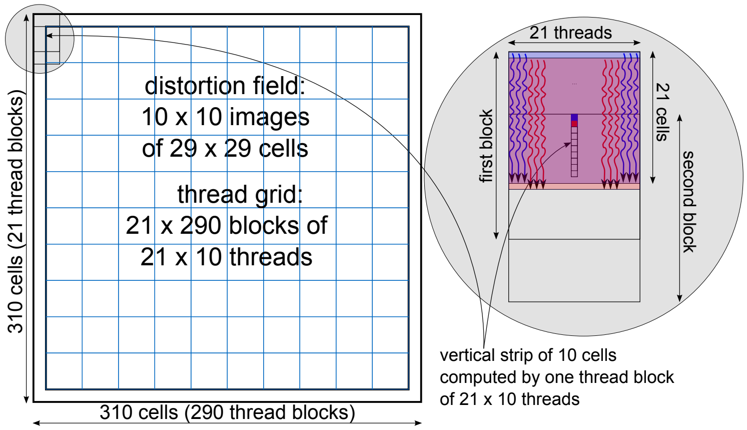


Fig. 2. Mapping the thread grid of convolution onto the distortion field.

margin effects when applying a Gaussian convolution kernel of size 21 × 21. The GPU program groups many threads into a block, where they share the same Gaussian kernel and parts of the random field. All 29 × 290 blocks contain 21 (the kernel size) ×10 threads, each computing a vertical strip of the convolution (Figure 2). Generating the elastic displacement field takes only 1.5 seconds. Deforming the whole training set is more than 10 times faster, taking 6.5 instead of the original 83 seconds. Further optimizations would be possible by porting all deformations onto GPU, and by using the hardware's interpolation capabilities to perform the final bilinear interpolation. We omitted these since deformations are already pretty fast (deforming all images of one epoch takes only 3-10 % of total computation time, depending on MLP size).

5 Forming a Committee

The training procedure of a single network of the committee is summarized in Figure 3. Each network is trained separately on normalized or original data. The normalization is done for all digits in the training set prior to training (normalization stage). For the network trained on original MNIST data the normalization step is omitted. Normalization of the original MNIST data is mainly motivated by practical experience. MNIST digits are already normalized such that the width or height of the bounding box equals 20 pixels. The variation of the aspect ratio for various digits is quite large, and we normalize the width of the bounding box to range from 10 to 20 pixels with a step-size of 2 pixels prior to training for all digits except ones. Normalizing the original MNIST training data results in 6 normalized training sets. In total we perform experiments with seven different data sets (6 normalized and the original MNIST).

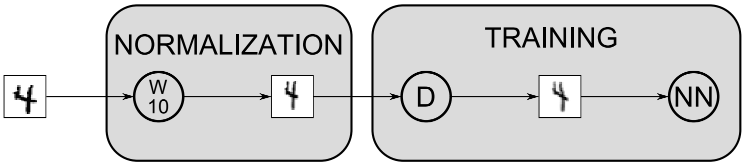


Fig. 3. Training a committee member. Original MNIST training data (left digit) is normalized (W10) prior to training (middle digit). The normalized data is distorted (D) for each training epoch and used as input (right digit) to the network (NN). Each depicted digit represents the whole training set.

We perform six experiments to analyze performance improvements due to committees. Each committee consists of seven randomly initialized one-hidden-layer MLPs with 800 hidden units, trained with the same algorithm on randomly selected batches. The six committees differ only in how the data are normalized (or not) prior to training and on how the data are deformed during training. The committees are formed by simply averaging the corresponding outputs as shown in Figure 4.

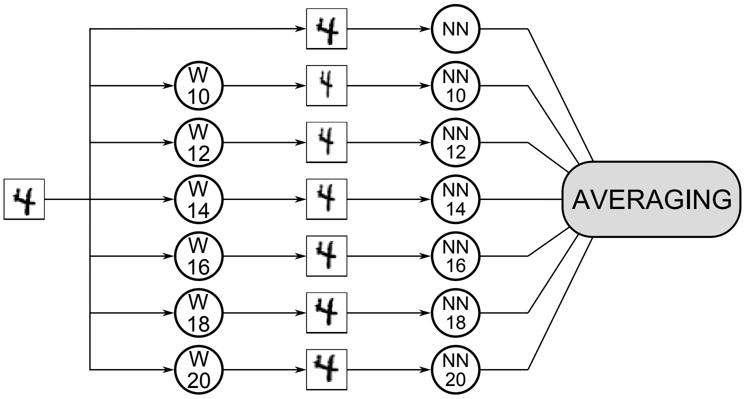


Fig. 4. Testing with a committee. If required, the input digits are width-normalized (W blocks) and then processed by the corresponding MLP. The committee is formed by averaging the outputs of all MLPs.

