LETTER Communicated by Yann LeCun

Deep, Big, Simple Neural Nets for Handwritten Digit Recognition

Dan Claudiu Cireşan
dan@idsia.ch
Ueli Meier
ueli@idsia.ch
Luca Maria Gambardella
luca@idsia.ch
Jürgen Schmidhuber
juergen@idsia.ch
IDSIA, 6928 Manno-Lugano, Switzerland, and University of Lugano, 6900 Lugano, Switzerland, and Sculoa universitaria professionale della svizzera italiana, 6928 Manno-Lugano, Switzerland

Good old online backpropagation for plain multilayer perceptrons yields a very low 0.35% error rate on the MNIST handwritten digits benchmark. All we need to achieve this best result so far are many hidden layers, many neurons per layer, numerous deformed training images to avoid overfitting, and graphics cards to greatly speed up learning.

1 Introduction

Automatic handwriting recognition is of academic and commercial interest. Current algorithms are already quite good at learning to recognize handwritten digits. Post offices use them to sort letters and banks to read personal checks. MNIST (LeCun, Bottou, Bengio, & Haffner, 1998) is the most widely used benchmark for isolated handwritten digit recognition. More than a decade ago, artificial neural networks called multilayer perceptrons (MLPs; Werbos, 1974; LeCun, 1985; Rumelhart, Hinton, & Williams, 1986) were among the first classifiers tested on MNIST. Most had few layers or few artificial neurons (units) per layer (LeCun et al., 1998), 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 (Simard, Steinkraus, & Platt, 2003). However, more complex methods listed on the MNIST Web page always seemed to outperform MLPs, and the general trend went toward more and more complex variants of support vector machines (SVMs; Decoste & Schölkopf, 2002) and combinations of neural networks (NN) and SVMs (Lauer, Suen, & Bloch, 2007). Convolutional neural networks (CNNs) achieved a record-breaking 0.40% error rate (Simard et al., 2003) using novel

elastic training image deformations. Recent methods pretrain 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 a 0.39% error rate (Ranzato, Poultney, Chopra, & LeCun, 2006; Ranzato, Huang, Boureau, & LeCun, 2007). The biggest MLP so far (Salakhutdinov & Hinton, 2007) also was pretrained 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 (Mohamed, Dahl, & Hinton, 2009).

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 (Bengio, Lamblin, Popovici, & Larochelle, 2006). Training them is hard, as backpropagated gradients quickly vanish exponentially in the number of layers (Hochreiter, 1991; Hochreiter, Bengio, Frasconi, & Schmidhuber, 2001; Hinton, 2007), just as the first recurrent neural networks (Hochreiter & Schmidhuber, 1997). Indeed, previous deep networks successfully trained with backpropagation (BP) either had few free parameters due to weight sharing (LeCun et al., 1998; Simard et al., 2003) or used unsupervised, layer-wise pretraining (Hinton & Salakhutdinov, 2006; Bengio et al., 2006; Ranzato, Poultney, Chopra, & LeCun, 2006). But is it really true that deep BP-MLPs do not work at all, or do they just need more training time? How can this be tested considering that online BP for hundreds or thousands of epochs on large MLPs may take weeks or months on standard serial computers? But can't one parallelize it? On computer clusters, this is hard due to communication latencies between individual computers. Parallelization across training cases and weight updates for mini-batches (Nair & Hinton, 2009) might alleviate this problem, but still leaves the task of parallelizing fully online BP. Only GPUs are capable of such finely grained parallelism. Multithreading on a multicore processor is not easy either. We may speed up BP using streaming single instruction, multiple data extensions either manually or by setting appropriate compiler flags. The maximum theoretical speed-up under single precision floating point, however, is four, which is not enough. And MNIST is large: its 60,000 images take almost 50 MB, too much to fit in the L2/L3 cache of any current processor. This requires continually accessing data in considerably slower RAM. To summarize, currently it is next to impossible to train big MLPs on CPUs.

We will show how to overcome all these problems by training large, deep MLPs on graphics cards.

2 Data

MNIST consists of two data sets: 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 online 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 (maximum 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 neural network input layer.

