Flexible, High Performance Convolutional Neural Networks for Image Classification

Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, Jürgen Schmidhuber
IDSIA, USI and SUPSI
Galleria 2, 6928 Manno-Lugano, Switzerland
{dan,ueli,jonathan,luca,juergen}@idsia.ch

Abstract

We present a fast, fully parameterizable GPU implementation of Convolutional Neural Network variants. Our feature extractors are neither carefully designed nor pre-wired, but rather learned in a supervised way. Our deep hierarchical architectures achieve the best published results on benchmarks for object classification (NORB, CIFAR10) and handwritten digit recognition (MNIST), with error rates of 2.53%, 19.51%, 0.35%, respectively. Deep nets trained by simple back-propagation perform better than more shallow ones. Learning is surprisingly rapid. NORB is completely trained within five epochs. Test error rates on MNIST drop to 2.42%, 0.97% and 0.48% after 1, 3 and 17 epochs, respectively.

1 Introduction

The human visual system efficiently recognizes and localizes objects within cluttered scenes. For artificial systems, however, this is still difficult due to viewpoint-dependent object variability, and the high in-class variability of many object types. Deep hierarchical neural models roughly mimick the nature of mammalian visual cortex, and by community consensus are among the most promising architectures for such tasks. The most successful hierarchical object recognition systems all extract localized features from input images, convolving image patches with filters. Filter responses are then repeatedly sub-sampled and re-filtered, resulting in a deep feed-forward network architecture whose output feature vectors are eventually classified. One of the first hierarchical neural systems was the Neocognitron [Fukushima, 1980] which inspired many of the more recent variants.

Unsupervised learning methods applied to patches of natural images tend to produce localized filters that resemble off-center-on-surround filters, orientation-sensitive bar detectors, Gabor filters [Schmidhuber *et al.*, 1996; Olshausen and Field, 1997; Hoyer and Hyvärinen, 2000]. These findings in conjunction with experimental studies of the visual cortex justify the use of such filters in the so-called *standard model* for object recognition [Riesenhuber and Poggio, 1999; Serre *et al.*, 2007; Mutch and Lowe, 2008], whose filters are fixed, in contrast to those of Convolutional Neural Networks (CNNs) [LeCun *et al.*, 1998; Behnke, 2003; Simard *et al.*, 2003], whose weights (filters) are randomly initialized and changed in a supervised way using back-propagation (BP).

Despite the hardware progress of the past decades, computational speed is still a limiting factor for CNN architectures characterized by many building blocks typically set by trial and error. To systematically test the impact of various architectures on classification performance, we present a fast CNN implementation on Graphics Processing Units (GPUs). Previous GPU implementations of CNNs [Chellapilla *et al.*, 2006; Uetz and Behnke, 2009; Strigl *et al.*, 2010] were hard-coded to satisfy GPU hardware constraints or use general purpose libraries, whereas our implementation is flexible and fully online (i.e., weight updates after each image). A notable exception is [Jarrett *et al.*, 2009] who performed a thorough analysis of the influence of all building blocks of a multistage architecture on recognition performance. Our implementation allows for training large CNNs within days instead of months, such that we can investigate the influence of various structural parameters by exploring large parameter spaces [Pinto *et al.*, 2009] and performing error analysis on repeated experiments.

We evaluate various networks on the handwritten digit benchmark MNIST [LeCun *et al.*, 1998] and two image classification benchmarks: NORB [LeCun *et al.*, 2004] and CIFAR10 [Krizhevsky, 2009].

2 Convolutional neural networks

CNNs are hierarchical neural networks whose convolutional layers alternate with subsampling layers, reminiscent of simple and complex cells in the primary visual cortex [Wiesel and Hubel, 1959]. CNNs vary in how convolutional and subsampling layers are realized and how the nets are trained.

2.1 Image processing layer

The image processing layer is an optional pre-processing layer of predefined filters that are kept fixed during training. Thus additional information besides the raw input image can be provided to the network, such as edges and gradients. In particular, we find that a contrast-extracting layer [Fukushima, 2003] helps to improve the recognition rate for NORB.