The first two experiments are performed on undeformed original MNIST images. We train a committee of seven MLPs on original MNIST and we also form a committee of MLPs trained on normalized data. In Table 1 the error rates are listed for each of the individual nets and the committees. The improvement of the committee with respect to the individual nets is marginal for the first experiment. Adding normalization, the individual experts as well as the corresponding committee of the second experiment achieve substantially better recognition rates.

Table 1. *Error rates of individual nets and of the two resulting committees. For experiment 1 seven randomly initialized nets are trained on the original MNIST, whereas for experiment 2 seven randomly initialized nets are trained on width-normalized data: WN x - Width Normalization of the bounding box to be x pixels wide; ORIG - original MNIST.*

	Error rate [%]	
	Exp. 1	Exp. 2
Net 1:	init 1: 1.79	WN 10: 1.62
Net 2:	init 2: 1.80	WN 12: 1.37
Net 3:	init 3: 1.77	WN 14: 1.48
Net 4:	init 4: 1.72	WN 16: 1.53
Net 5:	init 5: 1.91	WN 18: 1.56
Net 6:	init 6: 1.86	WN 20: 1.49
Net 7:	init 7: 1.75	ORIG: 1.79
Average:	**1.70**	**1.31**

To study the combined effect of normalization and deformation, we perform four additional experiments on deformed MNIST (Tab. 2). Unless stated otherwise, default elastic deformation parameters $\sigma = 6$ and $\alpha = 36$ are used. All experiments with deformed images use independent horizontal and vertical scaling of maximum 12.5% and a maximum rotation of ±12.5°. Experiment 3 is similar to Experiment 1, except that the data are continually deformed. Error rates of the individual experts are much lower than without deformation (Tab. 1). In experiment 4 we randomly reselect training and validation sets for each of the individual experts, simulating in this way the bootstrap aggregation technique [25]. The resulting committee performs slightly better than that of experiment 3. In experiment 5 we vary deformations for each individual network. Error rates of some of the nets are bigger than in experiments 3 and 4, but the resulting committee has a lower error rate. In the last experiment we train seven MLPs on width-normalized images that are also continually deformed during training. The error rate of the committee (0.43 %) is the best result ever reported for such a simple architecture. We conclude that width-normalization is essential for good committee performance, i.e. it is not enough to form a committee from trained nets with different initializations (experiment 3) or trained on subsets of the original dataset (experiment 4).

6 Using the GPU to train deep MLPs

Using simple tricks, such as creating a virtually infinite amount of training data through random distortions at the beginning of every epoch and forming a com-

Table 2. *Error rates of the individual nets and of the resulting committees. In experiments 3 and 4 seven randomly initialized nets are trained on deformed ($\sigma = 6$, $\alpha = 36$) MNIST, whereas in experiment 4 training and validation sets are reselected. In experiment 5 seven randomly initialized nets are trained on deformed (different σ, α) MNIST, and in experiment 6 seven randomly initialized nets are trained on width-normalized, deformed ($\sigma = 6$, $\alpha = 36$) MNIST. WN x - Width Normalization of the bounding box to be x pixels wide; ORIG - original MNIST.*

		Error rate [%]			
		Exp. 3	Exp. 4	Exp. 5	Exp. 6
Net 1:	init 1:	0.72	0.68	$\sigma = 4.5$ $\alpha = 36$: 0.69	WN 10: 0.64
Net 2:	init 2:	0.71	0.82	$\sigma = 4.5$ $\alpha = 42$: 0.94	WN 12: 0.78
Net 3:	init 3:	0.72	0.73	$\sigma = 6.0$ $\alpha = 30$: 0.55	WN 14: 0.70
Net 4:	init 4:	0.71	0.69	$\sigma = 6.0$ $\alpha = 36$: 0.72	WN 16: 0.60
Net 5:	init 5:	0.62	0.71	$\sigma = 6.0$ $\alpha = 42$: 0.60	WN 18: 0.59
Net 6:	init 6:	0.65	0.70	$\sigma = 7.5$ $\alpha = 30$: 0.86	WN 20: 0.70
Net 7:	init 7:	0.69	0.75	$\sigma = 7.5$ $\alpha = 36$: 0.79	ORIG: 0.71
Average:		**0.56**	**0.53**	**0.49**	**0.43**