3 Architectures

We train five MLPs with two to nine hidden layers and varying numbers of hidden units. Mostly, but not always, the number of hidden units per layer decreases toward the output layer (see Table 1). There are 1.34 to 12.11 million free parameters (or weights, or synapses).

We use standard online BP (Russell & Norvig, 2002), 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 $B = 0.6666$ (LeCun et al., 1998).

4 Deforming Images to Get More Training Instances

So far, the best results on MNIST were obtained by deforming training images, thus greatly increasing their number. This allows for training networks with many weights, making them insensitive to in-class variability. We combine affine (rotation, scaling, and horizontal shearing) and elastic deformations, characterized by the following real-valued parameters:

- σ and α: For elastic distortions emulating uncontrolled oscillations of hand muscles (for details, see Simard et al., 2003).
- 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 $[1 - \gamma/100, 1 + \gamma/100]$.

At the beginning of every epoch, the entire 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 or sheared less ($\beta = 7.5^\circ$) than other digits ($\beta = 15.0^\circ$).

Table 1: Error Rates on MNIST Test Set.

ID	Architecture (Number of Neurons in Each Layer)	Test Error for Best Validation (%)	Best Test Error (%)	Simulation Time (h)	Weights (Millions)
1	1000, 500, 10	0.49	0.44	23.4	1.34
2	1500, 1000, 500, 10	0.46	0.40	44.2	3.26
3	2000, 1500, 1000, 500, 10	0.41	0.39	66.7	6.69
4	2500, 2000, 1500, 1000, 500, 10	0.35	0.32	114.5	12.11
5	9×1000, 10	0.44	0.43	107.7	8.86

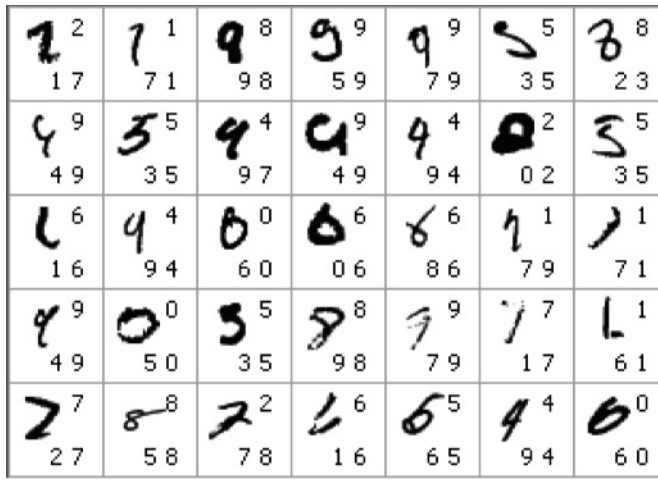

Figure 1: The 35 misclassified digits of the best network from Table 1, together with the two most likely predictions (bottom, from left to right) and the correct label according to MNIST (top, right).

5 Results

All simulations were performed on a computer with a Core2 Quad 9450 2.66 GHz processor, 3 GB of RAM, and a GTX280 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 40. Implementation details can be found in the appendix. We pick the trained MLP with the lowest validation error and evaluate it on the MNIST test set. Results are summarized in Table 1.

Most remarkable, the best network has an error rate of only 0.35% (35 out of 10,000 digits). This is significantly better than the best previously published results—0.39% by Ranzato et al. (2006) and 0.40% by Simard et al. (2003), both obtained by more complex methods. The 35 misclassified digits are shown in Figure 1. Many of them are ambiguous or uncharacteristic, with obviously missing parts or strange strokes. 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. 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 (see Table 1).

Networks with 12 million weights can successfully be trained by plain gradient descent to achieve test errors below 1% after 20 to 30 epochs in less than 2 hours of training. How can networks with so many parameters generalize well on the unseen test set? The answer is that the continual

deformations of the training set generate a virtually infinite supply of training examples, and the network rarely sees any training image twice.