2.2 Convolutional layer

A convolutional layer is parametrized by the size and the number of the maps, kernel sizes, skipping factors, and the connection table. Each layer has M maps of equal size (M_x, M_y). A kernel (blue rectangle in Fig 1) of size (K_x, K_y) is shifted over the valid region of the input image (i.e. the kernel has to be completely inside the image). The skipping factors S_x and S_y define how many pixels the filter/kernel skips in x- and y-direction between subsequent convolutions. The size of the output map is then defined as:

$$M_x^n = \frac{M_x^{n-1} - K_x^n}{S_x^n + 1} + 1; \quad M_y^n = \frac{M_y^{n-1} - K_y^n}{S_y^n + 1} + 1 \quad (1)$$

where index n indicates the layer. Each map in layer L^n is connected to at most M^{n-1} maps in layer L^{n-1}. Neurons of a given map share their weights but have different receptive fields.

2.3 Max-pooling layer

The biggest architectural difference between our implementation and the CNN of [LeCun *et al.*, 1998] is the use of a max-pooling layer instead of a sub-sampling layer. No such layer is used by [Simard *et al.*, 2003] who simply skips nearby pixels prior to convolution, instead of pooling or averaging. [Scherer *et al.*, 2010] found that max-pooling can lead to faster convergence, select superior invariant features, and improve generalization. A theoretical analysis of feature pooling in general and max-pooling in particular is given by [Boureau *et al.*, 2010]. The output of the max-pooling layer is given by the maximum activation over non-overlapping rectangular regions of size (K_x, K_y). Max-pooling enables position invariance over larger local regions and downsamples the input image by a factor of K_x and K_y along each direction.

2.4 Classification layer

Kernel sizes of convolutional filters and max-pooling rectangles as well as skipping factors are chosen such that either the output maps of the last convolutional layer are downsampled to 1 pixel per map, or a fully connected layer combines the outputs of the topmost convolutional layer into a 1D feature vector. The top layer is always fully connected, with one output unit per class label.

3 GPU implementation

The latest generation of NVIDIA GPUs, the 400 and 500 series (we use GTX 480 & GTX 580), has many advantages over older GPUs, most notably the presence of a R/W L2 global cache for device memory. This permits faster programs and simplifies writing the code. In fact, the corresponding transfer of complexity into hardware alleviates many software and optimization problems. Our experiments show that the CNN program becomes 2-3 times faster just by switching from GTX 285 to GTX 480.

Manual optimization of CUDA code is very time-consuming and error prone. We optimize for the new architecture, relying on the L2 cache for many of the device memory accesses, instead of manually writing code that uses textures and shared memory. Code obtained by this pragmatic strategy is fast enough. We use the following types of optimization: pre-computed expressions, unrolled loops within template kernels, strided matrices to obtain coalesced memory accesses and registers wherever possible. Additional manual optimizations are possible in case future image classification problems will require even more computing power.

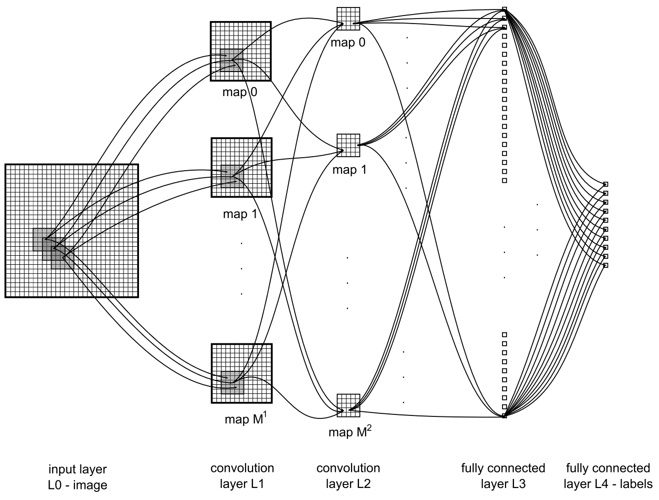

Figure 1: Architecture of a convolutional neural network with fully connected layers, kernel sizes of 5 x 5 and skipping factors of 1.