mittee of experts trained on differently preprocessed data, state-of-the art results are obtained on MNIST with a relatively small (800 hidden units) single hidden layer MLP. Here we report results using deep MLPs, with as many as 5 hidden layers and up to 12 millions of free parameters, that are prohibitive to train on current CPUs but can successfully be trained on GPUs in a few days. All simulations were performed on a computer with a Core i7 920 2.66GHz processor, 12GB of RAM, and a GTX 480 graphics card. The GPU accelerates the deformation routine by a factor of 10 (only elastic deformations are GPU-optimized); the forward propagation (FP) and BP routines are sped up by a factor of 50. We pick the trained MLP with the lowest validation error, and evaluate it on the MNIST test set.

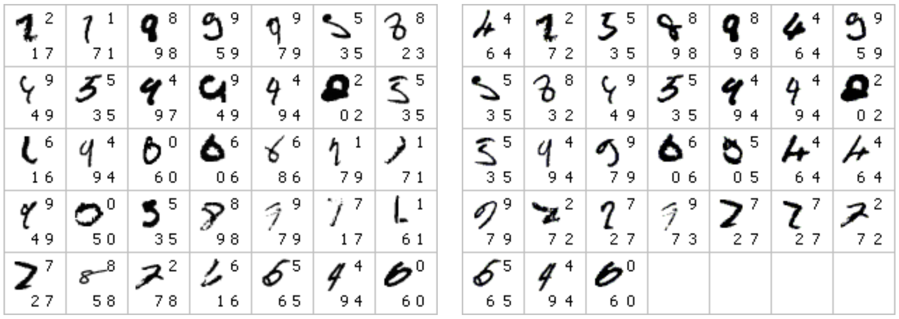
6.1 Single MLP

We train various MLP and summarize the results in Table 3. Training starts with a learning rate of 10^{-3} multiplied with 0.997 after every epoch until it reaches 10^{-6}, thus resulting in more than 2000 epochs, which can be computed in a few days even for the biggest net. The best network has an error rate of only 0.35% (35 out of 10,000 digits). This is better than the best previously published results, namely, 0.39% [11] and 0.40% [8], both obtained by more complex methods. The 35 misclassified digits are shown in Figure 5a. Many of them are ambiguous and/or uncharacteristic, with obviously missing parts or strange strokes etc. Interestingly, the second guess of the network is correct for 30 out of the 35 misclassified digits. The best test error of this MLP is even lower (0.32%) and may be viewed as the maximum capacity of the network, i.e. what it can learn if we do not get the result for the lowest error on validation set. Performance clearly profits from adding hidden layers and more units per layer. For example, network 5 has more but smaller hidden layers than network 4 (Table 3).

Networks with up to 12 million weights can successfully be trained by plain gradient descent to achieve test errors below 1% after 20-30 epochs in less than 2 hours of training. How can networks with so many parameters generalize well on the unseen test set? Answer: the continual deformations of the training set generate a virtually infinite supply of training examples, and the network rarely sees any training image twice. Without any distortions, the error for all networks is around 1.7-1.8% (last column in Table 3).

Table 3. Error rates on MNIST test set. Architecture: 841 input neurons, hidden layers containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs. TEfBV - test error for best validation, BTE - best test error.

ID	architecture	TEfBV	BTE	simulation	weights	test error [%]
	(number of neurons in each layer)	[%]	[%]	time [h]	[millions]	no distortion
1	1000, 500, 10	**0.49**	0.44	23.4	1.34	1.78
2	1500, 1000, 500, 10	**0.46**	0.40	44.2	3.26	1.85
3	2000, 1500, 1000, 500, 10	**0.41**	0.39	66.7	6.69	1.73
4	2500, 2000, 1500, 1000, 500, 10	**0.35**	0.32	114.5	12.11	1.71
5	9 × 1000, 10	**0.44**	0.43	107.7	8.86	1.81



(a)

(b)

Fig. 5. The misclassified digits, together with the two most likely predictions (bottom, from left to right) and the correct label according to MNIST (top, right): (a) the best network from Table 3. (b) the committee from Table 4.

