Large-scale Object Recognition with CUDA-accelerated Hierarchical Neural Networks

Rafael Uetz and Sven Behnke
Autonomous Intelligent Systems Group
Institute of Computer Science VI
University of Bonn, Germany
Email: {uetz, behnke}@ais.uni-bonn.de

***Abstract*—Robust recognition of arbitrary object classes in natural visual scenes is an aspiring goal with numerous practical applications, for instance, in the area of autonomous robotics and autonomous vehicles. One obstacle on the way towards human-like recognition performance is the limitation of computational power, restricting the size of the training and testing dataset as well as the complexity of the object recognition system. In this work, we present a hierarchical, locally-connected neural network model that is well-suited for large-scale, high-performance object recognition. By using the NVIDIA CUDA framework, we create a massively parallel implementation of the model which is executed on a state-of-the-art graphics card. This implementation is up to 82 times faster than a single-core CPU version of the system. This significant gain in computational performance allows us to evaluate the model on a very large, realistic, and challenging set of natural images which we extracted from the LabelMe dataset. To compare our model to other approaches, we also evaluate the recognition performance using the well-known MNIST and NORB datasets, achieving a testing error rate of 0.76 % and 2.87 %, respectively.**

I. Introduction

Vision clearly is an indispensable sense for humans. Without visual perception, orientation and movement are hardly possible. Another crucial ability of the visual system is the robust localization and recognition of arbitrary objects within the environment. This is an extremely difficult task due to different angles of view, lighting conditions, and partial occlusions.

In order to cover these variations, it is essential to provide an object recognition system with enough training examples containing objects under different conditions. However, most datasets commonly used for evaluating object recognition models do not meet these requirements. Objects of Caltech 101 [1], for instance, are view-normalized, i.e., all instances of an object class are depicted in the same angle of view. Hence, only one specific view of each class is learned. The NORB dataset [2], on the other hand, accounts for this problem by including many different angles of view for each object class. However, all images are created artificially and do not show natural scenes. Models achieving a good recognition performance on these and similar datasets are therefore not guaranteed to also achieve good results in real-life applications (see [3] for further discussion on this topic). To overcome these limitations, we created a new, realistic dataset by extracting a large number of objects from the LabelMe dataset of natural images [4]. Our dataset is described in Section II.

Another important issue we address here is the computational time required to train an object recognition system with a large dataset. Some classifiers, such as Support Vector Machines (SVMs), may considerably limit the maximum size of the training dataset because of the quadratic optimization during the training process. Neural Networks (NNs), on the other hand, often exhibit a sublinear runtime in the number of training samples as well as a structure that is well-suited for massively parallel implementation (see [2] and [5] for a comparison of the training runtime of SVMs and Convolutional Neural Networks).

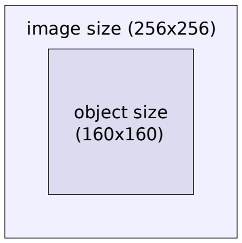
We propose a new neural network model which we call *Locally-connected Neural Pyramid* (*LCNP*). This model is optimized for large-scale, high-performance object recognition. Similar to models like the Neural Abstraction Pyramid [6] or Convolutional Neural Networks [7], we use a hierarchical structure of two-dimensional maps. This structure combines fast, inherently parallel processing with the ability to extract complex features at higher hierarchical layers. In contrast to other models, we propose a local connection structure where each neuron has its own weights, i.e., no weight sharing is applied. This allows for a fine-grained parallelization and gives very good recognition results. The model is described in more detail in Section III.

We implemented our model using the NVIDIA CUDA (Compute Unified Device Architecture) framework [8], allowing us to execute all time-critical functions in parallel on a state-of-the-art graphics card. The implementation is briefly described in Section IV.

The model is evaluated in Section V. We measure the speedup factor of the parallel implementation and the recognition performance by using the MNIST [9] and NORB normalized-uniform [2] datasets as well as our new dataset of natural images.