3.1 Data structures

Both outputs y and deltas δ of layer L^n are 2D strided. Their original size is $M_x \times MM_y$, but they are horizontally strided with a pitch of 32 floats (we use this stride for all 2D data), resulting in coalesced memory accesses. The vertical stride avoids additional bounding tests in CUDA kernels.

All connections between maps of consecutive layers L^{n-1} and L^n are stored in matrix C^n. Each row of C^n contains all connections that feed into a particular map in layer L^n. Because we aim for a flexible architecture with partially connected layers, in the first column we store the number of previous connections. This index is useful for Forward Propagation (FP) and Adjusting Weights (AW) CUDA kernels. The second column stores the number of connections, followed by corresponding indices of maps in L^{n-1} connected to the current map.

For BP and FP, analogous information about connections is needed. We therefore store backward connections in C_{BP}. AW requires a list of all map connections (see Subsection 3.4), stored as an array of map index pairs. Dealing with biases in BP kernel requires to know where the weights of particular connections start; this information is stored in a 2D array WIDX$_{BP}$ of size $M^n \times M^{n-1}$.

3.2 Forward propagation

A straightforward way of parallelizing FP is to assign a thread block to each map that has to be computed. For maps with more than 1024 neurons, the job is further split into smaller blocks by assigning a block to each line of the map, because the number of threads per block is limited (1024 for GTX 480). A one to one correspondence between threads and the map's neurons is assumed. Because of weight sharing, threads inside a block can access data in parallel, in particular the same weights and inputs from the previous layer. Each thread starts by initializing its sum with the bias, then loops over all map connections, convolving the appropriate patch of the input map with the corresponding kernel. The output is obtained by passing the sum through a scaled tanh activation function, and is then written to device memory.

3.3 Backward propagation

BP of deltas can be done in two ways: by pushing or by pulling. Pushing deltas means taking each delta from the current layer and computing the corresponding deltas for the previous layer. For an architecture with shared weights this has the disadvantage of being hard to code. Each delta from the current layer contributes to many deltas in the previous layer, which translates into a lot of programming. There are two ways of avoiding this: either writing partial deltas to a separated block of memory and then putting everything together by calling another kernel (slow because of a tremendous increase in the number of memory accesses, and the need of another kernel), or using atomic writes (to avoid data hazards) to update deltas (very slow because many writings are serialized). We implement pulling deltas, which has almost none of the above speed-limiting drawbacks, but is a bit more complicated.

The (uni- or bi-dimensional) thread grid assigns a (bi- or uni-dimensional) thread block to each map in the previous layer and a thread to each neuron in every map. Similar to FP, for maps with more than 1024 neurons, the 2D grid is further split into smaller 1D blocks by assigning a 2D block to each row of the map. Each thread computes the delta of its corresponding neuron by pulling deltas from the current layer. For every neuron in the previous layer we have to determine the list of neurons in the current layer which are connected to it. Let us consider neuron (i, j) from a map in layer L^{n-1}, and then assume that (x, y) are the coordinates of neurons in maps of L^n that contribute to the delta of neuron (i, j). The (x, y) neuron is connected to kernel size number neurons $(K_x \times K_y)$ from each connected map in the previous layer. The indices in L^{n-1} of the neurons connected through a kernel to the (x, y) neuron are:

$$\begin{aligned} x(S_x+1) \quad &\leq i \leq \quad x(S_x+1)+K_x-1, \\ y(S_y+1) \quad &\leq j \leq \quad y(S_y+1)+K_y-1. \end{aligned}$$

We can now compute the inequalities for (x, y):

$$\begin{aligned} \frac{i-K_x+1}{S_x+1} \quad &\leq x \leq \quad \frac{i}{S_x+1}, \\ \frac{j-K_y+1}{S_y+1} \quad &\leq y \leq \quad \frac{j}{S_y+1}. \end{aligned}$$

Because (x, y) has to be inside the map, the final inequalities are:

$$\max\left(\left\lceil \tfrac{i-K_x+1}{S_x+1} \right\rceil, 0\right) \quad \leq x \leq \quad \min\left(\left\lfloor \frac{i}{S_x+1} \right\rfloor, M_x-1\right),$$

$$\max\left(\left\lceil \tfrac{j-K_y+1}{S_y+1} \right\rceil, 0\right) \quad \leq y \leq \quad \min\left(\left\lfloor \frac{j}{S_y+1} \right\rfloor, M_y-1\right).$$

The above inequalities state that the delta of neuron (i, j) from L^{n-1} is computed from deltas of neurons in a rectangular area in maps of L^n (Fig. 2). After summing up the deltas, each thread multiplies the result by the derivative of the activation function.