6.2 Committee of MLP

Here we list results of a committee of MLP with the architecture that obtained 0.35% error rate on MNIST (841 neurons in the input layer, five hidden layers containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs). We train six additional nets with the same architecture on normalized data (the width of the digits is normalized prior to training) and form a committee by averaging the predictions of the individual nets (Table 4). The width-normalization is essential for good committee performance as shown in Section 5. All committee members distort their width-normalized training dataset before each epoch.

Table 4. Error rates of the individual nets and of the resulting committee. Architecture: 841 input neurons, hidden layers containing 2500, 2000, 1500, 1000 and 500 neurons, and 10 outputs. WN x—Width Normalization of the bounding box to be x pixels wide.

WN	10	12	14	16	18	20	ORIGINAL MNIST
test error [%]	0.52	0.45	0.44	0.49	0.36	0.38	0.35
committee error [%]	**0.31**						

Interestingly, the error of the extremely simple committee (0.31%) is lower than those of the individual nets. This is the best result ever reported on MNIST using MLP. Many of the 31 misclassified digits (Figure 5b) are ambiguous and/or uncharacteristic, with obviously missing parts or strange strokes etc. Remarkably, the committee's second guess is correct for 29 of the 31.

Discussion

In recent decades the amount of raw computing power per Euro has grown by a factor of 100-1000 per decade. Our results show that this ongoing hardware progress may be more important than advances in algorithms and software (although the future will belong to methods combining the best of both worlds). Current graphics cards (GPUs) are already more than 50 times faster than standard microprocessors when it comes to training big and deep neural networks by the ancient algorithm, on-line back-propagation (weight update rate up to $7.5\times10^9/s$, and more than 10^{15} per trained network). On the competitive MNIST handwriting benchmark, single precision floating-point GPU-based neural nets surpass all previously reported results, including those obtained by much more complex methods involving specialized architectures, unsupervised pre-training, combinations of machine learning classifiers etc. Training sets of sufficient size to avoid overfitting are obtained by appropriately deforming images. Of course, the approach is not limited to handwriting, and obviously holds great promise for many visual and other pattern recognition problems.

Although big deep MLP are very powerful general classifiers when combined with an appropriate distortion algorithm to enhance the training set, they cannot compete with dedicated architectures such as max-pooling convolutional neural networks on complex image classification problems. For tasks more difficult than handwritten digit recognition MLP are not competitive anymore, both in classification performance and required training time. We have recently shown [26] that large convolutional neural networks combined with max-pooling [27] improve the state-of-the-art by 30-80% for a plethora of benchmarks like Latin letters [28], Chinese characters [26], traffic signs [29, 30], stereo projection of 3D models [31, 26] and even small natural images [26].

Acknowledgments

Part of this work got started when Dan Cireşan was a PhD student at University "Politehnica" of Timişoara. He would like to thank his PhD advisor, Ştefan Holban, for his guidance, and Răzvan Moşincat for providing a CPU framework for MNIST. This work was partially supported by the Swiss Commission for Technology and Innovation (CTI), Project n. 9688.1 IFF: Intelligent Fill in Form., and by a FP7-ICT-2009-6 EU Grant, Project Code 270247: A Neurodynamic Framework for Cognitive Robotics: Scene Representations, Behavioral Sequences, and Learning.

Appendix - GPU implementation

Graphics Processing Unit

Until 2007 the only way to program a GPU was to translate the problem-solving algorithm into a set of graphical operations. Despite being hard to code and difficult to debug, several GPU-based NN implementations were developed when GPUs became faster than CPUs. Two layer MLPs [32] and CNNs [33] have been previously implemented on GPUs. Although speedups were relatively modest, these studies showed how GPUs can be used for machine learning. More recent GPU-based CNNs trained in batch mode are two orders of magnitude faster than CPU-based CNNs [27].

The GPU code is written using CUDA (Compute Unified Device Architecture), a C-like general programming language. GPU speed and memory bandwidth are vastly superior to those of CPUs, and crucial for fast MLP implementations. To fully understand our algorithm in terms of GPU / CUDA, please visit the NVIDIA website [34]. According to CUDA terminology, the CPU is called ***host*** and the graphics card ***device*** or ***GPU***.

Deformations

Only the most time-consuming part of the latter – convolution with a gaussian kernel [8] – is ported to the GPU. The MNIST training set is split into 600 sequentially processed batches. MNIST digits are scaled from the original 28×28 pixels to 29×29 pixels, to get a proper center, which simplifies convolution. An image grid has 290×290 cells, zero-padded to 300×300, thus avoiding margin effects when applying a Gaussian convolution kernel of size 21×21.