II. The LabelMe-12-50k dataset

Our main goal in creating the new dataset was to use natural images with a great variety of object instances, lighting conditions, and angles of view. We chose to extract all training and testing images from the LabelMe dataset [4], which consists of more than 175,000 natural images, most of them showing street and indoor scenes. Annotations (in the form of labeled

(a) 

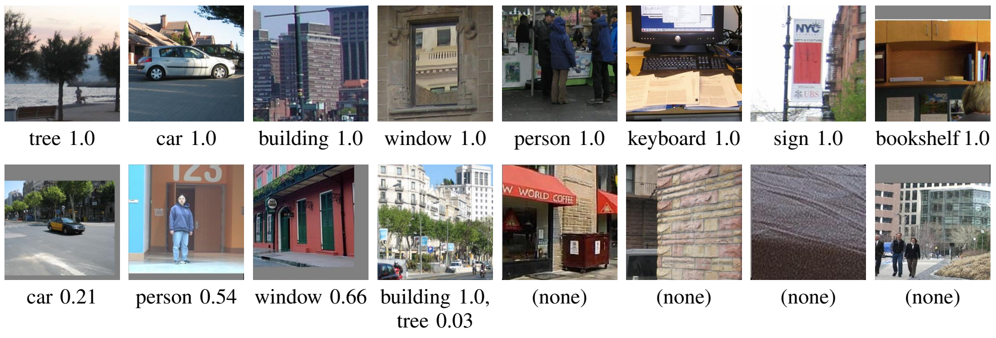
(b) 

Fig. 1. (a) Our dataset consists of JPEG images with a size of 256×256 pixels. Instances of the 12 object classes are resized to 160 pixels in their larger dimension, so there are at least 30 % of context information in each direction. (b) Some images of our training dataset. Class labels different from -1 are denoted below the images.

polygons) can be added by anyone using the online annotation tool on the project's web site *http://labelme.csail.mit.edu*.

We extracted 50,000 JPEG images in total, 40,000 of them for training and 10,000 for testing. We chose this size because larger training sets may be infeasible to train on current computers, mainly because of the restricted main memory. Potential future versions of the dataset could be much larger as the LabelMe dataset keeps growing.

There are images of 12 object classes and one additional "clutter" category in our dataset. Each image of the clutter category shows a randomly selected region of a randomly selected LabelMe image, whereas all the other images each show one centered object. The number of instances of each category is listed in Table I.

Our dataset was created as follows: We searched the whole LabelMe dataset for all instances of the 12 object classes using the MATLAB toolbox supplied with LabelMe. Instances smaller than 160 pixels in both dimensions (width and height) were discarded, the remaining instances were scaled down to have exactly 160 pixels in their larger dimension. We then randomly selected 25,000 of these instances (20,000 for the training set and 5,000 for the testing set). Each object was centered in a 256×256-pixel window by using the center of mass of its annotation polygon (see Figure 1 (a)). Areas of the window exceeding the image boundaries were colored gray. Figure 1 (b) shows the first 16 images of the training set.

The class labels of the created dataset were stored in a binary file as well as in a human-readable text file. For each image, these label files contain 12 successive float values (one for each class) with a range between -1 and 1. A value of 1 means that an instance of the corresponding class is exactly centered in the 160×160-pixel object area. This is obviously true for all extracted objects, so each of the object images (in contrast to the clutter images) has at least one class label with a value of 1. If one or more objects of a certain class overlap the object area of the current image, the label of the corresponding class is set to a value between 0 and 1, depending on the percentage of the overlapping and the size difference between the object and the object area of the image.

The dataset can be downloaded from our web site [10]. Its compressed size is about 460 MB. Note that much more main memory will be required (about 10 GB) if all JPEG images are loaded and uncompressed into memory.

#	object class	instances in training set	instances in testing set
1	person	4,855	1,180
2	car	3,829	974
3	building	2,085	531
4	window	4,097	1,028
5	tree	1,846	494
6	sign	954	249
7	door	830	178
8	bookshelf	391	100
9	chair	385	88
10	table	192	54
11	keyboard	324	75
12	head	212	49
	total no. of objects	20,000	5,000
	clutter images	20,000	5,000
	total no. of images	40,000	10,000

TABLE I
OBJECT CLASSES AND NUMBER OF INSTANCES IN THE LABELME-12-50K DATASET

III. THE LCNP MODEL

In this section, we describe our neural network model, which we call *Locally-connected Neural Pyramid* (*LCNP*). The following subsections describe the network structure, the neuron connections, the input encoding, and the training algorithm.