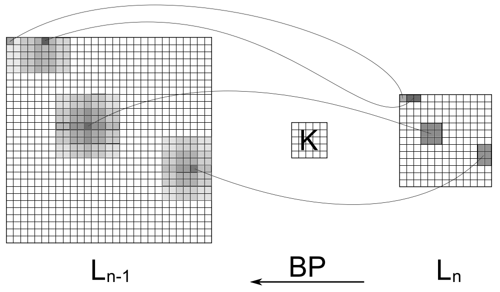

Figure 2: Back propagating deltas. A connection between two maps from two consecutive layers is displayed. The map in L^{n-1} has 29 x 29 neurons; the map in L^n has 13 x 13 neurons. They are linked through a 5 x 5 kernel K. Skipping factors of $S_x = 1$ and $S_y = 1$ are assumed. Arrows and colors depict the correspondence between neurons in L^{n-1} and their sources in L^n.

3.4 Adjusting weights

FP and BP have a grid over the list of maps, but the AW thread grid is over the list of kernels (filters) between maps of two consecutive layers. The 1D grid has a block for each connection between two maps. Thread blocks are 2D, with a corresponding thread for every kernel weight. The bias weight is included as an entire row of threads, thus requiring thread blocks to have $(K_x+1) \times K_y$ threads. Most of the time these additional K_y threads will wait, thread (0,0) being activated only for blocks that have to process the bias.

4 Experiments

We use a system with a Core i7-920 (2.66GHz), 12 GB DDR3 and four graphics cards: 2 x GTX 480 and 2 x GTX 580. The correctness of the CPU version is checked by comparing the analytical gradient with its finite difference approximation. On GPU this is not possible because all computations are performed with single precision floating point numbers. Hence the GPU implementation's correctness is checked by comparing its results to those of a randomly initialized net after training it for several epochs on the more accurate CPU version. Obtaining identical results after trillions of operations is a strong indication of correctness.

The implemented CNN's plain feed-forward architecture is trained using on-line gradient descent. All images from the training set are used for training and also for validation. If deformations are enabled, only the images from the training set will be deformed. Weights are initialized according to a uniform random distribution in the range $[-0.05, 0.05]$. Each neuron's activation function is a scaled hyperbolic tangent: $y(a) = 1.7159 \tanh(0.6666a)$ [LeCun *et al.*, 1998].

We pick the trained CNN with the lowest validation error, and evaluate it on the test set (Test for best Validation - TfbV). The best test error (bT) is also listed for all experiments. The reported computation times per epoch include training, validation and testing as well as all data transfers.

4.1 Experiments on MNIST

For the MNIST dataset the networks are trained on deformed images, continually generated in on-line fashion. Affine (translation, rotation, scaling, horizontal shearing) and elastic deformations [Simard *et al.*, 2003] are combined. We use a variable learning rate that shrinks by a multiplicative constant after each epoch, from 10^{-3} down to $3 \cdot 10^{-5}$ after 500 epochs.

Table 1: Error rates on MNIST test set for randomly connected CNNs with 2 to 6 convolutional layers with M Maps and an optional fully connected Layer with N Neurons, various kernel sizes and skipping factors were used.

#M, #N in Hidden Layers	TfbV [%]
20M-60M	1.02
20M-60M-150N	0.55
20M-60M-100M-150N	0.38
20M-40M-60M-80M-100M-120M-150N	**0.35**