Our GPU program groups many threads into a block, where they share the same gaussian kernel and parts of the random field. The blocks contain 21 (the kernel size) $\times 10$ threads, each computing a vertical strip of the convolution operation (Listing 1.1).

Training algorithm

We closely follow the standard BP algorithm [24], except that BP of deltas and weight updates are disentangled and performed sequentially. This allows for more parallelism within each routine.

Forward propagation

The algorithm is divided into two kernels. The weight matrix W is partitioned as illustrated in Figure 6.

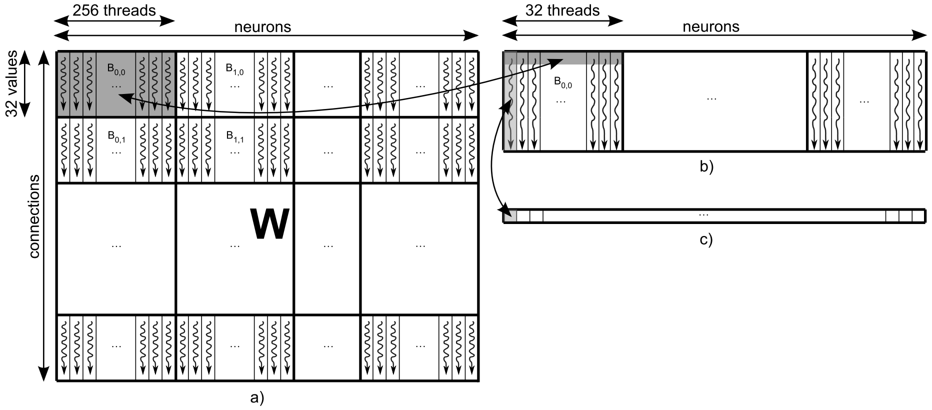


Fig. 6. Forward propagation: a) mapping of kernel 1 grid onto the padded weight matrix; b) mapping the kernel 2 grid onto the partial dot products matrix; c) output of forward propagation.

Kernel 1

Each block has 256 threads (Figure 6a), each computing a partial dot product of 32 component vectors. The dot products are stored in a temporary matrix (Figure 6b). This kernel has a very high throughput: average memory bandwidth is 115GB/s. This is possible because many relatively small blocks keep the GPU busy. Each block uses shared memory for storing the previous layer activations, which are simultaneously read by the first 32 threads of each block and then used by all 256 threads. After thread synchronization, the partial dot products are computed in parallel (Listing 1.2). The number of instructions is kept to a minimum by pre-computing all common index parts.

Kernel 2

The thread grid (Figure 6b) has only one row of blocks consisting of *warp* threads, since each thread has to compute a complete dot product (Figure 6c) and then pipe it into the activation function. This kernel (Listing 1.2) is inefficient for layers with fewer than 1024 incoming connections per neuron, especially for the last layer which has only ten neurons, one for each digit. That is, its grid will have only one block, occupying only 6% of the GTX 480 GPU.

Backward propagation

This is similar to FP, but we need W^T for coalesced access. Instead of transposing the matrix, the computations are performed on patches of data read from device memory into shared memory, similar to the optimized matrix transposition algorithm of [35]. Shared memory access is much faster, without coalescing restrictions. Because we have to cope with layers of thousands of neurons, back-propagating deltas uses a reduction method implemented in two kernels communicating partial results via global memory (Listing 1.3).

Kernel 1 The bi-dimensional grid is divided into blocks of *warp* (32) threads. The kernel starts by reading a patch of 32×32 values from W. The stride of the shared memory block is 33 ($warp + 1$), thus avoiding all bank conflicts and significantly improving speed. Next, 32 input delta values are read and all memory locations that do not correspond to real neurons (because of vertical striding) are zero-padded to avoid branching in subsequent computations. The number of elements is fixed to *warp* size, and the computing loop is unrolled for further speedups. Before finishing, each thread writes its own partial dot product to global memory.

Kernel 2

This kernel completes BP of deltas by summing up partial deltas computed by the previous kernel. It multiplies the final result by the derivative of the activation function applied to the current neuron's state, and writes the new delta to global memory.

Weight updating

The algorithm (Listing 1.4) starts by reading the appropriate delta, and precomputes all repetitive expressions. Then the first 16 threads read the states from global memory into shared memory. The "bias neuron" with constant activation 1.0 is dealt with by conditional statements, which could be avoided through expressions containing the conditions. Once threads are synchronized, each single thread updates 16 weights in a fixed unrolled loop.