A. Network structure

The basic structure of our model is depicted in Figure 2. Similar to the *Neural Abstraction Pyramid* [6], there are $L \geq 1$ *regular layers* and one *output layer*. Each regular layer $l \in \{0, \ldots, L-1\}$ consists of at least one *map*, each map being a two-dimensional, square array of $N_l \times N_l$ neurons. The neuron model is the same as the one used for multi-layer perceptrons [11], i.e., each neuron has an output value called activation which is calculated by applying a non-linear function (we use the hyperbolic tangent) to a weighted sum of input values.

Maps of a specific layer all have the same size, whereas the size of maps in consecutive layers is halved each time so that

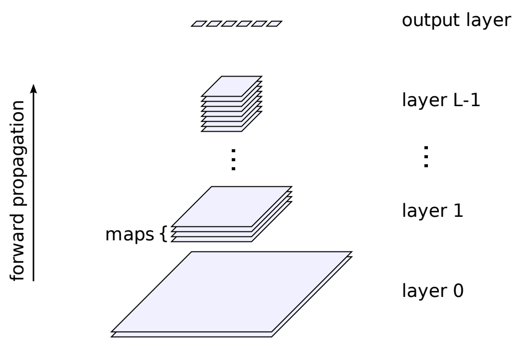


Fig. 2. Basic structure of our hierarchical neural network model. There are L regular layers and one output layer. Each regular layer consists of at least one map.

$N_l = \frac{1}{2}N_{l-1}$. The number of maps increases with each layer. This property ensures a decreasing spacial resolution while the number of features increases at the same time. Experiments have shown that doubling the number of maps in each layer seems to be a good trade-off between recognition performance and calculation speed.

In contrast to the regular layers, the output layer is a one-dimensional array of neurons, each representing one object class (*one-hot encoding*).

B. Neuron connections

Any two maps i and j of two consecutive regular layers l and $l+1$, $l \in \{0, 1, \ldots, L-2\}$, respectively, can potentially be connected by a *map connection*. A map connection is a local connection structure between two maps where each neuron of map j has connections to an adjacent set of neurons in map i, called the *receptive field* of the neuron in map j. We chose the size of the receptive field to be 4×4 neurons. As the maps of layer l are twice as large as the maps of layer $l+1$, the receptive fields overlap by $50\,\%$ in each direction (see Figure 3).

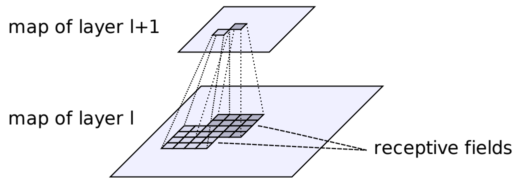


Fig. 3. Illustration of a map connection. Every neuron of the higher layer $l+1$ has local connections to neurons of the lower layer l.

The highest regular layer $L-1$ is fully connected to the output layer, i.e., each neuron of each map of layer $L-1$ is connected to each neuron of the output layer.

Unlike most other hierarchical neural network models, such as LeNet [7], our model does not employ weight sharing. That means each neuron's weights are independent of any other neuron's weights, resulting in a very large total number of free parameters. This design decision is motivated twofold: Firstly, the "blessing of dimensionality" gives hope to find good local minima when training the network via gradient descent algorithms. At the same time, the hierarchical, locally-connected network structure allows for a fast calculation compared to fully-connected structures like the multi-layer perceptron. Secondly, a simple reason to study local, non-shared weights is that this structure is massively used by the human brain [12]. To our knowledge, there is no work about hierarchical neural networks where such a tremendous amount of weights has been evaluated before.

C. Input encoding

Any map of each regular layer can be used as an *input map*. This is done by setting the activation values of its neurons to the desired input values.

To train and test the model with natural image datasets, the most obvious input method would be to use one input map for each of the red, green, and blue channels. However, such highly correlated values should be avoided as they may impair the training process [13]. Hence, we applied a principal component analysis (PCA) to a large sample of RGB values taken from all images of the LabelMe dataset. For the following measurements, we used the resulting three principal components to calculate three less correlated input channels from the RGB channels.