Fully connected convolutional layers lead to an exploding number of network connections and weights, making training of big and deep CNNs for hundreds of epochs impractical even on GPUs. Partial connectivity alleviates this problem and is also biologically more plausible. We reduce the number of connections between convolutional layers in a random way [LeCun *et al.*, 1998; Jarrett *et al.*, 2009]. Table 1 lists results of various networks with 2 to 7 hidden layers with random connections. Additional layers result in better networks, the best one achieving a test error of 0.35% for best validation and a best test error of 0.27%. The best previous CNN result on MNIST is 0.40% [Simard *et al.*, 2003]. A 0.35% error rate was recently also obtained by a big, deep MLP [Ciresan *et al.*, 2010] with many more free parameters. Deeper nets require more computation time to complete an epoch, but we observe that they also need fewer epochs to achieve good test errors. The deepest CNN from Table 1 reaches 2.42%, 0.97% and 0.48% after one, three and seventeen epochs, respectively. On the other hand, the network with 4 instead of 7 hidden layers reaches 4.71%, 1.58%, 0.68% after one, three and seventeen epochs, achieving a test error below 0.50% after only 34 epochs. This shows once more that deep networks, contrary to common belief, can be trained successfully by back-propagation. Despite the numerous free parameters, deep networks seem to learn faster (better recognition rates after fewer epochs) than shallow ones.

4.2 Experiments on NORB

NORB contains stereo images of 3D objects. Hence there are two maps on the input layer. Rotation, scaling, shearing and elastic distortions seem to have a negative impact on generalization. These deformations improve recognition rates for digits that are intrinsically 2D [Ciresan *et al.*, 2010], but seem inadequate for 3D objects.

Initial experiments on NORB show that unlike with MNIST where we use deformations, the CNN needs only 3 to 6 epochs to reach zero validation error. This allows us to quickly run numerous repetitive experiments with huge networks with hundreds of maps per layer. We decided to use a CNN with five hidden layers: layer1, a convolutional layer with 300 maps, kernel size 6×6 and skipping factors 1×1; layer2, a max-pooling layer over a 2×2 region; layer3, a convolutional layer with 500 maps, kernel size 4×4, skipping factors 0×0; layer4, a max-pooling layer over a 4×4 region; layer5, a fully connected layer with 500 neurons. The learning rate is initialized by 0.001 and multiplied by 0.95 after every epoch.

Table 2 summarizes the results of four different experiments by switching on/off translation as well as the fixed image processing layer. We report the average error rate as well as the standard deviation of N independent runs with identical architectures but different weight initializations. For the first experiment without translation and no image processing (IP), an average test error rate of 7.86% is obtained. With additional translations of at most 5%, the average error rate drops to 4.71%, contradicting the common belief that CNNs are translation invariant. These results are on par or better than others in the literature: 5.90% error rate for a combination of CNNs and SVMs [LeCun *et al.*, 2004] and 5.20% error rate for restricted Boltzman machines [Nair and Hinton, 2009].

Table 2: Average error rates and standard deviations of N runs for a five hidden layer CNN on the NORB test set (see text for details).

trans. [%]	IP	TfbV [%]	runs	time/epoch [s]
0	no	7.86 ± 0.55	50	1141
5	no	4.71 ± 0.57	50	1563
0	yes	3.94 ± 0.48	50	1658
5	yes	**2.53** $\pm$ 0.40	100	2080

The best previously published result on NORB (2.87%) was obtained by a hierarchical neural network which to every convolutional layer provides a subsampled version plus edge information of the original image [Uetz and Behnke, 2009]. This motivates us to implement a pre-processing layer with fixed filters. We try simple edge masks (Sobel, Scharr), but find that a contrast-extraction layer [Fukushima, 2003] realized by mexican hat shaped filters of size 21×21 works best. We use two filters, one with a concentric on-center receptive field and one with a concentric off-center receptive field.

The first filter extracts positive contrast in brightness, whereas the latter extracts negative contrast. Each image from the original NORB is filtered, consequently the input of the CNN has six maps, the original image plus the positive and negative contrast for each of the two stereo channels. Using such a pre-processing layer results in lower average error rates, 3.94% without translation and 2.53% with translation.

This result improves the previous state of the art on NORB [Uetz and Behnke, 2009].

Experience with other image datasets tells us that NORB is unusual. The training set has only five instances per class. The resulting poor training set variability makes the nets learn quickly but generalize badly. NORB is the only dataset that profits from a fixed pre-processing layer in a substantial way. For MNIST and CIFAR10 such pre-processing has little or no effect. It is also worth noting that NORB's standard error rate deviation is bigger than CIFAR10's (see Tables 2 and 3). Identical nets with different initializations do not produce very consistent results. The best net had an error rate of 1.72%, the worst 3.69%.