Listing 1.1. Convolution Kernel for elastic distortion.

```
__global__ void ConvolveField(float *randomfield, int width, int height, float *kernel, float *outputfield, float elasticScale){
    float sum=0;
    const int stride_k=GET_STRIDE(GAUSSIAN_FIELD_SIZE,pitch_x>>2); //stride for gaussian kernel
    __shared__ float K[GAUSSIAN_FIELD_SIZE][stride_k];    //kernel (21 x 32 values)
    __shared__ float R[GAUSSIAN_FIELD_SIZE+9][GAUSSIAN_FIELD_SIZE]; //random field (30 x 21 values)
    __shared__ float s[10][GAUSSIAN_FIELD_SIZE];    //partial sums (10 x 21 values)
    int stride_in=GET_STRIDE(width,pitch_x>>2);    //random field stride as a multiple of 32
    int stride_out=GET_STRIDE(width-GAUSSIAN_FIELD_SIZE+1,pitch_x>>2); //output stride as a multiple of 32

    //loading gaussian kernel into K (21 x 21 values)
    K[ 0+threadIdx.y][threadIdx.x] = kernel[( 0+threadIdx.y)*stride_k + threadIdx.x];//rows 0..9
    K[10+threadIdx.y][threadIdx.x] = kernel[(10+threadIdx.y)*stride_k + threadIdx.x];//rows 10..19
    if(threadIdx.y==0)
        K[20+threadIdx.y][threadIdx.x] = kernel[(20+threadIdx.y)*stride_k + threadIdx.x];//row 20

    //loading randomfield into R
    //0..9 x 21 values
    R[ 0+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+ 0+threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
    //10..19 x 21 values
    R[10+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+10+threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
    //20..29 x 21 values
    R[20+threadIdx.y][threadIdx.x] = randomfield[(10*blockIdx.y+20+threadIdx.y)*stride_in + blockIdx.x + threadIdx.x];
    __syncthreads(); //wait until everything is read into shared memory

    //computing partial sums
    #pragma unroll 21 //GAUSSIAN_FIELD_SIZE
    for(int i=0;i<GAUSSIAN_FIELD_SIZE;i++)
        sum += R[threadIdx.y + i][threadIdx.x] * K[i][threadIdx.x];
    s[threadIdx.y][threadIdx.x]=sum;
    __syncthreads();

    if(threadIdx.x==0){ //the first column of threads computes the final values of the convolutions
        #pragma unroll 20//GAUSSIAN_FIELD_SIZE-1
        for(int i=1;i<GAUSSIAN_FIELD_SIZE;i++) sum+=s[threadIdx.y][i];
        outputfield[(blockIdx.y*10+threadIdx.y)*stride_out + blockIdx.x] = sum * elasticScale;
    }
}
```

Listing 1.2. Forward propagation kernels.

```
__global__ void MLP_FP_reduction_Kernel1(float *prevLN, float *W, float *
    partialsum, unsigned int neurons, unsigned int prevneurons){
      const int threads=256;
      const int stride=GET_STRIDE(neurons,pitch_x>>2); //horizontal stride of
           W matrix
      int X=blockIdx.x*threads + threadIdx.x;  //precomputing expressions
      int Y=X+stride*blockIdx.y;
      int Z=blockIdx.y*pitch_y*stride + X;
      float sum=0.0f;
      __shared__ float output[pitch_y];
      if(blockIdx.y==0)
            if(threadIdx.x==0) output[0]=1.0f;
            else if(threadIdx.x<pitch_y) //there are only 32 values to read and
                128 threads
                  output[threadIdx.x] = threadIdx.x-1<prevneurons ?
                      prevLN[threadIdx.x-1] : 0.0f;
                  else;
      else if(threadIdx.x<pitch_y) //there are only 32 values to read and 128
          threads
                  output[threadIdx.x] = blockIdx.y*pitch_y+threadIdx.x-1<
                      prevneurons ?
                                                prevLN[blockIdx.y*pitch_y
                                                    +threadIdx.x-1] : 0.0f
                                                    ;
          else;
      __syncthreads();
      if(X<neurons){//compute partial sums
            //#pragma unroll 32
            int size=0;
            if((blockIdx.y+1)*pitch_y>=prevneurons+1)
                  size = prevneurons + 1 - blockIdx.y*pitch_y;
            else size=pitch_y;
            for (int ic=0; ic<size; ic++){
                  sum += output[ic] * W[Z];
                  Z+=stride;
            }
            partialsum[Y]=sum;
      }
}

__global__ void MLP_FP_reduction_Kernel2(float *currLN, float *partialsum,
    unsigned int neurons, unsigned int size){
      float sum=0.0f;
      int idx = blockIdx.x*(pitch_x>>2) + threadIdx.x;  //precomputed index
      unsigned int stride = GET_STRIDE(neurons,pitch_x>>2);  //stride for
          partialsum matrix

      if(idx>=neurons)return;      //is this thread computing a true neuron?
      for (int i=0; i<size; i++) sum += partialsum[i*stride+idx]; //computing
          the final dot product
      currLN[idx] = SIGMOIDF(sum);      //applying activation
}
```