Additionally, four edge channels are calculated by convolving with the filters

$$\begin{aligned} &\text{(i)} \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix} \quad \text{(ii)} \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix} \\ &\text{(iii)} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{(iv)} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \end{aligned} \tag{1}$$

to the grayscale values of the input image's pixels. The absolute values of the result are then used as activation values for four input maps. They resemble edge detectors of different orientations.

Another significant property of our model is the use of input maps in *every* regular layer. We do this by first subsampling the original input image multiple times to the size of all regular layer's maps. In the second step, we calculate the described input channels from these subsampled images. An advantage of this method is that the edge filters on lower layers extract fine-grained edges while those of higher layers extract coarse-grained edges. Similarly, the PCA channels of lower layers represent fine details, whereas those of higher layers represent coarse features. This approach turned out to improve the recognition performance significantly.

Summing up, there are seven input maps per regular layer for datasets consisting of RGB images: Three input maps resembling the color information by using a linear combination (calculated by a PCA) of the RGB channels and four input maps resembling different edge detectors. As the described preprocessing does not require any non-local memory accesses, it is well-suited for a massively parallel implementation.

D. Training algorithm

The model is a pure feed-forward architecture. Learning is supervised and done via "plain-vanilla" backpropagation of error. We use a fixed learning rate $\eta_l = 0.01 \cdot 2^{L-l}$ for map connections between layer $l-1$ and l, $l \in \{1, \ldots, L-1\}$, and $\eta_o = 0.0001$ for map connections between layer $L-1$ and the output layer. We tried regularization methods such as weight decay, but these did not improve the results (neither recognition rate nor training speed).

All connection weights of the network are randomly initialized in the range $\pm\frac{1}{\sqrt{n}}$, where n is the number of inbound connections for a specific neuron.

IV. CUDA IMPLEMENTATION

In this section, a brief introduction of the CUDA framework is given, followed by a sketch of our parallel implementation.

A. The NVIDIA CUDA framework

CUDA (Compute Unified Device Architecture) is a framework which allows to develop C/C++ programs that execute specific functions (so-called *kernels*) on a CUDA-compatible graphics card in parallel. This graphics card is called *device* in this context. The computer on which the device is installed is called *host*. An instance of a kernel is called a *grid*. It consists of an arbitrary number of *blocks*. Each block consists of the same number of *threads*, which all execute the kernel's code in parallel.

During the execution of a grid, blocks and threads are mapped to the *multiprocessors* of the GPU (Graphics Processing Unit) and their *(scalar) processors*, respectively.

A kernel may use multiple kinds of memory: *Registers*, *shared memory*, *texture cache* and *constant cache* are fast, but small on-chip memory, whereas *device memory* is much larger (up to 2 GB), but has a drastically higher latency. Registers are accessible only from the current thread, shared memory is accessible from all threads of one block. Data transfer between blocks as well as between the host and the device can only be accomplished via the device memory. The hardware model of CUDA-compatible graphics cards is depicted in Figure 4.

When developing CUDA kernels, one has to consider the hardware constraints of the graphics card in order to obtain good performance. Compared to developing a conventional CPU program, CUDA kernels require by far more manual optimization and their performance strongly depends on memory access patterns and a fine-grained parallelism.

There are two major challenges when implementing a program using CUDA. Firstly, one must decide how to parallelize the original, sequential program, i.e., how to map loops to threads, blocks, and loops in the host function and/or in the kernel function. Secondly, the access patterns must be optimized for the different kinds of memory. Accesses to the device memory, for example, should be *coalesced* for all threads of a *half-warp* (that is a batch of 16 threads executed on one multiprocessor concurrently). Coalesced means that all threads access memory in a certain area whose size and start offset depend on the size of the data type that is requested.