4.3 Experiments on CIFAR 10

CIFAR10 is a collection of natural color images of 32x32 pixels. It contains 10 classes, each of them with 5000 samples in the training set and 1000 in the test set. The images greatly vary inside each class. They are not necessarily centered, may contain only parts of the object, and have varying backgrounds. All of this makes CIFAR10 the hardest problem addressed in this paper. The CNN has three maps, one for each color channel (RGB). The CIFAR10 images are relatively small in comparison to NORB's, and force us to use small kernels. The tested CNNs differ only in the number of maps per convolutional and max-pooling layer. All have eight hidden layers: layer1, a convolutional layer with 3×3 kernels and skipping factor of 0; layer2, a max-pooling layer over a 3×3 region; layer3, a convolutional layer with 3×3 kernels and skipping factors of 0×0; layer4, a max-pooling over a 2×2 region; layer5, a convolutional layer with 3×3 kernels and a skipping factors of 0×0; layer6, a max pooling layer over a 2×2 region; layer7, a fully connected layer with 300 neurons; layer8, a fully connected layer with 100 neurons.

Like for MNIST, the learning rate is initialized by 0.001 and multiplied by 0.993 after every epoch. Results in Table 3 show that without translation the error rate does not drop below 28%; adding edge information does not help at all. Translations have a very positive effect, decreasing the error rate to almost 20%. Contrast extraction filters are better than the Sobel/Scharr filters but still worse than no pre-processing layer at all. Despite some CNN-inherent translation invariance, additional training image translations cause better generalization; additional image processing proved useless though.

To see if bigger nets are better, we increase the number of maps per layer from 100 to 200, 300 and 400, respectively (last three rows in Tab. 3). Training time increases exponentially, but the test error decreases, reaching a minimum for nets with 300 maps per layer. Our 19.51% error rate is better than the previous state of the art for this dataset, 20.40% [Coates *et al.*, 2010] and 25.50% [Yu and Zhang, 2010]. Unlike [Coates *et al.*, 2010], however, we use the original images without any particular input normalization. Note that the error rate standard deviations are smaller than those obtained on NORB, that is, different initializations yield consistent results.

Table 3: Average error rates and standard deviations for N runs of an eight hidden layer CNN on the CIFAR10 test set (see text for details). The first five nets have 100 maps per convolutional and max-pooling layer, whereas the sixth, seventh and eighth have 200, 300 and 400 maps per hidden layer, respectively. IP - image processing layer: edge - 3×3 Sobel and Scharr filters; hat - 13×13 positive and negative contrast extraction filters.

trans. [%]	IP	TfbV [%]	runs	time/epoch [s]
0; 100M	no	28.87 ± 0.37	11	93
0; 100M	edge	29.11 ± 0.36	15	104
5; 100M	no	20.26 ± 0.21	11	111
5; 100M	edge	21.87 ± 0.57	5	120
5; 100M	hat	21.44 ± 0.44	4	136
5; 200M	no	19.90 ± 0.16	5	248
5; 300M	no	**19.51** ± 0.18	5	532
5; 400M	no	19.54 ± 0.16	5	875

4.4 Speedup factor of GPU code

The GPU code scales well with network size. For small nets the speedup is small (but still over 10) since they fit better inside the CPU cache, and GPU resources are underutilized. For huge nets (ex: Table 2) the GPU implementation is more than 60 times faster than a compiler-optimized CPU version. Given the flexibility of our GPU version, this is a significant speedup. One epoch takes 35 GPU minutes but more than 35 CPU hours.

5 Conclusion

We presented high-performance GPU-based CNN variants trained by on-line gradient descent. Principal advantages include state-of-the-art generalization capabilities, great flexibility and speed. All structural CNN parameters such as input image size, number of hidden layers, number of maps per layer, kernel sizes, skipping factors and connection tables are adaptable to any particular application. We applied our networks to benchmark datasets for digit recognition (MNIST), 3D object recognition (NORB), and natural images (CIFAR10). On MNIST the best network achieved a recognition test error rate of 0.35%, on NORB 2.53% and on CIFAR10 19.51%. Our results are raising the bars for all three benchmarks. Currently the particular CNN types discussed in this paper seem to be the best adaptive image recognizers, provided there is a labeled dataset of sufficient size. No unsupervised pretraining is required. Good results require big and deep but sparsely connected CNNs, computationally prohibitive on CPUs, but feasible on current GPUs, where our implementation is 10 to 60 times faster than a compiler-optimized CPU version.