Listing 1.3. Backpropagating deltas kernels.

```
__global__ void backPropagateDeltasFC_A(float *indelta, float *weights, unsigned
      int ncon, unsigned int nrneur, float *partial){
    const int  px = pitch_x>>2;
    unsigned int stride_x = GET_STRIDE(nrneur,px);
    unsigned int stride_y = GET_STRIDE(ncon,pitch_y);
    float   outd = 0.0;
    int     idx = blockIdx.x*px+threadIdx.x;
    int     X = blockIdx.y*pitch_y*stride_x + idx;
    int     Y = threadIdx.x;
    __shared__ float w[32*33];    //pitch_y and px should be equal ! +1 to avoid
          bank conflict!
    __shared__ float id[px];    //input delta
            #pragma unroll 32 //read the weight patch in shared memory
    for(int i=0;i<pitch_y;i++){w[Y]=weights[X]; X+=stride_x; Y+=33;}
    //read the input delta patch in shared memory
    if(idx>=nrneur) id[threadIdx.x]=0; //a fake input delta for inexistent
          indelta
    else id[threadIdx.x]=indelta[idx];
    __syncthreads(); //not needed for block with warp number of threads: implicit
           synchronization
    #pragma unroll 32 //compute partial results
    for(int i=0;i<px;i++) outd+=w[threadIdx.x*33+i]*id[i];
    //write out the partial results
    partial[blockIdx.x*stride_y + blockIdx.y*pitch_y + threadIdx.x] = outd;
}
__global__ void backPropagateDeltasFC_B(float *outdelta,float *instates, unsigned
       int ncon, unsigned int nrneur, float *partial){
    int px=pitch_x>>2;
    unsigned int stride_x = GET_STRIDE(nrneur,px);
    unsigned int stride_y = GET_STRIDE(ncon,pitch_y);
    float outd = 0.0;
    int size=stride_x/px;
    int idx=blockIdx.x*pitch_y+threadIdx.x;
    if(idx==0); //true only for block and thread 0
    else{
            for(int i=0;i<size;i++)
                    outd+=partial[i*stride_y + idx];
            outdelta[idx-1] = outd * DSIGMOIDF(instates[idx-1]); //-1
                 BIAS ...
    }
}
```

Listing 1.4. Weights adjustment kernel.

```
__global__ void adjustWeightsFC(float *states,float *deltas, float *weights, float
   eta, unsigned int ncon, unsigned int nrneur){
      const int pitch_y=16;
      const int threads=256;
      unsigned int px = pitch_x >> 2;
      unsigned int stride_x = GET_STRIDE(nrneur,px);
      float etadeltak = eta*deltas[blockIdx.x*threads+threadIdx.x],t;
      int b=blockIdx.y*stride_x*pitch_y + threads*blockIdx.x + threadIdx.x;
      __shared__ float st[pitch_y]; //for states
      int cond1 = blockIdx.y || threadIdx.x;
      int cond2 = (blockIdx.y+1)*pitch_y<=ncon;
      int size = cond2 * pitch_y + !cond2 * (ncon%pitch_y);
      if(threadIdx.x<pitch_y) st[threadIdx.x] = cond1 * states[blockIdx.y*pitch_y
          + threadIdx.x - 1] + !cond1;
      __syncthreads();

      if (blockIdx.x*threads + threadIdx.x < nrneur){
            #pragma unroll 16
            for (int j=0; j<16; j++){
                  t=weights[b];
                  t-= etadeltak * st[j];
                  weights[b]=t;
                  b+=stride_x;}}
}
```