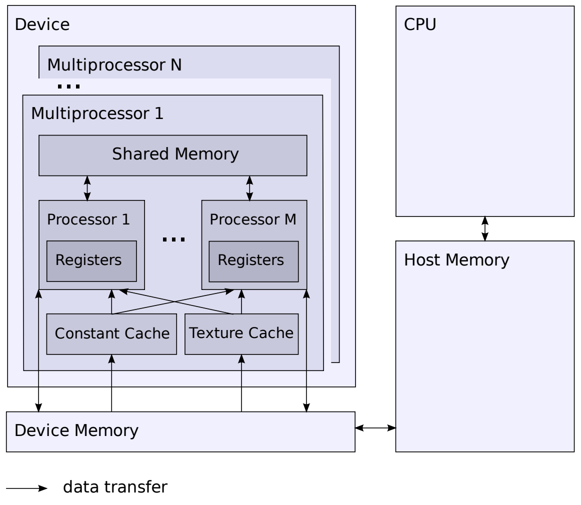


Fig. 4. Hardware structure of a CUDA graphics card. Adapted from [8].

B. Parallel implementation of our model

We implemented our model in C++ with CUDA under Linux. All performance-critical functions were programmed as a CPU- and a GPU version. This way, we were able to check the GPU version for correctness and to compare the execution speed of both versions directly. The CPU functions use one CPU core only and are not optimized manually (for example, by using the MMX or SSE instruction sets), so they only serve as a rough baseline for measuring the speedup factor of the parallel implementation. Compilation was done with `g++` and the `O2` flag set.

All attributes of the model's neurons are of the type `float`. Using `double` instead would drastically reduce the performance since twice the memory bandwidth would be required and the calculation would be slower.

The most performance-critical functions of our implementation are the forward and backward propagation steps. All other steps, like copying the input patterns from host to device, copying the results back, and calculating some statistics on the host require less than 10 % of the runtime in total. Hence, we only describe the implementation of the forward and backward propagation here.

Both functions process 16 patterns of an epoch concurrently, an epoch being one iteration of all training patterns. This method is called *mini-batch learning*. It is a trade-off between online learning (where all connection weights are updated after each pattern) and batch learning (where all patterns of the training set are processed before the connection weights are updated). Mini-batch learning generally requires less epochs for learning in comparison to batch learning and exhibits higher computational performance than online learning because the connection weights are saved less frequently.

We compared mini-batch learning (using batches of 16 patterns) and online learning directly and found both to be equally fast in terms of epochs required to obtain a certain testing error rate, but a lot faster in terms of runtime because of the better parallelization options.

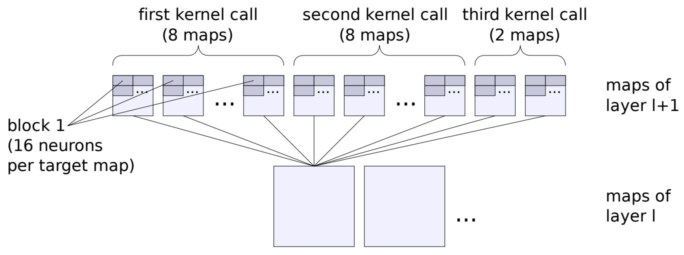


Fig. 5. Illustration of the forward propagation. For a source map of layer $l \in \{0, \dots, L-2\}$, up to eight target maps of layer $l+1$ are processed in parallel by one kernel call.

1) Forward propagation: The forward propagation processes all regular layers sequentially because all neuron activations of layer $l \in \{0, \dots, L-2\}$ are required to calculate the neuron activations of layer $l+1$. Within each loop iteration, there is another sequential loop over all maps of the current layer. For each map in layer l (*source map*), up to eight map connections to the maps of the next layer $l+1$ (*target maps*) are concurrently processed by one kernel call. The kernel processes these map connections for all 16 patterns of the current mini-batch in parallel.

Each block of a launched grid consists of 256 threads and represents the neurons of a specific area of all target maps that are currently processed, namely an array of 16×1 neurons (see Figure 5). Depending on the number of concurrently processed target maps, the threads perform different tasks: As there are 16 neuron positions and 16 patterns, each thread calculates the activation of exactly one neuron if only one target map is being processed by the kernel. If more target maps are being processed, the threads are subdivided by the number of target maps. The reduced number of threads for each target map is compensated by a loop within each thread so that each neuron's activation can be calculated.

This approach was chosen to minimize the memory latency: The more target maps are processed in parallel, the more often the same activation values of the source map are required. As we use texture functions to read the source map's activations more quickly, the on-chip texture cache is exploited better if the same values are read several times.