6 Conclusion

In recent decades the amount of raw computing power per euro has grown by a factor of 100 to 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 40 times faster than standard microprocessors when it comes to training big and deep neural networks by the ancient algorithm, online backpropagation (weight update rate up to 5×10^9/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 pretraining, and combinations of machine learning classifiers, for example. 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.

Appendix: GPU Implementation

A.1 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 neural network implementations were developed when GPUs became faster than CPUs. Two-layer MLPs (Steinkraus, Buck, & Simard, 2005) and CNNs (Chellapilla, Puri, & Simard, 2006) have been previously implemented on GPUs. Although speed-ups 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 (Scherer & Behnke, 2009).

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 and CUDA, visit the NVIDIA Web site (NVIDIA, 2009). According to CUDA terminology, the CPU is called host and the graphics card device or GPU.

A.2 Deformations. It takes 93 CPU seconds to deform the 60,000 MNIST training images, most of them (87) for elastic distortions. Only the most time-consuming part of the latter—convolution with a gaussian kernel (Simard et al., 2003)—is ported to the GPU. The MNIST training set is split into 600

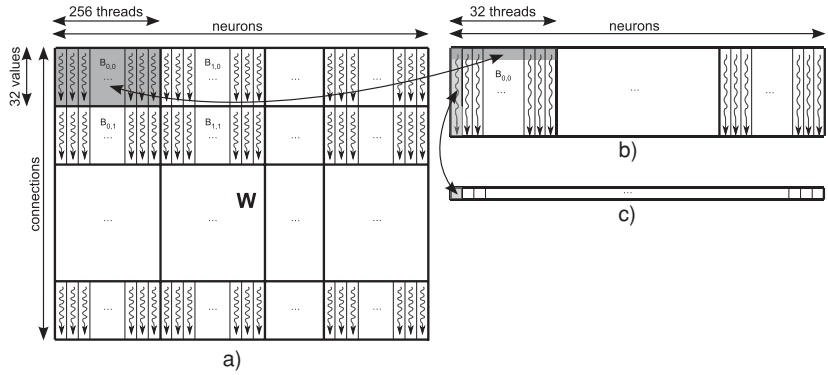

Figure 2: Forward propagation. (a) Mapping of kernel 1 grid onto the padded weight matrix. (b) Mapping the kernel 2 grid onto the partial dot products matrix. (c) Output of forward propagation.

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 (see algorithm 1).

Generating the elastic displacement field takes only 3 seconds. Deforming the whole training set is more than 10 times faster, taking 9 instead of the original 93 seconds. Further optimization would be possible by porting all deformations onto the GPU and using the hardware's interpolation capabilities to perform the final bilinear interpolation. We omitted this since deformations are already fast (deforming all images of one epoch takes only 5% to 15% of total computation time, depending on MLP size).

A.3 Training Algorithm. We closely follow the standard BP algorithm (Russell & Norvig, 2002), except that BP of deltas and weight updates are disentangled and performed sequentially. This allows for more parallelism within each routine.

A.3.1 Forward Propagation. The algorithm is divided into two kernels. The weight matrix W is partitioned as illustrated in Figure 2.

Each block of kernel 1 has 256 threads (see Figure 2a), each computing a partial dot product of 32 component vectors. The dot products are stored in a temporary matrix (see Figure 2b). This kernel has a very high throughput: average memory bandwidth is 115 GB/s. This is possible because many

Algorithm 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 compute 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;
  }
}
```

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 (see algorithm 2). The number of instructions is kept to a minimum by precomputing all common index parts.

The thread grid of kernel 2 (see Figure 2b) has only one row of blocks consisting of $warp$ threads, since each thread has to compute a complete dot product (see Figure 2c) and then pipe it into the activation function. This kernel (see algorithm 2) is inefficient for layers with fewer than 1024 incoming connections per neuron, especially for the last layer, which has only 10 neurons—one for each digit. That is, its grid will have only one block, occupying only 3% of the GTX280 GPU.

A.3.2 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 Ruetsch and Micikevicius (2009). Shared memory access is much faster without coalescing restrictions. Because we have to cope with layers of thousands of neurons, backpropagating deltas uses a reduction method implemented in two kernels communicating partial results via global memory (see algorithm 3).

The bidimensional grid of kernel 1 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 speed-ups. Before finishing, each thread writes its own partial dot product to global memory.

Kernel 2 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.

A.3.3 Weight Updating. Algorithm 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.

Algorithm 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
}
```