References

1. Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep, big, simple neural nets for handwritten digit recognition. Neural Computation **22**(12) (2010) 3207–3220
2. Ciresan, D.C., Meier, U., Gambardella, L.M., JürgenSchmidhuber: Handwritten Digit Recognition with a Committee of Deep Neural Nets on GPUs. Technical Report IDSIA-03-11, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA) (2011)
3. Meier, U., Ciresan, D.C., Gambardella, L.M., Schmidhuber, J.: Better digit recognition with a committee of simple neural nets. In: ICDAR. (2011) 1135–1139
4. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11) (November 1998) 2278–2324
5. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University (1974)
6. LeCun, Y.: Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). In: Proceedings of Cognitiva 85, Paris, France (1985) 599–604
7. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations. MIT Press, Cambridge, MA, USA (1986) 318–362
8. Patrice Y., S., Dave, S., John C., P.: Best practices for convolutional neural networks applied to visual document analysis. In: Seventh International Conference on Document Analysis and Recognition. (2003) 958–963
9. Decoste, D., Scholkopf, B.: Training invariant support vector machines. Machine learning (46) (2002) 161–190
10. Lauer, F., Suen, C., Bloch, G.: A trainable feature extractor for handwritten digit recognition. Pattern Recognition (40) (2007) 1816–1824
11. Ranzato, M., Poultney, C., Chopra, S., LeCun, Y.: Efficient learning of sparse representations with an energy-based model. In et al., J.P., ed.: Advances in Neural Information Processing Systems (NIPS 2006), MIT Press (2006)
12. Ranzato, M., Fu Jie Huang, Y.L.B., LeCun, Y.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. In: Proc. of Computer Vision and Pattern Recognition Conference. (2007)
13. Salakhutdinov, R., Hinton, G.: Learning a nonlinear embedding by preserving class neighborhood structure. In: Proc. of the International Conference on Artificial Intelligence and Statistics. Volume 11. (2007)
14. Mohamed, A., Dahl, G., Hinton, G.E.: Deep belief networks for phone recognition. In: Proc. of NIPS 2009 Workshop on Deep Learning for Speech Recognition and Related Applications. (2009)
15. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Neural Information Processing Systems. (2006)
16. Hochreiter, S.: Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München (1991) See www7.informatik.tu-muenchen.de/˜hochreit; advisor: J. Schmidhuber.
17. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer, S.C., Kolen,

J.F., eds.: A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press (2001)

18. Hinton, G.E.: To recognize shapes, first learn to generate images. Computational Neuroscience: Theoretical Insights into Brain Function (2007)
19. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9** (1997) 1735–1780
20. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313** (2006)
21. Nair, V., Hinton, G.E.: 3D object recognition with deep belief nets. In: Advances in Neural Information Processing Systems. (2009)
22. Chellapilla, K., Shilman, M., Simard, P.: Combining multiple classifiers for faster optical character recognition. In: Document Analysis Systems VII. Springer Berlin / Heidelberg (2006) 358–367
23. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
24. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edition edn. Prentice-Hall, Englewood Cliffs, NJ (2003)
25. Breiman, L.: Bagging predictors. Machine Learning **24** (1996) 123–140
26. Ciresan, D.C., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: Computer Vision and Pattern Recognition. (2012) 3642–3649
27. Scherer, D., Behnke, S.: Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors. In: Proc. of NIPS 2009 Workshop on Large-Scale Machine Learning: Parallelism and Massive Datasets. (2009)
28. Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Convolutional neural network committees for handwritten character recognition. In: International Conference on Document Analysis and Recognition. (2011) 1135-1139.
29. Ciresan, D.C., Meier, U., Masci, J., Schmidhuber, J.: A committee of neural networks for traffic sign classification. In: International Joint Conference on Neural Networks. (2011) 1918–1921
30. Ciresan, D.C., Meier, U., Masci, J., Schmidhuber, J.: Multi-column deep neural network for traffic sign classification. Neural Networks **32** (2012) 333338
31. Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: International Joint Conference on Artificial Intelligence. (2011) 1237–1242
32. Steinkraus, D., Simard, P.Y.: Gpus for machine learning algorithms. In: International Conference on Document Analysis and Recognition. (2005) 1115–1120
33. Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition. (2006)
34. NVIDIA: NVIDIA CUDA. Reference Manual. Volume 2.3. NVIDIA (2009)
35. Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in cuda. In: NVIDIA GPU Computing SDK. (2009) 1–2