The forward propagation kernel uses multiple kinds of memory. As already stated above, the activations of the source map's neurons (*source neurons*) are read via the texture cache. This is because the read accesses cannot be coalesced due to the structure of the activations in memory (which is optimized for the backpropagation step). Using the texture cache is especially advantageous if coalesced reading is not possible but the access pattern is local.

Connection weights are read from the device memory and then written to the shared memory. It is a common principle to first load all required values into the fast on-chip memory and then use them several times during the execution of a block. The target neurons' activations, calculated from a weighted sum of the connection weights and the source activations, are finally written back to the device memory. All of these accesses are coalesced in order to utilize the full device memory bandwidth.

After all activations of the regular layers' maps have been calculated, the full connections between the maps of layer $L-1$ and the output layer are calculated via matrix multiplications. For this step, we use the CUDA CUBLAS library, more precisely, the `cudaSgemm` general matrix multiplication function. This function allows to multiply the weight changes by a learning rate and to add the result to the current weights in one step.

2) Backpropagation: Similar to the forward propagation, the backpropagation processes all layers sequentially, but in reverse direction. The weight changes of the full connections between the output layer and the regular layer $L-1$ are again calculated via matrix multiplications. After that, each map connection of each regular layer is processed by one kernel call. In contrast to the previously described forward propagation, one kernel processes exactly one instead of up to eight map connections. This is due to the restricted hardware resources, namely registers, since the backpropagation requires more data.

During each kernel call, each block of the grid represents an array of 16×16 neurons on the lower layer's map. Each of the 256 threads within a block represents one of these neurons. Most of the required values are loaded directly from the device memory and are stored into registers. When the calculations are done, these values are written back to the device memory. All read and write accesses are coalesced.

V. Results

In this section, we present our results concerning the speedup factor of the parallel CUDA implementation and the recognition performance using the MNIST, NORB normalized-uniform, and LabelMe-12-50k datasets.

A. Speedup factor of the CUDA implementation

For our measurements, we used an Intel Core i7 940 CPU and an NVIDIA GeForce GTX 285 graphics card on a Linux system with 12 GB of RAM. With the network structure described in the next subsection, we obtained a speedup factor of 43.5 compared to the CPU version of the system. Using other network structures, the speedup factor is between 7.7 and 82.2 for very small and very large networks, respectively. The number of connection weight updates per second (WUPS) is up to $7.98 \cdot 10^9$. This number was calculated as if online learning was used, so every weight update of every pattern was counted (though the changes are only applied once for each mini-batch). This kind of calculation allows to compare the WUPS of different batch sizes directly.

The absolute time for one training epoch of the LabelMe-12-50k dataset is 70.2 seconds when using the network structure described in the next subsection. As one epoch consists of 40,000 images, the network processes about 570 images per second in the training phase. In the recall phase, the network processes about 1,356 images per second.

B. Recognition performance

We tested our system with three datasets: (1) The MNIST dataset [9], which consists of 60,000 grayscale images (50,000 for training and 10,000 for testing). Each image shows one handwritten digit. (2) The NORB normalized-uniform dataset [2], which consists of stereoscopic grayscale images of 50 toys, belonging to 5 categories. Each of the 48,600 images (24,300 for training and 24,300 for testing) shows one toy under different lighting conditions, elevations and azimuths. (3) The LabelMe-12-50k dataset described in Section II.

We used almost the same network structure and exactly the same parameters for these datasets. The only difference was the number of input maps per layer, depending on the number of color and/or stereoscopic channels of the input images.

The network we used for the measurements has five regular layers with the dimensions 256×256, 128×128, ..., 16×16. As described in Section III-C, we used seven input maps in each layer for training and testing the LabelMe-12-50k dataset. For the MNIST dataset, five input maps per layer were used (one grayscale channel and four edge channels). Accordingly, we used 10 input maps per layer for the NORB dataset (one grayscale and four edge channels for each of the left and right channels of the stereoscopic image).

Additionally, there are 2^l non-input maps in each regular layer $l \in \{1, \dots, L-1\}$, which have map connections to each map of layer $l - 1$. All patterns were initially stretched to 256×256 pixels in order to fit the input maps in layer 0. Each training image was randomly shifted by $\pm 5\,\%$ of the map size during the training phase in order to improve generalization and avoid overtraining. Testing images were not shifted. The order of the training patterns was permuted in each epoch.