Algorithm 3: Backpropagating Delta 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 ...
        }
}
```

Algorithm 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;}}
}
```

Acknowledgments

Part of this work got started when D.C. was a Ph.D. student at University Politehnica of Timişoara. He thanks his Ph.D. advisor, Ştefan Holban, for his guidance and Răzvan Moşincat for providing a CPU framework for MNIST. This work was supported by Swiss CTI, Commission for Technology and Innovation, Project n. 9688.1 IFF: Intelligent Fill in Form, and by Lifeware S.A.

References

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. In B. Schölkopf, J. Platt, & T. Hoffman (Eds.), *Advances in neural information processing systems, 19* (pp. 153–160). Cambridge, MA: MIT Press.

Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*. N.p.

Decoste, D., & Schölkopf, B. (2002). Training invariant support vector machines. *Machine Learning*, *46*, 161–190.

Hinton, G. (2007). To recognize shapes, first learn to generate images. In P. Cisek, T. Drew, & J. Kalaska (Eds.), *Computational neuroscience: Theoretical insights into brain function*. Burlington, MA: Elsevier.

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, *313*, 504–507.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen.* Diploma thesis, Technische Universität München.

Hochreiter, S., Bengio, Y., Frasconi, P., & Schmidhuber, J. (2001). Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. In S. C. Kramer & J. F. Kolen (Eds.), *A field guide to dynamical recurrent neural networks*. Piscataway, NJ: IEEE Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term Memory. *Neural Computation*, *9*, 1735–1780.

Lauer, F., Suen, C., & Bloch, G. (2007). A trainable feature extractor for handwritten digit recognition. *Pattern Recognition*, *40*, 1816–1824.

LeCun, Y. (1985). Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva, Paris*, *85*, 599–604.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 309–318.

Mohamed, A., Dahl, G., & Hinton, G. E. (2009). Deep belief networks for phone recognition. In *Proc. of NIPS 2009 Workshop on Deep Learning for Speech Recognition and Related Applications*. N.p.

Nair, V., & Hinton, G. E. (2009). Implicit mixtures of restricted Boltzmann machines. In D. Koller, D. Schuurmans, Y. Bengio, & L. Bottou (Eds.), *Advances in neural information processing systems, 21.* Cambridge, MA: MIT Press.

NVIDIA. (2009). *NVIDIA CUDA: Reference manual* (Version 2.3).

Ranzato, M., Huang, F., Boureau, Y., & LeCun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'07)*. San Mateo, CA: IEEE Computer Society Press.

Ranzato, M., Poultney, C., Chopra, S., & LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. In B. Schölkopf, J. Platt, & T. Hoffman (Eds.), *Advances in neural information processing systems, 19*. Cambridge, MA: MIT Press.

Ruetsch, G., & Micikevicius, P. (2009). *Optimizing matrix transpose in CUDA* (Tech. Rep.). Santa Clara, CA: NVIDIA.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Parallel distributed processing*, Cambridge, MA: MIT Press.

Russell, S., & Norvig, P. (2002). *Artificial intelligence: A modern approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Salakhutdinov, R., & Hinton, G. (2007). Learning a nonlinear embedding by preserving class neighborhood structure. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*. San Francisco: Morgan Kaufmann.

Scherer, D., & Behnke, S. (2009). 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*. N.p.

Simard, P. Y., Steinkraus, D., & Platt, J.C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Intl. Conf. Document Analysis and Recognition* (pp. 958–962). San Mateo, CA: IEEE Computer Society Press.

Steinkraus, D., Buck, I., & Simard, P. Y. (2005). GPUs for machine learning algorithms. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition* (pp. 1115–1120). San Mateo, CA: IEEE Computer Society Press.

Werbos, P. J. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Unpublished doctoral dissertation, Harvard University.

Received February 19, 2010; accepted May 25, 2010.