Acknowledgment

This work was partially funded by the Swiss Commission for Technology and Innovation (CTI), Project n. 9688.1 IFF: Intelligent Fill in Form.

References

[Behnke, 2003] Sven Behnke. *Hierarchical Neural Networks for Image Interpretation*, volume 2766 of *Lecture Notes in Computer Science*. Springer, 2003.

[Boureau *et al.*, 2010] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A Theoretical Analysis of Feature Pooling in Visual Recognition. In *International Conference on Machine Learning*, 2010.

[Chellapilla *et al.*, 2006] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[Ciresan *et al.*, 2010] Dan C. Ciresan, Ueli Meier, Luca M. Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, 2010.

[Coates *et al.*, 2010] Adam Coates, Honglak Lee, and Andrew Ng. An analysis of single-layer networks in unsupervised feature learning. In *Advances in Neural Information Processing Systems*, 2010.

[Fukushima, 1980] Kunihiko Fukushima. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[Fukushima, 2003] Kunihiko Fukushima. Neocognitron for handwritten digit recognition. *Neurocomputing*, 51:161–180, 2003.

[Hoyer and Hyvärinen, 2000] Patrik O. Hoyer and Aapo Hyvärinen. Independent component analysis applied to feature extraction from colour and stero images. *Network: Computation in Neural Systems*, 11(3):191–210, 2000.

[Jarrett *et al.*, 2009] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision*, 2009.

[Krizhevsky, 2009] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Computer Science Department, University of Toronto, 2009.

[LeCun *et al.*, 1998] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[LeCun *et al.*, 2004] Yann LeCun, Fu-Jie Huang, and Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proc. of Computer Vision and Pattern Recognition Conference*, 2004.

[Mutch and Lowe, 2008] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *Int. J. Comput. Vision*, 56(6):503–511, 2008.

[Nair and Hinton, 2009] Vinod Nair and Geoffrey E. Hinton. 3d object recognition with deep belief nets. In *Advances in Neural Information Processing Systems*, 2009.

[Olshausen and Field, 1997] Bruno A. Olshausen and David J. Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, December 1997.

[Pinto *et al.*, 2009] Nicolas Pinto, David Doukhan, James J. DiCarlo, and David D Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS computational biology*, 5(11):e1000579, November 2009.

[Riesenhuber and Poggio, 1999] Maximiliam Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nat. Neurosci.*, 2(11):1019–1025, 1999.

[Scherer *et al.*, 2010] Dominik Scherer, Adreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*, 2010.

[Schmidhuber *et al.*, 1996] J. Schmidhuber, M. Eldracher, and B. Foltin. Semilinear predictability minimization produces well-known feature detectors. *Neural Computation*, 8(4):773–786, 1996.

[Serre *et al.*, 2007] Thomas Serre, Lior Wolf, and Tomaso Poggio. Object recognition with features inspired by visual cortex. In *Proc. of Computer Vision and Pattern Recognition Conference*, 2007.

[Simard *et al.*, 2003] P.Y. Simard, D. Steinkraus, and J.C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Seventh International Conference on Document Analysis and Recognition*, 2003.

[Strigl *et al.*, 2010] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. Performance and scalability of GPU-based convolutional neural networks. In *18th Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 2010.

[Uetz and Behnke, 2009] Rafael Uetz and Sven Behnke. Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, 2009.

[Wiesel and Hubel, 1959] D. H. Wiesel and T. N. Hubel. Receptive fields of single neurones in the cat's striate cortex. *J. Physiol.*, 148:574–591, 1959.

[Yu and Zhang, 2010] Kai Yu and Tong Zhang. Improved local coordinate coding using local tangents. In *Proceedings of the International Conference on Machine Learning*, 2010.