Dataset	Training error rate	Testing error rate
MNIST	0.03 %	0.76 %
NORB normalized-uniform	0.05 %	2.87 %
LabelMe-12-50k	3.77 %	16.27 %

TABLE II
TRAINING AND TESTING ERROR RATES

The results are shown in Table II. The training and testing error rates denote the percentage of incorrectly classified patterns of the whole training and testing set, respectively. To verify the correctness of the output during the LabelMe-12-50k testing phase, we first determined the output neuron having the largest activation value (*winning neuron*). If the winning neuron's activation exceeded a threshold of 0, the pattern was classified as the winning neuron's class, else it was classified as belonging to the clutter category.

All results were measured after training for 1000 epochs and are averaged over 10 epochs. However, similar recognition rates appeared much earlier. The testing error rate of LabelMe-12-50k, for instance, did not change significantly after having trained for 100 epochs. The testing error rate of MNIST was $3.86\,\%$ after one epoch and $< 1\,\%$ after about 35 epochs. For NORB, it was $18.35\,\%$ after one epoch and $< 5\,\%$ after about 20 epochs. We did not observe any significant overtraining.

VI. Conclusion

In this paper, we introduced a large, realistic dataset of natural images which was extracted from LabelMe [4]. Our dataset consists of 50,000 256×256-pixel images of natural scenes, showing objects of 12 classes with a great variety of object instances, lighting conditions, and angles of view. The dataset can be downloaded from our web site [10].

Furthermore, we presented a neural network model called Locally-connected Neural Pyramid (LCNP) and described its massively parallel implementation using the NVIDIA CUDA framework. The main features of our model are its hierarchical structure, the local connectivity without weight sharing, the use of subsampled inputs in all hierarchical layers, and the local preprocessing of the input patterns.

The recognition performance of our model is competitive with state-of-the-art approaches. For the MNIST dataset, most approaches achieving better results (see [9] for a comprehensive list) use sophisticated preprocessing algorithms and/or unsupervised pretraining. For the NORB dataset, our results are (to our knowledge) the best ones achieved so far. As the parallel implementation of our model runs extremely fast, it allows for very large-scale object recognition systems and an unprecedented size of training and testing datasets.

Future work will focus on training with larger datasets by keeping compressed JPEG images in the main memory and uncompressing them "on the fly" during the training phase. We are also planning to employ de-noising autoencoders for unsupervised pretraining and recurrent network structures for an iterative improvement of the output.

References

[1] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," *Computer Vision and Image Understanding*, vol. 106, no. 1, pp. 59–70, 2007.

[2] Y. LeCun, F. J. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Proceedings of CVPR04*. IEEE Press, 2004.

[3] N. Pinto, D. Cox, and J. DiCarlo, "Why is real-world visual object recognition hard," *PLoS Computational Biology*, vol. 4, no. 1, 2008.

[4] B. C. Russell, A. Torralba, K. Murphy, and W. T. Freeman, "LabelMe: a database and web-based tool for image annotation," *International Journal of Computer Vision*, vol. 77, no. 1–3, pp. 157–173, 2008.

[5] F.-J. Huang and Y. LeCun, "Large-Scale Learning with SVM and Convolutional Nets for Generic Object Categorization," in *Proc. CVPR'06*. IEEE Press, 2006.

[6] S. Behnke, *Hierarchical Neural Networks for Image Interpretation*, ser. Lecture Notes in Computer Science. Springer, 2003, vol. 2766.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] NVIDIA Corporation, *CUDA Programming Guide, version 2.2*, 2009.

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

[10] R. Uetz, "The LabelMe-12-50k dataset," http://www.ais.uni-bonn.de/download/datasets.html.

[11] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, 1958.

[12] E. R. Kandel, J. H. Schwartz, and T. M. Jessel, *Principles of Neural Science*, 4th ed. McGraw-Hill, 2000.

[13] Y. LeCun, L. Bottou, G. Orr, and K. Müller, "Efficient backprop," in *Neural Networks: Tricks of the trade*. Springer, 1998.